2019 Spring ICSII

计算机体系结构实验

实验四

32位MIPS流水线处理器

（带高速缓冲CACHE）

实验报告

姓名：田睿

学号17300750084

目录

1 实验目的

通过设计高速缓冲cache，掌握cache的结构与原理；

2 32位MIPS单周期处理器（带高速缓冲）的设计思路

2.1 处理器的总体结构

与实验三相似，我对流水线各阶段的数据通路没有改动，只是在此基础上添加了指令内存的高速缓冲CacheF以及数据内存的高速内存CacheF；

因此，在接下来的实验报告中，我将不对流水线处理器进行展开的分析，只介绍两个高速缓存的实现机制；

2.2 高速缓存 cache

2.2.1 cache 顶层模块的信号

由于在实验三设计的数据通路中，有两个不同的内存模块，为了便于cache管理数据，我在两个内存上分别接了两个cache；

cache 的顶层模块中的输入输出信号如下图所示：

```
module CacheF(
    input clk,
    input reset,
    input en,
    input wen,
    input stall,
    input [31:0] add,
    input [31:0] data,
    input [63:0] updatedata,
    output [31:0] rdata,
    output [63:0] wdata,
    output [5:0] wadd,
    output wbk,
    output stallC);
```

输入信号 clk，时钟信号，只有在时钟上升沿到达时，cache内部数据才允许发生更改；

输入信号 reset，当reset信号为变为1，cache 与数据通路中其他模块一样被清零；

输入信号en，使能信号帮助判断只有在cache被需要使用的时候，cache才会进入活动状态（即可以判断hit 和 miss）；

输入信号 wen 为写使能，在需要写入内存（高速缓存）时为1；

输入信号 stall 帮助cache判断是否更新，即当stall为1时，表明内存所在流水线阶段不更新，cache也不需要改变当前状态；

输入信号 $add_{[31:0]}$ 为地址信号，cache将读取以 add 为地址的指定内存，或写入指定地址；

输入信号 $data_{[31:0]}$ 为待写入内存的数据信号，近当 wen 为 1 时，data被写入内存；

输入信号 $updatedata_{[63:0]}$ 为从内存中读取的更新信息，当cache miss 后，更新，updatedata 更新cache；

输出信号 $rdate_{[31:0]}$ 为从 cache 中根据地址 add 读取的数据信号；

输出信号 $wdate_{[63:0]}$ 由于 cache 更新，若被替换的数据已经被更改过，则需要写回内存，而 wdata 即记录了这一数据；

输出信号 $wadd_{[5:0]}$ 为写回数据的地址；

输出信号 wbk 则输出写回使能信号，该信号会作为相应内存的写使能信号；

输出信号 stallC 输出至 hazard， 当cache在更新或写回时，我采用阻塞全流水线的方法保证处理器不会出错， stallC 为1 时， hazard会收到这一信号， 并给五个阶段流水线的寄存器都穿出 stall 信号；

2.2.2 cache 的组织结构

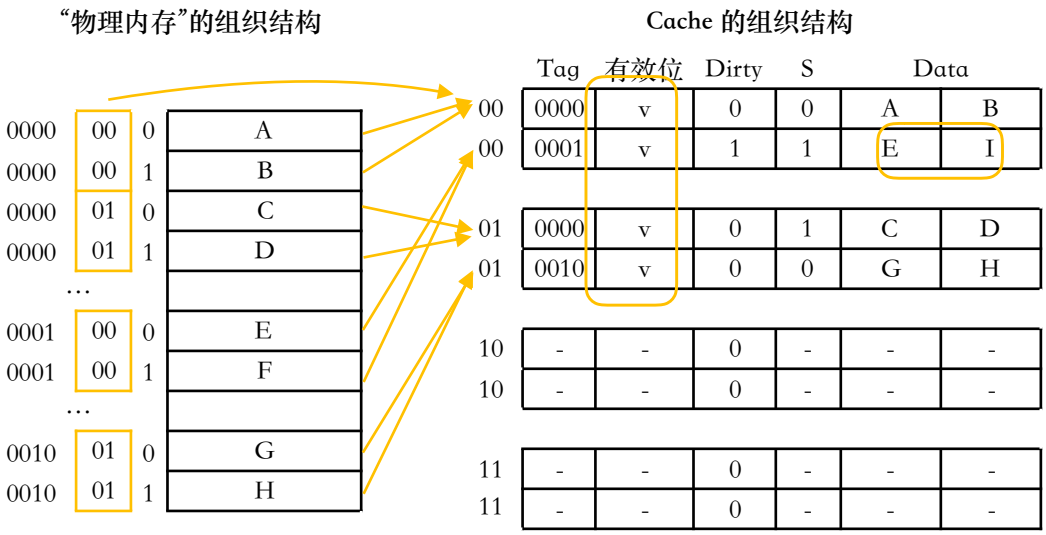


图2.2.1 cache的匹配

我实现的cache为组相连，共有四组，即四组的标识符分别为：00、01、10、11（$2^2 = 4$），并且每组共有两行，每行可存放两个 32-bit 的数据，总共可存储 $2^2 * 2 * 2 = 16$ 块数据。在mips

32 位的系统中，内存存储的数据单位为 32 位的数据；我设计的（有效）内存地址（指令内存/数据内存）为7位，因此cache采用的映射规则为：根据物理地址的第3位-第2位定位当前数据属于cache的哪一行，即用add[2:1]作为index，由此物理地址的第7位-第4位将作为标识符tag(add[6:4])；而地址的第一位决定数据在cache中存储的偏移量；

当 cache 中传入地址 add ，它将先根据 add[2:1] 定位一个list，此后，再扫描 list，看 cache line 中是否有 Tag 与 add[6:4] 一致，如果一致，再根据 add[0] 在 cache line 中定位数据；cache初始化时，有效位为 0 ，当数据加载入cache后，有效位v变为1；Dirty 位默认为 0，若cache中数据被改动，Dirty 位变为 1；S 位则记录cache中数据近期使用的情况，在更近期被读取或写入的数据的 S 位为 1，反之则为 0；

2.2.3 cache 顶层模块的实现原理

```
reg [1:0] state;
reg [1:0] next_state;

parameter  read      = 2'b00,
           writeback = 2'b01,
           updatew   = 2'b10,
           stop      = 2'b11;
```

图2.2.3 cache顶层模块中的状态

为了更方便地判断cache当前运行所执行的操作：读取、更新、写回；我在顶层模块中使用状态机，共设置四个状态：

① read：cache 在这一状态下处于工作态，读取或写入数据，此时若 miss，cache 将会相应地在下一周期执行更新（updatew）或写回（writeback）的操作

② writeback：cache 在这一状态下处于“悬停”状态，等待被改写的数据写回内存，在下一周期更新从内存中读取的数据；

③ updatew：cache在这一状态下更新 miss 的数据；

④ stop：此时 cache 虽然依旧根据输入信号输出数据，但处于未工作态，输出数据为无效数据，不被使用。与此同时，miss 信号不会使得 cache 在下一周期执行更新或写回操作。

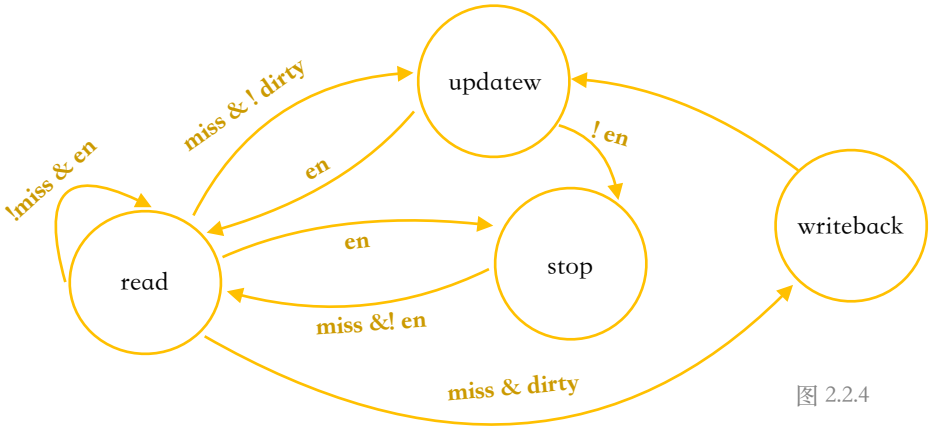


图 2.2.4

```
always @(*)
begin
  case(state)
  read:       begin
              if(miss & !dirty) next_state = updatew;
              else if(miss & dirty) next_state = writeback;
              else if(!miss & en) next_state = read;
              else next_state = stop;
                end
   writeback: next_state = updatew;
   updatew:    begin
                if(en) next_state = read;
                else next_state = stop;
               end
   stop:      begin
               if(en) next_state = read;
               else next_state = stop;
              end
  endcase
end
```

图 2.2.5

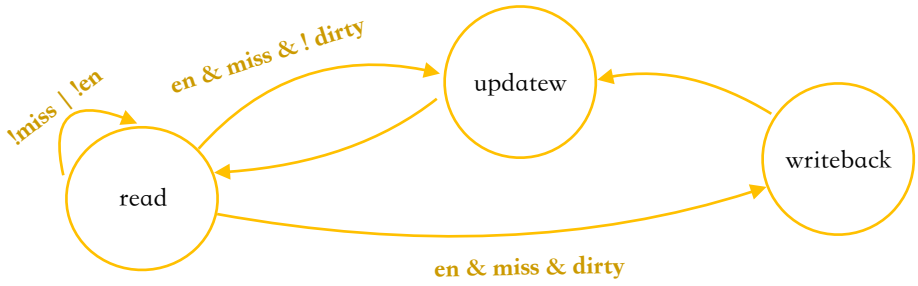


图 2.2.5

```
always @(*)
begin
  case(state)
  read:       begin
              if(en & miss & !dirty) next_state = updatew;
              else if(en & miss & dirty) next_state = writeback;
              else next_state = read;
                end
   writeback: next_state = updatew;
   updatew:   next_state = read;
   default:   next_state = read; // never happen
  endcase
end
```

图 2.2.6

简化后的 cache 状态图如图*2.2.5*所示，取消stop状态；仅在updatew 以及 writeback 状态时， cache 才会执行特殊的操作。miss 信号用以记录失配情况，以判断是否要更新 cache；dirty 信号记录失配时需要被替换的数据是否是“脏数据”，以判断是否要写回内存；

```
always @(*)
begin
  case(state)
  read:      begin update = 0; writebk = 0; end
  writeback: begin update = 0; writebk = 1; end
  updatew:   begin update = 1; writebk = 0; end
  default:   begin update = 0; writebk = 0; end // never happen
  endcase
end

assign stallc = ((next_state == writeback) | (next_state == updatew)) & ~stall;
assign wbk = writebk;
```

图 2.2.7

update 信号用于表示是否更新高速缓存，仅在 updatew 状态时，update 信号为1；

writebk 信号用于表示是否写回脏数据，仅在 writeback 状态时， writebk 信号为1, writebk 信号将作为内存的写使能；

```
always @(posedge clk or posedge reset)
begin
  if(reset)
      state <= stop;
  else if(~stall)
      state <= next_state;
end
```

图 2.2.8

可知仅当输入信号 **stall** 为 0 时，即所在流水线未被阻塞时，cache 状态可以更新：

但值得考量的是，这个 stall 信号是经过过滤的。当 cache 的状态即将跳转到 updatew 或 writeback 状态时，由于 cache 自身发出 stallC 信号（阻塞流水线处理器，以便与内存交互后数据不会出错），当前阶段流水线寄存器接收到的 stall 信号有大概率是 1，而此时cache 状态却需要被更新；并且，由综合逻辑输出的信号再次经逻辑组合作为输入信号，将会产生 ***Timing Loop***；因此我们应另输入经过处理的stall信号，将经过转发产生的 stallC 信号过滤，只考虑由其他原因造成的stall信号。此时，cache的更新是正常的。

e.g. 以CacheF为例，其传入stall信号为：**stallF & ～stallcF**

（不考虑 CacheF 传出的阻塞信号，考虑全部其他造成 Fetch 阻塞的信号）

stallC 信号传出，经由hazard转发给流水线各阶段。以便在cache更新以及写回数据的时候

阻塞流水线的全部五个阶段。而数据内存以及指令内存分别位于 Memory、Fetch 两阶段，当 Memory、Fetch两流水线被阻塞时，即cache接收到阻塞信号时，两个cache的状态是否更新以及如何输出stallC信号应当分情况讨论：

① Cache的下一状态为 read 时：

此时无论 Fetch 阶段或 Memory 阶段， cache 输出的 stallC 一定为0（无需与内存读写则不必阻塞流水线）；而如图*2.2.8*所示，cache的状态是否更新取决于接收到的 stall 信号；

② Cache的下一状态为 writebk 或 memory 时：

可知此时按照设想，stallC 信号应为1，阻塞个阶段流水线；

- **当只有一个高速缓存需要更新或写回：**

  此时两高速缓存都按照正常的逻辑执行。当进入下一周期，其中一个缓存进入更新/写回；而另一个缓存由于被阻塞将不会更新状态

- **当两个高速缓存都需要更新或写回：**

  经过仿真发现，当 Fetch 以及 Memory 阶段的指令数据高速缓存及数据高速缓存在同一个周期内都将转入更新或者写回状态时，它们会阻塞彼此，类似于产生了一个死锁。

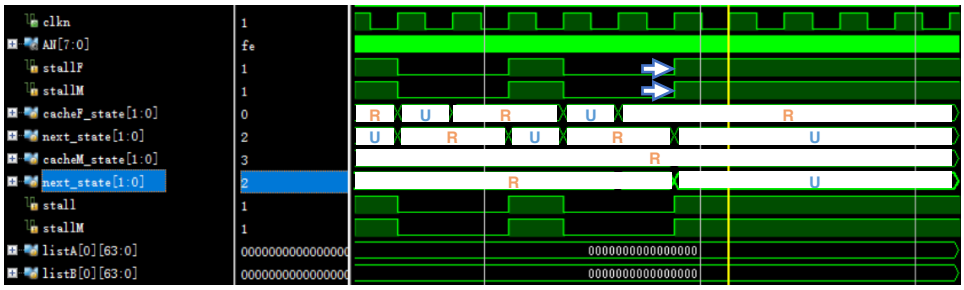


图 2.2.9 当两个cache同时需要更新时，出现“死锁”

但由于它们并不访问一个共同资源（两片内存是分开的），不存在竞争的情况。则在此时无需对 Fetch 以及 Memory 阻塞。我退而求其次，为了保证不会出现 timing loop 的情况，我设置了一个类似于“锁”的机制：当CacheM（Memory 阶段高速缓存）释放 stallC信号时，CacheF（Fetch 阶段高速缓存） 则默认不释放 stallC 信号：

```
assign stallc = ((next_state == writeback) | (next_state == updatew)) & ~stall;  (CacheF)
assign stallC = (next_state == writeback) | (next_state == updatew);  (CacheM)
```

由于 Memory 阶段的 stallC 信号已经保证除自己以外的四阶段流水线不更新，Fetch

阶段的 stallC 信号将不必做重复的工作阻塞Decode、Execute以及Writeback 阶段；而 CacheM 的阻塞信号仅仅依赖于 Fetch 阶段释放的 stallC信号，此时 CacheM将获得“锁”，访问内存资源；当 CacheM 不需要进入更新以及写回状态后，其 stall 信号变为0，CacheF 释放的 stallC 信号生效，下一周期中，Memory 以及其他三阶段寄存器被阻塞，等待 Fetch 阶段高速缓存结束访问内存资源后。整个流水线开始自由转动；

需要考虑的是，这里只可以对 CacheF stallC 信号加以约束。如果对换两者角色，将会出错。因为即便 Memory 阶段传出的 stallC 信号为0，Fetch阶段依旧可能因为 lw 或 branch 指令而进入阻塞，而此时全流水线都“不动”，将依旧僵持在死锁的状态中。

*在这里的设计中，我的stall信号机制略有些繁琐，但是在现有的条件下可以保证所有的流程都有很好的鲁棒性，不会出现问题。

2.2.4 cache 内部的实现策略

基于 cache 的组织结构以及 cache 的状态转换，我在 cache 内部采用组合逻辑以及时序逻辑设计结合的方式处理传入的信息。

cache 只在时钟上升沿到达时才会更新/写入数据；因此对cache的tag、valid位、dirty位、数据的使用频率优先级改动（s 位）的修改也只在时序逻辑中设计；

而读取数据完全依赖于组合逻辑。对于数据的 **Hit or Miss** 完全通过数据通路判断;

```
always @(*)
begin
  miss = 1'b0; dirty = 1'b0;
  wbadd = 6'b0; wbdata = 64'b0;
  case(add[2:1])  ①
  2'b00:
    begin
      if(update | ((add[6:3] == tag[0][3:0]) & v[0][0]) | ((add[6:3] == tag[0][7:4]) & v[0][1])) ②
        begin
          miss = 0;
          dirty = 0;
        end
      else
        begin  ③
          miss = 1'b1;
          if(~s[0][0]) begin dirty = d[0][0]; wbadd = {tag[0][3:0], 2'b00}; wbdata = listA[0]; end
          else         begin dirty = d[0][1]; wbadd = {tag[0][7:4], 2'b00}; wbdata = listB[0]; end
        end
    end
```

图 2.2.10

由于顶层模块中的状态机判断下一状态需要时时更新的Hit or Miss信息，而将这一判断放入时序逻辑，只有当下一周期到来时，dirty 和 miss 寄存器才会发生改变，将影响状态转换的正确性；因此，首要任务就是判断需要读取的地址是 hit 还是 miss ，以下将用文字顺序和代码对

应，模拟该逻辑的执行步骤：

① 根据输入地址判断数据应当存放在哪一组（add[2:1]）

② 在组内的两行中寻找是否有tag与地址（add[6:4]）一致的

⑴ 若update信号为1，代表已经进入更新，此时不列入 Miss

⑵ 若没有一致的：Miss

⑶ 若找到一致的，但是行中数据无效（invalid） Miss

⑷ 其余情况：Hit

③ 当地址 Miss 时：

⑴ 判断哪一行数据是最近使用较少的（s 位为0）

⑵ 将该行数据计入writeback寄存器，将地址也计入wbkadd寄存器

⑶ 将 dirty 寄存器更新为该行数据的 dirty 位

其次，需要考虑的是如何写入以及更新数据，这一部分将在时序逻辑中完成：

```
always @(posedge clk or posedge reset)
  begin
   if(reset)
     begin
       tag[0] <= 8'b0; tag[1] <= 8'b0; tag[2] <= 8'b0; tag[3] <= 8'b0;
       listA[0] <= 64'b0; listA[1] <= 64'b0; listA[2] <= 64'b0; listA[3] <= 64'b0;
       listB[0] <= 64'b0; listB[1] <= 64'b0; listB[2] <= 64'b0; listB[3] <= 64'b0;
       v[0] <= 2'b0; v[1] <= 2'b0; v[2] <= 2'b0; v[3] <= 2'b0;
       d[0] <= 2'b0; d[1] <= 2'b0; d[2] <= 2'b0; d[3] <= 2'b0;
       s[0] <= 2'b0; s[1] <= 2'b0; s[2] <= 2'b0; s[3] <= 2'b0;
     end
```

图 2.2.11

当传入reset信号时，cache清零，所有line中数据清楚，valid位置0，dirty位置0；

```
else if(update)
   case(add[2:1])
   2'b00:
     begin
       if(s[0][0]) begin tag[0][3:0] <= add[6:3];  v[0][0] <= 1; s[0][0] <= 1; s[0][1] <= 0;
                   if(WE) begin if(~add[0]) begin listA[0][31:0]  <= wdata; listA[0][63:32] <= updated[63:32]; d[0][0] <= 1; end
                                  else        begin listA[0][63:32] <= wdata; listA[0][31:0]  <= updated[31:0];  d[0][0] <= 1; end end
                   else   begin listA[0] <= updated; d[0][0] <= 0; end end
       else        begin tag[0][7:4] <= add[6:3];  v[0][1] <= 1; s[0][1] <= 1; s[0][0] <= 0;
                   if(WE) begin if(~add[0]) begin listB[0][31:0]  <= wdata; listB[0][63:32] <= updated[63:32]; d[0][1] <= 1; end
                                  else        begin listB[0][63:32] <= wdata; listB[0][31:0]  <= updated[31:0];  d[0][1] <= 1; end end
                   else   begin listB[0] <= updated; d[0][1] <= 0; end end
```

图 2.2.12

如果 update 信号为1，即表明 cache 的状态为 **updatew** 。那么 cache 中执行如下操作：

① 根据输入地址判断更新数据应当存放在哪一组（add[2:1]）

② 在组内的两行中寻找需要被替换的行（近期没有使用，s位为0）

③ 将tag位更新为输入地址信号所对应的字段（add[2:1]）

④ 将相应行更新为从内存中读取的数据，valid 位置1，dirty 位置0，将该行 s 位置1，表明近期使用过，另一行的 s 位置0；

```
else if (en)
  case(add[2:1])
  2'b00:
    begin
      if(add[6:3] == tag[0][3:0] & v[0][0])      begin s[0][0] <= 1; s[0][1] <= 0;  end
      else if(add[6:3] == tag[0][7:4] & v[0][1]) begin s[0][0] <= 0; s[0][1] <= 1;  end
    end
  2'b01:
    begin
      if(add[6:3] == tag[1][3:0] & v[1][0])      begin s[1][0] <= 1; s[1][1] <= 0; end
      else if(add[6:3] == tag[1][7:4] & v[1][1]) begin s[1][0] <= 0; s[1][1] <= 1; end
    end
  2'b10:
    begin
      if(add[6:3] == tag[2][3:0] & v[2][0])      begin s[2][0] <= 1; s[2][1] <= 0; end
      else if(add[6:3] == tag[2][7:4] & v[2][1]) begin s[2][0] <= 0; s[2][1] <= 1; end
    end
  2'b11:
    begin
      if(add[6:3] == tag[3][3:0] & v[3][0])      begin s[3][0] <= 1; s[3][1] <= 0; end
      else if(add[6:3] == tag[3][7:4] & v[3][1]) begin s[3][0] <= 0; s[3][1] <= 1; end
    end
  endcase
end
```

图 2.2.13

否则，如果en为1，即表明 cache 所在阶段正在执行 lw 或 sw 指令（Fetch 阶段 en 默认为1）；那么cache中执行如下操作：

① 根据输入地址判断读取或写入的数据目前存放在哪一组的哪一行

② 如果数据存在于cache中，则更改其所在行的最近读取符号位 s 为1；将另一行 s 位改为0；

最后，需要考虑输出读取的数据：

```
always @(*)
begin
  if (update)
    case(add[0])
    1'b0: data = updated[31:0];
    1'b1: data = updated[63:32];
    endcase
  else
    case(add[2:1])
    2'b00:
      begin
        if(add[6:3] == tag[0][3:0] & v[0][0]) begin data = add[0] == 0 ? listA[0][31:0] : listA[0][63:32]; end
        else                                  begin data = add[0] == 0 ? listB[0][31:0] : listB[0][63:32]; end
      end
```

图 2.2.14

在之前多周期的实验中（详见实验二实验报告），我设计的cache的状态转移规则为，在update后重新回到read状态，再依次执行之后的状态；而为了不浪费CPU，在流水线的处理器中，我并不打算等待更新的信息写入后，再回到read状态读取（这样会stall两个周期）。因此，我直接添加判断当前是否为更新状态的逻辑：

若为更新状态，则根据行内偏移量直接输出更新的数据；

若不为更新状态，则直接按照地址映射规则以及tag是否匹配的逻辑判断，读取存储在cache中的数据。

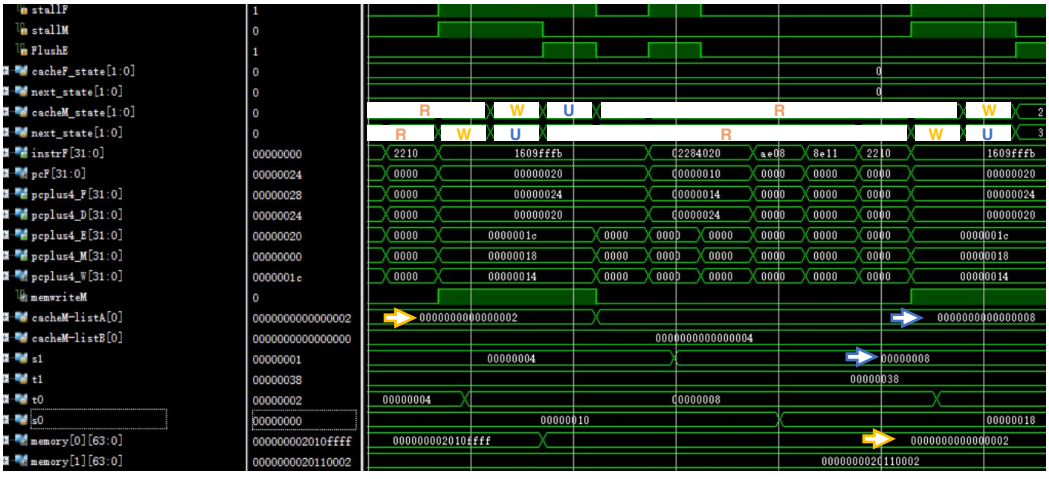
3 针对高速缓冲的仿真

仿真代码

```
0x0:  addi $t0, $0,  1          0x10: add  $t0, $t0, $s1
0x4:  addi $t1, $0,  56         0x14: sw   $t0, $s0, 0
0x8:  addi $s0, $0,  0          0x18: lw   $s1, $s0, 0
0xc:  addi $s1, $0,  1          0x1c: addi $s0, $s0, 8
0x10:  LOOP                     0x20: bne  $s0, $t1, LOOP
```

仿真代码不断计算2的幂次方，并将2^n存放在地址为8*（n-1）的内存处；

可见t0中数据成功写入Cache中



可见当Cache中两行都有数据时，近期未被使用的数据被替换，写回内存；

4 申A理由

- **对Cache的设计的结构合理，可以有效地避免CPU的资源浪费：**
- **对Cache的内部设计鲁棒性良好，可保证在多种代码情况下，不会出现问题：**

（详见前文的实验报告）

以方针代码的实验为例，如果不采用cache作为高速缓存，那么每次获取内存中的数据都将使用2周期（从内存中读取数据的时间长度）；而使用cache后，只需要两次读取同一地址数据只需要2周期（如果hit）或者3周期（miss）。如果讲cache中行数增加，这一数据差距可以更加明显直观。