《数据库系统》实验报告

年级、专业、班级	2021 级计算机科学与技术(卓越)02 班	**姓名**	文红兵
实验题目	Cache 实验		
实验时间	2022.5.26	**实验地点**	DS1410
实验成绩		**实验性质**	□验证性 □设计性 ☑综合性
评价教师签名： □算法/实验过程正确； □源程序/实验内容提交； □程序结构/实验步骤合理; □实验结果正确； □语法、语义正确； □报告规范; 其他: 评价教师:			

报告完成时间: 2023 年 5 月 27 日

1 实验目的

1. 加深对 Cache 原理的理解
2. 通过使用 verilog 实现 Cache，加深对状态机的理解

2 实验项目内容

1. 最低要求：参考指导书中直接映射写直达 Cache 的实现，实现写回策略的 Cache
2. 替换实验环境中的 Cache 模块，并通过仿真测试
3. [选做][较高要求] 性能优化，实现 2 路组相联的 Cache
4. [选做][更高要求] 性能优化，使用伪 LRU 等替换策略实现 4 路以上组相联的 Cache

3 实验过程或者算法

```
//Cache配置
parameter  INDEX_WIDTH  = 10, OFFSET_WIDTH = 2;
localparam TAG_WIDTH    = 32 - INDEX_WIDTH - OFFSET_WIDTH;
localparam CACHE_DEEPTH = 1 << INDEX_WIDTH;

//Cache存储单元
reg                 cache_valid [CACHE_DEEPTH - 1 : 0];
reg                 cache_dirty [CACHE_DEEPTH - 1 : 0];
reg [TAG_WIDTH-1:0] cache_tag   [CACHE_DEEPTH - 1 : 0];
reg [31:0]          cache_block [CACHE_DEEPTH - 1 : 0];

//访问地址分解
wire [OFFSET_WIDTH-1:0] offset;
wire [INDEX_WIDTH-1:0] index;
wire [TAG_WIDTH-1:0] tag;

assign offset = cpu_data_addr[OFFSET_WIDTH - 1 : 0];
assign index = cpu_data_addr[INDEX_WIDTH + OFFSET_WIDTH - 1 : OFFSET_WIDTH];
assign tag = cpu_data_addr[31 : INDEX_WIDTH + OFFSET_WIDTH];
```

图 1: Cache 配置

这段代码定义了 Cache 的配置和存储单元，以及实现了访问地址分解的功能，将地址分解成 tag，index 和 offset。

```
//访问Cache line
wire c_valid;
wire c_dirty;
wire [TAG_WIDTH-1:0] c_tag;
wire [31:0] c_block;

assign c_valid = cache_valid[index];
assign c_dirty = cache_dirty[index];
assign c_tag   = cache_tag  [index];
assign c_block = cache_block[index];

//判断是否命中
wire   hit, miss, dirty, clean;
assign hit = c_valid & (c_tag == tag);  //cache line的valid位为1，且tag与地址中tag相等
assign miss = ~hit;
assign dirty = c_valid & c_dirty;
assign clean = ~dirty;

//读或写
wire read, write;
assign write = cpu_data_wr;
assign read = ~write;
```

图 2: 命中判断

这段代码实现了命中判断的功能，根据当前索引 index 和访问地址中的偏移量、索引和标记，访问 Cache line，并根据有效位和标记判断是否命中 Cache 和是否脏位有效

```
//FSM
parameter IDLE = 2'b00, RM = 2'b01, WM = 2'b11;
reg [1:0] state;
always @(posedge clk) begin
    if(rst) begin
        state <= IDLE;
    end
    else begin
        case(state)
            IDLE:   state <= cpu_data_req & hit ? IDLE :
                             cpu_data_req & miss & clean ? RM :
                             cpu_data_req & miss & dirty ? WM : IDLE;
            RM:     state <= cache_data_data_ok ? IDLE : RM;
            WM:     state <= cache_data_data_ok ? RM   : WM;
        endcase
    end
end
```

图 3: FSM

这段代码实现了有限状态机的功能，三个状态 $IDLE, RM, WM$，分别表示空闲状态、读取内存状态和写入内存状态。这三个状态的含义如下：

IDLE：空闲状态，表示当前没有读写操作，等待下一次读写请求。在该状态下，根据命中情况决定是否需要进行 Cache 读写操作，具体如下：

- 如果当前访问命中 Cache，直接返回 Cache 中的数据，状态保持为 IDLE。
- 如果当前访问未命中 Cache，并且对应的 Cache line 是干净行，则需要进行读内存操作，状态转换为 RM。

- 如果当前访问未命中 Cache，并且对应的 Cache line 是脏行，则需要先将数据写回内存，再进行读内存操作，状态转换为 WM。

RM：读取内存状态，表示当前正在进行 Cache 读取操作，等待读取完成。在该状态下，等待 cache_data_data_ok 信号的到来，表示 Cache 数据已经准备好，可以将数据返回给 CPU 或写入 Cache。具体如下：

- 如果 cache_data_data_ok 信号为 1，表示 Cache 数据已经准备好，可以将数据返回给 CPU 或写入 Cache。状态转换为 IDLE。
- 如果 cache_data_data_ok 信号为 0，表示 Cache 数据还未准备好，继续等待数据准备完成，状态保持为 RM。

WM：写入内存状态，表示当前正在进行 Cache 写入操作，等待写入完成。在该状态下，等待 cache_data_data_ok 信号的到来，表示数据已经写入 Cache。具体如下：

- 如果 cache_data_data_ok 信号为 1，表示数据已经写入 Cache，可以继续进行下一次操作。状态转换为 RM。
- 如果 cache_data_data_ok 信号为 0，表示数据还未写入 Cache，继续等待写入完成，状态保持为 WM。

```
//读内存（RM）
//变量read_req, addr_rcv, read_finish用于构造类sram信号。
wire read_req;      //一次完整的读事务，从发出读请求到结束
reg addr_rcv;       //地址接收成功(addr_ok)后到结束
wire read_finish;   //数据接收成功(data_ok)，即读请求结束
always @(posedge clk) begin
    addr_rcv <= rst ? 1'b0 :
                read_req & cache_data_req & cache_data_addr_ok ? 1'b1 :
                read_finish ? 1'b0 : addr_rcv;
end
assign read_req = state == RM;
assign read_finish = read_req & cache_data_data_ok;

//写内存（WM）
wire write_req;
reg waddr_rcv;
wire write_finish;
always @(posedge clk) begin
    waddr_rcv <= rst ? 1'b0 :
                write_req & cache_data_req & cache_data_addr_ok ? 1'b1 :
                write_finish ? 1'b0 : waddr_rcv;
end
assign write_req = state == WM;
assign write_finish = write_req & cache_data_data_ok;
```

图 4: 内存读写操作

这段代码实现了对内存读写操作的编码，利用类 sram 信号的方式对读写事务进行控制。具体来说，对于读内存(RM)和写内存(WM)两种状态，分别定义了两个信号：

读内存信号：包括 read_req、addr_rcv 和 read_finish 三个信号。其中，read_req 信号表示是否需

要进行读内存操作，根据当前状态是否为 RM 确定；addr_rcv 信号表示当前读内存操作是否完成地址接收，即 cache_data_addr_ok 信号是否为 1；read_finish 信号表示当前读内存操作是否完成数据接收，即 cache_data_data_ok 信号是否为 1。这些信号的组合可以表示一次完整的读事务，从发出读请求到数据接收完成。

写内存信号：包括 write_req、waddr_rcv 和 write_finish 三个信号。其中，write_req 信号表示是否需要进行写内存操作，根据当前状态是否为 WM 确定；waddr_rcv 信号表示当前写内存操作是否完成地址接收，即 cache_data_addr_ok 信号是否为 1；write_finish 信号表示当前写内存操作是否完成数据写入，即 cache_data_data_ok 信号是否为 1。这些信号的组合可以表示一次完整的写事务，从发出写请求到数据写入完成。

```
//output to mips core
assign cpu_data_rdata   = hit ? c_block : cache_data_rdata; // 读命中就返回cache内容(c_block)，否则继续等待
assign cpu_data_addr_ok = cpu_data_req & hit | cache_data_addr_ok & read_req & cache_data_req;
assign cpu_data_data_ok = cpu_data_req & hit | cache_data_data_ok & read_req;

// address of replace block
wire [31:0] write_address, read_address;
assign write_address = {c_tag, index, 2'b00};
assign read_address = {cpu_data_addr[31:2], 2'b00};

//output to axi interface
assign cache_data_req   = read_req & ~addr_rcv | write_req & ~waddr_rcv  ; // 读请求并且地址未收到（写请求并且地址未收到）
assign cache_data_wr    = write_req;
assign cache_data_size  = 2'b10 ;
assign cache_data_addr  = write_req ? write_address : read_address;
assign cache_data_wdata = c_block ;
```

图 5: 对 MIPS 核心和 AXI 接口的输出控制

这段代码实现了对 MIPS 核心和 AXI 接口的输出控制，将 Cache 的读写结果返回给 MIPS 核心，并根据读写请求生成 AXI 接口的信号。

```
integer t;
always @(posedge clk) begin
    if(rst) begin
        for(t=0; t<CACHE_DEEPTH; t=t+1) begin   //刚开始将Cache置为无效
            cache_valid[t] <= 0;
            cache_dirty[t] <= 0;
        end
    end
    else begin
        if(read_finish) begin //缺失，访存结束时
            cache_dirty[index_save] <= 1'b0;
            cache_valid[index_save] <= 1'b1;              //将Cache line置为有效
            cache_tag  [index_save] <= tag_save;
            cache_block[index_save] <= cache_data_rdata; //写入Cache line
        end
        else if(write & state == IDLE & hit) begin //写命中时直接写Cache
            cache_dirty[index] <= 1'b1;
            cache_block[index] <= write_cache_data;
        end
    end
end
```

图 6: Cache 内存读写

这段代码实现了 Cache 的读写操作，将读取到的数据块存储到 Cache 中，并将写入的数据块更新到 Cache 中。同时，根据 Cache line 的状态更新 cache_valid 和 cache_dirty 信号，以便后续的访问。

4 实验结果

```
================================================================
Test begin!
shell1 test begin.

dgemm PASS!

Total Count(SoC count) = 0x2d2d6

Total Count(CPU count) = 0x14890

================================================================
Test end!
----PASS!!!
```

图 7: TCL 控制台输出结果

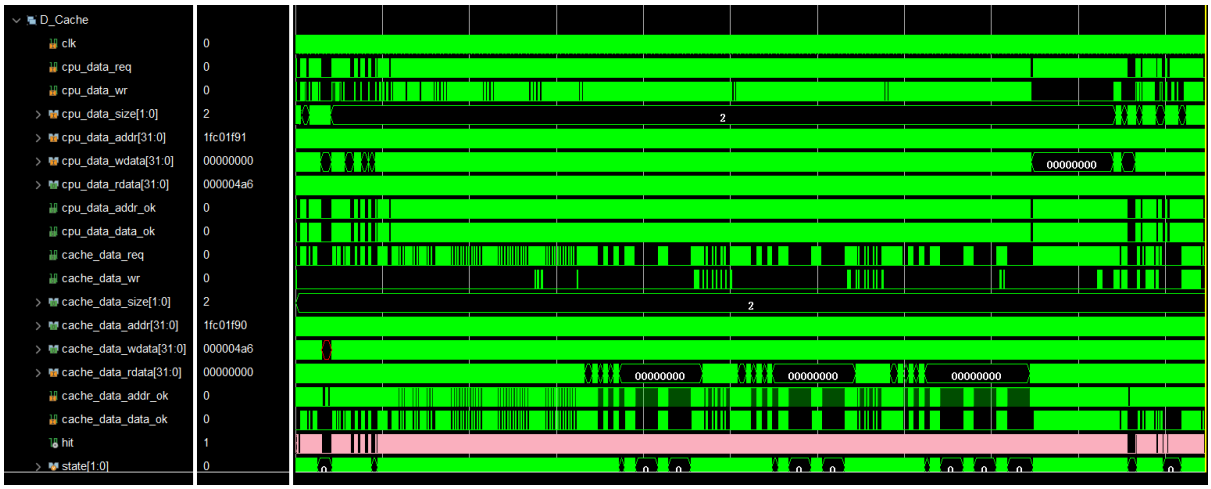

图 8: D_Cache 仿真图

5 实验心得

非常熟练的掌握了 Cache 的工作原理,也同时知道了 AXI 接口,熟悉了各种信号。

A 附录一

```
module d_cache (
    // 时钟信号和复位信号
    input wire clk, rst,
    // MIPS处理器核心输入信号
    input          cpu_data_req     , // CPU数据请求信号，表示CPU正在请求访问数据缓
        存
    input          cpu_data_wr      , // CPU数据写信号，表示CPU正在执行一次数据写操
        作
    input  [1 :0] cpu_data_size     , // CPU数据大小，用于表示数据访问的字节大小（例
        如：00表示1字节，01表示2字节，10表示4字节）
    input  [31:0] cpu_data_addr     , // CPU数据访问地址
    input  [31:0] cpu_data_wdata    , // CPU数据写入的数据
    output [31:0] cpu_data_rdata    , // CPU从数据缓存读取的数据
    output         cpu_data_addr_ok , // CPU数据地址确认信号，表示地址访问已成功
    output         cpu_data_data_ok , // CPU数据读/写确认信号，表示数据读取或写入已
        成功

    // AXI接口输出信号
    output          cache_data_req     , // 缓存数据请求信号，表示缓存正在请求访问
        AXI总线
    output          cache_data_wr      , // 缓存数据写信号，表示缓存正在执行一次数据
        写操作
    output  [1 :0] cache_data_size    , // 缓存数据大小，用于表示数据访问的字节大小
        （例如：00表示1字节，01表示2字节，10表示4字节）
    output  [31:0] cache_data_addr    , // 缓存数据访问地址
    output  [31:0] cache_data_wdata   , // 缓存数据写入的数据
    input   [31:0] cache_data_rdata   , // 缓存从AXI总线读取的数据
    input          cache_data_addr_ok , // 缓存数据地址确认信号，表示地址访问已成功
    input          cache_data_data_ok   // 缓存数据读/写确认信号，表示数据读取或写
        入已成功
);
    // 写回实现：增加脏位

    // 读命中：直接返回cache内容。
    // 读缺失：访问内存，写入cache
    // 写命中：只写入cache。
    // 写缺失：访问内存，写入cache
    // 替换：dirty写入后再替换

    //Cache配置
    parameter  INDEX_WIDTH  = 10, OFFSET_WIDTH = 2;
    localparam TAG_WIDTH    = 32 - INDEX_WIDTH - OFFSET_WIDTH;
    localparam CACHE_DEEPTH = 1 << INDEX_WIDTH;

    //Cache存储单元
    reg                  cache_valid [CACHE_DEEPTH - 1 : 0];
    reg                  cache_dirty [CACHE_DEEPTH - 1 : 0];
    reg [TAG_WIDTH-1:0] cache_tag   [CACHE_DEEPTH - 1 : 0];
    reg [31:0]           cache_block [CACHE_DEEPTH - 1 : 0];
```

```
//访问地址分解
wire [OFFSET_WIDTH-1:0] offset;
wire [INDEX_WIDTH-1:0] index;
wire [TAG_WIDTH-1:0] tag;

assign offset = cpu_data_addr[OFFSET_WIDTH - 1 : 0];
assign index = cpu_data_addr[INDEX_WIDTH + OFFSET_WIDTH - 1 : OFFSET_WIDTH];
assign tag = cpu_data_addr[31 : INDEX_WIDTH + OFFSET_WIDTH];

//访问Cache line
wire c_valid;
wire c_dirty;
wire [TAG_WIDTH-1:0] c_tag;
wire [31:0] c_block;

assign c_valid = cache_valid[index];
assign c_dirty = cache_dirty[index];
assign c_tag   = cache_tag  [index];
assign c_block = cache_block[index];

//判断是否命中
wire   hit, miss, dirty, clean;
assign hit = c_valid & (c_tag == tag);  //cache line的valid位为1, 且tag与地址中
    tag相等
assign miss = ~hit;
assign dirty = c_valid & c_dirty;
assign clean = ~dirty;


//读或写
wire read, write;
assign write = cpu_data_wr;
assign read = ~write;

//FSM
parameter IDLE = 2'b00, RM = 2'b01, WM = 2'b11;
reg [1:0] state;
always @(posedge clk) begin
    if(rst) begin
        state <= IDLE;
    end
    else begin
        case(state)
            IDLE:   state <= cpu_data_req & hit ? IDLE :
                             cpu_data_req & miss & clean ? RM :
                             cpu_data_req & miss & dirty ? WM : IDLE;
            RM:     state <= cache_data_data_ok ? IDLE : RM;
            WM:     state <= cache_data_data_ok ? RM   : WM;
```

```
        endcase
    end
end

// 读缺失或者写缺失时 dirty 位有效

//读内存 (RM)
//变量read_req, addr_rcv, read_finish用于构造类sram信号。
wire read_req;      //一次完整的读事务, 从发出读请求到结束
reg addr_rcv;       //地址接收成功(addr_ok)后到结束
wire read_finish;   //数据接收成功(data_ok), 即读请求结束
always @(posedge clk) begin
    addr_rcv <= rst ? 1'b0 :
                read_req & cache_data_req & cache_data_addr_ok ? 1'b1 :
                read_finish ? 1'b0 : addr_rcv;
end
assign read_req = state == RM;
assign read_finish = read_req & cache_data_data_ok;

//写内存 (WM)
wire write_req;
reg waddr_rcv;
wire write_finish;
always @(posedge clk) begin
    waddr_rcv <= rst ? 1'b0 :
                 write_req & cache_data_req & cache_data_addr_ok ? 1'b1 :
                 write_finish ? 1'b0 : waddr_rcv;
end
assign write_req = state == WM;
assign write_finish = write_req & cache_data_data_ok;

//output to mips core
assign cpu_data_rdata   = hit ? c_block : cache_data_rdata; // 读命中就返回
    cache内容(c_block), 否则继续等待
assign cpu_data_addr_ok = cpu_data_req & hit | cache_data_addr_ok & read_req &
    cache_data_req;
assign cpu_data_data_ok = cpu_data_req & hit | cache_data_data_ok & read_req;

// address of replace block
wire [31:0] write_address, read_address;
assign write_address = {c_tag, index, 2'b00};
assign read_address = {cpu_data_addr[31:2], 2'b00};

//output to axi interface
assign cache_data_req   = read_req & ~addr_rcv | write_req & ~waddr_rcv  ; //
    读请求并且地址未收到 (写请求并且地址未收到)
assign cache_data_wr    = write_req;
assign cache_data_size  = 2'b10 ;
assign cache_data_addr  = write_req ? write_address : read_address;
```

```
assign cache_data_wdata = c_block ;

//写入Cache
//保存地址中的tag, index, 防止addr发生改变
reg [TAG_WIDTH-1:0] tag_save;
reg [INDEX_WIDTH-1:0] index_save;
always @(posedge clk) begin
    tag_save   <= rst ? 0 :
                  cpu_data_req ? tag : tag_save;
    index_save <= rst ? 0 :
                  cpu_data_req ? index : index_save;
end

wire [31:0] write_cache_data;
wire [3:0] write_mask;

//根据地址低两位和size, 生成写掩码 (针对sb, sh等不是写完整一个字的指令), 4位对
    应1个字 (4字节) 中每个字的写使能
assign write_mask = cpu_data_size==2'b00 ?
                        (cpu_data_addr[1] ? (cpu_data_addr[0] ? 4'b1000 : 4'
                            b0100):
                                            (cpu_data_addr[0] ? 4'b0010 : 4'
                                                b0001)) :
                        (cpu_data_size==2'b01 ? (cpu_data_addr[1] ? 4'b1100 :
                            4'b0011) : 4'b1111);

//掩码的使用: 位为1的代表需要更新的。
//位拓展: {8{1'b1}} -> 8'b11111111
//new_data = old_data & ~mask | write_data & mask
assign write_cache_data = cache_block[index] & ~{{8{write_mask[3]}}, {8{
    write_mask[2]}}, {8{write_mask[1]}}, {8{write_mask[0]}}} |
                          cpu_data_wdata & {{8{write_mask[3]}}, {8{write_mask
                              [2]}}, {8{write_mask[1]}}, {8{write_mask[0]}}};

integer t;
always @(posedge clk) begin
    if(rst) begin
        for(t=0; t<CACHE_DEEPTH; t=t+1) begin    //刚开始将Cache置为无效
            cache_valid[t] <= 0;
            cache_dirty[t] <= 0;
        end
    end
    else begin
        if(read_finish) begin //缺失, 访存结束时
            cache_dirty[index_save] <= 1'b0;
            cache_valid[index_save] <= 1'b1;              //将Cache line置为有效
            cache_tag  [index_save] <= tag_save;
            cache_block[index_save] <= cache_data_rdata; //写入Cache line
        end
```

```
            else if(write & state == IDLE & hit) begin //写命中时直接写Cache
                cache_dirty[index] <= 1'b1;
                cache_block[index] <= write_cache_data;
            end
        end
    end
endmodule
```