中山大学移动信息工程学院本科生实验报告

（2013 学年春季学期）

课程名称：**计算机组成与设计实验**　　任课教师：**黄凯**　　助教：**黄秋鹏**

年级&班级	12 级 1201 班	专业（方向）	软件工程（移动信息工程）
学号	12353022	姓名	陈胜杰
电话	13631203625	Email	1109197209@qq.com
开始日期	2014.5.19	完成日期	2014.6.13

一、实验目的

1、深入理解cache的结构和工作原理；

2、用Verilog设计cache并仿真验证功能；

3、掌握改进cache性能的方法并尝试改进。

二、实验内容

- **实验步骤**

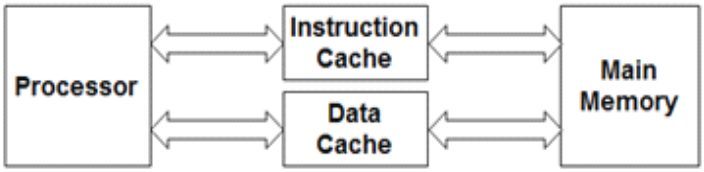
1、第一周，通过阅读给定I_Cache实现以及进行仿真分析，学习并理解较为简单的I_Cache工作原理以及Verilog实现，为D_Cache的实现打好基础；

2、第二周，在给定的工程下，模拟I_Cache实现D_Cache，并对其进行功能仿真、分析与调试；

3、第三、四周，在理论学习指导下，对I_Cache以及D_Cache进行优化，最后总结实现过程中遇到的问题以及本次project学到的经验、思考。

- **实验原理**

1、Cache 简介。



Cache 是位于 CPU 与内存之间的高速缓冲存储器，是一种特殊的存储器子系统。由静态存储芯片(SRAM)组成，容量比较小（一般为几十千字节到几兆字节），但速度比主存高得多，接近于 CPU 的速度。Cache 的功能是利用程序的空间局部性和时间局部性，用来存放那些近期需要运行的指令与数据，以提高 CPU 对存储器的访问速度。

【注】 本次试验的工程中，Cache 和 dram 的都是通过 register file 实现，而通过对于 dram 访问的一定延时控制(dram controller)来模拟现实中访问速度较为缓慢的内存。

2、Cache 映射规则。

① 直接映射规则。

```
/*-----------------------------------------------------------------------------
     采用直接相连映射方式下，cpu指定的32-bit的地址分布情况:
 |31-------------14-13---------------5-4-----------------2-1--------------0|
 |       Tag(18)           Index(9)       Word-Offset(3)     Byte-offset(2) |
 |-------------------------------------------------------------------------|
  $ ARM按照四个字节对齐     ====> Byte-offset[1:0]
  $ 每个块有8个字，即2^3    ====> Word-Offset[4:2]
  $ Cache有512个块，即2^9 ====> Index[13:5]
  $ 32-2-9-3 = 18          ====> Tag[31:14]
-----------------------------------------------------------------------------*/
```

直接映射规则，即对于内存中的任何一个数据，在 Cache 中有且只有一个块与之对应。实现中即是对于一个内存地址，只对应于一个 Cache 索引。

② 全相联映射规则。

全相联，对于内存中的一个地址，对应于 Cache 中的任何一个位置，类似于将内存中的数据直接搬入 Cache，而 Cache 作为一个中转站。

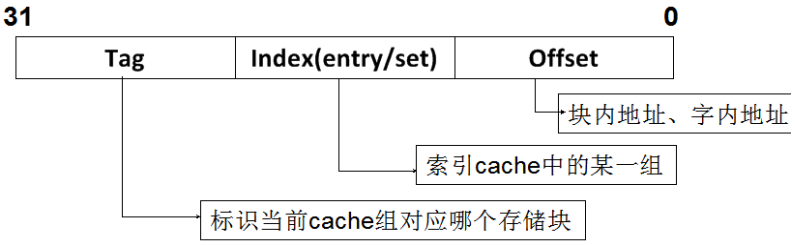
【分析】 直接映射规则能够高效进行内存与 Cache 间数据的转移以及地址的转换，而全相联则能提供一个较高的命中率。组相联是这两种方式的折中。

③ 组相联映射规则。

```
/*-----------------------------------------------------------------------------
   采用二路组相连映射方式下，同时块大小为8 words，cpu指定的32-bit的地址分布情况：
 |31-------------13-12---------------5-4-----------------2-1---------------0|
 |       Tag(19)          Index(8)       Word-Offset(3)     Byte-offset(2) |
 |-------------------------------------------------------------------------|
  $ ARM按照四个字节对齐       ====> Byte-offset[1:0]
  $ 每个块有8个字，即2^3      ====> Word-Offset[4:2]
  $ Cache有512个块，即2*2^8 ====> Index[12:5]
  $ 32-2-8-3 = 19           ====> Tag[31:13]
-----------------------------------------------------------------------------*/
```

组相联映射，结合了上述两种映射规则，一个内存地址索引到 Cache 的一个组，这里是直接映射；而具体到一个组的哪一个块则采用全相联的方式。组相联映射，折中了二者的优点，是本工程改进 Cache 性能的有效途径之一。

对于不同映射方式，我们需要对内存地址做一个合理的转换，转换关系大致如下(本工程采用 32-bit 地址线)：



3、Cache 替换策略。

当 Cache 缺失，而缺失所在块(组)存在数据，我们需要一定策略选择一个替换块来存放我们从主存中载入的数据，以使我们在未来的指令操作中，竟可能减少 Cache 缺失的可能。

① FIFO，先到先置换算法，在一开始我们实现的直接相连映射下的覆盖方式类似于这种方法；

② 随机算法，即随机置换一个块；

③ 最近最少使用算法(LRU)，这一算法运用了程序时间局部性的原理，在置换的时候，选择组中最近不怎么使用的块，将其置换，可以竟可能达到满意的命中率。在后面改进的 D_Cache 中我们模拟实现了一种近似的 LRU 算法。

4、I_Cache。

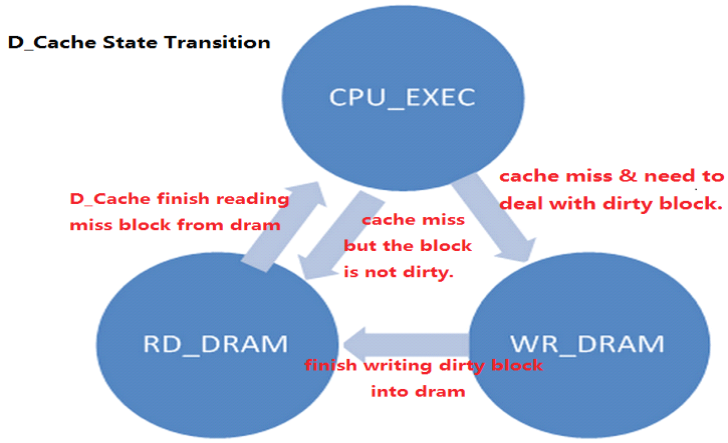
Instruction Cache 只需要完成响应 cpu 取指令操作，以及 cache miss 的时候能够从 dram 中加载相应的指令两个功能就可以了，这是因为对于指令只有读操作，不存在写操作。

【注意】当取指缺失时，I_Cache 需要产生一个 rom_abort 信号，控制 cpu_en 信号来让 cpu 暂停工作，等待 I_Cache 从主存中读取数据；同时 I_Cache 向 dram 发出读取数据的请求，完成缺失块加载到 I_Cache 中这一操作。

5、D_Cache。

与 I_Cache 相比，D_Cache 还需要提供写操作，即 cpu 写入数据到 D_Cache 以及发生置换时，写入 D_Cache 的数据需要能够写入 dram 保存(实际上使用了写回法)。因此，D_Cache 与 cpu 以及 dram 交互时均需要提供读/写的信号，从而进行相应操作的交互。

D_Cache 中的状态转换：



由于 D_Cache 的操作比较多，我们通过一个简单的三状态 FSM 来控制 D_Cache，让其在不同的状态做出相应的反应。

【注】 对于 D_Cache 的写操作，有两种比较常见的策略：①写直达法，即向 D_Cache 写入数据时，直接将数据写入主存，同时采用写不分配法，写数据的时候不将写缺失块载入 Cache，等到读缺失时再将其载入；②写回法，向 D_Cache 写入数据时，采用写分配法，为其在 Cache 中分配空间，直接在 Cache 中写入数据，当其需要被置换时，再将其写入主存。这两种方法都能保持 Cache 与主存的一致性，实验中采用较为容易实现的写回法。

6、Dram controller。

最后，说说模拟内存存取的控制器。其实是通过一个多状态的 FSM 控制其状态，让从 Cache 的数据访问在等待一定时钟周期后才得到响应。同时，支持 Cache 的以块为存取单元的操作，只需要给定块的起始地址，就能够存取一定大小的块的数据。

三、实验结果

1、编写.c 源代码，在 Ubuntu 系统下按照以下指令获取用于工程仿真的汇编指令，将指令导入工程的 program.mif 中，以便用于仿真测试。

```
shengjie@Tarantula-7:/mnt/hgfs/win7share/Today$ arm-elf-gcc -c cache_test.s
shengjie@Tarantula-7:/mnt/hgfs/win7share/Today$ arm-elf-ld.real -o cache_test.o
t cache_test.o -lc
arm-elf-ld.real: warning: cannot find entry symbol _start; defaulting to 000080
0
shengjie@Tarantula-7:/mnt/hgfs/win7share/Today$ arm-elf-objdump -S cache_test.o
t > program.s
shengjie@Tarantula-7:/mnt/hgfs/win7share/Today$
```

2、I_Cache、D_Cache 未进行优化的仿真过程与结果分析。

【实现方案】 一开始，在给定工程基础上，对 I_Cache 未做大改动，只是进行相关注释以及某些变量重命名(代码请见)；仿照 I_Cache，实现 D_Cache(代码请见)。未进行优化的一些重要参数主要有，I_Cache、D_Cache 的块大小均为 8 个 word，都是用直接映射相联的方式，对于置换，因为只有一个块，所以直接进行操作，无需考虑。下面是 I_Cache、D_Cache 实现的总体描述，具体实现请参见代码：

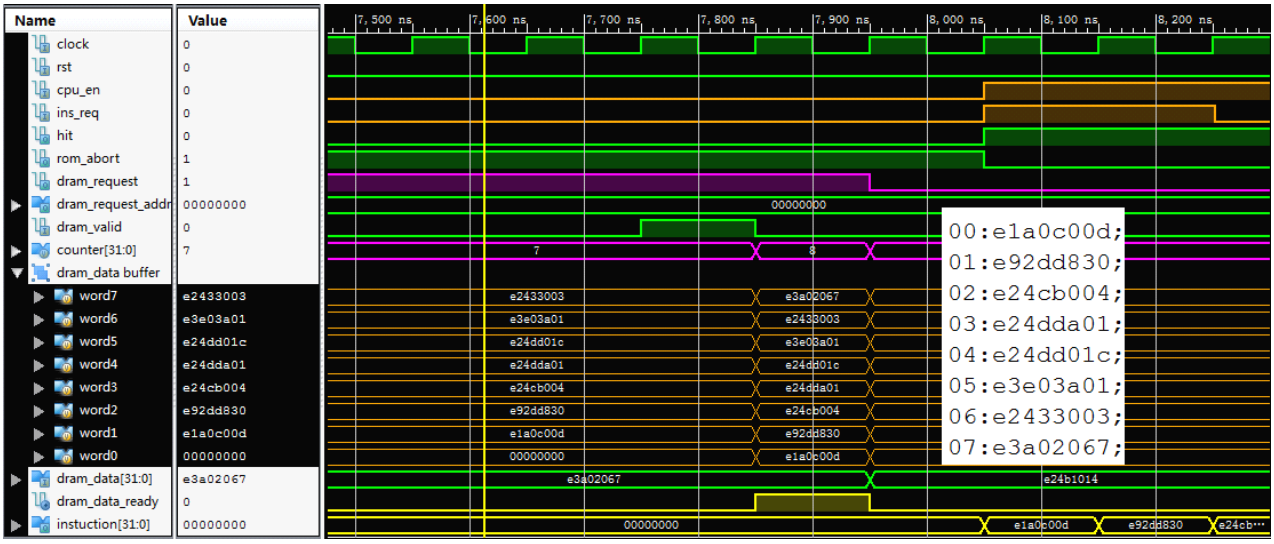
① 在 I_Cache 实现中，实现 cpu 读取指令操作，主要需要解决如何通过 cpu 地址得到对应块的内容以及通过地址在块中得到对应的指令传送给 cpu；发生 I_Cache 读取缺失的时候，如何将 cpu 的地址转为 dram 地址，交由 dram controller 进行数据读取，I_Cache 在得到 dram 返回的数据时需要对连续得到的数据进行块的构建，最后将整个块的内容写入 cpu 地址指定的 I_Cache 块中。此外，当 I_Cache 发生取指缺失时需要发出一个 rom_abort 信号让 cpu 暂停运行。

② D_Cache 的读取以及读缺失时从 dram 中载入缺失块的实现与 I_Cache 基本一致。cpu 除了读取 D_Cache 外，还能够往 D_Cache 中写数据，这里主要涉及到要写到哪个块中的哪个字段的问题。当读或者写缺失时，同 I_Cache 一样，D_Cache 需要从 dram 载入数据，但是，由于 D_Cache 不像 I_Cache 一样，其在程序运行中会被修改，我们在每个块都设置了一个脏位，当检测到要被替换的块被修改过时，使用写回法和写分配法的策略，先将该块写入 dram，即将 D_Cache 地址映射到 dram，给 dram controller 一个写请求的信号，然后将块中的 8 个字先后传送到 dram 控制器，让其完成数据写入。完成写回操作后，再将缺失块从 dram 中载入 D_Cache。

实现之后，我们通过功能仿真进行调试，分析运行下面程序的仿真结果。

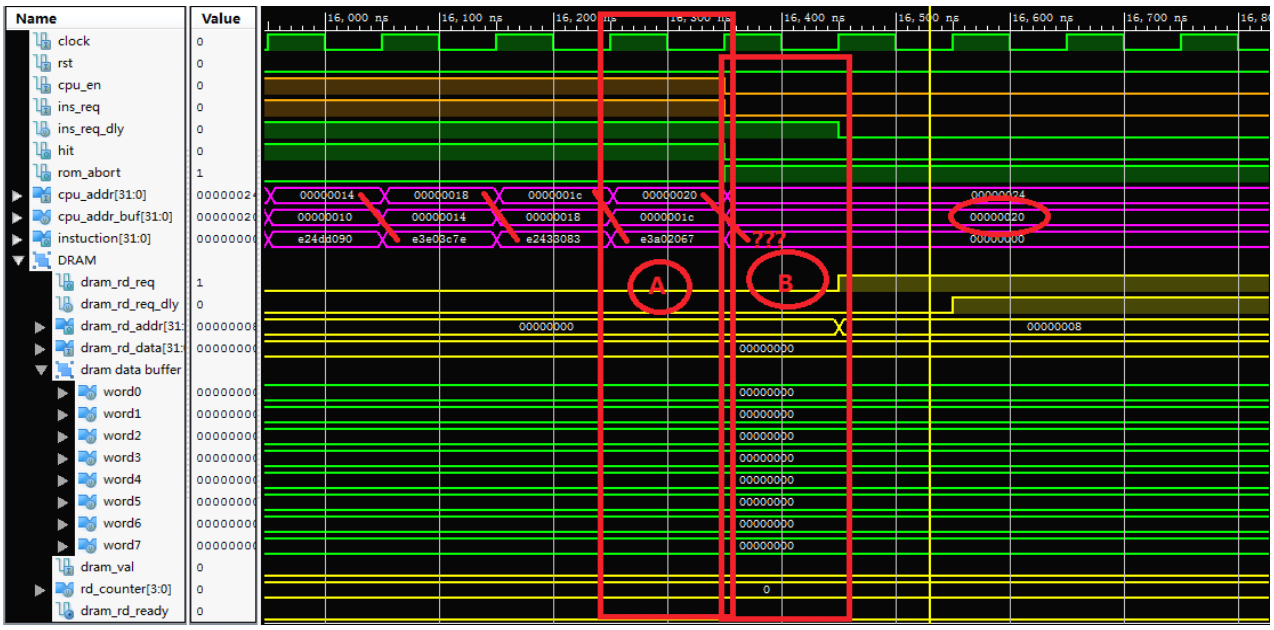
```
int __gccmain(){
    int ranNum[8096];
    int seed = 103;
    int a = 29, b = 37;
    int i;

    ranNum[0] = seed;
    for(i = 1; i < 8096; ++i){
      ranNum[i] = (a*ranNum[i-1] + b) % 1024;
    }

    return 0;
}
```

下图显示 I_Cache miss，需要从 dram 中读取块进入 I_Cache 的过程。最终 8 个字成功写入 I_Cache，在这过程中 dram_data_buffer 起到缓冲从 dram 读到的数据的作用。我们注意，这里由于 I_Cache 块大小为 8 个 word，所以一次读取只载入 8 个 word。



Figure：I_Cache 读 dram

进一步分析 I_Cache 读指令缺失时怎么处理？



Figure：cpu 从 I_Cache 取指发生缺失

首先应该明确，cpu address 总是指向下一条指令的地址。在 A 周期，当前执行指令为 0xe3a00067，我们程序的第 8 条指令。此时，结合流水线实现，cpu 同时准备从 I_Cache 中读取地址为 0x20 的指令，

所以在 B 周期 cpu 读 I_Cache 发现缺失。所以，在 A 周期结束时，rom_abort 置 1，cpu_en、ins_req 立马被拉低。B 周期的下一个周期，dram_rd_req=1，I_Cache 请求从 dram 中搬数据，通过 cpu_addr_dly 我们保留发生 cache miss 时的指令地址(我们看到，由于 pc 指令计数器在每个 CPU 周期会自动+4) 0x20，其对应的物理地址是 0x08，该地址也为其所在块的首地址。开始进行 8 个字(一个块)的数据搬移。

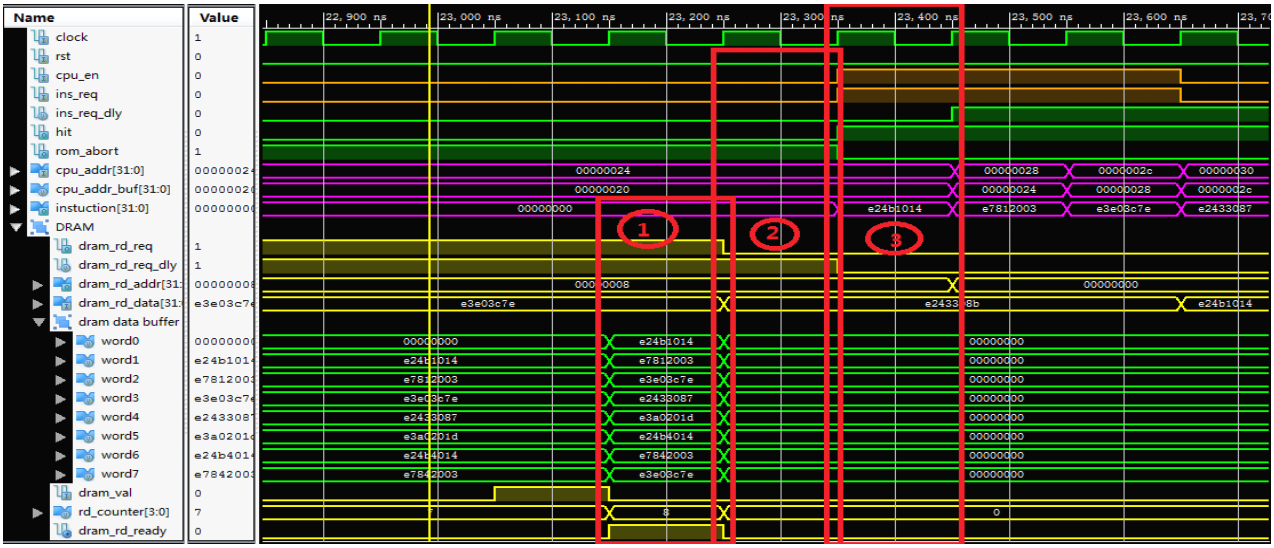


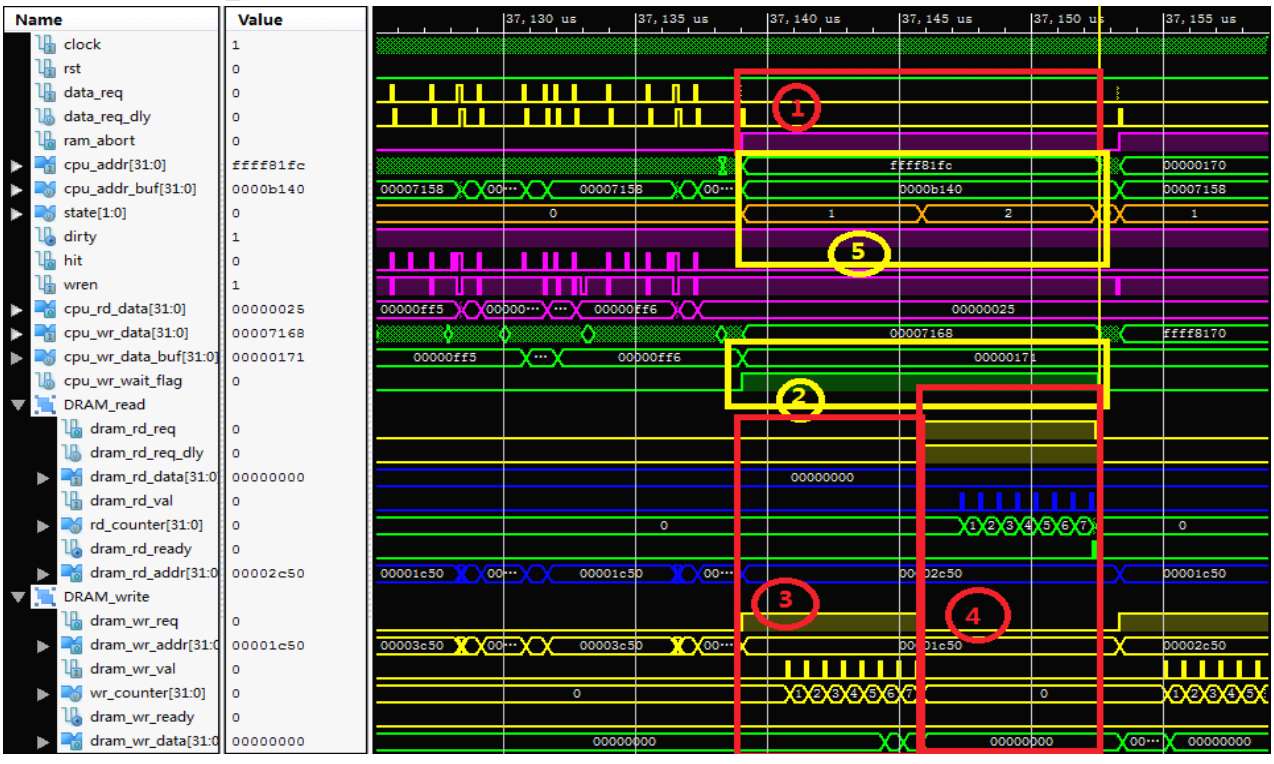
Figure: I_Cache 完成从 dram 中取指

到了仿真时间 23150ns，最后一个字搬移完成(由于模拟 dram 存取，这一块相对于 Cache 的存取，需要的时间长很多)。**Figure: Instructions in DRAM.**验证了从 dram 中进行数据搬移的正确性。该块内容在周期①写入 I_Cache。在周期②中，instruction 取得 0x20(cpu_addr_buf)对应的指令并在周期③之前传到 cpu，cpu 在周期③重新运行，正确执行下一条指令，instruction 中的 e24b1014 指令，我们程序的第 9 条指令。

	0	1	2	3	4	5	6	7
0x0	E1A0C00D	E92DD830	E24CB004	E24DDC7E	E24DD090	E3E03C7E	E2433083	E3A02067
0x8	E24B1014	E7812003	E3E03C7E	E2433087	E3A0201D	E24B4014	E7842003	E3E03C7E
0x10	E243308B	E3A02025	E24B5014	E7852003	E3E03C7E	E243307F	E3E02C7E	E2422083
0x18	E24BC014	E79C1002	E24B2014	E7821003	E3E03C7E	E243308F	E3A02001	E24B4014
0x20	E7842003	E3E02C7E	E242208F	E24B5014	E7953002	E3A02D7E	E282201F	E1530002
0x28	DA000000	EA000033	E3E02C7E	E242207F	E3E03C7E	E243308F	E24BC014	E79C1003
0x30	E1A0C001	E1A0310C	E08B1002	E2412014	E3E01C7E	E2411087	E3E0CC7E	E24CC07F

Figure: Instructions in DRAM.

下图显示了 D_Cache 向 dram 写入数据后接着从 dram 中读取缺失块的过程：



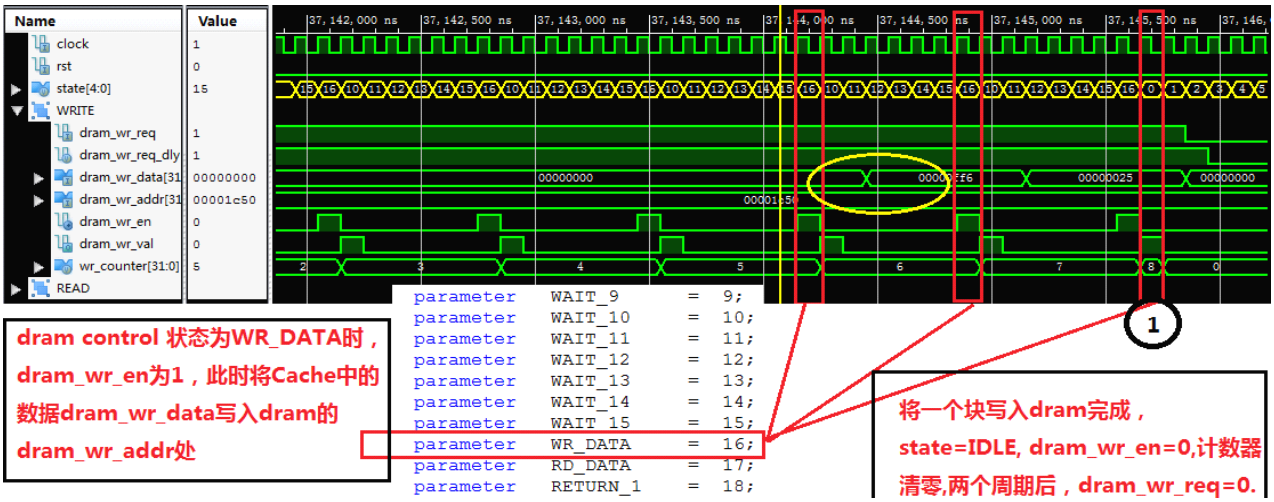
①中的 data_req=1,⑤中 wren=1，cpu 对 D_Cache 进行写操作。⑤中的 hit=0，表明未命中，ram_abort=1，②中 cpu_wr_wait_flag=1，cpu 等待 D_Cache 完成操作。

由于⑤中的 dirty=1，所以需要先将 cpu_addr 索引的块写入 dram，③中的 dram_wr_req=1；再将目标块从 dram 读入 D_Cache，④中的 dram_rd_req=1。⑤中 D_Cache 的状态 state 也反映了这个过程。在这个过程中，通过②中的 cpu_wr_data_buf 保存 cpu 要写入 D_Cache 的数据，其对应的地址则是通过⑤中的 cpu_addr_buf 保存。

③④中的读写 dram 的操作我们看到,通过 dram 返回的 valid 信号我们对写入/读取的数据进行计数,最终完成一个块的传送。

完成上述所有操作后，即 dram_wr_req、dram_rd_req 相继复 0，ram_abort 复 0，cpu_wr_wait_flag 复 0，cpu 继续执行，数据已经成功写入指定 Cache 块的正确位置。

我们对上述操作进行进一步较为细致的分析。我们分析一下 dram controller 在 D_Cache 向 dram 写入过程中的相关操作(见下图)。



Figure：dram controller 中 D_Cache 向 dram 写入数据

从上图我们发现,对于每次读取操作,需要 dram_control 发出一个写完成的 dram_wr_val 信号给 D_Cache,D_Cache 获取这个信号后，在下个周期将其该块写入字的计数器+1，通过计数器获取要写的下一个字的内容 dram_wr_data，并在下一个周期发送给 dram_control.在上图①周期，dram_control 发出最后一个字写完的有效信号，D_Cache 获取该信号后计数器+1，达到 BLOCK_SIZE，D_Cache 的 dram_wr_ready 信号置 1，D_Cache 在下个周期自动跳转到 RD_DRAM 状态，不再需要进行写 dram 操作，所以在下个周期 dram_wr_req 信号置 0，开始进行 read dram 操作。

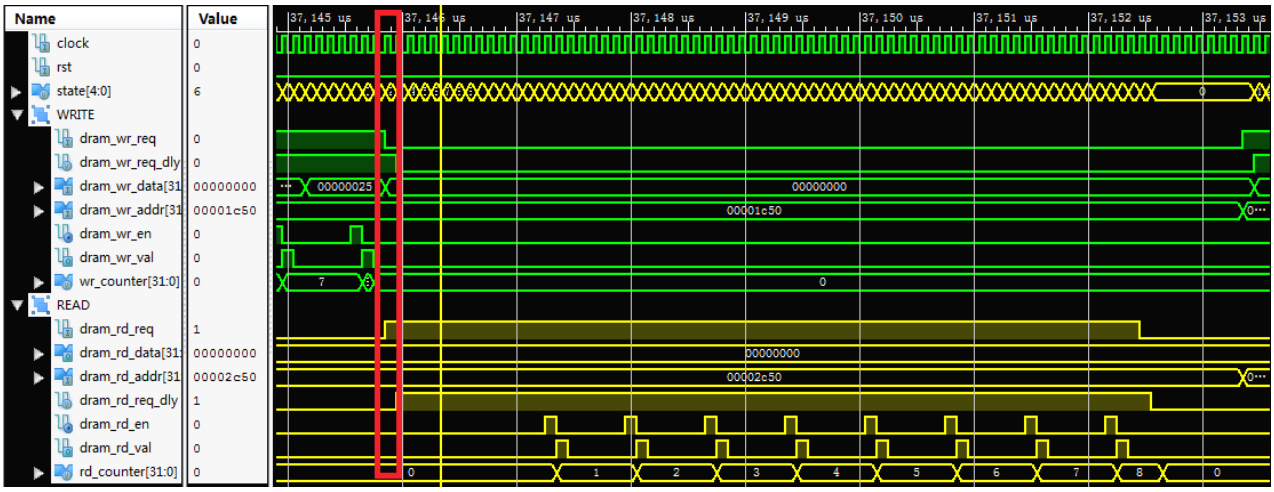
D_Cache 写 dram 操作对应下面的实现代码：

```
// Cache need to write dram when in WR_DRAM state.
   assign dram_wr_req = (state == WR_DRAM);
// get the physical dram address of the block in D_Cache to write.
   assign dram_wr_addr = {2'b0, wr_block_tag, cpu_addr_buf[13:5],3'b0};
// assignment for word to write into Dram.
   always@(posedge clock)
     begin
       if(rst) dram_wr_data <= 32'b0;
       else if(dram_wr_req) begin
         case(wr_counter[2:0])
           0: dram_wr_data <= D_SRAM[cpu_addr_buf[13:5]][31:0];
           1: dram_wr_data <= D_SRAM[cpu_addr_buf[13:5]][63:32];
           2: dram_wr_data <= D_SRAM[cpu_addr_buf[13:5]][95:64];
           3: dram_wr_data <= D_SRAM[cpu_addr_buf[13:5]][127:96];
           4: dram_wr_data <= D_SRAM[cpu_addr_buf[13:5]][159:128];
           5: dram_wr_data <= D_SRAM[cpu_addr_buf[13:5]][191:160];
           6: dram_wr_data <= D_SRAM[cpu_addr_buf[13:5]][223:192];
           7: dram_wr_data <= D_SRAM[cpu_addr_buf[13:5]][255:224];
           default: dram_wr_data <= 32'b0;    /* 还是选择第一个字?*/
         endcase
       end
     end
```

Figure：D_Cache 向 dram 写入数据部分代码实现

在完成写入 dram 的操作后，D_Cache 便开始将缺失块从 dram 读入 Cache，步骤如与 I_Cache 一致。下图

显示由于 dram 中 0x2c50 为起始地址的 8 个字内容均为 0，故写入 D_Cache 的 dram_rd_data 在写 Cache 的过程中始终是 0：



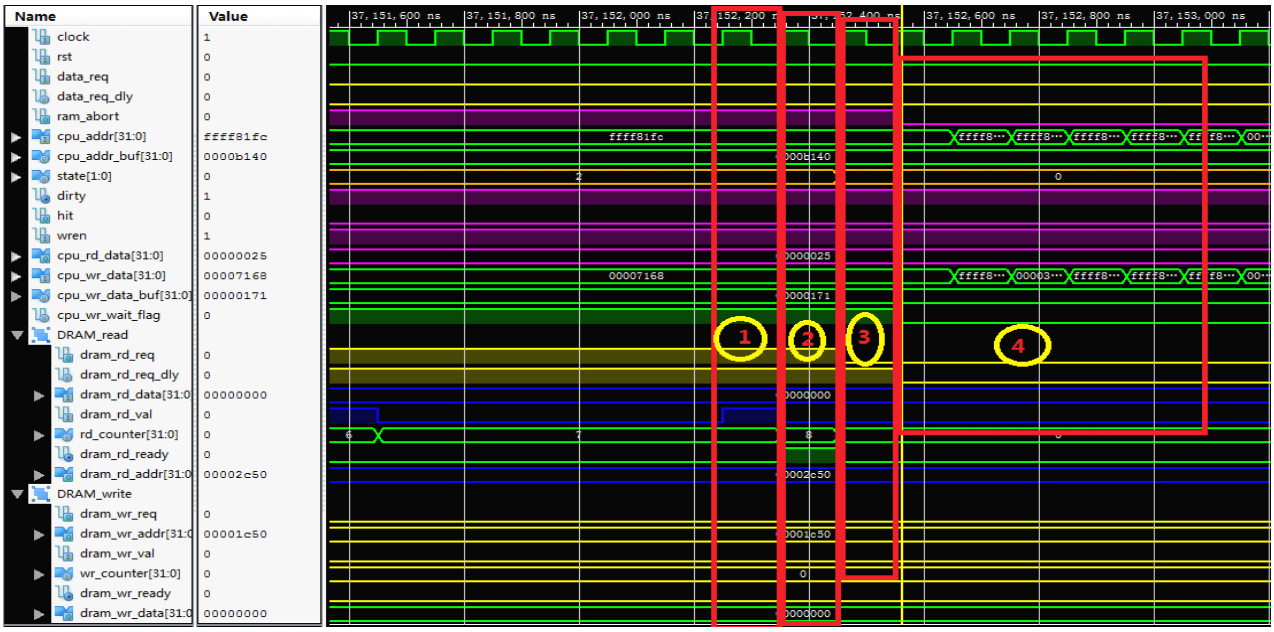
Figure：D_Cache 从 dram 中读取数据操作

仔细分析一下下面部分代码对应的仿真结果，见 **Figure：D_Cache 完成从 dram 读取数据的操作：**

```
    end
                                    // cache-miss的数据搬入cache完成后cpu写Cache.
    else if(cpu_wr_wait_flag & {dram_rd_req, dram_rd_req_dly}==2'b01) begin
      D_SRAM[cpu_addr_buf[13:5]][274] <= 1'b1;    // set the dirty bit.
      case(cpu_addr_buf[4:2])                     // 将字写入对应的位置
        0: D_SRAM[cpu_addr_buf[13:5]][31:0] <= cpu_wr_data_buf;
        1: D_SRAM[cpu_addr_buf[13:5]][63:32] <= cpu_wr_data_buf;
        2: D_SRAM[cpu_addr_buf[13:5]][95:64] <= cpu_wr_data_buf;
        3: D_SRAM[cpu_addr_buf[13:5]][127:96] <= cpu_wr_data_buf;
        4: D_SRAM[cpu_addr_buf[13:5]][159:128] <= cpu_wr_data_buf;
        5: D_SRAM[cpu_addr_buf[13:5]][191:160] <= cpu_wr_data_buf;
        6: D_SRAM[cpu_addr_buf[13:5]][223:192] <= cpu_wr_data_buf;
        7: D_SRAM[cpu_addr_buf[13:5]][255:224] <= cpu_wr_data_buf;
        default: ;                                 // otherwise, D_SRAM never changes.
      endcase
    end
  end
```

Figure：cache miss 之后，cpu 向 D_Cache 写入数据部分代码实现

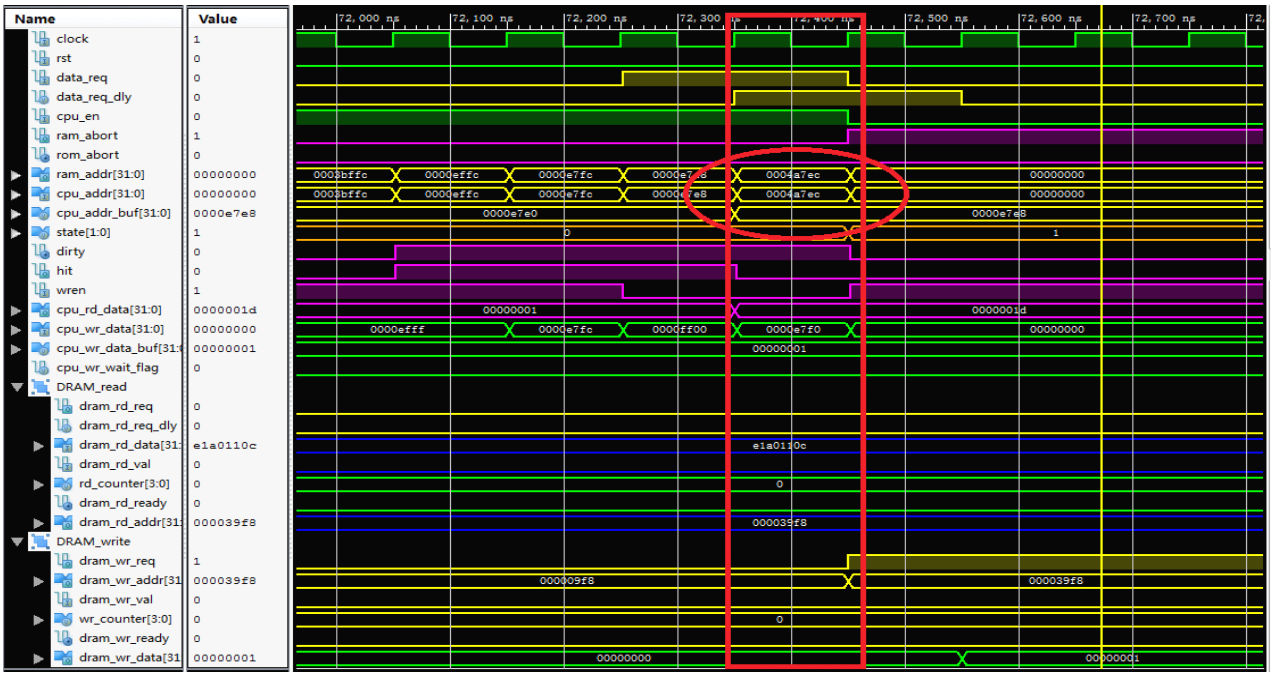
周期①中最后一个 dram_rd_val 信号表示已经将目标块最后一个字的内容读入 dram data buffer 了，从而在周期②dram_rd_ready=1，同时将目标块从 dram data buffer 写入 D_Cache，dram_rd_req 也复位。在周期③，cpu_wr_data_buf 写入 cpu_addr_buf 对应的 D_Cache 位置中，dram_rd_ready 复位。dram_rd_req_dly 也复位从而拉低 ram_abort，cpu 得以继续执行指令。



Figure：D_Cache 完成从 dram 读取数据的操作

周期③对应于上面的 cache miss，从 dram 载入对应块之后的写 Cache 操作，几个重要信号，cpu_wr_wait_flag 从检测到 cache miss 后一直保持为 1；dram_rd_req=0 表示数据已经读取完毕，不再需要从 dram 读取数据；dram_rd_req_dly=1，为 dram_rd_req 的一个延时信号；cpu_wr_data_buf 保存着一开始 cpu 要写入 cache 的数据内容；cpu_addr_buf 则是该内容指定要写入的地址；这样，cpu 需要写入的数据通过写回法、写分配法实现中成功写入 D_Cache 的正确位置。

仿真时间 723,000ns 时，连续发生两次 D_Cache 读取请求，第一次命中，第二次未能命中。仿真结果波形图如下：



Figure：D_Cache 缺失时对于缺失数据地址的解释出现错误

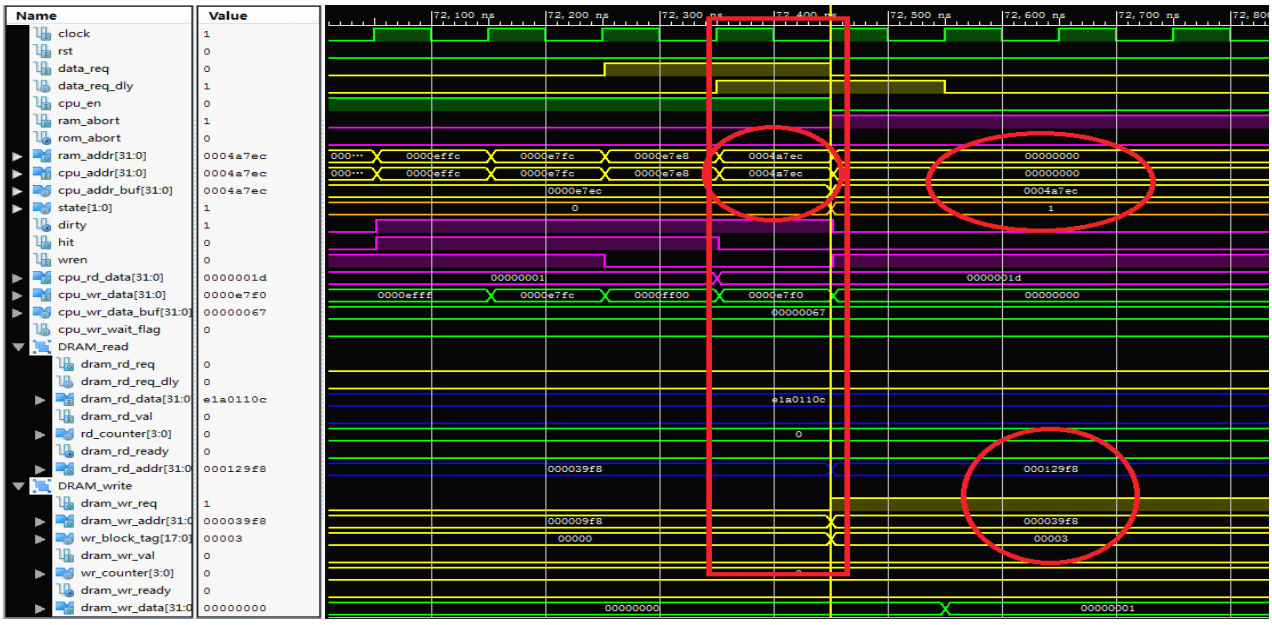
我们看到，cpu_addr_buf 不能够准确保持发生 cache miss 时的 cpu 地址 0x4a7ec，保持的是前一条并未发生 cache miss 的 cpu 地址。显然，后面的一系列操作都不能正确执行。上图仿真结果出错对应的代码如下：

```
// assignment for cpu_addr_buf, buffer for cpu_addr.
   always@(posedge clock)
     begin
       if(rst)
         cpu_addr_buf <= 0;
       else if((data_req_dly & ~hit) || dram_rd_req || dram_wr_req)
         cpu_addr_buf <= cpu_addr_buf;
       else if(data_req)
         cpu_addr_buf <= cpu_addr;
   end
```

在 if 的条件判断中出现错误，导致对于连续的 data request 操作，cpu address buffer 不能准确保存 D_Cache miss 对应的地址！经过分析之后，我们对用于保存 cache miss 时的 cpu address 还有 cpu write data 的两个寄存器的赋值作出如下修改：

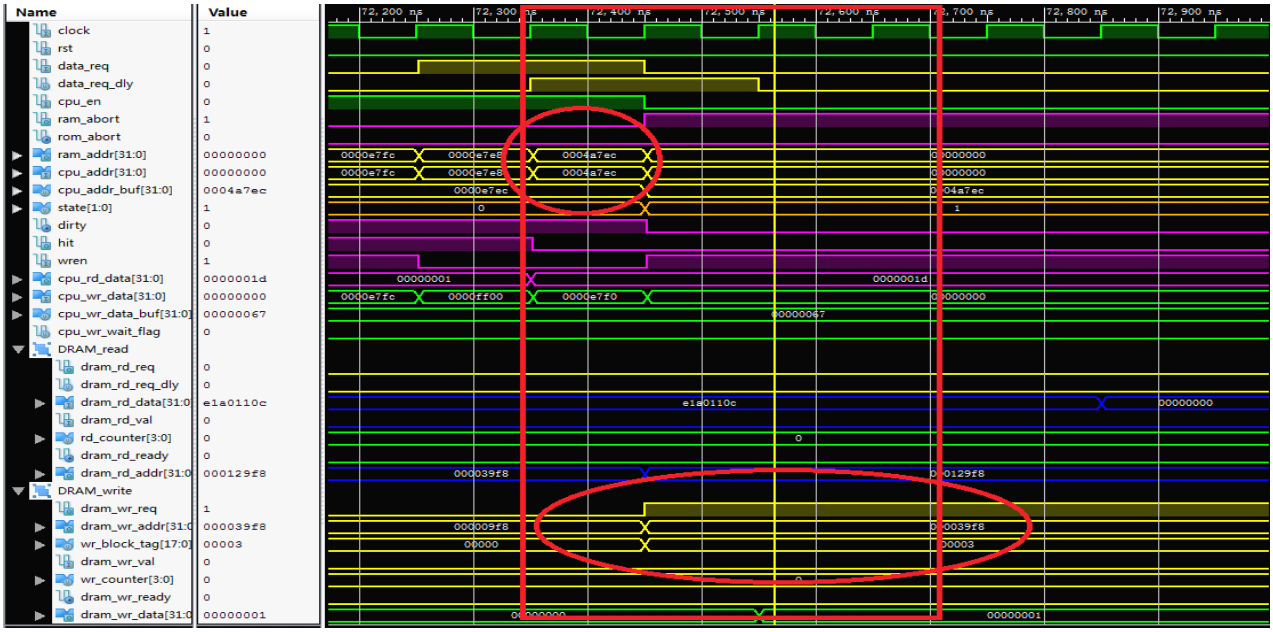
```
// assignment for cpu_addr_buf, buffer for cpu_addr.
   always@(posedge clock)
     begin
       if(rst)
         cpu_addr_buf <= 0;
       /*else if(({data_req, data_req_dly} == 2'b10) & ~hit) || dram_rd_req || dram_wr_req)
         cpu_addr_buf <= cpu_addr_buf;*/
       else if(~hit & data_req)
         cpu_addr_buf <= cpu_addr;
   end

// assignment for cpu_wr_data_buf, buffer for cpu_wr_data.
   always@(posedge clock)
      begin
         if(rst)
            cpu_wr_data_buf <= 0;
         /*else if((data_req_dly & ~hit) || dram_rd_req || dram_wr_req)
            cpu_wr_data_buf <= cpu_wr_data_buf;*/
         else if(data_req & wren & ~hit)
            cpu_wr_data_buf <= cpu_wr_data;
      end
```

让其能够正确保持发生 cache miss 时对应的 cpu address 还有 cpu write data。修改后的仿真结果如下图：



Figure：修改后，D_Cache 缺失时对于缺失数据地址能够正确解释

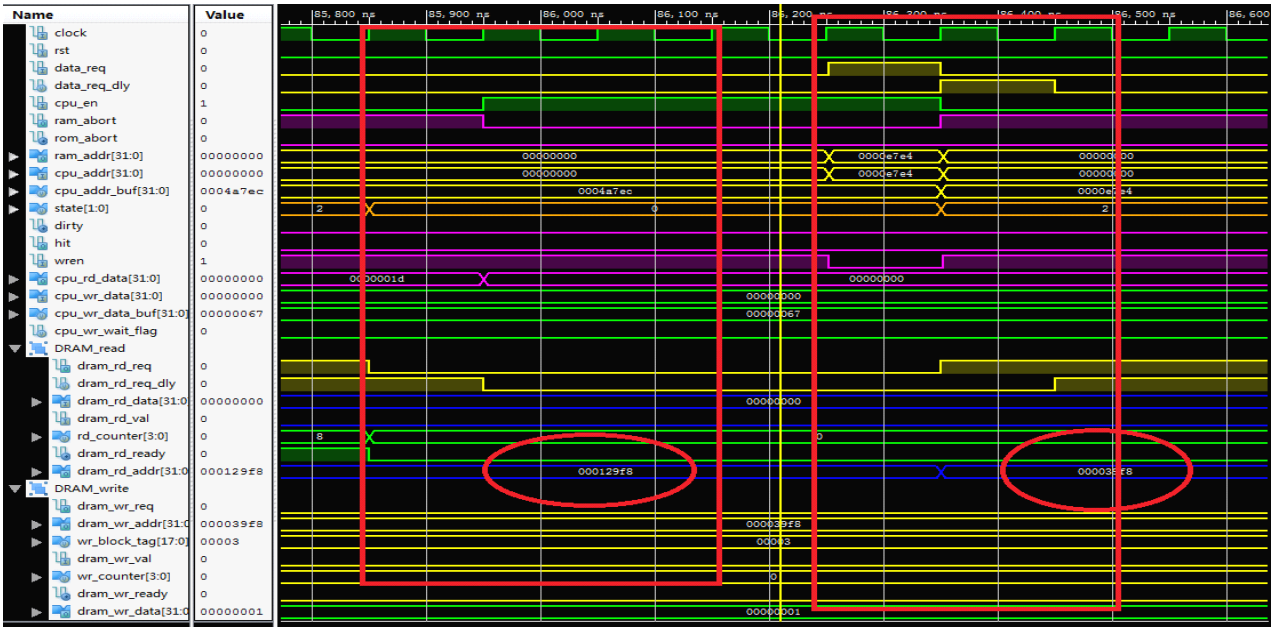
到这里，我们就分析了 I_Cache 以及 D_Cache 几大重要功能的仿真结果，通过分析仿真中出现的异常，我们做出适当的修改，最终完成所有功能。我们的实现对于几个重要功能都能够仿真正确。



Figure：D_Cache 发生缺失，先将 dirty 块写入 dram

仿真过程中，我们发现这样一个有趣的现象，仿真时间 724,000ns(见上图 **Figure：D_Cache 发生缺失，先将 dirty 块写入 dram**)，cpu 读取数据地址 0x4A7eEC 发生缺失，缺失块存放数据发生修改，在从 DRAM 中读取缺失块载入 D_Cache 之前，需要先将其写入 dram，写入 dram 地址为 0x39f8。但是，仿真时间 86250ns 时(见下图)，即将缺失块从 dram 读入 D_Cache 进行一次读取后，又需要读取刚刚写入 dram 的 cpu 地址为 0xE7E4 的数据，又不得不将其从 dram 读入(如下图 **Figure：D_Cache 发生缺失，将缺失块读入块**所示)。这一来一回，由于对于内存的存取是比较消耗时钟周期的，所以造成很多时间上的损耗，严重影响了性能，我们有必要改进 D_Cache 实现来提高命中率。

考虑到对于 D_Cache 的读取不像 I_Cache 那么较为连续，D_Cache 具有更多的随机跳变，增加块大小不能取得比较好效果，我们看到这 D_Cache 中使用组相联映射方式的必要性。所以对于性能的改进，在 I_Cache 中通过增加块的大小以减少从 dram 中取指令的平均等待时间(只需要一次较长等待，便能连续在多次较短等待中读取指令)；D_Cache 通过二路组相联的方式降低由于 Cache miss 带来的代价。



Figure：D_Cache 发生缺失，将缺失块读入块

3、I_Cache、D_Cache 进行优化后的仿真过程与结果分析。

【实现方案】 上述过程中我们已经实现了普通的 I_Cache 还有 D_Cache 并通过功能仿真验证了实现的正确性。在此基础上，我们需要改进二者的实现以求带来更好的性能。上述已经提到，通过增加 I_Cache 的块的大小可以充分利用指令的空间局部性原来，明显缩短取指消耗的时钟周期。对于 D_Cache，我们通过改变映射方式，增加 cache 命中率，减少因 cache 缺失带来的额外代价，从另一个方面进行改进。

① 首先，为了增加块大小，我们需要对 dram 控制器进行必要的修改，将 dram 对于 I_Cache 以及 D_Cache 的数据存取控制进行分开，主要涉及其对来自 I_Cache/D_Cache 的读数据操作进行区分(显然只有 D_Cache 会请求写操作)，采用两种计数机制分别对应不同类型存取操作。

为 dram controller 添加不同的状态，标记读操作是来自 I_Cache 还是 D_Cache：

```
parameter   WAIT_15     =  15;
parameter   WR_DATA     =  16;
parameter   RD_DATA_I   =  17;      /* state for I_Cache read dram.*/
parameter   RETURN_1    =  18;
parameter   RETURN_2    =  19;
parameter   RETURN_3    =  20;
parameter   RD_DATA_D   =  21;      /* Added one state for D_Cache read dram.*/
```

通过一个请求信号区分读操作来源，对块首地址的偏移量进行选择：

```
assign rd_addr_offset = idrd_req ? ird_counter : drd_counter;
```

最后，通过不同的计数机制计数不同读操作的地址偏移量，从而得到要读取数据的 dram 地址：

```
reg   [31:0]   wr_counter, ird_counter, drd_counter;      // counter for block.
wire           wr_last_data, ird_last_data, drd_last_data;// block finished.
```

② 实现了面向 I_Cache 和 D_Cache 的不同机制的 dram 控制器后，我们需要 I_Cache 实现中增加块的大小，因此带来的问题主要有：

a. 需要增加一位标记每个块中的 word offset，这就造成 tag 的位数降为 17 位：

```
/*-------------------------------------------------------------------------------
    采用直接相连映射方式下，同时块大小为16 words，cpu指定的32-bit的地址分布情况：
 |31-------------15-14---------------6-5-----------------2-1--------------0|
 |      Tag(17)           Index(9)       Word-Offset(4)     Byte-offset(2)  |
 |-----------------------------------------------------------------------|
  $ ARM按照四个字节对齐      ====> Byte-offset[1:0]
  $ 每个块有16个字，即2^4    ====> Word-Offset[5:2]
  $ Cache有512个块，即2^9 ====> Index[14:6]
  $ 32-2-9-4 = 17          ====> Tag[31:15]
-------------------------------------------------------------------------------*/
```

b. 修改从 dram 读取数据缓冲区的大小，有 8 个字扩充为 16 个字：

```
    end
    /* 通过移位寄存器保存读取指令，dram data buffer由低到高依次为先后读取的16个字的指令.*/
    else if(dram_val)
    begin
       dram_data_buf[0] <= dram_data_buf[1];
       dram_data_buf[1] <= dram_data_buf[2];
       dram_data_buf[2] <= dram_data_buf[3];
       dram_data_buf[3] <= dram_data_buf[4];
       dram_data_buf[4] <= dram_data_buf[5];
       dram_data_buf[5] <= dram_data_buf[6];
       dram_data_buf[6] <= dram_data_buf[7];
       dram_data_buf[7] <= dram_data_buf[8];
       dram_data_buf[8] <= dram_data_buf[9];
       dram_data_buf[9] <= dram_data_buf[10];
       dram_data_buf[10] <= dram_data_buf[11];
       dram_data_buf[11] <= dram_data_buf[12];
       dram_data_buf[12] <= dram_data_buf[13];
       dram_data_buf[13] <= dram_data_buf[14];
       dram_data_buf[14] <= dram_data_buf[15];
       dram_data_buf[15] <= dram_rd_data;
    end
  end
```

c. CPU 地址到 dram 地址的映射发生相应改变，其他涉及对 cpu 地址的映射的也作出相应的调整：

```
// phisical address of instructions, address from I_Cache to dram.
   always@(posedge clock)
   begin
      if(rst)
        dram_rd_addr <= 0;
      /*所谓的块是以{cpu_addr_dly[31:6], 4'b0}为起始地址，长度为8的一块dram.*/
      /*本来应该是cpu_address[31:2],去掉低两位的字节偏移量.*/
      else if(~hit & ins_req_dly)
        dram_rd_addr <= {2'b0, cpu_addr_buf[31:6], 4'b0};
      else if(hit & ins_req)
        dram_rd_addr <= 0;
   end
```

③ 在 D_Cache 中实现二路组相联，同时实现 LRU 的置换算法。

a. 首先，需要对 cpu 地址进行不同方式的映射：

```
/*-------------------------------------------------------------------------------
   采用二路组相连映射方式下, 同时块大小为8 words, cpu指定的32-bit的地址分布情况:
 |31-------------13-12---------------5-4----------------2-1---------------0|
 |       Tag(19)          Index(8)       Word-Offset(3)     Byte-offset(2) |
 |-----------------------------------------------------------------------|
  $ ARM按照四个字节对齐       ====> Byte-offset[1:0]
  $ 每个块有8个字，即2^3      ====> Word-Offset[4:2]
  $ Cache有512个块，即2*2^8 ====> Index[12:5]
  $ 32-2-8-3 = 19           ====> Tag[31:13]
-------------------------------------------------------------------------------*/
```

我们将整个 D_Cache 分成上下两个半区，每一区里面都有每个组中的一路数据，标记 D_Cache 地址最高位为 0 的为上路，为 1 的为下路，因此通过 8 位的索引，我们可以找到分布在上下两个区的一组数据。

b. 其次，需要控制对两路数据中哪一路数据进行操作。

```
/* Description: Cpu can write D_Cache, given the cpu write data and address.
      However, when cache miss happen, miss block should be taken into D_Cache,
      then to write data can be written into it. Besides, when Cache reading miss,
      miss block should also be taken into before reading.
 */
  reg cpu_wr_wait_flag;                    // cpu写Cache等待标志.
  reg [277:0] entry_data[1:0];             // entry data according to the cpu address.
  wire hit0, hit1;                         // Upper entry hit or bottom.
  reg [8:0] hit_block_index, miss_block_index; // choose which block in an entry.
  wire replace_bottom;                     // signal for replace bottom entry block.
  wire dirty0, dirty1;                     // block dirty flag.
```

其中，我通过 entry data 保存当期输入的 cpu 地址指向的组的内容：

```
// entry data is always corresponding to the cpu_addr entry.
   always@(*) begin
     entry_data[0] <= D_SRAM[{0, cpu_addr[12:5]}];
     entry_data[1] <= D_SRAM[{1, cpu_addr[12:5]}];
   end
```

我们需要不同信号提示哪一路数据命中、哪一路数据被修改。

```
// which block in the same entry hit?
assign hit0 = entry_data[0][277] & (cpu_addr[31:13]==entry_data[0][274:256]);
assign hit1 = entry_data[1][277] & (cpu_addr[31:13]==entry_data[1][274:256]);

// assignment for output signals to cpu, hit or not?
assign hit = hit0 | hit1;
```

hit or not, which entry

```
124
125  // assignment for dirty, the writing block's dirty bit.
126  assign dirty0 = entry_data[0][276];
127  assign dirty1 = entry_data[1][276];
128  assign dirty = ((dirty0 & ~replace_bottom) | (dirty1 & replace_bottom));
```

dirty

我们需要获取当前操作，包括读、写的操作目标块，而这个目标块通过 D_Cache 的索引指定。命中的时候，我们通过 hit block index 指定目标块：

```
106
107  // hit block index is always corresponding to the index of D_SRAM when hit.
108     always@(*) begin
109       if(hit & hit0 & data_req) hit_block_index <= {1'b0, cpu_addr[12:5]};
110       else if(hit & hit1 & data_req) hit_block_index <= {1'b1, cpu_addr[12:5]};
111       else hit_block_index = 0;
112     end
```

hit block index

;

缺失的时候，我们通过 miss block index 指定目标块：

```
113
114  // miss block index is used to get the miss D_SRAM index.
115     always@(posedge clock) begin
116       if(rst) miss_block_index <= 0;
117       else begin
118         if(~hit & data_req & replace_bottom)
119           miss_block_index <= {1'b1, cpu_addr[12:5]};
120         else if(~hit & data_req & ~replace_bottom)
121           miss_block_index <= {1'b0, cpu_addr[12:5]};
122       end
123     end
```

miss block index

我们通过 replace bottom 信号控制在发生置换是选择哪一块进行置换：

```
100  // assignment for bottom block replace signal.
101  assign replace_bottom = ((entry_data[0][276] == 1'b1) & (entry_data[1][276] == 1'b0))
102                        | ((entry_data[0][275] == 1'b1) & (entry_data[1][275] == 1'b0));
103  /* when the upper block is valid but the bottom,
104     or both are valid but bottom block is never used recently, replace the bottom one.
105   */
```

当两个块都有效时，我们通过每个块的使用位选择一个最近最少使用的块进行置换，为了达到这个目的，我们还需要维护该使用位。对某个块进行读/写操作时，其使用位需要置一：

```
end
else if(hit & data_req & wren) begin
  D_SRAM[hit_block_index][276] <= 1'b1;       // set the dirty bit.
  D_SRAM[hit_block_index][275] <= 1'b1;       // recently used.
  case(cpu_addr[4:2])                         // 将字写入对应的位置
```

假如同时发现同组的另一块使用位也为 1，需要将其清零，以此保证每个周期某个组至多只有一个块的使用位被标记。

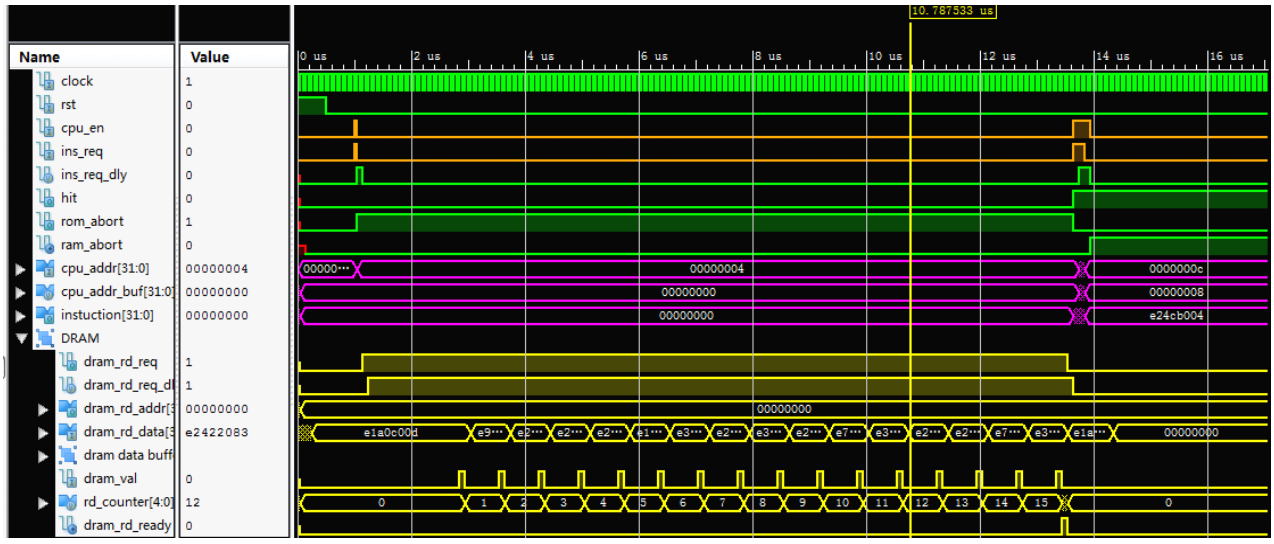
```
    endcase

    if(D_SRAM[{~hit_block_index[8], hit_block_index[7:0]}][275] == 1'b1) begin
      D_SRAM[{~hit_block_index[8], hit_block_index[7:0]}][275] <= 1'b0;
    end
  end
```

进行上述优化之后，我们选择一下程序进行功能仿真，该程序产生 512 个随机数并对其进行排序（**见附件 cache_test_512**）：

```c
int __gccmain(){
    int ranNum[512];
    int seed = 103;
    int a = 29, b = 37;
    int i, k, j, tmp;
    // Initialize the random number.
    for(i = 0; i < 512; ++i){
      ranNum[i] = 0;
    }
    // Get the random number in [0, 1023]
    ranNum[0] = seed;
    for(i = 1; i < 512; ++i){
      ranNum[i] = (a*ranNum[i-1] + b) % 1024;
    }
    // Sort the random number in nondecreasing order.
    for(i = 0; i < 511; i++){
     k = i;
     for(j = i+1; j < 512; j++){
        if(ranNum[k] > ranNum[j]){
              k = j;
         }
     }
     if(k != i){
       tmp = ranNum[k];
       ranNum[k] = ranNum[i];
       ranNum[i] = tmp;
     }
    }

    return 0;
}
```

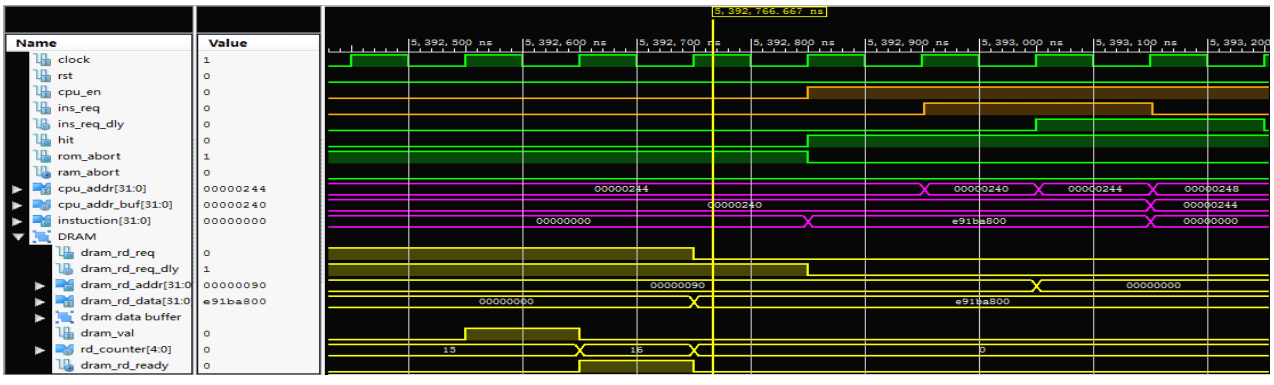
接下来，我们对仿真结果进行分析。

我们看到 I_Cache 每次取指令从原先的 8 个字成功扩展到 16 个字。

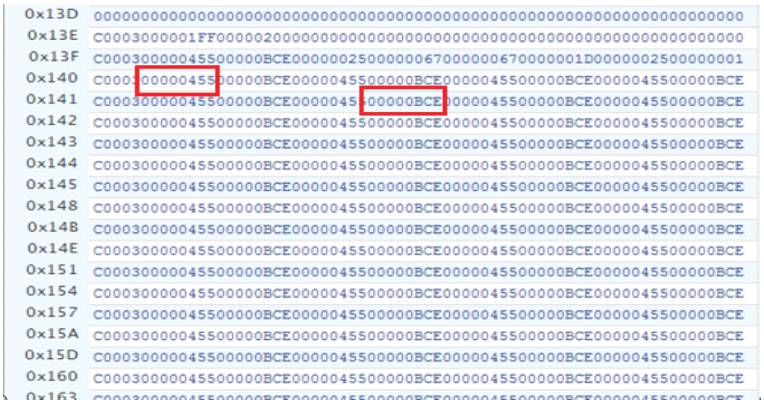


Figure：cpu 从 I_Cache 取指发生缺失，I_Cache 一次从 dram 加载一个块数据(16 条指令)

为了不重复前面使用的通过波形图进行分析的方法，这一次我们通过分析内存变化对程序运行进行分析。一开始我们直接 run all，查看程序运行结束时 D_Cache 中的数据是否正确。



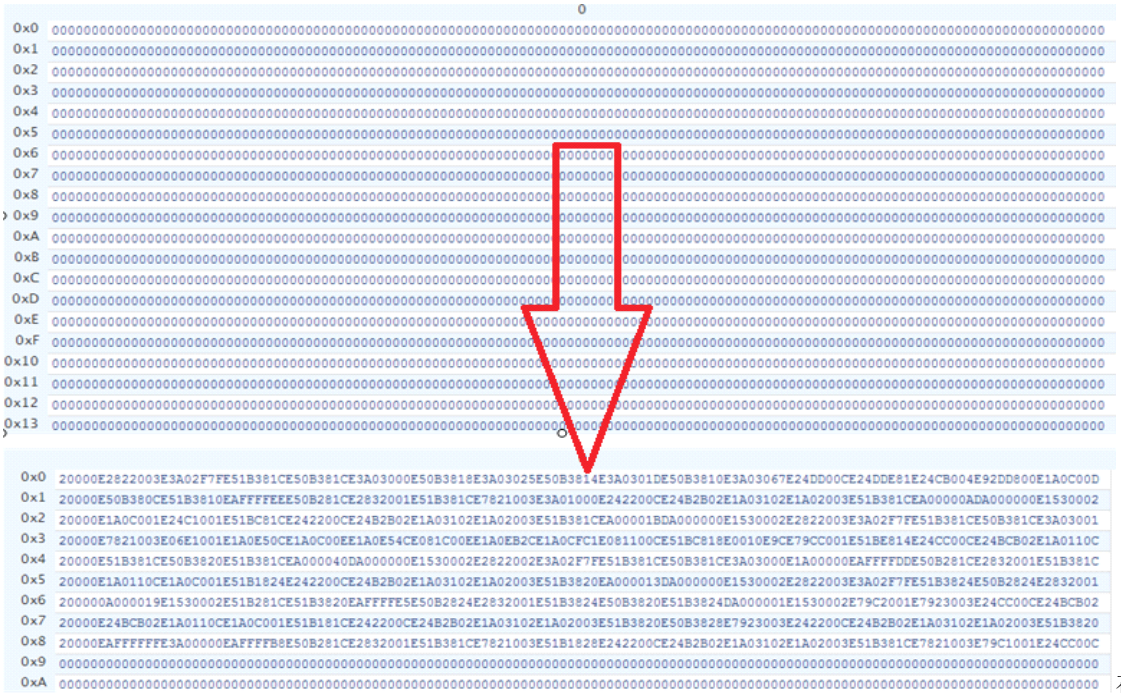
上图的仿真波形显示，pc 已经到最后一条指令了，程序运行完毕。此时，我们查看 D_Cache 内存情况：



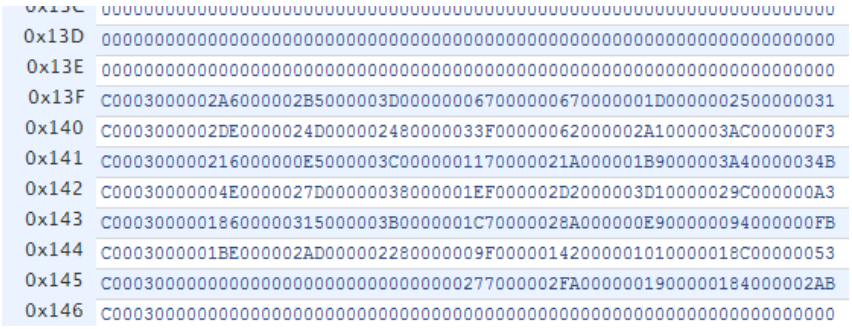
发现并没有得到我们需要的结果，虽然产生了一堆连续数据，但是数据并却没有排好序，而且这些数据都是两个奇奇怪怪的数据(0x445=1093；0xBCE=3022)，猜测是某处实现出现了错误。查找发现以下代码段存在明显的编写疏忽：

```
/* assignment for instruction.*/
   always@(*)
      case(cpu_addr_buf[5:2])                    // 块中对哪个字?
         0:  instuction = block_data[31:0];
         1:  instuction = block_data[63:32];
         2:  instuction = block_data[95:64];
         3:  instuction = block_data[127:96];
         4:  instuction = block_data[159:128];
         5:  instuction = block_data[191:160];
         6:  instuction = block_data[223:192];
         7:  instuction = block_data[255:224];
         8:  instuction = block_data[287:256];
         9:  instuction = block_data[319:288];
         10: instuction = block_data[351:320];
         11: instuction = block_data[383:352];
         12: instuction = block_data[415:384];
         13: instuction = block_data[447:446];
         14: instuction = block_data[479:448];
         15: instuction = block_data[511:480];
         default:instuction = 0;
      endcase
```

正因为这个错误，导致运行时候没能够正确得到指令，从而准确执行，我们对其进行修改。没有找到其他明显错误后，我们进行进一步仿真，最终在程序运行完所有指令后，我们看到 D_Cache 中有一堆数据，而且有序存放，我们猜测仿真结果准确。重新运行，我们作进一步的分析。仿真 512 个随机数生成后进行排序，运行一段时间后，所有程序指令从 dram 成功载入 I_Cache(如下图所示)：



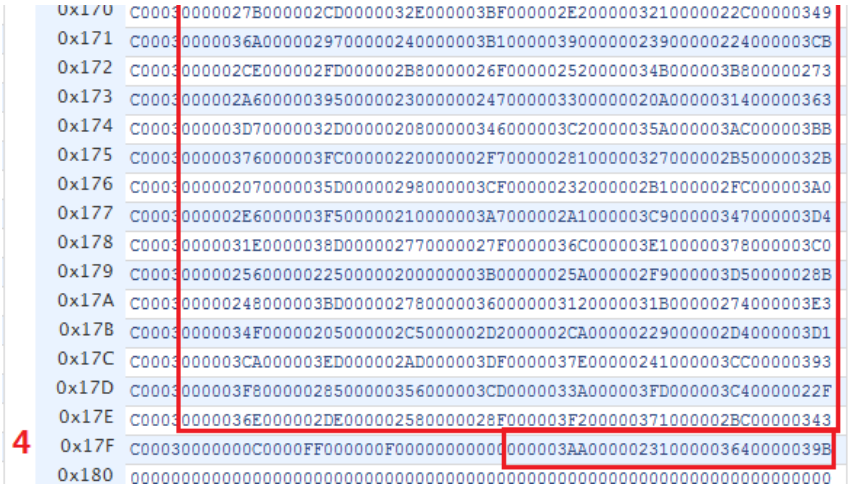
在此过程中，D_Cache 中 512 个随机数的变化如下：



【注】上图是一开始生成部分随机数截图，在后面我们将进一步分析该随机数组在 D_Cache 中的地址范围。

继续执行，下图显示全部 512 个随机数已经生成，并进行了部分排序。由于采用的是选择排序，每次都选择最小那个数，我们看到，D_Cache 中该数组开头部分已经成功排序。

最终，所有 512 个随机数成功排序：



【注】这里显示了该数组后半部分数据的内容。

我们对该数组进行分析，在 D_Cache 中，0x13F 显示的 4 个字数据，加上 0x17F 显示的 4 个字数据，以及从 0x140~0x17E 这 8*（0x17E - 0x140 + 1）= 504 个字，总共 512 个字，即该元素个数为 512 的 int 随机数数组。这样，我们就从内存上验证了仿真结果的正确性。

D_Cache 修改成二路组相联，对上述 512 个随机数产生与排序的程序进行仿真，我们发现，从头到尾 D_Cache 都不需要一次 write dram 操作，这是因为通过二路组相联，降低了 miss 率，而 D_Cache 具有许多关于数据的读取修改操作，一旦数据发生修改，又发生了 cache miss，需要替换掉发生修改的块的位置，我们需要做的不仅仅是将目标块从 dram 读入，还需要先将修改块写入 dram。所以，D_Cache 发生一次 cache miss 的代价是相当昂贵的，提高其命中率显得更加关键。考虑到数据不像指令那么具有空间局部性，所以增加块大小带来的性能提高可能不大，我们使用二路组相联的方式，来降低 miss 率。最终也能成功对这 512 个随机数进行排序(如下图所示)！

	0
0x3B	0000000000000000000000000000000000000000000000000000000000000000
0x3C	0000000000000000000000000000000000000000000000000000000000000000
0x3D	0000000000000000000000000000000000000000000000000000000000000000
0x3E	380007000001FF00000200000003FC0000000000000000000000000000000000
0x3F	3800070000006000000040000000300000001000000670000001D00000025000001FF
0x40	3800070000001D0000001900000018000000170000001400000012000000C00000009
0x41	380007000000370000003500000033000002D0000002B0000002A000000210000001E
0x42	3800070000004400000043000000420000003E0000003C0000003B0000003A00000038
0x43	3800070000005500000054000000530000005100000004E0000004C0000004A00000045
0x44	380007000006B0000006800000067000000660000006200000005D0000005B00000057
0x45	3800070000008600000084000000830000007C0000007600000072000000710000006E
0x46	380007000000920000009100000008E0000008D0000008C0000008900000088000000087
0x47	38000700000009E0000009B0000009A0000009900000096000000950000009400000093
0x48	380007000000AA000000A9000000A8000000A7000000A5000000A3000000A20000009F
0x49	380007000000C1000000BF000000BE000000BD000000BA000000B9000000B4000000AF
0x4A	380007000000CF000000CC000000CB000000C9000000C8000000C7000000C4000000C2
0x4B	380007000000E5000000E3000000E1000000DF000000DC000000DB000000D9000000D3
0x4C	380007000000F3000000F2000000EF000000EE000000ED000000E9000000E8000000E7
0x4D	3800070000010500000104000001010000000FF000000FB000000F6000000F5000000F4
0x4E	380007000001 0F0000010E0000010C0000010B0000010A000001080000010700000106
0x4F	3800070000012200000011F0000011D0000011C0000011A000001170000011600000111
0x50	380007000001340000013300000131000012F0000012A000001280000012600000123
0x51	380007000001480000014500000144000001420000013E0000013B0000013700000136
0x52	380007000001 5C000001590000015700000155000001520000015100000 14D0000014C
0x53	3800070000017400000173000001720000016B000001690000016800000165000001 5E
0x54	380007000001830000018200000181000017D0000017B0000017A0000017900000177
0x55	380007000001960000018E0000018C0000018A000001890000018800000186000000184
0x56	380007000001A6000001A5000001A2000001A10000019C0000019A0000019900000197
0x57	380007000001B6000001B5000001B4000001AE000001AD000001AB000001A9000001A8
0x58	380007000001C7000001C6000001C5000001C3000001C1000001BE000001B9000001B7
0x59	380007000001E0000001DE000001DD000001DC000001DB000001D3000001D2000001C8
0x5A	380007000001F0000001EF000001EC000001EB000001EA000001E8000001E7000001E4
0x5B	3800070000020500000202000002000000001FF000001FE000001FA000001F4000001F1
0x5C	380007000002100000020F0000020E0000020D0000020B0000020A0000020800000207
0x5D	3800070000021F0000021C0000021B0000021A0000021600000215000002130000021 1
0x5E	380007000002280000022700000226000002250000022400000223000002220000220
0x5F	3800070000023400000232000002310000023000000022F0000022E0000022C00000229
0x60	380007000002470000024600000241000002400000023F0000023D0000023900000236
0x61	3800070000025600000252000002500000024F0000024D0000024B0000024900000248
0x62	380007000002610000026000000025F0000025E0000025C0000025A0000025900000258
0x63	3800070000026F0000026D0000026C0000026A0000026900000265000002640000263
0x64	380007000002 7A0000027900000278000002770000027500000274000002730000270
0x65	3800070000028500000282000002810000028000000027F0000027E0000027D0000027B
0x66	380007000002 9D0000029C0000029800000297000002900000028F0000028B0000028A
0x67	380007000002AE000002AD000002AC000002AB000002A6000002A4000002A1000002A0
0x68	380007000002B8000002B7000002B6000002B5000002B3000002B2000002B1000002B0
0x69	380007000002CD000002CA000002C6000002C5000002C3000002C0000002BC000002BB
0x6A	380007000002D7000002D6000002D5000002D4000002D2000002D1000002D0000002CE
0x6B	380007000002E6000002E4000002E2000002E0000002DE000002DD000002DA000002D8
0x6C	380007000002F9000002F8000002F7000002F1000002F0000002EC000002EB000002EA
0x6D	3800070000030900000303000003020000030000000 2FE000002FD000002FC000002FA
0x6E	380007000003190000031800000315000003140000031300000312000003100000030D
0x6F	3800070000032900000327000003250000032400000321000003200000031E0000031B
0x70	380007000003380000033500000332000003300000032E0000032D0000032C0000032B
0x71	3800070000034300000341000003400000033F0000033D0000033C0000033A00000339
0x72	380007000003500000034F0000034E0000034B0000034A000003490000034700000346
0x73	380007000003 5F0000035D0000035B0000035A0000003580000003560000035400000353
0x74	380007000003 6A0000036700000366000003640000036300000362000003610000036 0
0x75	3800070000037600000375000003710000037000000 36F0000036E0000036D0000036C
0x76	380007000003 8B0000038700000385000003800000037F0000037E0000037C00000378
0x77	3800070000039500000394000003930000039200000391000003900000038F0000038D
0x78	380007000003A4000003A3000003A00000039F0000039E0000039D0000039B00000398
0x79	380007000003B3000003B2000003B1000003B0000003AF000003AC000003AA000003A7
0x7A	380007000003C2000003C0000003BF000003BD000003BC000003BB000003BA000003B8
0x7B	380007000003CF000003CE000003CD000003CC000003CB000003CA000003C9000003C4
0x7C	380007000003D9000003D8000003D7000003D6000003D5000003D4000003D1000003D0
0x7D	380007000003E9000003E6000003E5000003E3000003E2000003E1000003DF000003DA
0x7E	380007000003F8000003F7000003F6000003F5000003F3000003F2000003EE000003ED
0x7F	3800070000000C0000FF000000F0000000000000003FD000003FC000003FB000003F9

考虑到我们在测试程序的过程只是出现了：

1、cache miss时cache读dram;

2、cpu/dram写cache;

3、cpu读cache

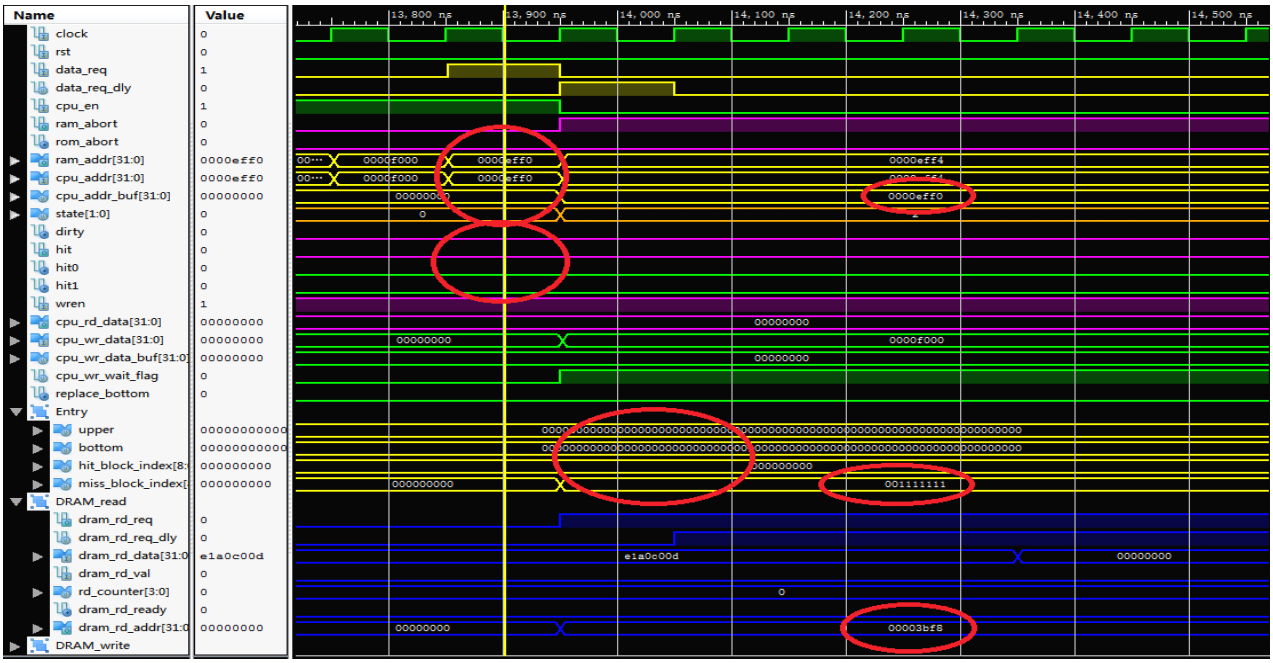
并没有出现cache写dram这种情况，为了验证我们实现的准确性与完备性，我们需要测试有更大存储空间需求的程序(能将cache填满)下，D_Cache能否正确运行。

我们采用一个下面程序作为测试程序**(见附件 cache_test_8096)**，看到，数组大小为 8096，将填满整个

D_Cache：

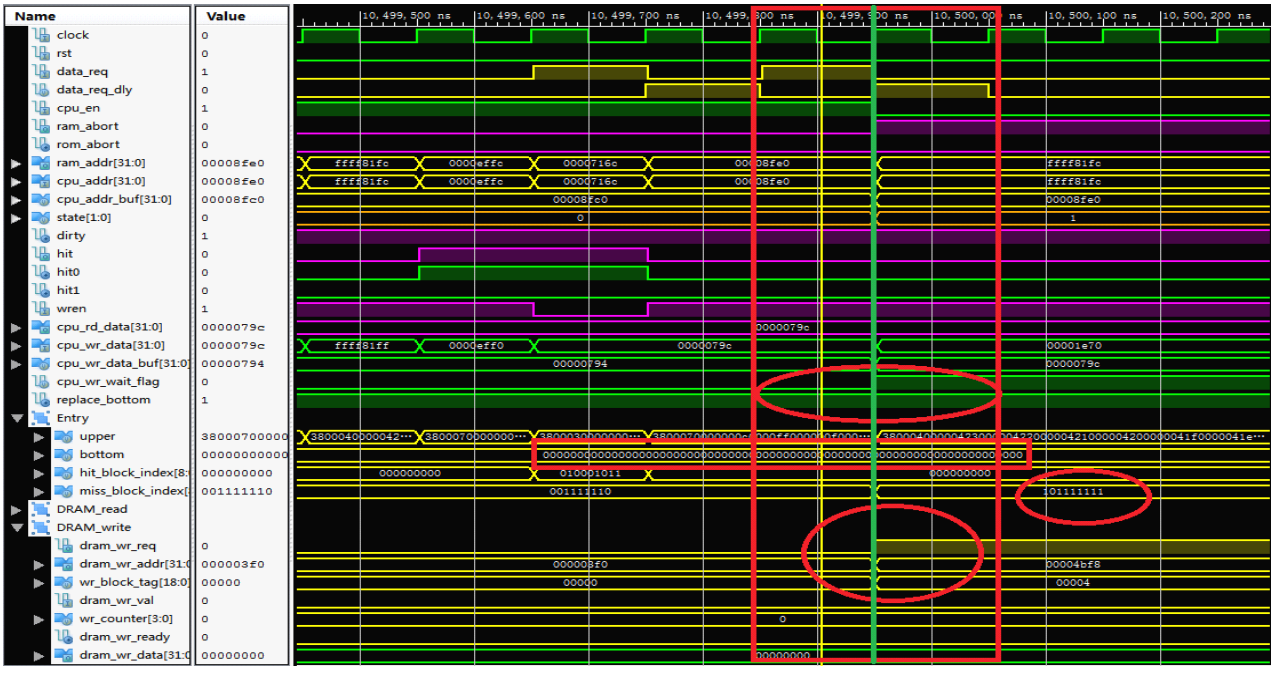
```
int __gccmain(){
    int ranNum[8096];
    int i;
    for(i = 0; i < 8096; ++i){
      ranNum[i] = i;
    }
    return 0;
}
```

下图显示了采用二路组相联后 write miss 的情况，我们对其进行分析：



hit=0，cpu 读 D_Cache 缺失，而此时 dirty=0，所以直接在下个周期，D_Cache 请求读 dram。miss_block_index 能够对应于缺失的 cpu_addr 缺失的 cpu 地址也成功转为 dram 地址，最终，在一段时间之后，缺失块被加载到该路的第一个块中。

继续执行指令，我们看到在处理 D_Cache 缺失的时候，出现错误。



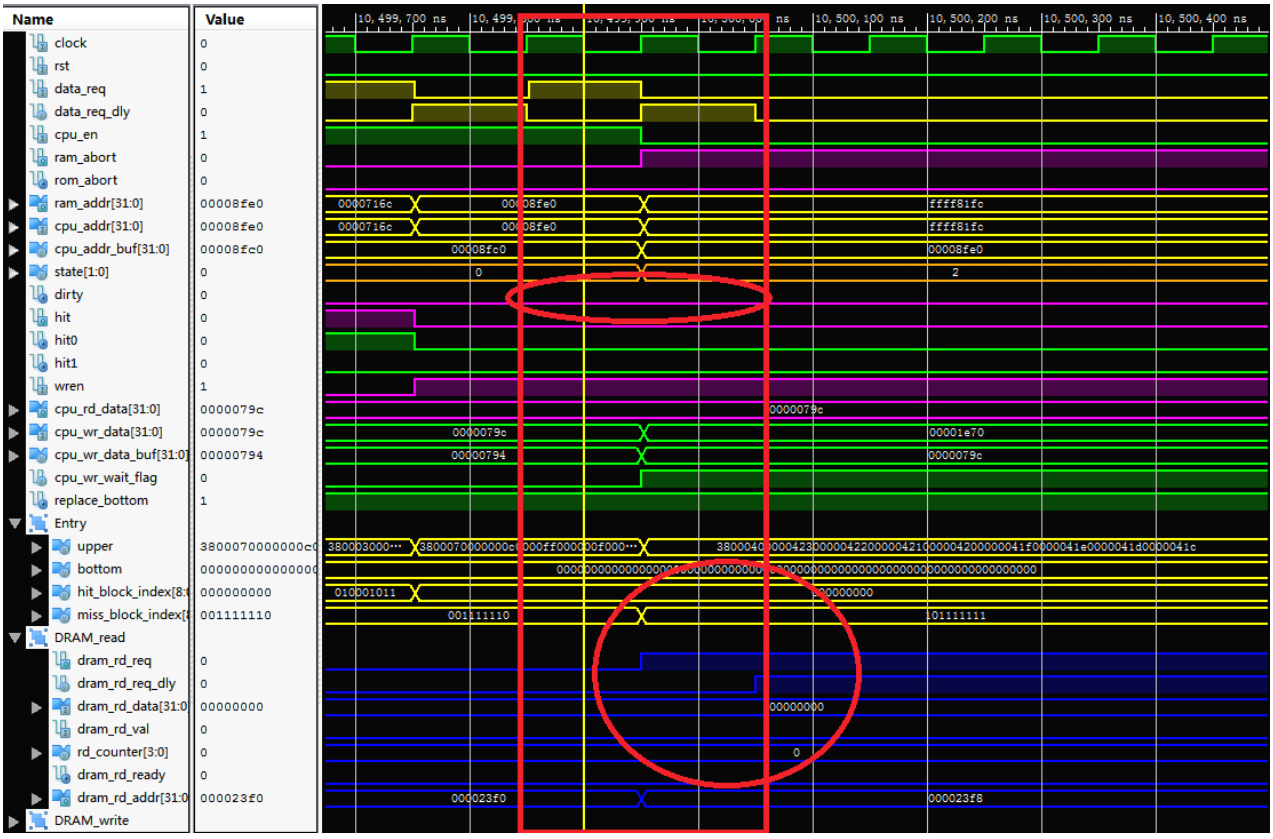
上图我们发现，写的是下路，而从该路数据显示可以知道这路数据是无效的，我们可以直接覆盖掉他。波形图中 replace_bottom 还有 miss_block_index 可以正确指出，但是，我们看到居然出现了 dram read 请求信号，这是错误的！我们看到代码，D_Cache 中的状态机 Verilog 实现如下：

```
/*---------------------- D_Cache FSM ---------------------------------------*/
   reg [1:0] state;                        // D_Cache state.
   always@(posedge clock)
     begin
       if(rst) state <= CPU_EXEC;
       else begin
         case(state)                       // 不命中时CPU需要等待D_Cache 读/写 Dram.
           CPU_EXEC : if(~hit & dirty & data_req) state <= WR_DRAM;
                      else if(~hit & data_req) state <= RD_DRAM;
           WR_DRAM  : if(dram_wr_ready) state <= RD_DRAM;
           RD_DRAM  : if(dram_rd_ready) state <= CPU_EXEC;
           default  : state <= CPU_EXEC;
         endcase
       end
     end

// assignment for dirty, the writing block's dirty bit.
assign dirty = entry_data[0][276] | entry_data[1][276];
```

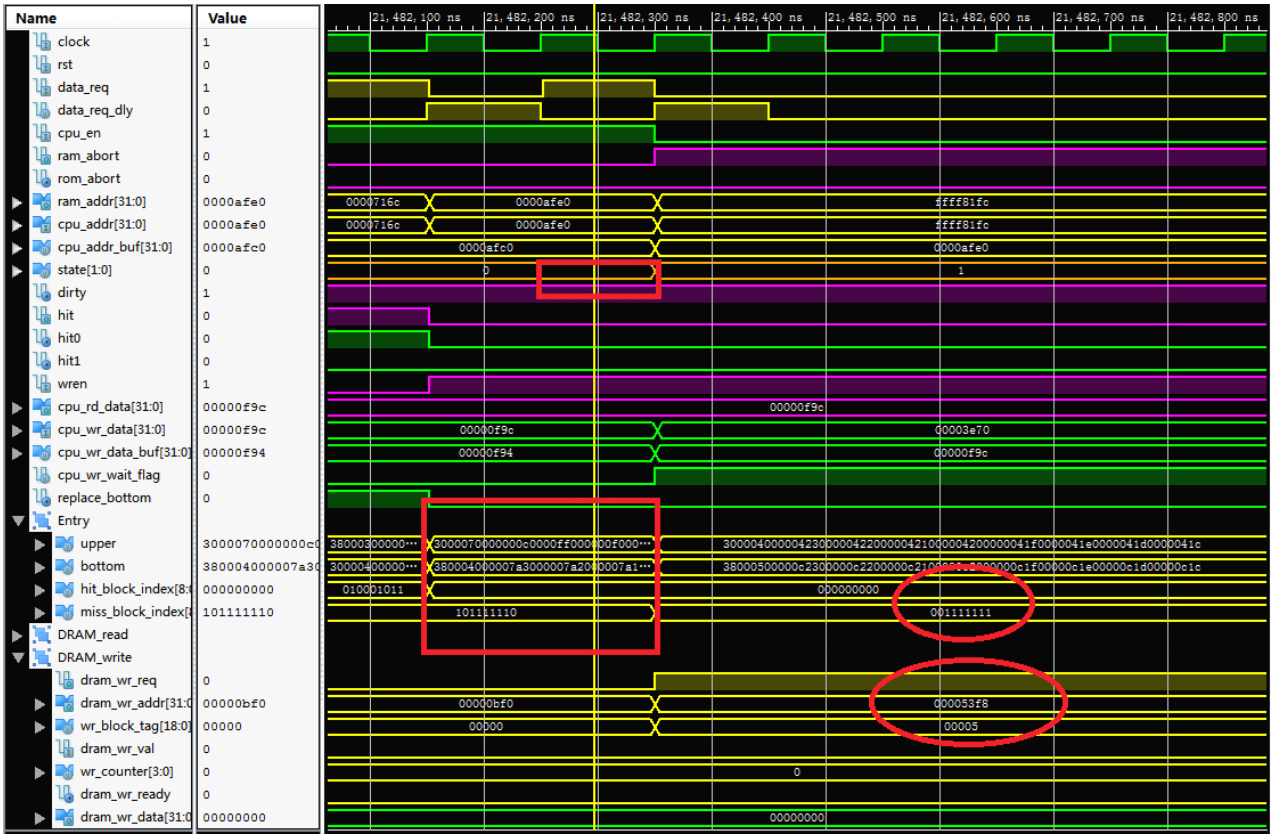
显然，这里对于 dirty 没有区分，上图中，上路数据缺失 dirty，但是下路数据并不 dirty。我们应该通过 replace_bottom 信号将 dirty 信号与具体的一路信号联系起来，对 dirty 信号的赋值具体修改如下：

```
// assignment for dirty, the writing block's dirty bit.
assign dirty0 = entry_data[0][276];
assign dirty1 = entry_data[1][276];
assign dirty = ((dirty0 & ~replace_bottom) | (dirty1 & replace_bottom));
```

经过修改之后，同样的仿真时间下，我们看到：

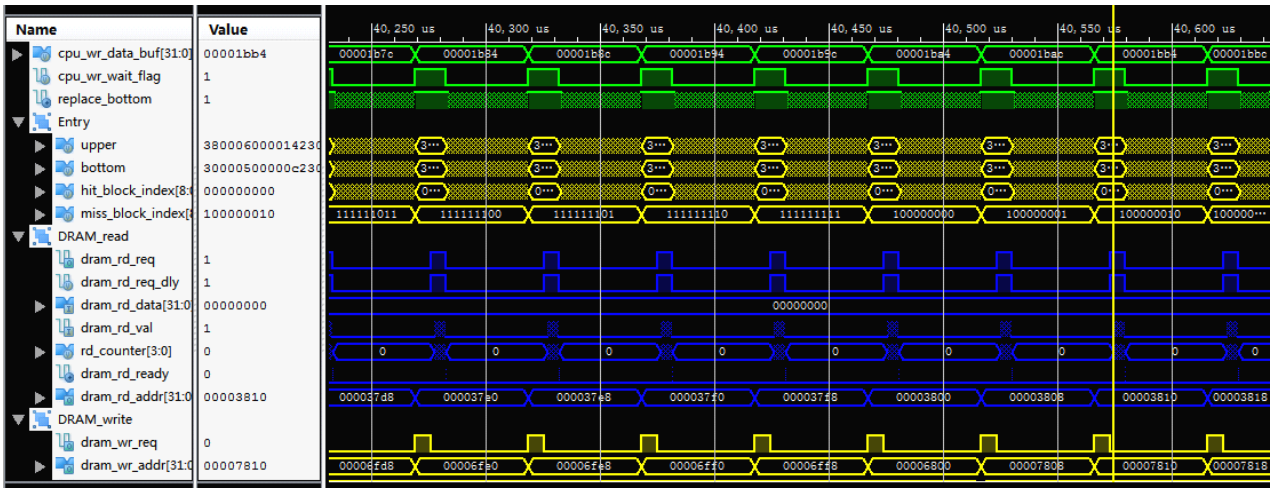


dirty 位并没有置一，而且直接从 dram 加载缺失块到 D_Cache 中，缺失块存放地址 miss_block_index 也正确选择了没有有效数据的第 2 个块。在仿真时间 21,482,300ns，出现 D_Cache 向 dram 写数据的操作，如下图所示：



我们看到，此时该路对应的两个块都存放了有效数据，而且 bottom，即下路第二块最近使用(upper=0x30...表示前三位为 3'b110，bottom=0x38...则表示前三位为 3'b111)，所以选择置换第一块，dirty 信号置一，miss_block_index 也成功与第一块对应，dram_wr_addr 正确转换。

此外，程序执行到后面，即将整个 D_Cache 填满后，我们发现(如下图)多次出现 cache miss，而且这种 miss 的代价比较昂贵的，不仅需要从 dram 中加载数据，而且需要先将数据写入 dram，成本比较大。这跟之前通过直接映射实现的 D_Cache 情况相似，因为整个 D_Cache 已经被填满，每多发生一次 cache 中数据之外的数据访问，就必须带来 dram 的读写操作，解决这个问题的方法是增加 D_Cache 的大小，让其满足程序的数据空间要求。这也是在选择 Cache 容量时经常考虑的问题。



我们同时留意了上述过程中 D_Cache 内存变化。

开始执行程序，我们在 D_Cache 中看到从 0 开始的一段连续的+1 递增的数据，这就是我们那个长度为 8096 的数组的数据：

0x8B	380003000000030000000200000001000000000000002BB0000000000000000000000000
0x8C	380003000000B0000000A0000000900000008000000070000000600000005000000004
0x8D	380003000000130000001200000011000000100000000F0000000E0000000D0000000C
0x8E	380003000000B0000001A0000001900000018000000170000001600000015000000014
0x8F	380003000000230000002200000021000000200000001F0000001E0000001D0000001C
0x90	380003000002B0000002A0000002900000028000000270000002600000025000000024
0x91	380003000003300000032000000310000003000000002F0000002E0000002D0000002C
0x92	380003000003B0000003A0000003900000038000000370000003600000035000000034
0x93	380003000004300000042000000410000004000000003F0000003E0000003D0000003C
0x94	380003000004B0000004A0000004900000048000000470000004600000045000000044
0x95	380003000005300000052000000510000005000000004F0000004E0000004D0000004C
0x96	380003000005B0000005A0000005900000058000000570000005600000055000000054
0x97	380003000006300000062000000610000006000000005F0000005E0000005D0000005C
0x98	380003000006B0000006A0000006900000068000000670000006600000065000000064
0x99	380003000007300000072000000710000007000000006F0000006E0000006D0000006C
0x9A	380003000007B0000007A0000007900000078000000770000007600000075000000074
0x9B	380003000008300000082000000810000008000000007F0000007E0000007D0000007C
0x9C	380003000008B0000008A0000008900000088000000870000008600000085000000084
0x9D	380003000009300000092000000910000009000000008F0000008E0000008D0000008C
0x9E	380003000009B0000009A0000009900000098000000970000009600000095000000094
0x9F	380003000000A3000000A2000000A1000000A0000000 9F0000009E0000009D0000009C
0xA0	380003000000AB000000AA000000A9000000A8000000A7000000A6000000A5000000A4

当整个 D_Cache 被填满后，我们看到，在 dram 中也出现一段连续的数据(下图为一小部分截图)，这些数据也是符合我们的数据要求的(0x6C60 之后的数据从 0xFFC 开始+1 递增，前一部分则是 0x6C58 之前从 FFC+1 递增)，这部分就是由于 cache miss，从 D_Cache 中写入到 dram 的部分。

	0	1	2	3	4	5	6	7
0x6C50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x6C58	00000FFC	00000FFD	00000FFE	00000FFF	00001000	00001001	00001002	00001003
0x6C60	00000804	00000805	00000806	00000807	00000808	00000809	0000080A	0000080B
0x6C68	0000080C	0000080D	0000080E	0000080F	00000810	00000811	00000812	00000813
0x6C70	00000814	00000815	00000816	00000817	00000818	00000819	0000081A	0000081B
0x6C78	0000081C	0000081D	0000081E	0000081F	00000820	00000821	00000822	00000823
0x6C80	00000824	00000825	00000826	00000827	00000828	00000829	0000082A	0000082B
0x6C88	0000082C	0000082D	0000082E	0000082F	00000830	00000831	00000832	00000833
0x6C90	00000834	00000835	00000836	00000837	00000838	00000839	0000083A	0000083B
0x6C98	0000083C	0000083D	0000083E	0000083F	00000840	00000841	00000842	00000843
0x6CA0	00000844	00000845	00000846	00000847	00000848	00000849	0000084A	0000084B
0x6CA8	0000084C	0000084D	0000084E	0000084F	00000850	00000851	00000852	00000853
0x6CB0	00000854	00000855	00000856	00000857	00000858	00000859	0000085A	0000085B
0x6CB8	0000085C	0000085D	0000085E	0000085F	00000860	00000861	00000862	00000863

这样，我们也验证了通过二路组相联实现的 D_Cache 在写 dram 部分功能正确。

四、实验感想

1、继上学期末一个月左右的 cpu 工程学习之后，这学期也在学期末迎来 cache 工程的学习。显然，通过自己用 Verilog 语言实现 Cache，一方面学习、理解理论课上关于 cache 的知识点，并通过实际实现有了自己独特的认识；另一方面，通过这一较大规模的工程的实现，从组合逻辑上、时序逻辑上、模块与模块之间的联系、各个模块中不同部分之间的联系以及一些时钟同步、信号保持的技巧上有了进一步学习，经过这一工程的锻炼，在 ISE 下使用 Verilog 硬件语言实现硬件这方面造诣有了提升。

2、 实验过程中再次体会到理论对实践的指导作用。在任何实现之前，必须要有充分的知识作为基础，结合自己的实现目标从而指定出自己的实现方案。实现之后，就需要不断的通过仿真(实验中主要通过波形仿真分析程序执行情况以及结合数据内存分布分析程序运行结果两种)对我们的实现进行验证、调试与 debug。实验中对于波形的分析是费时费精力了，但个人觉得只有通过这种方式，才能确保功能实现准确，这也是一种直接且较为高效的解决 bug 的方法。同时，在波形分析中能够使自己对实现有更深的认识，同时能够在更好的体验理论的基础上对实现中的漏洞做出修补，对不足做出改进。

3、任何实现都可能是不完美的。本次实验中，我们先是实现了基本的，能完成功能要求的 Cache，但是其在许多方面存在可以改进的地方。所以，在后面，我们基于该实现进行了改进，在改进过程同样需要仿真与分析，从而在某些方面对我们的实现策略做出总结。比如，实验中在实现 LRU 的时候，产生两

种想法：第一种，通过计数器在每个一定的时钟周期之后就对所有块的使用位清零，每个块在使用的时候才对其置 1，通过使用位情况决定置换哪个快；第二种，则是每个块在使用时使用位置 1，同时，若检测到同一路的另一个块使用位为 1，则将其清零，依次保证每个组中，使用位为 1 的块是最近使用过的块。最后，通过分析对比，选择了实现成本较低，更加符合 LRU 规则的第二种想法。

4、最后提交的有优化的实现仍存在可以进一步改进的地方，如可以增加 D_Cache 块的大小；增加 D_Cache 的容量；D_Cache 置换时考虑进置换成本，选择置换块的时候，有限选择非 dirty 块.....

附录：

1) 12353022_陈胜杰_Cache.pdf …… 本次试验实验报告
2) Cache_v0 …… 未进行优化的实现
   a. D_Cache.v …… D_Cache 模块.v 文件
   b. dram_ctrl_sim.v …… dram controller 模块.v 文件
   c. I_Cache.v …… I_Cache 模块.v 文件
   d. stimulus.v …… 仿真测试文件
3) Cache_v1 …… 进行优化的实现
   a. D_Cache.v …… D_Cache 模块.v 文件
   b. dram_ctrl_sim.v …… dram controller 模块.v 文件
   c. I_Cache.v …… I_Cache 模块.v 文件
   d. stimulus.v …… 仿真测试文件
4) Test …… 用于测试的测试用例
   a. cache_test_512 …… 512 个随机数生成与排序
   b. cache_test_8096 …… 长度 8096 的数组简单赋值
   c. cache_test_high …… 8096 个随机数生成与排序