实验 16 报告

学号　2019K8009906001
2019K8009929051
姓名　孙彬　代瀚堃
箱子号　49

一、实验任务（10%）

1. 理解 Cache 的组织结构和工作机理；
2. 设计一个 Cache 模块，将其集成到模块级验证环境中，并通过仿真和上板验证。

二、实验设计（40%）

（一）总体设计思路

组织一个容量为 8KB，两路组相联，行大小为 16B 的 Cache，采用写回写分配、伪随机替换的策略。Tag 域和 V 域用两个宽度为 21bit，深度为 256 项的同步 RAM 存储，data 域分为两路，每路分 4 个 bank，每个 bank 用一个宽度为 32bit，深度为 256 项的同步 RAM 存储，而 dirty 域则用两个位宽为 256bit 的 reg 型变量存储。当收到读写请求时，以输入的 index 为索引，用输入的 tag 来和两路的 Tag 域进行匹配，如果读命中，则将命中那一路 data 域输出，如果写命中，则将待写数据写入相应的 Cache 行中；如果不命中，则随机替换出一路（如果这一路为脏，需要将其写回至内存），从内存读回缺失的数据，根据所接受的请求是读还是写，以及输入的字节使能，生成重填的数据写入 data 域中，并生成读写请求的响应信号。

（二）重要模块 1 设计：Cache 状态机

1. 工作原理

设置一个主状态机 main 和一个写命中状态机 sub。其中，main 状态机有 IDLE，LOOKUP，MISS，REPLACE 和 REFILL 五个状态，其状态转移条件如下页图 1 所示；sub 状态机，即 Write Buffer 状态机，有 S_IDLE 和 S_WRITE 两个状态，其状态转移条件如下页图 2 所示。各个状态的含义讲义中已经详述，此处不再重复。hit_wr_conflict 表示有 Hit Write 冲突，需要将主状态机阻塞，不接受读请求。

两个状态机通过 Hit Write 相联系。如果 main 在 LOOKUP 下识别出写命中，则在 S_IDLE→S_WRITE 的一拍发出写 data_ram 的使能信号，进入到 S_WRITE 时已经完成写入，可以接收新的写命令。

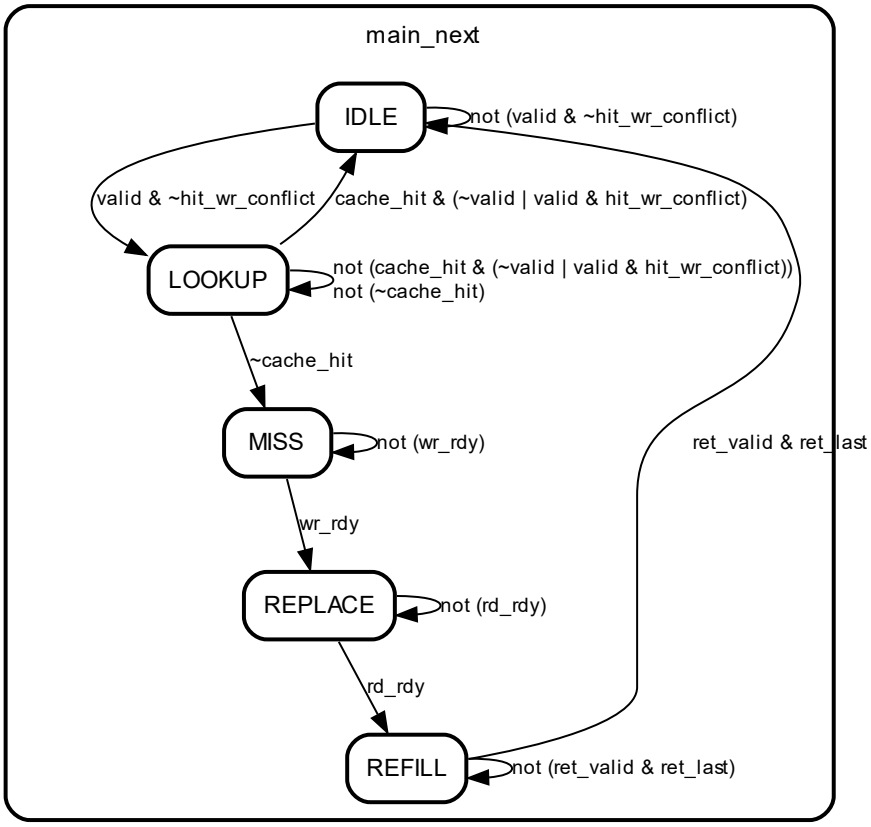

图 1　Cache 模块主状态机

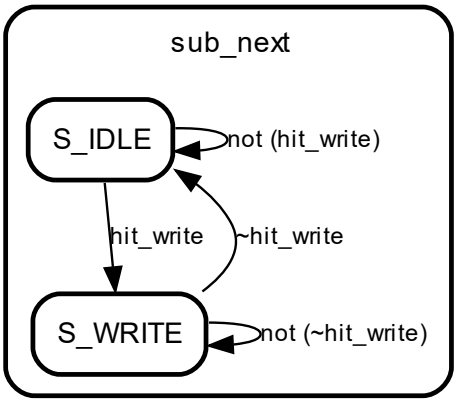

图 2　Write Buffer 状态机

2、接口定义

我们在此列出 Cache 模块的接口信号，如表 1：

名称	方向	位宽	功能描述
clk_g	input	1	时钟信号

resetn	input	1	复位信号，低电平有效
valid	input	1	与流水线交互的端口，op 为 0 表示读请求，op 为 1 表示写请求
op	input	1	
index	input	8	
tag	input	20	
offset	input	4	
addr_ok	output	1	
data_ok	output	1	
rdata	output	32	
wstrb	input	4	
wdata	input	32	
rd_req	output	1	对 AXI 总线的读通道
rd_type	output	3	
rd_addr	output	32	
rd_rdy	input	1	
ret_valid	input	1	
ret_last	input	1	
ret_data	input	32	
wr_rdy	input	1	对 AXI 总线的写通道
wr_req	output	1	
wr_type	output	3	
wr_addr	output	32	
wr_wstrb	output	4	
wr_data	output	128	

表 1　Cache 模块的接口定义

3. 功能描述

复位后，主状态机进入 IDLE，等待一个有效的读写请求，Write Buffer 状态机进入 S_IDLE，等待发生写命中。当流水线发出一个有效的读写请求时，Cache 向内部存储{tag, v}和 data 的 RAM 发起读请求，查询 Tag 域和 V 域的信息，发出 addr_ok 信号，表示接收到请求的地址，将请求信息记录在 Request Buffer 中，并跳转至 LOOKUP 状态。

当 main 进入 LOOKUP 状态时，已经读取出了 Tag 域和 V 域的信息，和 TLB 转换出的物理 tag 进行比对，在当前拍用组合逻辑判断出两路中是否有命中。如果读命中，则将命中那一路的 data 根据请求的地址选出 32 位后输出；如果写命中，则触发 Write Buffer 转移至 S_WRITE，将数据写入命中的 Cache 行。在 LOOKUP 或 IDLE 状态下，流水线此时有可能发起一个有效的读请求，如果 Write Buffer 中还有数据没有完成写入，且读请求的地址和待写数据的地址存在相关，则还不能接收这一请求，需要回到 IDLE 状态，等待数据写入后再接收这一请求。

如果两路都没有命中，则需要随机选出一路（replace_way）进行替换，以重填入缺失的数据，并跳转至 MISS 状态。在 MISS 状态下，main 等待 AXI 总线发出 wr_rdy 信号，当 wr_rdy=1 时跳转至 REPLACE 状态，如果被替换出的 Cache 行 dirty 位有效，则还需要发出 wr_req 信号，将 replace_data 写回至内存，如果无效，可以直接丢弃。main 在 REPLACE 状态下等待 AXI 总线接收读请求，此时发出 rd_req 信号，当 rd_rdy=1 时，完成握手，跳转至 REFILL 状态，等待返回数据。在 REFILL 状态下，ret_valid 每有一个周期为高，就表示返回了一个有效数据，将这些数据依次填入被替换的 Cache 行的第 0 ~ 3 号 Bank，当 ret_last 和 ret_valid 同时为高时，表示正在返回最后一个有效数据，可以跳转至 IDLE 状态等待下一个请求。

总的来看，读和写操作的状态转移基本一致，其区别在于命中时，写操作会触发 Write Buffer 状态机将待写入的数据写到命中的 Cache 行的某些字节上，读操作可以直接将数据输出；而在未命中时，写操作会将重填入 Cache 数据中的某些字节替换为待写入的数据，而读操作则不需要。

（三）重要模块 2 设计：Request Buffer 的设计

1. 工作原理

由于读写同步 RAM 都需要跨越两拍，且请求端口上只有 valid=1 时，输入的信息才是有效的，如果 Cache 接受了一个读写请求，则在向{tag, v}的 RAM 发起读请求的同时，应当锁存下读写请求的信息，这样当 main 状态机进入 LOOKUP 状态后，就能通过比对 tag_req 和 RAM 中读出的 Tag 来判断是否命中。

另一方面，如果读写不命中，也需要根据 Request Buffer 中锁存的读写信息，将重填的数据写入到指定行中，并将其中的某些字节用 wdata_req 来替换。

2. 功能描述

Request Buffer 包含以下部分：

op_req：存储输入的 op，用于区分目前执行的是写操作还是读操作，据此判断是否写命中，产生 hit_write 信号，并在不命中的重填时更新 dirty 域；

index_req：替换出脏块时用于拼合写回的地址，重填时用于索引，更新对应的 Cache 行；

tag_req：寄存读写请求的指定的 tag，用于比对、替换和重填；

offset_req：用于拼合内存读写地址，在写命中时，用于判断应该写入 4 个 Bank 中的哪一个，也用于判断是否存在 Hit Write 冲突，在重填时，用于判断当前填入的 Bank 是否也是要写请求对应的 Bank；

wstrb_req：用于指定一个 Bank 中的哪些字节应当被替换为 wdata 的相应字节；

wdata_req：寄存要写入的数据。

（四）重要模块 3 设计：Cache 表的实现

1. 工作原理

Tag 域和 V 域：两个域的更新条件完全相同，可以合在一起，用两个位宽为 21bit，深度为 256 项的同步 RAM 来实现；

Dirty 域：用两个位宽为 256bit 的寄存器存储；

Data 域：每一路 Cache 分为 4 个 Bank，每一个 Bank 为一个位宽为 32bit，深度为 256 项的同步 RAM，即可以用行号和 Bank 号确定出一个 32 位的字。

2. 接口定义

我们此处列出使用到的同步 RAM 接口：

名称	方向	位宽	功能描述
clka	input	1	时钟信号
wea	input	1	写使能
addra	input	32	读写地址
dina	input	32	写数据
douta	output	32	读数据

表 2　同步 RAM 使用到的接口

3. 功能描述

Tag 域用于存储地址中除了 index 和 offset 以外的信息，以判断是否命中，V 域表示该 Cache 行是否有效，该同步 RAM 的读写均需跨越两拍，另外，V 域，或者整个 RAM，应该初始化为 0；

Dirty 域用于标记该 Cache 行是否被写过，我们采用写回策略，如果被写过，该行被替换出去时应当写回到内存，以维护数据的一致性，由于其容量较小，可以用 reg 或 reg_file 实现；

Data 域用于存储某一地址上的数据，为了减少不必要的阻塞，即存在对同一 Cache 行的写（命中）后读但两次请求不在同一个字时，不判断为冲突，我们将每一路 Cache 划分为 4 个 Bank，每一个 Cache 行都等分为 4 份，

作为某一 Bank 在该地址上的一个字。

另外，为了避免引入从 RAM 输出到输入的路径，我们在添加 IP 时勾选 always enable 选项，将 RAM 的读写使能始终置为 1，读和写通过 wea 端口来区分。

（五）重要模块 4 设计：伪随机数发生器

1. 工作原理和功能描述

我们采用 LFSR（线性反馈移位寄存器）的方式来产生伪随机数，将一个寄存器的某些位输入到一个线性表达式中，输出经过移位后又作为这排触发器的输入，以产生一个伪随机数序列。

比如，我们设置一个位宽为 3bit 的 reg 型变量 random，其初值为 3'b111，每次发生替换后，将 random 的第 0 位和第 2 位做异或，放在原来第 0 位的位置，然后再循环左移 1 位，作为新的 random 值。而 replace_way 取其中的某一位即可，我们此处取最低位。

2. 代码实现

```
reg [2:0] random;
always @(posedge clk)
begin
    if(rst)
    begin
        random <= 3'b111;
    end
    else if(from_REFILL_to_IDLE)
    begin
        random <= {random[1], random[0]^random[2], random[2]};
    end
end
assign replace_way = random[0];
```

三、实验过程（50%）

（一）实验流水账

2021/12/16	13:30 ~ 17:00	阅读讲义
2021/12/16	20:00 ~ 23:00	组织 Cache 表
2021/12/17	13:00 ~ 17:00	设计 Cache 内部的数据通路及控制逻辑

2021/12/17　　19:00 ~ 22:00　调试

2021/12/18　　08:00 ~ 12:00　撰写实验报告

（二）错误记录

1、错误 1：没有为 index_req 赋初值

（1）错误现象

复位后 data_ok 和 addr_ok 出现不定值，如图 3：

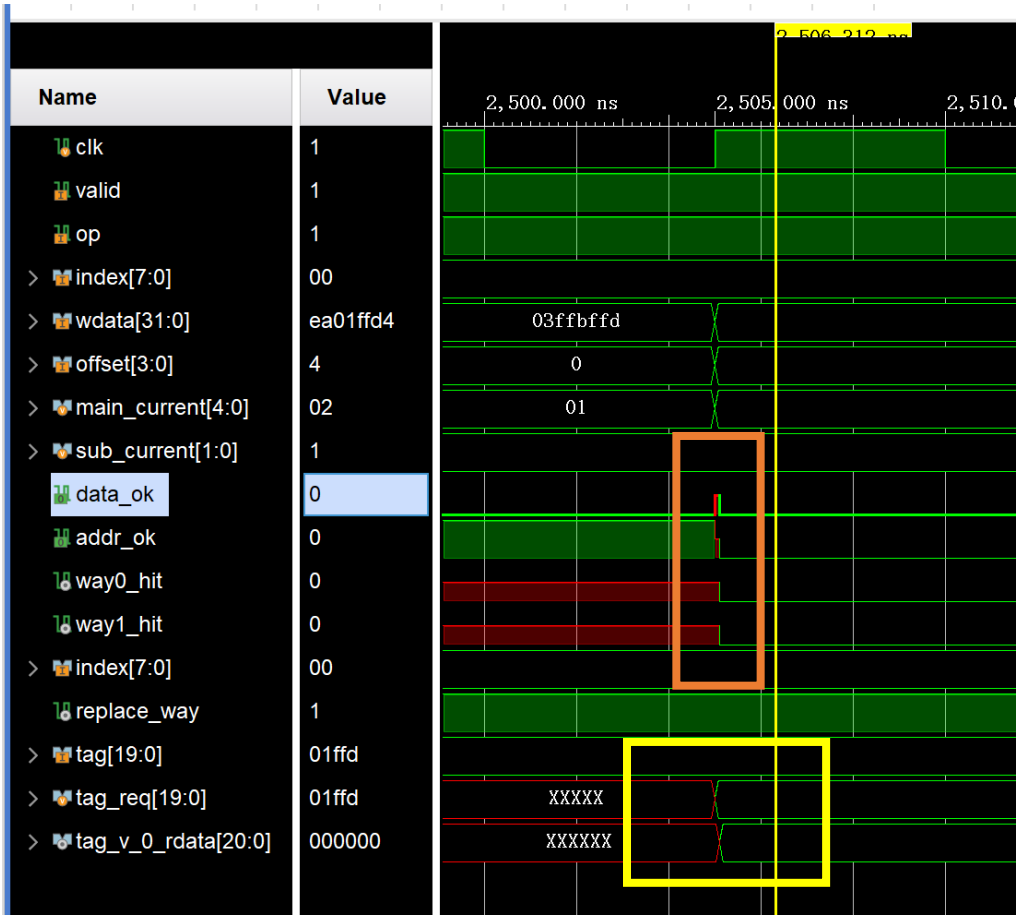

图 3　错误 1 的现象

（2）分析定位过程

此时主状态机从 IDLE 转移至 LOOKUP，应该先接收地址，所以我们先来看 addr_ok，其产生逻辑如下：

```
assign addr_ok   = main_current[0] | from_LOOKUP_to_LOOKUP;
```

main_current 是确定的，所以 from_LOOKUP_to_LOOKUP 应该是不定态，后者的产生逻辑为：

```
assign from_LOOKUP_to_LOOKUP    = main_current[1] & cache_hit & valid & ~hit_wr_conflict;
```

依次排查，发现 cache_hit 为不定态，如上图中橙色框的 way0_hit 和 way1_hit，按理来说我们已经初始化了 V 域，不可能出现这两个信号为不定态的情况，于是我们进一步查看 way0_hit 的产生：

```
assign way0_hit  = way0_v && (way0_tag == tag_req);
```

way0_v 和 way0_tag 都来自 tag_v_0_rdata，我们将其添加到波形图中，如图中黄色框。可以看到，复位后 tag_req 推迟 tag 一拍，没有问题，但 tag_v_0_rdata 相比 tag_req 却略有滞后，这中间相差的一小段不定态应该就是造成 data_ok 和 addr_ok 出现短暂不定态的原因。我们再来查看发起请求的地址 tag_v_0_addr：

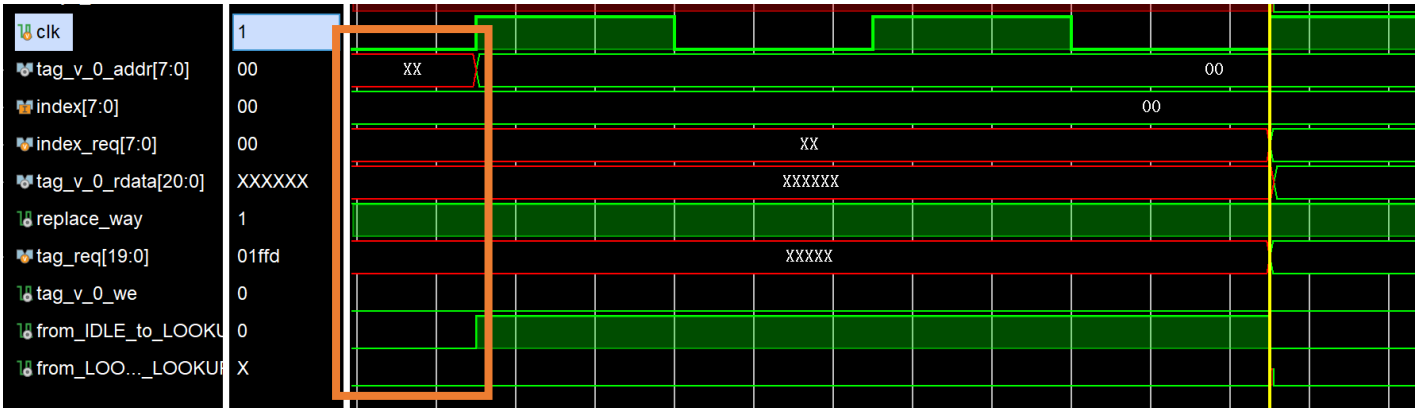

图 4　错误 1 的定位

tag_v_0_addr 只有 index 和 index_req 两个来源：

```
assign tag_v_0_addr  = (from_IDLE_to_LOOKUP | from_LOOKUP_to_LOOKUP) ? index : index_req;
```

所以 tag_v_0_addr 前面的一小段不定态应该是由 index_req 的不定态带来的，而后者出现不定态是因为我们在复位后没有为它初始化，只有接收一个读写请求时才将 index 更新到 index_req 中。

（3）错误原因

复位后没有将 index_req 初始化，导致读到了不确定的 V 域，最终产生不确定的握手信号。而至于为什么 tag_v_0_rdata 会略晚于时钟上升沿，我们猜测这和同步 RAM 这个 IP 本身有关。

（4）修正效果

复位时将 index_req 置为 8'b0，再次仿真，上述不定态去除（事实上将 index_req 复位为 0 ~ 255 中的一个值就可以，因为此时所有 Cache 行都是无效的）。

（5）总结

在本实验的环境下，上面这个错误其实并没有影响我们仿真的通过情况，但复位后握手信号出现不定值是有可能导致严重后果的，应当予以消除。

2、错误 2：没有考虑重填数据来自 wdata 的情况

（1）错误现象

在 index=00 时，第一次替换出的数据错误，见下页图 5：

```
relaunch_sim: Time (s): cpu = 00:00:02 ; elapsed = 00:00:09 . Memory (MB): peak = 1939.219 ; gain = 0.000
run all
replace wrong at index 00
=============================================================
Test end!
----FAIL!!!
$finish called at time : 2795 ns : File "C:/Users/Admin/Desktop/20211215/cache_verify/testbench/testbench.v" Line 43
```

图 5　错误 2 的现象

（2）分析定位过程

查看 testbench 判断错误的逻辑，应该是我们在发生替换时，写回到内存的数据有误，我们在波形图中查看 wr_rdy，wr_req 和 wr_data，用橙色框标出：

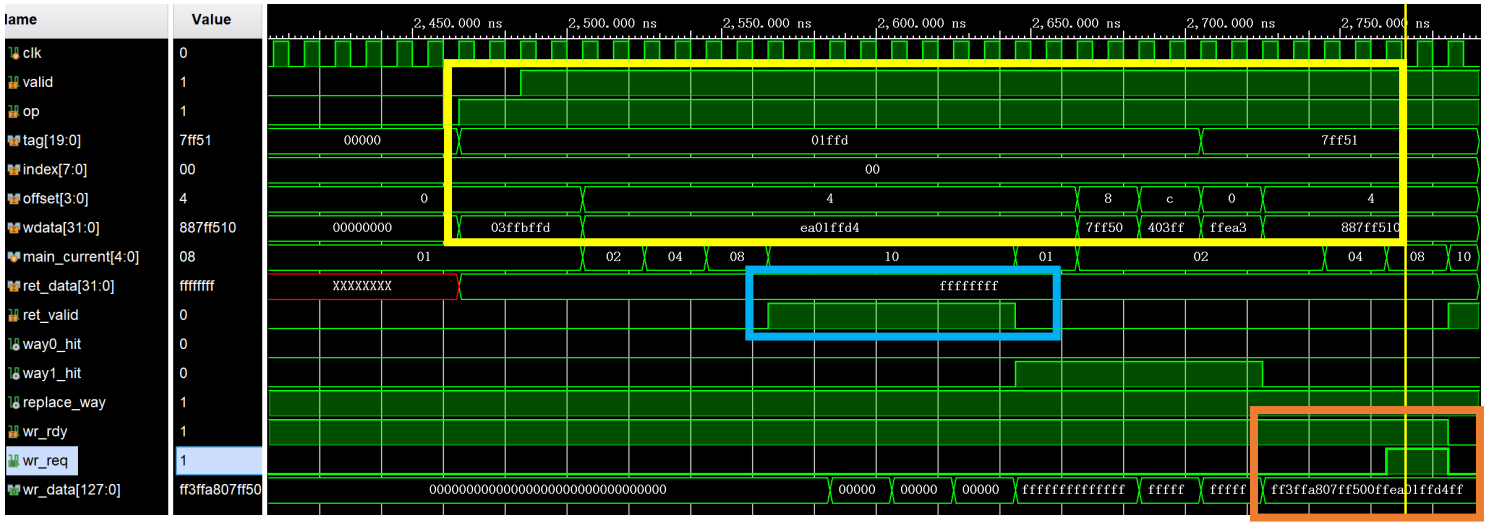

图 6　错误 2 的波形图

即此时写回的 0xff3ffa80_7ff500ff_ea01ffd4_ffffffff 是错误的。这一错误发生在第一次对 index=0 发起访问后不久，所以我们来查看 index=0 处的访问，如图中黄色框。我们首先向 tag=0x01ffd 处，分别以 offset=0x0, 0x4, 0x8, 0xc 写入 4 个数据，按照 replace_way，写入在第一路中。然后更换 tag=0x7ff51，此时根据 replace_way=1，我们将刚才写入的第一路替换出去。所以在替换前写入的 4 个数可能没有正确地写入到 Cache 中，或者第一次写入发生写不命中时，从内存取回的数没有正确地写入 Cache。根据 main_current 的状态转移，在 main_current=0x10 时进行重填，此时从内存取回 4 个 0xffffffff，如图中蓝色框。此后的 4 次写入则将这一 128bit 数的第 0～3 个字依次替换为 0x03ffbffd, 0xea01ffd4, 0x7ff500ff, 0x403ffa80，根据 piazza 上的提问，我们可以先忽略最高的一个字节，所以最高的 3 个字都写入正确了，而最低的一个字没有写入到 Cache 行中，替换出来的还是原来从内存中取回的内容。查看代码，我们向 data 域的 RAM 写入的 wdata 为：

```
assign data_way0_bank0_wdata = (from_S_IDLE_to_S_WRITE | from_S_WRITE_to_S_WRITE) ?
wdata_req : ret_data;
```

所以答案已经很明显了，我们在 REFILL 状态下直接将 AXI 总线返回的 ret_data 写入到了 RAM 中，而没有考虑写

不命中下待写入的数据 wdata_req。

（3）错误原因

写不命中时没有将写请求的数据填入 Cache

（4）修正效果

根据从 AXI 总线返回的字节数、offset_req 和 wstrb_req 来产生重填的数据 refill_data，在 REFILL 时填入 Cache 行中：

```
    assign refill_data[ 7: 0] = (ret_count==offset_req[3:2] && wstrb_req[0]) ?
wdata_req[ 7: 0] : ret_data[ 7: 0];
    assign refill_data[15: 8] = (ret_count==offset_req[3:2] && wstrb_req[1]) ?
wdata_req[15: 8] : ret_data[15: 8];
    assign refill_data[23:16] = (ret_count==offset_req[3:2] && wstrb_req[2]) ?
wdata_req[23:16] : ret_data[23:16];
    assign refill_data[31:24] = (ret_count==offset_req[3:2] && wstrb_req[3]) ?
wdata_req[31:24] : ret_data[31:24];
```

改正后，仿真继续前进到下一个错误点

3、错误 3：误把 offset_req 作为 dirty 域的索引

（1）错误现象

在 index=0x04 时，替换出的数据错误：

```
run all
index 00 finished
index 00 finished
index 01 finished
index 01 finished
index 02 finished
index 02 finished
index 03 finished
index 03 finished
replace wrong at index 04
==============================================================
Test end!
----FAIL!!!
$finish called at time : 10255 ns : File "C:/Users/Admin/Desktop/20211215/cache_verify/testbench/testbench.v" Line 43
```

图 7　错误 3 的错误现象

（2）分析定位过程

同样地，我们来查看 wr_rdy 和 wr_req 完成握手的周期，如下页图 8：

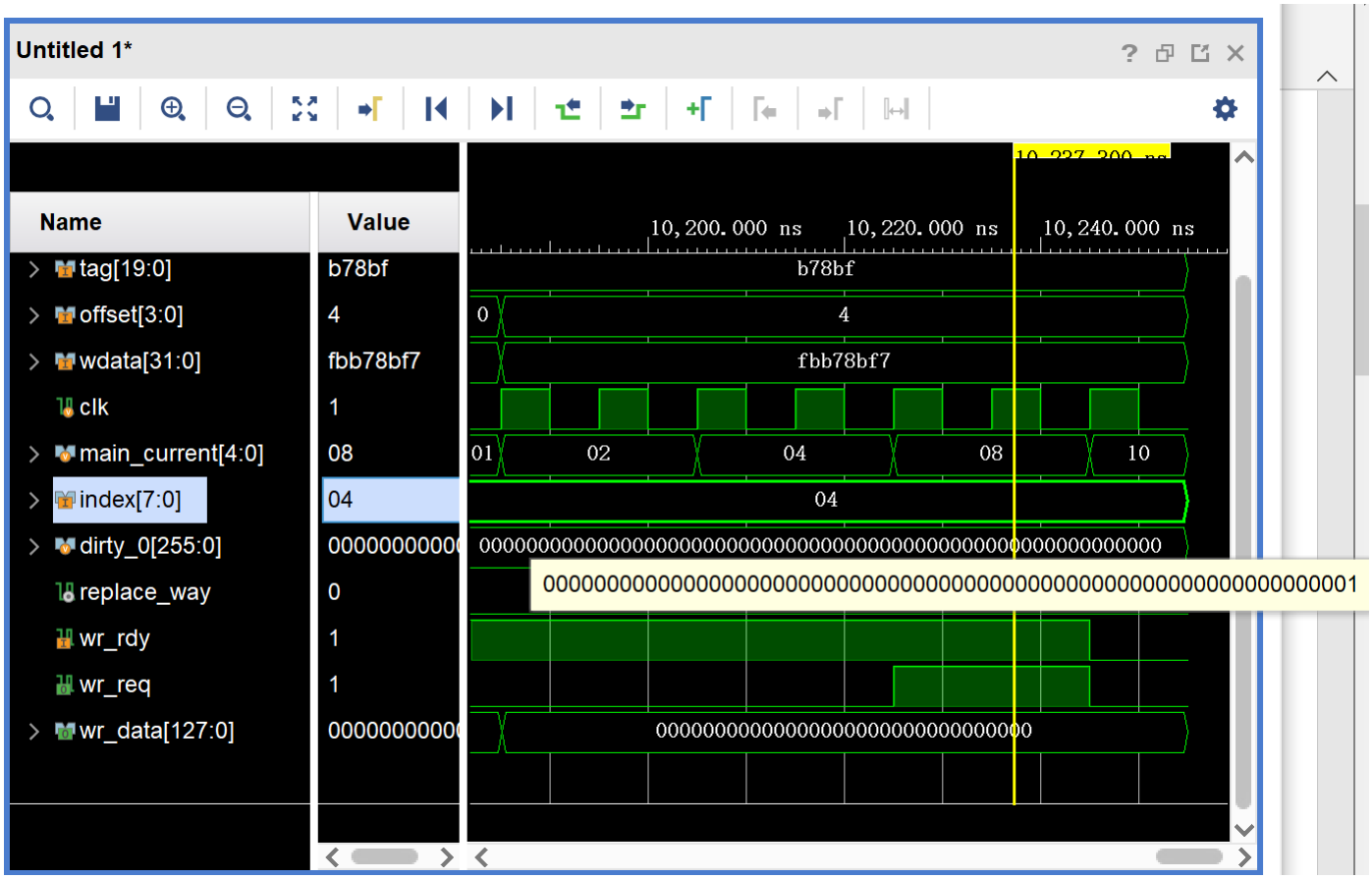

图 8　错误 3 的波形图

此时 replace_way=0 而 index=0x04，说明我们将第 0 路中 index=0x4 的数据替换出去了，将 dirty_0 添加至波形图中，以检查该数据是否应该被写回。可以看出，index=0x4 处的 dirty 位为 0，所以可以不用写回，而直接将该数据丢弃，即 wr_req 不用更新为 1，我们来查看 wr_req 的更新逻辑：

```
    reg wr_req_r;
    always @(posedge clk)
    begin
        if(rst)
        begin
            wr_req_r <= 1'b0;
        end
        else if(from_MISS_to_REPLACE)
        begin
            wr_req_r <= (replace_way == 1'b0 && dirty_0 == 1'b1 || replace_way == 1'b1 &&
dirty_1 == 1'b1);
        end
        else if(main_current[3])
        begin
            wr_req_r <= 1'b0;
```

```
        end
    end
    assign wr_req = wr_req_r;
```

发现我们直接把整个 dirty_0 用来作比较了，在我们的设计中，dirty_0 是一个 256bit 的寄存器，所以相当于我们只用了最低位来判断。顺带地，我们检查了此前是否也是按照该写法来更新 dirty_0 的，发现 dirty_0 的更新也有问题，我们将 dirty[offset_req]更新为 op_req，但实际上每一个 Cache 行有一个 dirty 位，所以一路有 256 个 dirty 位，应当以 index_req 为索引。

（3）错误原因

没有区分清楚 index_req 和 offset_req，错误使用 dirty 位。

（4）修正效果

修改以上两个错误点后，仿真通过。

四、实验总结

1. 本次实验总体看来比较简单。但状态机的设计思路和我们在组成原理中 Cache 状态机的设计有较大区别，甚至与我们平时设计状态机的思路也有一定的差别，我们平时倾向于在状态转移时发起读写请求，这样当进入下一状态时，对 RAM 或 reg_file 的读写已经完成。所以在阅读讲义时，我们对主状态机 5 个状态的作用有些摸不着头脑，所以一直有些焦虑。而在后来看到状态转移图和对外信号的产生这一部分才逐渐理解讲义中的设计思路。最后，在调试时也没有遇到太大的困难，比较顺利。