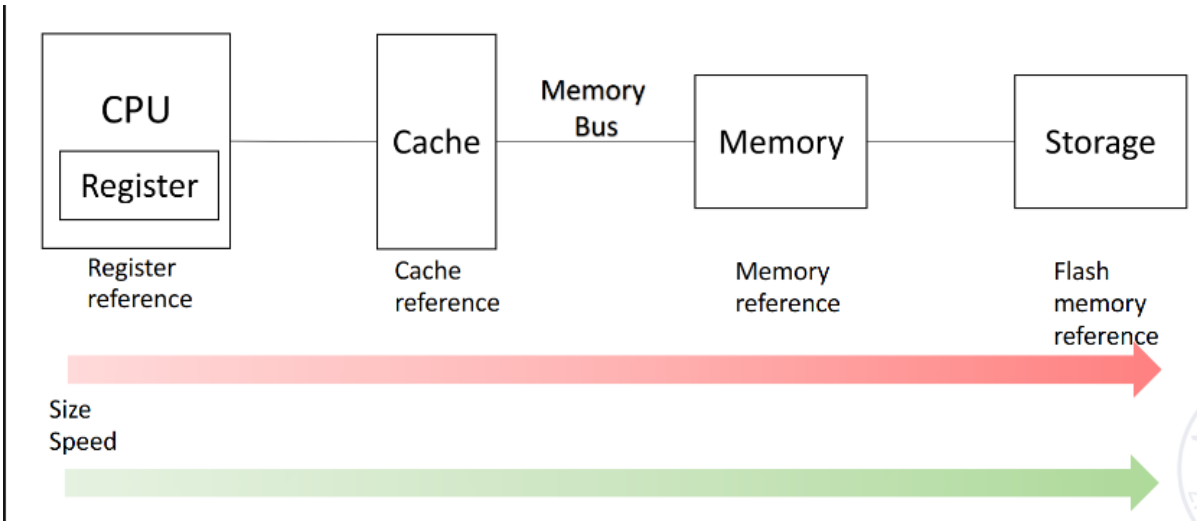
Lab6: Cache 设计和实现

3220106039 李瀚轩

一、操作方法与实验步骤

1.1 Cache 的原理

CPU 访问内存的速度是比较慢的，但现实中我们会有很多的读写操作，这造成了性能的瓶颈。因此我们引入 cache, 将数据缓存在 CPU 附件，利用数据访问的时间和空间的局部性原理来减少 CPU 访问内存的次数，提高性能。



本次实验我实现了一个使用二路组相联策略的 cache, 使用 write back 和 write allocate 策略，并采取 LRU 替换策略决定 cache block 的替换。

1.1.1 set associative

内存中的一块可以映射到的位置不止一个，同一个内存块可以映射到的所有位置称为一个组。如下图是一个四路组相联的 Cache 的原理图。同一个内存块可以存在一个组中的四个 Cache line 中。对于映射到相同位置的不同内存不必要将原有的数据覆盖，减少了 Cache miss 的概率。

1.1.2 Write back 和 Write allocate

当 CPU 需要将数据写入内存时，分为两种情况，即 write hit 和 write miss。对于 write hit 的情形，我们采取 write back 策略，即将数据写到 cache 中，此时并不写回内存，而是等到该 block 将被替换出去的时候写回内存。对于 write miss 的情况，我们从内存中将对应的数据块先搬到 cache 中，再执行写操作。

1.1.3 LRU 替换策略

LRU 即 Least Recently Used，当一个 set 里的 block 全都满了，需要替换时，我们将访问时间最早的那一块替换出去。

1.2 Cache 的实现

首先我们需要确定每个 block 需要存储数据的 bit 数目。

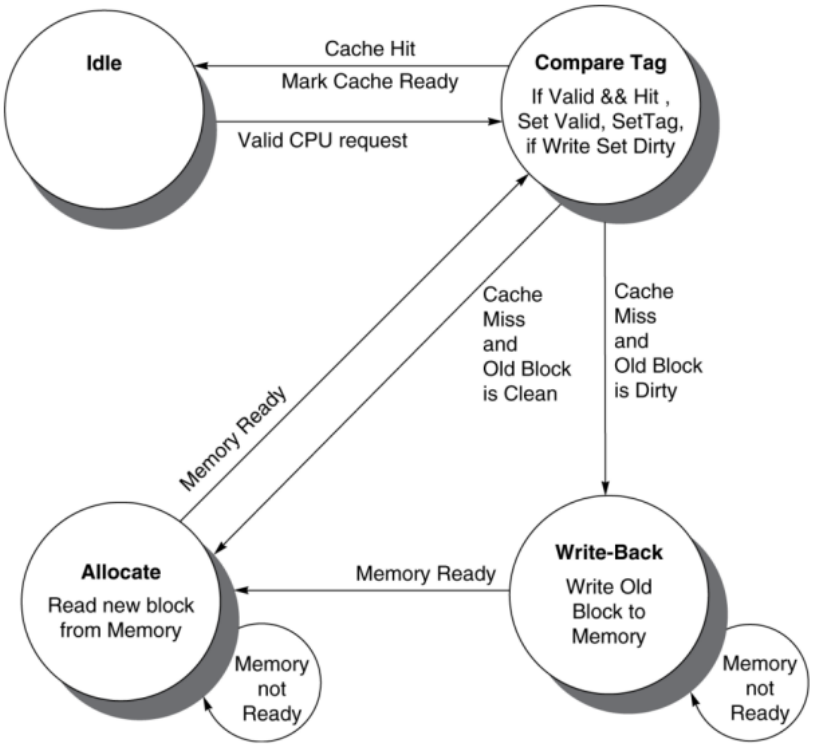
我设计的 cache 总共有 32 个 set, 每个 set 里面包含 2 个 block, 一个 block 含有一个 word。我们知道 cache 具有 byte_offset + index + tag 这几个基本概念。由于每个 block 仅有一个 word, 因此这里就没有 byte_offset。总共 32 个 set, 因此是 5 位 index。tag 的位数就是地址位数减去其余两个量的位数，也就是 32 - 5 = 27 位。除此之外，我们需要一个 valid-bit 代表这个 block 目前是有效的，即可以进行读写操作；一个 dirty-bit 标志该 block 的数据被更改了，在替换出去的时候需要将数据写到内存中；一个 LRU-bit 来指示该 block 是否被访问，如果被最新访问则置为 1。

因此可以得到 cache 中每一个 set 的每一个 block 的大小：32(数据) + 27(tag) + 1 + 1 + 1 = 62 bit；

接着，我们就可以进行 cache 逻辑的设计与实现。我将整个 cache 定义了四种状态：

- IDLE: 初始默认状态。如果 CPU 上层发出 Read 或者 Write 信号，则进入 compare_tag 状态。
- COMPARE_TAG：判断读写是 Hit 还是 Miss。如果 Hit 则正常访问 cache, 对于 miss 的情况，如果是脏页的话则进入 write back 状态将脏页的数据写回，如果不是脏页的话则进入 ALLOCATE 状态从内存中将对应的数据页拿到 cache 中。
- ALLOCATE：将内存对应的 block 搬到 cache 中，倘若 cache 对应的 set 两个 block 都满了，则根据 LRU 替换策略将其中一个替换出去。
- WRITE_BACK：将脏页的数据写回内存的对应位置中。

状态图如下图所示：



至此 cache 的基本实现完成，代码如下：

```
module cache(
    input clk,
    input rst,
    input [31:0] cpu_addr,
    input [31:0] write_data,
    input [31:0] block_data, // from memory
    input Read, //from cpu
```

```
    input Write, // from cpu
    input ack, // memory ack
    output [31:0] data, // 读出来的data
    output [31:0] mem_data, //write-back data
    output [31:0] mem_addr,
    output MemRead, //Cache miss: memory read
    output MemWrite, // Cache miss: memory write
    output ready // cache 读写完成
    );

    `define IDLE 2'd0
    `define COMPARE_TAG 2'd1
    `define ALLOCATE 2'd2
    `define WRITE_BACK 2'd3

    // one word per block (32-bit) + (32 sets, 5 index bits) 27 tag bits
    //+1 valid bit + 1 dirty bit + 1 LRU bit
    reg [61:0] cache_data[31:0][1:0]; // 2-set associative cache

    wire [4:0] index = cpu_addr[4:0];
    wire [26:0] tag = cpu_addr[31:5];
    reg [1:0] state;

    reg MemRead_out;
    reg MemWrite_out;
    reg [31:0] data_r;
    reg [31:0] mem_data_r;
    reg [31:0] mem_addr_r;
    reg ready_r;
    integer i;

    always @(posedge clk or posedge rst) begin
        if(rst) begin
            for(i = 0; i < 32; i = i + 1) begin
                cache_data[i][0] <= 0;
                cache_data[i][1] <= 0;
            end
            state <= `IDLE;
            ready_r <= 0;
        end
        else begin
            case(state)
                `IDLE: begin
                    MemRead_out <= 0;
                    MemWrite_out <= 0;
                    ready_r <= 0;
                    if(Read || Write) begin
                        state <= `COMPARE_TAG;
                    end
                    else begin
                        state <= `IDLE;
                    end
                end

                `COMPARE_TAG: begin
```

```
                    if(cache_data[index][0][61] == 1'b1 && cache_data[index][0][58:32]
== tag) begin
                        if(Write) begin
                            cache_data[index][0][31:0] <= write_data;
                            cache_data[index][0][60] <= 1'b1;
                            cache_data[index][0][59] <= 1'b1;
                        end
                        else begin
                            data_r <= cache_data[index][0][31:0];
                        end
                        state <= `IDLE;
                        ready_r <= 1;
                    end
                    else if(cache_data[index][1][61] == 1'b1 && cache_data[index][1]
[58:32] == tag) begin
                        if(Write) begin
                            cache_data[index][1][31:0] <= write_data;
                            cache_data[index][1][60] <= 1'b1;
                            cache_data[index][1][59] <= 1'b1;
                        end
                        else begin
                            data_r <= cache_data[index][1][31:0];
                        end
                        state <= `IDLE;
                        ready_r <= 1;
                    end
                    else begin
                        if(cache_data[index][0][60] == 1 || cache_data[index][1][60] ==
1) begin //dirty, need write back
                            state <= `WRITE_BACK;
                            MemWrite_out <= 1;
                            MemRead_out <= 0;
                        end
                        else begin
                            state <= `ALLOCATE;
                            mem_addr_r <= cpu_addr;
                            MemRead_out <= 1;
                            MemWrite_out <= 0;
                        end
                    end
                end

                `ALLOCATE: begin
                    mem_addr_r <= cpu_addr;
                    if(ack) begin
                        if(cache_data[index][0][59] == 1) begin
                            cache_data[index][0][59] <= 1'b0;
                            cache_data[index][1][59] <= 1'b1;
                            cache_data[index][1][61] <= 1'b1;
                            cache_data[index][1][60] <= 1'b0;
                            cache_data[index][1][58:32] <= tag;
                            cache_data[index][1][31:0] <= block_data;
                        end
                        else begin
                            cache_data[index][1][59] <= 1'b0;
                            cache_data[index][0][59] <= 1'b1;
```

```
                        cache_data[index][0][61] <= 1'b1;
                        cache_data[index][0][60] <= 1'b0;
                        cache_data[index][0][58:32] <= tag;
                        cache_data[index][0][31:0] <= block_data;
                    end
                    state <= `COMPARE_TAG;
                end
                else begin
                    state <= `ALLOCATE;
                end
            end

            `WRITE_BACK: begin
                if(ack) begin
                    MemWrite_out <= 1;
                    if(cache_data[index][0][60] == 1) begin
                        mem_data_r <= cache_data[index][0][31:0];
                        cache_data[index][0][60] <= 0;
                        mem_addr_r <= {cache_data[index][0][58:32], index};
                        state <= `ALLOCATE;
                    end
                    else begin
                        mem_data_r <= cache_data[index][1][31:0];
                        mem_addr_r <= {cache_data[index][1][58:32], index};
                        cache_data[index][1][60] <= 0;
                        state <= `ALLOCATE;
                    end
                end
                else begin
                    state <= `WRITE_BACK;
                end
            end
        endcase
    end
end

assign MemRead = MemRead_out;
assign MemWrite = MemWrite_out;
assign data = data_r;
assign mem_data = mem_data_r;
assign mem_addr = mem_addr_r;
assign ready = ready_r;

endmodule
```

需要注意的是，本次实验的 cache 是需要适配框架中的数据内存的，因此我们需要传出内存的使能信号、访问内存的地址，以及要写入内存的数据。并且由于数据内存的读写并不是一个时钟周期就能完成的，因此我们 cache 的相关操作需要等内存操作完之后，即 ack 信号值为 1 时才能进行。

二、实验结果与分析

如下是我设计的仿真代码，包含了 read/write miss/hit 的所有情况，以及验证了 write back, write allocate 和 LRU 替换策略的正确性：

```
module cache_sim(

    );
    reg clk;
    reg rst;
    reg [31:0] cpu_addr;
    reg [31:0] write_data;
    reg Read;
    reg Write;
    wire [31:0] data;
    wire ready;

    wire [31:0] mem_data;
    wire [31:0] block_data;
    wire [31:0] mem_addr;
    wire MemRead;
    wire MemWrite;
    wire ack;


    cache m0(
        .clk(clk),
        .rst(rst),
        .cpu_addr(cpu_addr),
        .write_data(write_data),
        .block_data(block_data),
        .Read(Read),
        .Write(Write),
        .ack(ack),
        .data(data),
        .mem_data(mem_data),
        .mem_addr(mem_addr),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .ready(ready)
    );

    DMem m1(
        .clk(clk),
        .rst(rst),
        .wen(MemWrite),
        .ren(MemRead),
        .addr(mem_addr),
        .i_data(mem_data),
        .o_data(block_data),
        .ack(ack)
    );

    initial begin
        rst = 1'b1;
```

```
        Read = 0;
        Write = 0;
        #100;
        //Read miss
        rst = 0;
        cpu_addr = 32'h00000000;
        Read = 1; Write = 0;
        #100;
        //Read miss
        cpu_addr = 32'h00000004;
        Read = 1;
        Write = 0;
        #100;
        //Read hit
        cpu_addr = 32'h00000000;
        #100;
        cpu_addr = 32'h00000004;
        #100;
        //Write hit
        Read = 0;
        Write = 1;
        cpu_addr = 32'h00000000;
        write_data = 32'hAAAAAAAA;
        #100;
        //Read hit
        Read = 1;
        Write = 0;
        cpu_addr = 32'h00000000;
        #100;
        //Write hit
        Read = 0;
        Write = 1;
        cpu_addr = 32'h00000000;
        write_data = 32'hBBBBBBBB;
        #100;
        //Read Miss
        Read = 1;
        Write = 0;
        cpu_addr = 32'h00000020;
        #200;
        //Read Hit
        cpu_addr = 32'h00000000;
        #100;
        cpu_addr = 32'h00000020;
        #100;
        //Write back
        Read = 1;
        Write = 0;
        cpu_addr = 32'h00000060;
        #100;
        Read = 1;
        Write = 0;
        cpu_addr = 32'h00000000;
        #100;
        //Write miss
        Read = 0;
```

```
        Write = 1;
        cpu_addr = 32'h00000008;
        write_data = 32'hCCCCCCCC;
        #100;
        //Read hit
        Read = 1;
        Write = 0;
        cpu_addr = 32'h00000008;
        #100;

    end

    always begin
        clk <= 1'b1; #5;
        clk <= 1'b0; #5;
    end

endmodule
```

首先访问地址为 `00000000` 的数据，此时为 read miss, 则将内存地址为 `00000000` 的数据搬到 cache 的第 0 个 set 中，并将对应的 valid bit 设置为 1。其次访问地址 `0x00000004` 同样也是 read miss 的情况。接下依次访问地址 `00000000` 和 `00000004` ，此时为 read hit, 直接从 cache 中读取数据输出。

接下来将数据 `AAAAAAAA` 写到地址 `0x00000000` ，由于之前 read 的时候将这块地址的数据搬到了 cache 中，因此此时 write hit, 但由于修改了该地址的数据值，因此此时需要将 dirty-bit 设置为 1。接下来验证该数据被成功写入，即读取地址为 `0x00000000` 的数据，此时为 read hit, 成功读取数据 `AAAAAAAA` 。接下来向 `0x00000000` 的地址写入数据 `0xBBBBBBBB` ，此时同样为 write hit，并且标志为 dirty bit, 之后需要通过 write back 策略写回内存。

接下来验证，write back 和 LRU 替换的正确性。首先 Read 地址为 `0x00000020` 的数据，此时 read miss, 对应的数据被搬到 cache 中 index 为 0 的 set 里，此时该 set 的数据有两个： `0x0000000` 的数据以及 `0x00000020` 的数据。此时 `0x00000020` 的数据 LRU bit 被标为 1，另一个 block 则被标为 0。接下来我们再读取 `0x00000060` 地址的数据，此时该地址也会被映射到 index 为 0 的 set 里，但是该 set 目前两个 block 已经满了，我们需要根据 LRU 替换策略替换掉一个 block, 由于之前 `0x00000020` 的 block 被标为 1，我们替换掉另一个 block，即 `0x00000000` 。接下来通过 Read `0x00000000` 的数据来判断是否成功写回内存，可以验证到读到了数据 `0xbbbbbbbb` 的数据。

最后验证 write miss 的情况，向 `0x00000008` 的地址写入数据 `0xcccccccc` , 此时为 write miss,首先将对应的 Block 搬到 cache 中，然后写入。最后访问该地址的数据，此时为 read hit, 也可以验证读到了正确的数据，验证了 write allocate 的正确性。

至此我们的测试结果全部符合预期，仿真波形已经在验收时得到确认，可以验证 cache 实现的正确性。

三、思考题

> 我们可以使用一个 Cache Line 中存放多个 word 的策略，从而利用数据访问的空间局部性，但是使用该策略的 Cache 性能一定会比一个 Cache Line 存放一个 word 的 Cache 好么？会不会出现性能更差的情况?

1. 如果我们的程序片段不具有良好的空间局部性，一个 cache line 存放多个 words 会可能会使得数据变的很分散，cache 的命中率可能会降低，导致性能变差。
2. 实际的 cache 中，整个 cache 的存储空间是固定的，因此如果一个 cache line 存放多个 word, 可能会导致整个 cache 的 set 数变少，这样可能会增加访存的冲突，导致接下来可能访问的数据因冲突被替换写回，导致性能变差。