Lab 9 报告

学号：2020K8009929034、2020K8009929019、2020K8009929037
姓名：胡康、李子恒、吕星宇
箱子号：05

一、实验任务（10%）

Cache 访存速度和容量都介于寄存器和内存之间，可以有效提升访存速度，缓解访存速度与运算速度之间形成的“剪刀差”。本次实验的主要任务就是设计一个 Cache 模块。

二、实验设计（40%）

（一）总体设计思路

1. **Cache 的组织方式**：两路组相联。

2. **Cache 的访问类型**：可以从顶层上划分为四个部分，分别是 Look Up, Hit Write, Replace, Refill。Look Up 阶段，要根据 Tag, Index 和 Offset，从 Cache 中查找数据，判断是否命中。对于命中的读操作，直接返回从 Cache 中读出的数据；对于命中的写操作，进入 Hit Write 阶段，将数据写入 Cache 并设置 D 位。对于未命中的读操作和写操作，进入 Replace 和 Refill 阶段，将数据从内存中取出来，替换掉一个 Cache 行，然后根据读操作还是写操作来设置 D 位和 data 信息。

3. **Cache 表的组织结构**：每个 Cache 行含有 Tag 域, V 位, D 位, 以及 Data 域，因为 Cache 行占 16 字节，即 4 个字，所以 Data 域中保存了四个地址上的信息，可以进一步拆分为 4 个子表：Bank0, Bank1, Bank2 和 Bank3。Tag 域和 V 位表现相同，合并为一个{Tag, V}表。D 位单独组织成一个表。

3. **Cache 的数据通路**：分为五个核心部分，分别是 Request Buffer, Tag Compare, Data Select, Miss Buffer 和 Write Buffer。这一部分会在后面详细展开。

4. **Cache 的控制逻辑**：为 Cache 设计了两个状态机。所有读写操作都要经过 Look Up 阶段。对于读命中，它会直接将数据返回；对于读不命中和写不命中，它们会复用相同的数据通路来进行 Cache 行的替换；对于写命中，它需要另有通路来专门向 Cache 行中写入数据。因此为主要的通路设计了主状态机来进行控制，有五个状态：IDLE, LOOKUP, MISS, REPLACE, REFILL，而对写命中设计了 Write Buffer 状态机来进行控制，有两个状态：IDLE 和 WRITE。

（二）重要模块 1 设计：Cache 数据通路

1、工作原理

通过状态机控制，完成 Cache 的读写通路，实现 Cache 的基本功能。

2、功能描述

下面结合代码介绍一下 Cache 数据通路上的五个核心部分，Request Buffer, Tag Compare, Data Select, Miss Buffer 和 Write Buffer。

（1）Request Buffer：在 IDLE 阶段，如果发现了 Cache 请求，主状态机会进入 LOOKUP 阶段。这个时候要向 Request Buffer 中填入请求信息，包括 op, index, tag, offset 等，同时向 Cache RAM 发起读访问。代码如下图所示：

```
238  /* Request Buffer */
239  // When load/store request happens, take down request info
240  assign request_happen = (m_current_state == IDLE) && valid;
241  assign request_buffer = {op,
242                           index,
243                           tag,
244                           offset,
245                           wstrb,
246                           wdata};
247  always @(posedge clk) begin
248      if(!resetn)
249          request_buffer_r <= {69{1'b0}};
250      else if(request_happen)
251          request_buffer_r <= request_buffer;
252  end
253  assign {op_r,
254          index_r,
255          tag_r,
256          offset_r,
257          wstrb_r,
258          wdata_r} = request_buffer_r;
```

图 1：Request Buffer 设计代码

（2）Tag Compare：用于比较 tag，判断是否命中。命中的条件为：请求的 tag 和读出的两路 Cache 行中的某一行相等，且该行 V 位为 1，即该行有效。代码如下图所示：

```
422  /* Tag Compare */
423  assign way0_hit  = way0_v && (way0_tag == tag_r);
424  assign way1_hit  = way1_v && (way1_tag == tag_r);
425  assign cache_hit = way0_hit || way1_hit;
```

图 2：Tag Compare 设计代码

（3）Data Select：选出前面读出的两路 Cache 行的 Data 信息，产生读或写操作的结果。如果是读命中，只需要把命中的 Cache 行中，offset 所指示的那个字选择出来。如果是读写不命中，需要把 LFSR 所指示的那一路 Cache，也就是要被替换的 Cache 中的全部数据读出来。这里写命中没有考虑，因为它有专门的数据通路，由 Write Buffer 状态机进行管理，用 Write Buffer 存储内容。代码如下图所示：

```
427  /* Data Select */
428  assign sel_bank0 = (offset_r[3:2] == 2'b00);
429  assign sel_bank1 = (offset_r[3:2] == 2'b01);
430  assign sel_bank2 = (offset_r[3:2] == 2'b10);
431  assign sel_bank3 = (offset_r[3:2] == 2'b11);
432  assign way0_load_word = ({4{sel_bank0}} & way0_bank0_rdata) |
433                          ({4{sel_bank1}} & way0_bank1_rdata) |
434                          ({4{sel_bank2}} & way0_bank2_rdata) |
435                          ({4{sel_bank3}} & way0_bank3_rdata) ;
436  assign way1_load_word = ({4{sel_bank0}} & way1_bank0_rdata) |
437                          ({4{sel_bank1}} & way1_bank1_rdata) |
438                          ({4{sel_bank2}} & way1_bank2_rdata) |
439                          ({4{sel_bank3}} & way1_bank3_rdata) ;
440  assign load_res = ({32{way0_hit}} & way0_load_word) |
441                    ({32{way1_hit}} & way1_load_word) |
442                    ({32{~cache_hit && ret_valid}} & ret_data);
443  assign replace_data = replace_way ?
444                        {way1_bank3_rdata, way1_bank2_rdata, way1_bank1_rdata, way1_bank0_rdata} :
445                        {way0_bank3_rdata, way0_bank2_rdata, way0_bank1_rdata, way0_bank0_rdata} ;
```

图 3：Data Select 设计代码

（4）Miss Buffer：读写不命中时要替换，Miss Buffer 记录要替换的路的信息，包括要替换哪一路（replace_way），从 AXI 总线上返回的数据个数（ret_data_num），以及要替换的数据（miss_bank0/1/2/3）。其中，replace_way 直接由 LFSR 生成，ret_data_num 由 AXI 上返回的有效数据来计数，miss_bank 中存储某个 bank 要写回 Cache 的数据，它的来源有两个，一是来自于 AXI 返回的数据，也就是从内存中读出来的字；另一个是要向 Cache 中写的数据，这是不命中的 store 操作要写的数据。代码如下图所示（图中只展示了 miss_bank0，其余三个 miss_bank 代码类似）：

```
456  /* Miss Buffer */
457  assign replace_way = LFSR[0];
458  always @(posedge clk) begin
459      if(!resetn)
460          ret_data_num <= 2'b00;
461      else if(ret_valid && ret_last)
462          ret_data_num <= 2'b00;
463      else if(ret_valid)
464          ret_data_num <= ret_data_num + 2'b01;
465  end
466  // 如果是写store操作，且要写的bank是该miss_bank，那么把传入的数据
467  // 写入Cache RAM，否则还是写从Cache中读出来的数据
468  always @(posedge clk) begin
469      if (!resetn)
470          miss_bank0 <= 32'b0;
471      else if (ret_valid && ret_data_num==2'b00)
472          miss_bank0 <= (op_r == 1'b1 && sel_bank0) ? wdata_r : ret_data;
473  end
```

图 4：Miss Buffer 设计代码

（5）Write Buffer：这个是为写命中 Hit Write 专门准备的，前面已提到过多次。当 Look Up 阶段发现写命中的时候，会把 store 操作要写入的 Cache 块的相关信息先寄存起来，Write Buffer 状态机转为 WRITE 状态，再写回。代码如下图所示：

```
496  assign hit_write = (op_r == 1'b1) &&        // write
497                     (m_current_state == LOOKUP) && cache_hit ; // hit
498  assign write_buffer = {
499      way1_hit,       // way
500      offset_r[3:2],  // bank
501      wdata_r,        // data
502      wstrb_r,        // wstrb
503      index_r         // index
504  };
505  always @(posedge clk) begin
506      if(!resetn)
507          write_buffer_r <= 47'b0;
508      else if(hit_write)
509          write_buffer_r <= write_buffer;
510  end
511  assign {hw_way,
512          hw_bank,
513          hw_data,
514          hw_wstrb,
515          hw_index} = write_buffer_r;
516
```

图 5：Write Buffer 设计代码

（三）重要模块 2 设计：Cache 表读写信号生成

1、工作原理

Cache 表分为七部分，分别是{Tag, V}表，D 表和四个 bank 表。又因为是两路组相联结构，因此有两个{Tag, V}表，八个 bank 表和两个 D 表，其中{Tag, V}表使用 21×256 的同步 RAM，bank 表使用 32×256 的同步 RAM，D 表使用两个寄存器堆。这些 RAM 和寄存器堆需要有专门的读写信号来修改 Cache 的内容，从而实现 Cache 的具体行为。

2、功能描述

（1）D 表：写命中时， Cache 中写入了未被保存到内存的新值，需要将 D 位置为 1。写不命中时，从内存取出来的值在被放入 Cache 之前，要用待写入的字换掉原有的那个字，因此也需要将 D 位置为 1。读不命中时，从内存取出来的值直接被放入 Cache，并且回传到 CPU 中，没有置新值，因此 D 位置 0。综上，实现 D 位读写的代码逻辑如下（仅展示零路 Cache 的 D 表行为，一路同理）：

```
404  /* Dirty */
405  always @(posedge clk) begin
406      if(!resetn)
407          way0_d <= 256'b0;
408      else if(w_current_state == WRITE && way0_hit)
409          way0_d[index_r] <= 1'b1;    // hit write, so this line will be dirty
410      else if(m_current_state == REFILL && !replace_way)
411          way0_d[index_r] <= op_r;
412  end
```

图 6：D 表读写逻辑代码

（2）{Tag, V}表：重填时要更新 tag 表，并将 V 置为 1，地址要用寄存的 index_r。非重填时，读数的地址从 index 中直接取出即可，因为它和填入 Request Buffer 是在同一拍内进行的。代码如下图所示：

```
293   assign way0_tagv_addr = way0_tagv_we ? index_r : index;
294   assign way0_tagv_wdata = {tag_r, 1'b1};
295   assign way0_tagv_we = (m_current_state == REFILL && replace_way == 1'b0);
296   assign {way0_tag, way0_v} = way0_tagv_rdata;
```

图 7：{Tag, V}表读写逻辑代码

（3）bank 表：对于 we 写使能信号，它需要在写命中和重填时拉高。写命中时，用 wstrb 控制其使能位，重填时，需要全部赋值成 1。对于 wdata 写入数据，它需要在写命中时赋成 CPU 上传来的 wdata，而重填时需要赋成重填数据，它是 CPU 上传来的字覆盖 AXI 总线上传来的 Data 信息后生成的新信息。这里由于 Data 被拆分成了四个子表，所以直接用多路选择器和 offset 位判断各子表的赋值来源是 CPU 还是 AXI 送来的数据即可。对于地址 addr 信号，同样是来自 index 信号，根据是否命中判断地址来源即可。最终各信号的赋值方法如下图所示：

```
369   // we condition: hit write OR refill
370   assign way0_bank0_we = {4{(w_current_state == WRITE && hw_bank == 2'b00 && hw_way == 1'b0)
371                         || (ret_valid && ret_data_num == 2'b00 && replace_way == 1'b0)}}
372                        & ((ret_valid && ret_data_num == 2'b00) ? refill_bank_wstrb : hw_wstrb);
373   assign way0_bank1_we = {4{(w_current_state == WRITE && hw_bank == 2'b01 && hw_way == 1'b0)
374                         || (ret_valid && ret_data_num == 2'b01 && replace_way == 1'b0)}}
375                        & ((ret_valid && ret_data_num == 2'b01) ? refill_bank_wstrb : hw_wstrb);
376   assign way0_bank2_we = {4{(w_current_state == WRITE && hw_bank == 2'b10 && hw_way == 1'b0)
377                         || (ret_valid && ret_data_num == 2'b10 && replace_way == 1'b0)}}
378                        & ((ret_valid && ret_data_num == 2'b10) ? refill_bank_wstrb : hw_wstrb);
379   assign way0_bank3_we = {4{(w_current_state == WRITE && hw_bank == 2'b11 && hw_way == 1'b0)
380                         || (ret_valid && ret_data_num == 2'b11 && replace_way == 1'b0)}}
381                        & ((ret_valid && ret_data_num == 2'b11) ? refill_bank_wstrb : hw_wstrb);
382   assign way0_bank0_addr = ret_valid ? index_r : index;
383   assign way0_bank1_addr = ret_valid ? index_r : index;
384   assign way0_bank2_addr = ret_valid ? index_r : index;
385   assign way0_bank3_addr = ret_valid ? index_r : index;
386   assign way0_bank0_wdata = (ret_valid && ret_data_num == 2'b00) ? refill_bank_data : hw_data;
387   assign way0_bank1_wdata = (ret_valid && ret_data_num == 2'b01) ? refill_bank_data : hw_data;
388   assign way0_bank2_wdata = (ret_valid && ret_data_num == 2'b10) ? refill_bank_data : hw_data;
389   assign way0_bank3_wdata = (ret_valid && ret_data_num == 2'b11) ? refill_bank_data : hw_data;
```

图 8：4 个 bank 表读写逻辑代码

三、实验过程（50%）

（一）实验流水账

12 月 12 日　20：00-23：00　阅读讲义，理清 Cache 结构、读写操作过程、数据通路和控制逻辑，标记一些值得注意的细节。

12 月 14 日　14：00-18：00，20：00-22：00 完成初步代码设计。

12 月 15 日　13：30-15：00　仿真与 debug。

12 月 16 日　 8：00-12：30　撰写报告，完成上板验证。

（二）错误记录

1、错误 1：信号值为“Z”

（1）错误现象

如图所示，ret_last 信号的值恒为 Z：

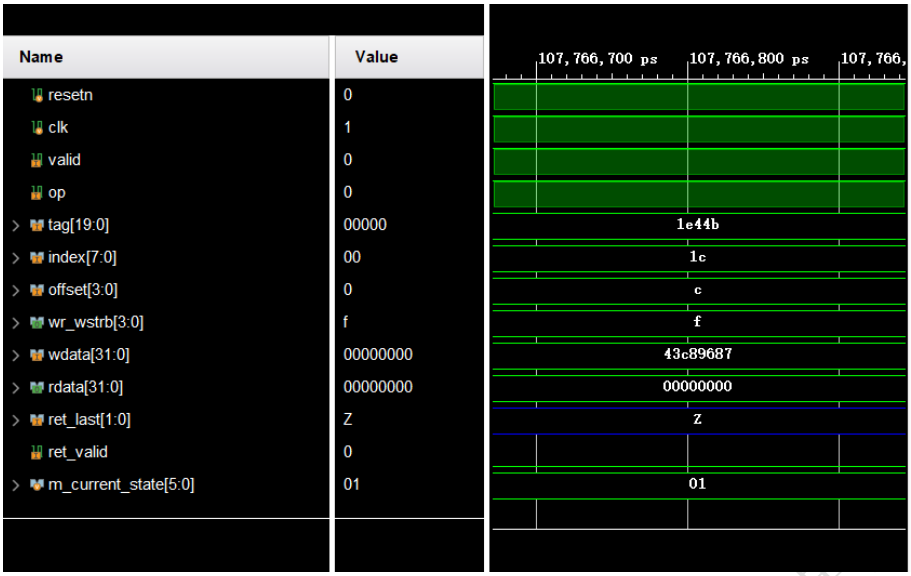

图 9：ret_last 信号值为“Z”错误波形图

（2）分析定位过程

展开这个信号，发现它的低位是有效值：

图 10：ret_last 各比特位信号值展开图

想到，可能是因为信号位的赋值不对，或者位宽搞错了。于是我找到了 cache_top.v 文件，看看顶层是怎么定义的，发现顶层文件中这个信号只有一位：

```
78   wire         rd_req;
79   wire [  2:0] rd_type;
80   wire [ 31:0] rd_addr;
81   wire         rd_rdy;
82   wire         ret_valid;
83   wire         ret_last;
84   wire [ 31:0] ret_data;
```

图 11：ret_last 在 cache_top.v 顶层文件中位宽定义为 1 位

果然是位宽搞错了，这里是讲义的位宽给错了，我直接照搬了讲义。改成 1 位之后就没有这个问题了。

（3）错误原因

信号位宽定义错误。

（4）修正效果

修改位宽为 1 位，很快便不再有 Z 值了。

（5）归纳总结

不能完全照搬讲义。当然这有时候也难以避免，最好的办法就是，不能假定讲义一定是对的，出错误的时候也要确认一下讲义是不是一定没出现手误搞错的地方。

2、错误 2：仿真通过，上板不过。

（1）错误现象

仿真虽然通过了，但是上板时数码管一直显示为 00，不递增到 FF。

（2）分析定位过程

刚开始的两天毫无头绪，后来实在没办法了，找到了一位同学，想比照一下他的代码设计，看看我的问题可能出在哪里。对照他的设计，修改了几处可能的“错误”，但是依然是仿真通过，上板不过的问题。

后来去问另一位同学，他说问题可能出在某些信号的值为 X 或者 Z 上，这样的信号在仿真阶段是 X 或 Z，但是在上板的时候一定是一个确定的信号，0 或者 1，这就有可能导致条件判断上仿真和上板结果不一样。但是我把出现信号为 X 的全部修正掉之后，也没有解决这个问题。

最后，我去和一个组员一起 debug，这时他发现我和我一开始比照的另一个同学的代码，有些端口的输入输出方向不一样。一看讲义，真的是我搞错了。我把写错的 input 和 output 方向修改过来，上板一验证，果然通过了。

（3）错误原因

部分端口信号输入输出方向定义错误。

（4）修正效果

把写错的 input 和 output 方向修改过来，仿真和上板都能通过。

（5）归纳总结

一定要细心，不能犯这种低级错误。可以写完代码之后对照讲义看一下，看看自己是否把讲义中提到的核心设计理念和设计要求都给实现了。