Lab3-Cache Report

> 实验目标：使用`systemverilog`实现组相连Cache
>
> 实验环境和工具：
>
> 操作系统：Windows 10 (64-bits)
> 综合工具：Vivado 2017.2
> 姓名：陈炳楠
> 学号：PB17111650

1.第一阶段实验内容和过程

N路组相连Cache设计与说明

设计思路：

> 1.N路组相连和直接映射cache的区别在于每一组中有多个cache line，所以在设计相应的数据大小时，应该加上`WAY_CNT`一维；在进行数据匹配时也需要便利一组中的所有数据进行比对。
> 2.对于FIFO算法的实现

具体设计：

```
module cache #(
    parameter  LINE_ADDR_LEN = 3, // line内地址长度，决定了每个line具有2^3个word
    parameter  SET_ADDR_LEN  = 3, // 组地址长度，决定了一共有2^3=8组
    parameter  TAG_ADDR_LEN  = 6, // tag长度
    parameter  WAY_CNT       = 3  // 组相连度，决定了每组中有多少路line，这里是直接映射型cache，因此该参数没用到
)(
    input  clk, rst,
    output miss,               // 对CPU发出的miss信号
    input  [31:0] addr,        // 读写请求地址
    input  rd_req,             // 读请求信号
    output reg [31:0] rd_data, // 读出的数据，一次读一个word
    input  wr_req,             // 写请求信号
    input  [31:0] wr_data      // 要写入的数据，一次写一个word
);

localparam MEM_ADDR_LEN    = TAG_ADDR_LEN + SET_ADDR_LEN ; // 计算主存地址长度 MEM_ADDR_LEN，主存大小=2^MEM_ADDR_LEN个line
localparam UNUSED_ADDR_LEN = 32 - TAG_ADDR_LEN - SET_ADDR_LEN - LINE_ADDR_LEN - 2 ;       // 计算未使用的地址的长度

localparam LINE_SIZE       = 1 << LINE_ADDR_LEN  ;         // 计算 line 中 word 的数量，即 2^LINE_ADDR_LEN 个word 每 line
localparam SET_SIZE        = 1 << SET_ADDR_LEN   ;         // 计算一共有多少组，即 2^SET_ADDR_LEN 个组

reg [           31:0] cache_mem    [SET_SIZE][WAY_CNT][LINE_SIZE]; // SET_SIZE个line，每个line有LINE_SIZE个word
reg [TAG_ADDR_LEN-1:0] cache_tags   [SET_SIZE][WAY_CNT];            // SET_SIZE个TAG
reg                    valid        [SET_SIZE][WAY_CNT];            // SET_SIZE个valid(有效位)
reg                    dirty        [SET_SIZE][WAY_CNT];            // SET_SIZE个dirty(脏位)

wire [             2-1:0]   word_addr;                   // 将输入地址addr拆分成这5个部分
wire [  LINE_ADDR_LEN-1:0]   line_addr;
wire [   SET_ADDR_LEN-1:0]    set_addr;
wire [   TAG_ADDR_LEN-1:0]    tag_addr;
wire [UNUSED_ADDR_LEN-1:0] unused_addr;

enum  {IDLE, SWAP_OUT, SWAP_IN, SWAP_IN_OK} cache_stat;    // cache 状态机的状态定义
                                                           // IDLE代表就绪，SWAP_OUT代表正在换出，SWAP_IN代表正在换入，SWAP_IN_OK代表换入后进行一周期的写入cache操

reg  [   SET_ADDR_LEN-1:0] mem_rd_set_addr = 0;
reg  [   TAG_ADDR_LEN-1:0] mem_rd_tag_addr = 0;
wire [   MEM_ADDR_LEN-1:0] mem_rd_addr = {mem_rd_tag_addr, mem_rd_set_addr};
reg  [   MEM_ADDR_LEN-1:0] mem_wr_addr = 0;

reg  [31:0] mem_wr_line [LINE_SIZE];
wire [31:0] mem_rd_line [LINE_SIZE];

wire mem_gnt;      // 主存响应读写的握手信号
```

```
assign {unused_addr, tag_addr, set_addr, line_addr, word_addr} = addr;  // 拆分 32bit ADDR

reg cache_hit = 1'b0;

enum {FIFO, LRU} swap_out_strategy;
integer time_cnt;//LRU时间记录

reg [WAY_CNT-1, 0] way_addr;//空闲位置
reg [WAY_CNT-1, 0] out_way; //换出位置

reg [15:0] LRU_record[SET_SIZE][WAY_CNT]; //每一项的LRU记录
reg [WAY_CNT:0] FIFO_record[SET_SIZE][WAY_CNT];//每一组内的FIFO排位记录


always @ (*) begin              // 判断输入的address 是否在 cache 中命中,如果没有命中，需要在之后设置换出策略
    cache_hit = 1'b0;
    for(integer i=0; i<WAY_CNT; i++)
        begin
        if(valid[set_addr][i] && cache_tags[set_addr][i] == tag_addr)   // 如果 cache line有效，并且tag与输入地址中的tag相等，则命中
            begin
            cache_hit = 1'b1;
            way_addr = i;
            end
        end
end

always @ (*) begin
    if(~cache_hit & (wr_req | rd_req)) begin
    //LRU策略选出冲突时换处的块
        if(swap_out_strategy == LRU) begin
            for (integer i = 0; i < WAY_CNT; i++) begin
                out_way = 0;
                if (LRU_record[set_addr][i] < LRU_record[set_addr][out_way]) begin
                    out_way = i;
                end
            end
        end
    //FIFO策略选出冲突时换出的块
        else begin
            integer free = 0;//是否有空闲块的标志
            for (integer i = 0;i < WAY_CNT ;i++ ) begin
                if(FIFO_record[set_addr][i] == 0)begin
                    out_way = i;
```

```
                    free = 1;
                    break;
                end
            end
            if(free == 0) begin
                for (integer i = 0;i < WAY_CNT ;i++ ) begin//此时说明时最早进去的
                    if(FIFO_record[set_addr][i] == WAY_CNT) begin
                        out_way = i;
                        FIFO_record[set_addr][i] = 0;/*my adding*/
                        break;
                    end
                end
            end
            if(FIFO_record[set_addr][out_way] == 0) begin
                for (integer i = 0;i < WAY_CNT ;i++ ) begin
                    if(FIFO_record[set_addr][i] != 0]) begin
                        FIFO_record[set_addr][i] = FIFO_record[set_addr][i] + 1;
                    end
                end
            end
            FIFO_record[set_addr][out_way] = 1;
        end
    end
end

always @ (posedge clk or posedge rst) begin     // ?? cache ???
    if(rst) begin
        cache_stat <= IDLE;
        time_cnt <= 0;
        swap_out_strategy <= LRU;
        //swap_out_strategy <= FIFO;
        for(integer i = 0; i < SET_SIZE; i++) begin
            for (integer j = 0;j < WAY_CNT ;j++ ) begin
                dirty[i][j] = 1'b0;
                valid[i][j] = 1'b0;
                LRU_record[i][j] = 0;
                FIFO_record[j][j] = 0;
            end
        end
        for(integer k = 0; k < LINE_SIZE; k++)
            mem_wr_line[k] <= 0;
        mem_wr_addr <= 0;
        {mem_rd_tag_addr, mem_rd_set_addr} <= 0;
        rd_data <= 0;
```

```
    end else begin
        time_cnt ++;
        case(cache_stat)
        IDLE:       begin
                        if(cache_hit) begin
                            if(rd_req) begin    // 如果cache命中，并且是读请求，
                                rd_data <= cache_mem[set_addr][way_addr][line_addr];   //则直接从cache中取出要读的数据
                            end else if(wr_req) begin // 如果cache命中，并且是写请求，
                                cache_mem[set_addr][way_addr][line_addr] <= wr_data;   // 则直接向cache中写入数据
                                dirty[set_addr][way_addr] <= 1'b1;                     // 写数据的同时置脏位
                            end
                            LRU_record[set_addr][way_addr] <= time_cnt;
                        end else begin
                            if(wr_req | rd_req) begin   // 如果 cache 未命中，并且有读写请求，则需要进行换入
                                if(valid[set_addr][out_way] & dirty[set_addr][out_way]) begin    // 如果 要换入的cache line 本来有效，且脏，则需要先将它换出
                                    cache_stat  <= SWAP_OUT;
                                    mem_wr_addr <= {cache_tags[set_addr][out_way], set_addr};
                                    mem_wr_line <= cache_mem[set_addr][out_way];
                                end else begin                                      // 反之，不需要换出，直接换入
                                    cache_stat  <= SWAP_IN;
                                end
                                {mem_rd_tag_addr, mem_rd_set_addr} <= {tag_addr, set_addr};
                            end
                        end
                    end
        SWAP_OUT:   begin
                        if(mem_gnt) begin           // 如果主存握手信号有效，说明换出成功，跳到下一状态
                            cache_stat <= SWAP_IN;
                        end
                    end
        SWAP_IN:    begin
                        if(mem_gnt) begin           // 如果主存握手信号有效，说明换入成功，跳到下一状态
                            cache_stat <= SWAP_IN_OK;
                        end
                    end
        SWAP_IN_OK: begin           // 上一个周期换入成功，这周期将主存读出的line写入cache，并更新tag，置高valid，置低dirty
                        for(integer i=0; i<LINE_SIZE; i++)
                        cache_mem[mem_rd_set_addr][out_way[i] <= mem_rd_line[i];
                        cache_tags[mem_rd_set_addr][out_way] <= mem_rd_tag_addr;
                        valid     [mem_rd_set_addr][out_way] <= 1'b1;
                        dirty     [mem_rd_set_addr][out_way] <= 1'b0;
                        LRU_record[mem_rd_set_addr][out_way] <= time_cnt;
                        cache_stat <= IDLE;         // 回到就绪状态
                    end
```

```
        endcase
    end
end

wire mem_rd_req = (cache_stat == SWAP_IN );
wire mem_wr_req = (cache_stat == SWAP_OUT);
wire [   MEM_ADDR_LEN-1 :0] mem_addr = mem_rd_req ? mem_rd_addr : ( mem_wr_req ? mem_wr_addr : 0);

assign miss = (rd_req | wr_req) & ~(cache_hit && cache_stat==IDLE) ;     // 当 有读写请求时，如果cache不处于就绪(IDLE)状态，或者未命中，则miss=1

main_mem #(     // 主存，每次读写以line 为单位
    .LINE_ADDR_LEN  ( LINE_ADDR_LEN          ),
    .ADDR_LEN       ( MEM_ADDR_LEN           )
) main_mem_instance (
    .clk            ( clk                    ),
    .rst            ( rst                    ),
    .gnt            ( mem_gnt                ),
    .addr           ( mem_addr               ),
    .rd_req         ( mem_rd_req             ),
    .rd_line        ( mem_rd_line            ),
    .wr_req         ( mem_wr_req             ),
    .wr_line        ( mem_wr_line            )
);

endmodule
```

测试样例及实现结果

FIFO测试结果

- 可以看出最终变量值回到-1，FIFO策略实现正确

LRU测试结果

- 可以看出最终变量值回到-1，LRU策略实现正确

2.第二阶段实验内容和过程

核心代码设计

```
module WB_Data_WB(
    input wire clk, bubbleW, flushW,
    input wire wb_select, csrwb_select,
    input wire [2:0] load_type,
    input  [3:0] write_en, debug_write_en,
    input  [31:0] addr,
    input  [31:0] debug_addr,
    input  [31:0] in_data, debug_in_data,
    input [31:0] csrRegmem1,
    output wire [31:0] debug_out_data,
    output wire [31:0] data_WB,
    output wire Cachemiss
    );

    wire [31:0] data_raw;
    wire [31:0] data_WB_raw;

    reg [31:0] hit_count = 0;
    reg [31:0] miss_count = 0;
    reg [31:0] last_addr;
    wire cache_rd_wr = wb_select | (|write_en);

    always @ (posedge clk or posedge flushW) begin
        if(flushW) begin
            last_addr <= 0;
        end else begin
            if(cache_rd_wr) begin
                last_addr <= addr;
            end
        end
    end

    always @ (posedge clk or posedge flushW) begin
        if(flushW) begin
            hit_count <= 0;
            miss_count <= 0;
        end
        else begin
            if(cache_rd_wr & (last_addr != addr)) begin
                if (Cachemiss) begin
                    miss_count <= miss_count +1;
                end
                else hit_count <= hit_count + 1;
            end
```

```
        end
    end

    cache #(
    .LINE_ADDR_LEN  ( 3             ),
    .SET_ADDR_LEN   ( 2             ),
    .TAG_ADDR_LEN   ( 12            ),
    .WAY_CNT        ( 3             )
    ) cache_test_instance (
    .clk            ( clk           ),
    .rst            ( flushW        ),
    .miss           ( Cachemiss          ),
    .addr           ( addr          ),
    .rd_req         ( wb_select        ),
    .rd_data        ( data_raw       ),
    .wr_req         ( |write_en        ),
    .wr_data        ( in_data << (8 * addr[1:0])       )
);

    // Add flush and bubble support
    // if chip not enabled, output output last read result
    // else if chip clear, output 0
    // else output values from cache

    reg bubble_ff = 1'b0;
    reg flush_ff = 1'b0;
    reg wb_select_old = 0;
    reg csrwb_select_old = 0;
    reg [31:0] data_WB_old = 32'b0;
    reg [31:0] csrRegmem1_old = 32'b0;
    reg [31:0] addr_old;
    reg [2:0] load_type_old;

    DataExtend DataExtend1(
        .data(data_raw),
        .addr(addr_old[1:0]),
        .load_type(load_type_old),
        .dealt_data(data_WB_raw)
    );

    always@(posedge clk)
    begin
        bubble_ff <= bubbleW;
        flush_ff <= flushW;
```

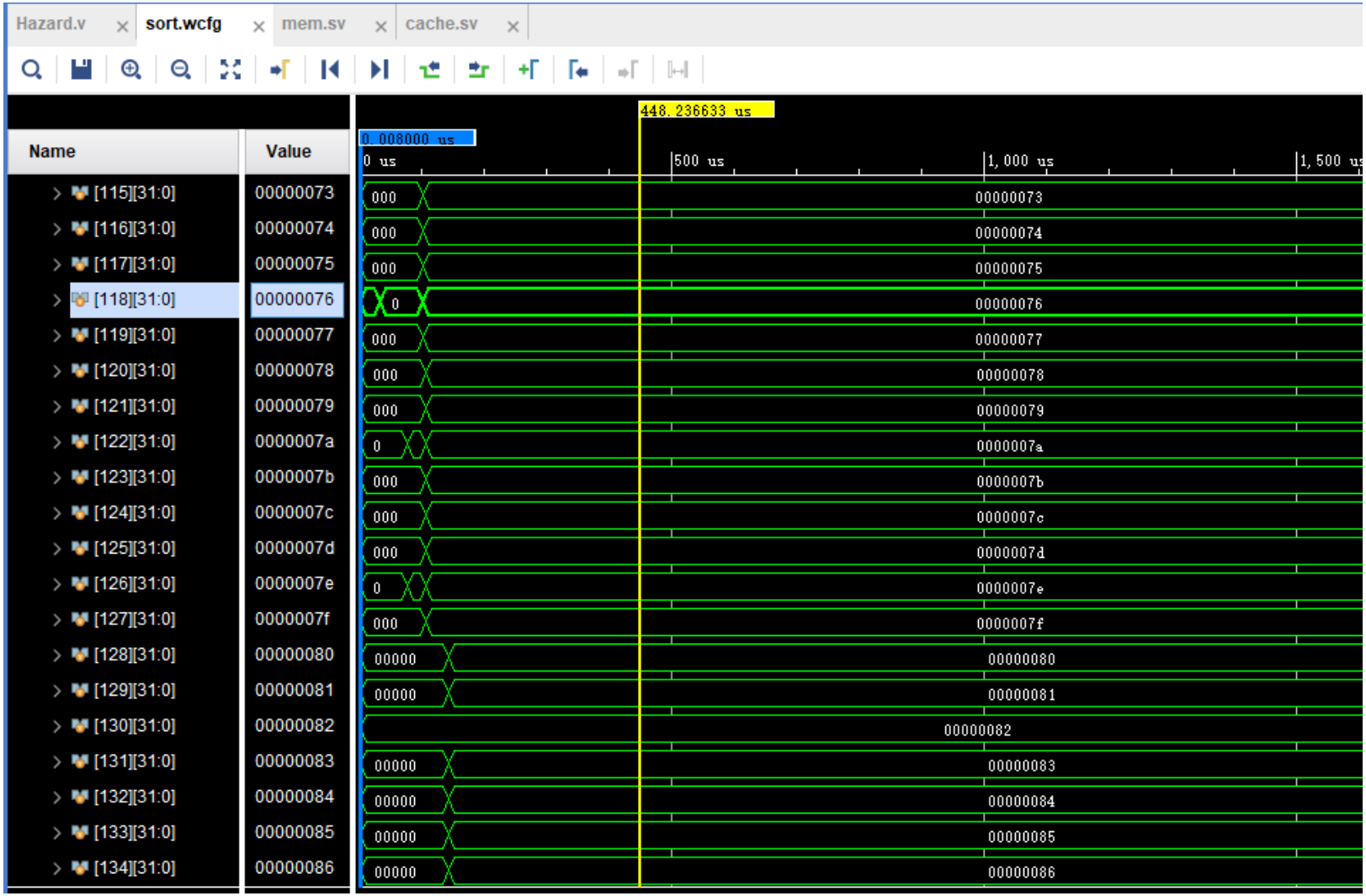
```
        data_WB_old <= data_WB;
        addr_old <= addr;
        wb_select_old <= wb_select;
        csrwb_select_old <= csrwb_select;
        load_type_old <= load_type;
        csrRegmem1_old <= csrRegmem1;
    end

    wire [31:0] data_WB_first;
    assign data_WB_first = bubble_ff ? data_WB_old :
                                (flush_ff ? 32'b0 :
                                            (wb_select_old ? data_WB_raw :
                                                             addr_old));
    assign data_WB = csrwb_select_old ? csrRegmem1_old : data_WB_first;

endmodule
```

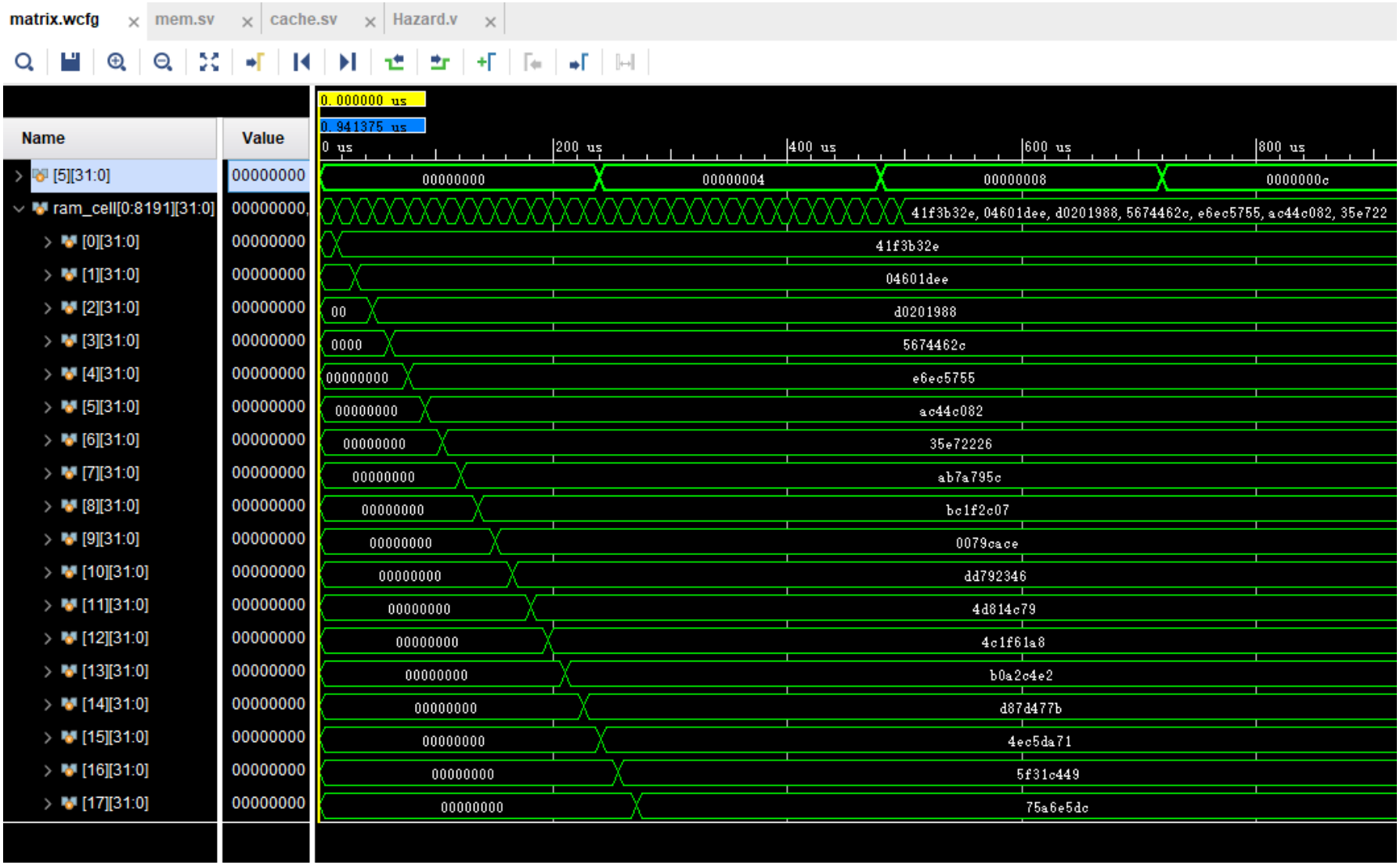
测试样例及实现结果

快排结果



部分结果展示

矩阵乘法结果



部分结果展示

3.实验数据采集及性能分析

LRU策略

*Cachesize = number of sets * way per sets * word per line*

设计	benchmark	Cachesize	组相连度	仿真周期数	缺失率	LUT	FF	IO	BRAM	BUFG
原始设定	MatMul.S(16*16)	4*4*8	4	328863	4664/(3784+4664)=0.552083	2211	3300	171	8	1
组相连度x2	MatMul.S(16*16)	4*8*8	8	328863	4664/(3784+4664)=0.552083	2177	3299	171	8	2
组相连度x4	MatMul.S(16*16)	4*16*8	16	328863	4664/(3784+4664)=0.552083	1983	3295	171	8	2
组数x2	MatMul.S(16*16)	8*4*8	4	327945	4647/(4647+3801)=0.550071	2395	5545	171	8	2
组数x4	MatMul.S(16*16)	16*4*8	4	142367	1317/(1317+7131)=0.155895	4508	10004	171	8	2
line字节数x2	MatMul.S(16*16)	4*4*16	4	318693	4475/(4475+3973)=0.529711	3919	6108	171	8	2
line字节数x4	MatMul.S(16*16)	4*4*32	4	150235	1356/(1356+7092)=0.160511	8210	11730	171	8	2
组数x4+组相连度x4	MatMul.S(16*16)	16*16*8	16	142357	1317/(1317+7130)=0.155913	7735	18002	171	8	2
组数x4+line字节数x4	MatMul.S(16*16)	16*4*32	4	65110	24/(24+8424)=0.002814	15645	36885	171	8	2
line字节数x4+组相连度x4	MatMul.S(16*16)	4*16*32	16	150235	1356/(1356+7092)=0.160511	6710	11746	171	8	2
原始设定	QuickSort.S(256)	4*4*8	4	61728	225/(5242+225)=0.041156	2211	3300	171	8	1
组相连度x4	QuickSort.S(256)	4*16*8	16	61728	225/(225+5242)=0.041156	2211	3300	171	8	1
组数x4	QuickSort.S(256)	16*4*8	4	46083	72/(72+5395)=0.013170	4508	10004	171	8	2
line字节数x4	QuickSort.S(256)	4*4*32	4	42395	22/(22+5445)=0.004024	8210	11730	171	8	2
组数x4+组相连度x4	QuickSort.S(256)	16*16*8	16	46083	72/(72+5395)	7735	18002	171	8	2
组数x4+line字节数x4	QuickSort.S(256)	16*4*32	4	40913	9/(9+5458)=0.001646	15645	36885	171	8	2
line字节数x4+组相连度x4	QuickSort.S(256)	4*16*32	16	42395	22/(22+5445)=0.004024	6710	11746	171	8	2

LRU策略数据分析

通过实验数据的分析，我们可以得出以下结论：

- 提高组相连度对于提升程序速度，降低缺失率作用有限，但可以起到简化电路的作用

- 增加组数和增加line内字数对于提升程序速度，降低缺失率有很大作用，具体哪一种更好取决于程序/代码的特性。两者一起使用可以进一步降低缺失率，优化访存效率，这两种方法会成比例增加电路面积，电路方面的成本会有明显增加
- 矩阵乘法代码的局部性比快排差，因此在增加cache面积的时候表现出较大的优化比例

FIFO策略

*Cachesize = number of sets * way per sets * word per line*

设计	benchmark	Cachesize	组相连度	仿真周期数	缺失率	LUT	FF	IO	BRAM	BUFG
原始设定	MatMul.S(16*16)	448	4	281233	3781/(3781+4667)=0.447561	1103	2089	171	8	1
组相连度x2	MatMul.S(16*16)	488	8	151581	1381/(1381+7067)=0.163471	1103	2089	171	8	1
组相连度x4	MatMul.S(16*16)	4168	16	93802	434/(434+8014)=0.051373	859	2089	171	8	1
组数x2	MatMul.S(16*16)	848	4	153957	1425/(1425+7023)=0.168679	1186	3145	171	8	1
组数x4	MatMul.S(16*16)	1648	4	97744	507/(507+7941)=0.060014	1968	5250	171	8	1
line字节数x2	MatMul.S(16*16)	4416	4	158802	1515/(151+6933)=0.179332	2364	3879	171	8	1
line字节数x4	MatMul.S(16*16)	4432	4	105898	658/(658+7790)=0.077888	3978	7485	171	8	1
组数x4+组相连度x4	MatMul.S(16*16)	16168	16	68998	96/(96+8352)=0.011364	1846	5247	171	8	1
组数x4+line字节数x4	MatMul.S(16*16)	16432	4	65110	24/(24+8424)=0.002841	6856	19834	171	8	1
line字节数x4+组相连度x4	MatMul.S(16*16)	41632	16	65110	24/(24+8424)=0.002841	4006	7467	171	8	1
原始设定	QuickSort.S(256)	448	4	67973	284/(284+5183)=0.051948	1103	2089	171	8	1
组相连度x4	QuickSort.S(256)	4168	16	42533	39/(39+5428)=0.007134	859	2089	171	8	1
组数x4	QuickSort.S(256)	1648	4	42533	39/(39+5428)=0.007134	1968	5250	171	8	1
line字节数x4	QuickSort.S(256)	4432	4	40913	9/(9+5458)=0.001646	3978	7485	171	8	1
组数x4+组相连度x4	QuickSort.S(256)	16168	16	42533	39/(39+5428)=0.007134	1846	5247	171	8	1
组数x4+line字节数x4	QuickSort.S(256)	16432	4	40913	9/(9+5458)=0.001646	6856	19834	171	8	2

设计	benchmark	Cachesize	组相连度	仿真周期数	缺失率	LUT	FF	IO	BRAM	BUFG
line字节数x4+组相连度x4	QuickSort.S(256)	41632	16	40913	9/(9+5458)=0.001646	4006	7467	171	8	1

FIFO策略数据分析

通过对FIFO策略数据的简单分析，我们可以得出以下结论：

- 访存的优化效果比较：增加组相连度 > 增加组数 > 增加line内字数
- 增加组相连度还是可以起到降低电路面积的作用
- 三种方法配合使用可以进一步提高访存优化效果
- 快排代码基本达到优化极限，继续进一步优化的空间很小

4.实验总结

实验中踩过的坑

- 较重要的一个问题是关于`Hazard.v`模块，在Lab2中对该模块的处理不完善，主要是当存在Load方面的数据相关时只生成了`bubbleF=1, bubbleD=1`的信号，没有设置`flushE=1`的信号，导致数据相关时出现问题，程序运行失败。
- 信号`Cachemiss`的优先级应该较高，即最后应该是类似如下结构

```
if (rst) begin ... end
else if (Cachemisss) begin ... end
else begin ..其它.. end
```

实验总结

通过这次Cache部分的实验，了解了Cache中LRU和FIFO的实现机制。并且通过测试数据，我们看出FIFO策略的访存优化效果较好，采用`组数x4+组相连度x4`的方法可以较大程度地提高访存效率又可以控制电路面积相对较小，选择这一组参数设置比较合理。