**《计算机系统结构》实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **姓名** | | **黄昊、陈秦、蒋胜、**  **邓晨程** | | **年级** | | **2020** |
| **学号** | | **20204205、20204220、20204203、20203085** | | **专业、班级** | | **20计算机科学与技术（卓越）01、02** |
| **实验题目** | **实验2：Cache策略** | | | | | |
| **实验时间** | **DS3305** | | **实验地点** | | **第三实验楼** | |
| **实验成绩** |  | | **实验性质** | | **□验证性 🗹设计性 □综合性** | |
| 教师评价：  □算法/实验过程正确； □源程序/实验内容提交 □程序结构/实验步骤合理；  □实验结果正确； □语法、语义正确； □报告规范；  其他：  评价教师签名 | | | | | | |
| 一、实验目的  1. 加深对 Cache 原理的理解  2. 通过使用 verilog 实现 Cache，加深对状态机的理解 | | | | | | |
| 二、实验项目内容  1. 最低要求：参考指导书中直接映射写直达 Cache 的实现，实现写回策略的 Cache  2. 替换实验环境中的 Cache 模块，并通过仿真测试  3. [选做][较高要求] 性能优化，实现 2 路组相联的 Cache  4. [选做][更高要求] 性能优化，使用伪 LRU 等替换策略实现 4 路以上组相联的 Cache  5. [选做][最高要求] 实现其它 Cache 性能优化方法，如 axi burst 传输 | | | | | | |
| 三、实验过程或算法（源程序）  本次实验我们完成了前四个部分的要求，下面是原理阐释：  （一）写回和写分配  **写回**的策略是在每个Cacheline增加一个dirty位，标识该Cache块的内容在读出内存后是否被写过。如果该Cacheline需要换出，则先检查该Cacheline的dirty位是否为1，如果不为1，则直接覆盖掉；否则要先将Cacheline读入内存后再从内存中读出值进行覆盖。这是写回的原理。  **写分配**的策略是发生写缺失时，同时将数据写入cache和内存中。写不分配则是直接写入内存。  本次实验的最低要求就是写回+写分配。  考虑到CPU，cache和内存的主频各不相同，数据交换采用**握手机制**：主方发送数据请求和数据操作类型标识，从方得到数据地址后返回应答型号，找到数据后返回数据时也产生一个应答信号。这就是握手机制的步骤。  为了实现数据的正确传输，cache的信号需要通过一个**有限状态机**进行控制。本次实验我们设计的状态机如下所示：    *图1 有限状态机设计*  由于发生读缺失，且为脏位的时候情况最复杂：先将对应的块读入内存，再从内存中读出需要的块。因此写内存WM的状态被拆分成了两个状态：读时写内存RWM和写时写内存WWM。IDLE状态代表无请求状态，RM为读内存状态。转移条件如箭头所标。  （二）组相联    *图2 组相联示意图*  组相联的地址被编码，高位为tag，后面跟着index，最后为字节偏移。当发生寻址时，同时在多路Cache块中查找并同时比对tag，进而判断是否命中。  （三）伪LRU  伪LRU类似于哈夫曼树，每个路数可以视为二进制编码。如四路组相联，编码分别为00，01，10，11。对于哈夫曼树而言，从上而下搜索，如果编码为0则向左走，反之则向右走。  而PLRU与哈夫曼树的区别在于每个节点需要维护一个值，记录了最近的查找信息。如果命中，从根节点到访问路数的编码对应的节点之间路径上的节点，根据路径改成相应的值；如访问第三路时，编号为0的节点改为1，编号为2的节点改为0，编号为1的节点不变。  如果发生缺失时，被选择替换的块从根节点开始，根据节点记录的值的相反方向走。考虑这样一颗树：节点0为1，节点1为0为节点0的左儿子，节点2为1，为节点0的右儿子。此时发生缺失，查看节点0，记录为1，则按反方向查找到节点1，记录为0，于是查找到第二路并将对应的块替换掉。  （四）源代码  本次实验修改的为i-cache和d-cache。代码如下所示：  i-cache.v  module i\_cache (      input wire clk, rst,  *//mips core*      input         cpu\_inst\_req     ,      input         cpu\_inst\_wr      ,      input  [1 :0] cpu\_inst\_size    ,      input  [31:0] cpu\_inst\_addr    ,      input  [31:0] cpu\_inst\_wdata   ,      output [31:0] cpu\_inst\_rdata   ,      output        cpu\_inst\_addr\_ok ,      output        cpu\_inst\_data\_ok ,  *//axi interface*      output         cache\_inst\_req     ,      output         cache\_inst\_wr      ,      output  [1 :0] cache\_inst\_size    ,      output  [31:0] cache\_inst\_addr    ,      output  [31:0] cache\_inst\_wdata   ,      input   [31:0] cache\_inst\_rdata   ,      input          cache\_inst\_addr\_ok ,         *// axi接收到地址*      input          cache\_inst\_data\_ok           *// 返回了data*  );  *//Cache配置*      parameter  INDEX\_WIDTH  = 10, OFFSET\_WIDTH = 2, WAY\_WIDTH = 2;      localparam TAG\_WIDTH    = 32 - INDEX\_WIDTH - OFFSET\_WIDTH;      localparam CACHE\_DEEPTH = 1 << INDEX\_WIDTH;      localparam WAY\_NUM = 1 << WAY\_WIDTH;    *// 由于语法问题不得不前置的变量*      reg [WAY\_WIDTH - 1 : 0] victim\_index;      reg [WAY\_WIDTH - 1 : 0] choose\_way\_save;  *//Cache存储单元*      reg                 cache\_valid[WAY\_NUM - 1 : 0][CACHE\_DEEPTH - 1 : 0];      reg                 cache\_dirty[WAY\_NUM - 1 : 0][CACHE\_DEEPTH - 1 : 0];      reg [TAG\_WIDTH-1:0] cache\_tag  [WAY\_NUM - 1 : 0][CACHE\_DEEPTH - 1 : 0];      reg [31:0]          cache\_block[WAY\_NUM - 1 : 0][CACHE\_DEEPTH - 1 : 0];      reg [WAY\_NUM - 2:0] tree\_table                  [CACHE\_DEEPTH - 1 : 0];  *//访问地址分解*      wire [OFFSET\_WIDTH-1:0] offset;      wire [INDEX\_WIDTH-1:0] index;      wire [TAG\_WIDTH-1:0] tag;        assign offset = cpu\_inst\_addr[OFFSET\_WIDTH - 1 : 0];      assign index = cpu\_inst\_addr[INDEX\_WIDTH + OFFSET\_WIDTH - 1 : OFFSET\_WIDTH];      assign tag = cpu\_inst\_addr[31 : INDEX\_WIDTH + OFFSET\_WIDTH];  *//访问Cache line*      reg [WAY\_NUM - 1 : 0] c\_valid;      reg [WAY\_NUM - 1 : 0] c\_dirty;      reg [TAG\_WIDTH-1:0] c\_tag [WAY\_NUM - 1 : 0];      reg [31:0] c\_block [WAY\_NUM - 1 : 0];      wire [WAY\_NUM - 2 : 0] c\_tree;      integer c\_way\_index;  *// 组合逻辑描述各路取值状况*      assign c\_tree = tree\_table[index];      always @(\*) begin          if (rst) begin              for (c\_way\_index = 0; c\_way\_index < WAY\_NUM ; c\_way\_index = c\_way\_index + 1) begin                  c\_valid[c\_way\_index]  = 1'b0;                  c\_dirty[c\_way\_index]  = 1'b0;                  c\_tag[c\_way\_index]    = 0;                  c\_block[c\_way\_index]  = 0;              end          end          else begin              for (c\_way\_index = 0; c\_way\_index < WAY\_NUM ; c\_way\_index = c\_way\_index + 1) begin                  c\_valid[c\_way\_index]  = cache\_valid[c\_way\_index][index];                  c\_dirty[c\_way\_index]  = cache\_dirty[c\_way\_index][index];                  c\_tag  [c\_way\_index]  = cache\_tag  [c\_way\_index][index];                  c\_block[c\_way\_index]  = cache\_block[c\_way\_index][index];              end          end      end  *//判断是否命中*      reg [WAY\_NUM - 1 : 0] w\_hit, w\_miss;    *// 各路的命中情况*      reg [WAY\_WIDTH - 1 : 0] hit\_index;       *// 命中所在的路下标*      wire [WAY\_WIDTH - 1 : 0] choose\_way;      wire hit, miss;                         *// 所有路是否有命中的，全未命中则miss为1*      wire dirty, valid;      wire [31:0] wc\_block;      integer signal\_way\_index;      always @(\*) begin          if (rst) begin              w\_hit       = 0;              w\_miss      = 0;              hit\_index   = 0;          end          else begin              for (signal\_way\_index = 0; signal\_way\_index < WAY\_NUM ; signal\_way\_index = signal\_way\_index + 1) begin                  w\_hit[signal\_way\_index]     = c\_valid[signal\_way\_index] & (c\_tag[signal\_way\_index] == tag);                  w\_miss[signal\_way\_index]    = ~w\_hit[signal\_way\_index];                  hit\_index                   = w\_hit[signal\_way\_index] == 1'b1 ? signal\_way\_index : hit\_index;       *// 命中的路数*              end          end      end      assign hit  = |w\_hit;        *//存在命中的则为命中信号，注意这里有个位缩减运算符*      assign miss = ~hit;      assign dirty = hit ? c\_dirty[hit\_index] : c\_dirty[victim\_index];      assign valid = hit ? c\_valid[hit\_index] : c\_valid[victim\_index];      assign wc\_block = c\_block[choose\_way];      assign choose\_way = hit ? hit\_index : victim\_index;  *//读或写*      wire    read, write;      assign  write   = cpu\_inst\_wr;      assign  read    = ~write;  *// 鉴别状态*      reg in\_RM;  *// FSM有限状态机*      parameter IDLE = 2'b00, RM = 2'b01;      reg [1:0] state;      always @(posedge clk) begin          if(rst) begin              state <= IDLE;              in\_RM <= 1'b0;          end          else begin              case(state)                  IDLE:                  begin                      state <= cpu\_inst\_req & (hit | write & ~hit & ~dirty)   ?   IDLE  :                               cpu\_inst\_req & read & ~hit & ~dirty        ?   RM    :                      in\_RM <= 1'b0;                  end                  RM:                  begin                      state <= cache\_inst\_data\_ok  ? IDLE : RM;                      in\_RM <= 1'b1;                  end              endcase          end      end  *//读内存，为端口信号做准备*  *//变量read\_req, addr\_rcv, read\_finish用于构造类sram信号。*      wire read\_req;      *//一次完整的读事务，从发出读请求到结束*      reg addr\_rcv;       *//地址接收成功(addr\_ok)后到结束, 1表示握手成功（和MEMORY）*      wire read\_finish;   *//数据接收成功(data\_ok)，即读请求结束，1表示*      always @(posedge clk) begin          addr\_rcv <= rst ? 1'b0 :                      read & cache\_inst\_req & cache\_inst\_addr\_ok ? 1'b1 :                      read\_finish ? 1'b0 : addr\_rcv;      end      assign read\_req = state==RM;      assign read\_finish = read & cache\_inst\_data\_ok;     *//1表示从Memory读完了*  *//写内存，为端口信号做准备*      wire write\_req;      reg waddr\_rcv;      wire write\_finish;      always @(posedge clk) begin          waddr\_rcv <= rst ? 1'b0 :                       write & cache\_inst\_req & cache\_inst\_addr\_ok ? 1'b1 :                       write\_finish ? 1'b0 : waddr\_rcv;      end      assign write\_req = 1'b0;      assign write\_finish = write & cache\_inst\_data\_ok;   *//1表示已经向Memory写完了*  *//output to mips core*      assign cpu\_inst\_rdata   = hit ? c\_block[choose\_way] : cache\_inst\_rdata;          *// hit命中就算读命中，否则读缺失，读取cache的data上来*      assign cpu\_inst\_addr\_ok = cpu\_inst\_req & hit | cache\_inst\_req & cache\_inst\_addr\_ok ;      assign cpu\_inst\_data\_ok = cpu\_inst\_req & hit | cache\_inst\_data\_ok ;  *//output to axi interface*      assign cache\_inst\_req   =   read\_req & ~addr\_rcv | write\_req & ~waddr\_rcv;      assign cache\_inst\_wr    =   write\_req;      assign cache\_inst\_size  =   cpu\_inst\_size;      assign cache\_inst\_addr  =   cache\_inst\_wr ? {c\_tag[choose\_way\_save], index, offset}:                                                  cpu\_inst\_addr;      assign cache\_inst\_wdata =   wc\_block;           *// 写回是从cache读出去，到内存中*  *//写入Cache*  *//保存地址中的tag, index，防止addr发生改变*      reg [TAG\_WIDTH-1:0] tag\_save;      reg [INDEX\_WIDTH-1:0] index\_save;      always @(posedge clk) begin          tag\_save   <= rst ? 0 :                        cpu\_inst\_req ? tag : tag\_save;          index\_save <= rst ? 0 :                        cpu\_inst\_req ? index : index\_save;      end      wire [31:0] write\_cache\_data;      wire [3:0] write\_mask;  *//根据地址低两位和size，生成写掩码（针对sb，sh等不是写完整一个字的指令），4位对应1个字（4字节）中每个字的写使能*  *// write-mask 1位对应对应的字节*      assign write\_mask = cpu\_inst\_size==2'b00 ?                              (cpu\_inst\_addr[1] ? (cpu\_inst\_addr[0] ? 4'b1000 : 4'b0100):                                                  (cpu\_inst\_addr[0] ? 4'b0010 : 4'b0001)) :                              (cpu\_inst\_size==2'b01 ? (cpu\_inst\_addr[1] ? 4'b1100 : 4'b0011) : 4'b1111);  *//掩码的使用：位为1的代表需要更新的。*  *//位拓展：{8{1'b1}} -> 8'b11111111*  *//new\_data = old\_data & ~mask | write\_data & mask*      assign write\_cache\_data = wc\_block & ~{{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}} |       *//这个写进Cache的原因预测与57条指令有关，先别动*                                cpu\_inst\_wdata & {{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}};            *// 从cpu来的data*  *// 读，写缺失，需要选择一个Victim Block*      reg  [31:0] write\_cache\_inst\_save;      always @(\*) begin          if (rst) begin              victim\_index = 0;          end          else if (miss & cpu\_inst\_req) begin              victim\_index = c\_tree == 3'b000 ? 2'b11 :                             c\_tree == 3'b001 ? 2'b10 :                             c\_tree == 3'b010 ? 2'b11 :                             c\_tree == 3'b011 ? 2'b10 :                             c\_tree == 3'b100 ? 2'b01 :                             c\_tree == 3'b101 ? 2'b01 :                             c\_tree == 3'b110 ? 2'b00 :                                                2'b00 ;          end          else begin              victim\_index = victim\_index;          end      end  *// 保存victim\_index及写入的数据*      always @(posedge clk) begin          write\_cache\_inst\_save <= rst ? 0 : cpu\_inst\_req ? write\_cache\_data : write\_cache\_inst\_save;          choose\_way\_save <= rst? 0 : cpu\_inst\_req ? choose\_way : choose\_way\_save;      end      integer t, s;      always @(posedge clk) begin     *// 修改cache\_valid的具体位置*          if(rst) begin              for(t = 0; t < CACHE\_DEEPTH; t = t + 1) begin   *//刚开始将Cache置为无效，以及dirty位置为0*                  for (s = 0; s < WAY\_NUM; s = s + 1) begin                      cache\_valid[s][t] <= 0;                      cache\_dirty[s][t] <= 0;                      cache\_block[s][t] <= 0;                      cache\_tag  [s][t] <= 0;                  end                  tree\_table [t]    <= 0;              end          end          else begin              if(read\_finish & in\_RM) begin *//读缺失，访存结束时，read隐含之意就是缺失了,read\_finish就是对于cache,Memory的数据读出来了*  *// 特别要注意！要从RM回来才写，不然可能发生意想不到的错误*                  cache\_valid[choose\_way][index] <= 1'b1;             *//将Cache line置为有效*                  cache\_dirty[choose\_way][index] <= 1'b0;                  cache\_tag  [choose\_way][index] <= tag\_save;                  cache\_block[choose\_way][index] <= cache\_inst\_rdata; *//写入Cache line*              end          end          if (cpu\_inst\_req & hit) begin   *// 没有Hit的话就别更新了，找到victim\_block替换后，hit了再更*              tree\_table[index] <= choose\_way == 2'b00 ?                                      tree\_table[index][2] == 1'b0 ? 3'b000 : 3'b001 :                                   choose\_way == 2'b01 ?                                      tree\_table[index][2] == 1'b0 ? 3'b010 : 3'b011 :                                   choose\_way == 2'b10 ?                                      tree\_table[index][1] == 1'b0 ? 3'b100 : 3'b110 :                                      tree\_table[index][1] == 1'b0 ? 3'b101 : 3'b111 ;          end      end  endmodule  d-cache.v  module d\_cache (      input wire clk, rst,  *//mips core*      input         cpu\_data\_req     ,      input         cpu\_data\_wr      ,      input  [1 :0] cpu\_data\_size    ,      input  [31:0] cpu\_data\_addr    ,      input  [31:0] cpu\_data\_wdata   ,      output [31:0] cpu\_data\_rdata   ,      output        cpu\_data\_addr\_ok ,      output        cpu\_data\_data\_ok ,  *//axi interface*      output         cache\_data\_req     ,      output         cache\_data\_wr      ,      output  [1 :0] cache\_data\_size    ,      output  [31:0] cache\_data\_addr    ,      output  [31:0] cache\_data\_wdata   ,      input   [31:0] cache\_data\_rdata   ,      input          cache\_data\_addr\_ok ,      input          cache\_data\_data\_ok  );  *//Cache配置*      parameter  INDEX\_WIDTH  = 10, OFFSET\_WIDTH = 2, WAY\_WIDTH = 2;      localparam TAG\_WIDTH    = 32 - INDEX\_WIDTH - OFFSET\_WIDTH;      localparam CACHE\_DEEPTH = 1 << INDEX\_WIDTH;      localparam WAY\_NUM = 1 << WAY\_WIDTH;    *// 由于语法问题不得不前置的变量*      reg [WAY\_WIDTH - 1 : 0] victim\_index;  *//Cache存储单元*      reg                 cache\_valid[WAY\_NUM - 1 : 0][CACHE\_DEEPTH - 1 : 0];      reg                 cache\_dirty[WAY\_NUM - 1 : 0][CACHE\_DEEPTH - 1 : 0];      reg [TAG\_WIDTH-1:0] cache\_tag  [WAY\_NUM - 1 : 0][CACHE\_DEEPTH - 1 : 0];      reg [31:0]          cache\_block[WAY\_NUM - 1 : 0][CACHE\_DEEPTH - 1 : 0];      reg [WAY\_NUM - 2:0] tree\_table                  [CACHE\_DEEPTH - 1 : 0];  *//访问地址分解*      wire [OFFSET\_WIDTH-1:0] offset;      wire [INDEX\_WIDTH-1:0] index;      wire [TAG\_WIDTH-1:0] tag;        assign offset = cpu\_data\_addr[OFFSET\_WIDTH - 1 : 0];      assign index = cpu\_data\_addr[INDEX\_WIDTH + OFFSET\_WIDTH - 1 : OFFSET\_WIDTH];      assign tag = cpu\_data\_addr[31 : INDEX\_WIDTH + OFFSET\_WIDTH];  *//访问Cache line*      reg [WAY\_NUM - 1 : 0] c\_valid;      reg [WAY\_NUM - 1 : 0] c\_dirty;      reg [TAG\_WIDTH-1:0] c\_tag [WAY\_NUM - 1 : 0];      reg [31:0] c\_block [WAY\_NUM - 1 : 0];      wire [WAY\_NUM - 2 : 0] c\_tree;      integer c\_way\_index;  *// 组合逻辑描述各路取值状况*      assign c\_tree = tree\_table[index];      always @(\*) begin          if (rst) begin              for (c\_way\_index = 0; c\_way\_index < WAY\_NUM ; c\_way\_index = c\_way\_index + 1) begin                  c\_valid[c\_way\_index]  = 1'b0;                  c\_dirty[c\_way\_index]  = 1'b0;                  c\_tag[c\_way\_index]    = 0;                  c\_block[c\_way\_index]  = 0;              end          end          else begin              for (c\_way\_index = 0; c\_way\_index < WAY\_NUM ; c\_way\_index = c\_way\_index + 1) begin                  c\_valid[c\_way\_index]  = cache\_valid[c\_way\_index][index];                  c\_dirty[c\_way\_index]  = cache\_dirty[c\_way\_index][index];                  c\_tag  [c\_way\_index]  = cache\_tag  [c\_way\_index][index];                  c\_block[c\_way\_index]  = cache\_block[c\_way\_index][index];              end          end      end  *//判断是否命中*      reg [WAY\_NUM - 1 : 0] w\_hit, w\_miss;    *// 各路的命中情况*      reg [WAY\_WIDTH - 1 : 0] hit\_index;       *// 命中所在的路下标*      wire [WAY\_WIDTH - 1 : 0] choose\_way;      wire hit, miss;                         *// 所有路是否有命中的，全未命中则miss为1*      wire dirty, valid;      wire [31:0] wc\_block;      integer signal\_way\_index;      always @(\*) begin          if (rst) begin              w\_hit       = 0;              w\_miss      = 0;              hit\_index   = 0;          end          else begin              for (signal\_way\_index = 0; signal\_way\_index < WAY\_NUM ; signal\_way\_index = signal\_way\_index + 1) begin                  w\_hit[signal\_way\_index]     = c\_valid[signal\_way\_index] & (c\_tag[signal\_way\_index] == tag);                  w\_miss[signal\_way\_index]    = ~w\_hit[signal\_way\_index];                  hit\_index                   = w\_hit[signal\_way\_index] == 1'b1 ? signal\_way\_index : hit\_index;       *// 命中的路数*              end          end      end      assign hit  = |w\_hit;        *//存在命中的则为命中信号，注意这里有个位缩减运算符*      assign miss = ~hit;      assign dirty = hit ? c\_dirty[hit\_index] : c\_dirty[victim\_index];      assign valid = hit ? c\_valid[hit\_index] : c\_valid[victim\_index];      assign wc\_block = c\_block[choose\_way];      assign choose\_way = hit ? hit\_index : victim\_index;  *//读或写*      wire    read, write;      assign  write   = cpu\_data\_wr;      assign  read    = ~write;  *// 鉴别状态*      reg in\_RM, in\_RWM;  *// FSM有限状态机*      parameter IDLE = 2'b00, RM = 2'b01, RWM = 2'b11, WWM = 2'b10;      reg [1:0] state;      always @(posedge clk) begin          if(rst) begin              state <= IDLE;              in\_RM <= 1'b0;              in\_RWM <= 1'b0;          end          else begin              case(state)                  IDLE:                  begin                      state <= cpu\_data\_req & (hit | write & ~hit & ~dirty)   ?   IDLE  :                               cpu\_data\_req & read & ~hit & ~dirty        ?   RM    :                               cpu\_data\_req & read & ~hit & dirty         ?   RWM   :                               cpu\_data\_req & write & ~hit & dirty        ?   WWM   : IDLE;                      in\_RM <= 1'b0;                      in\_RWM <= 1'b0;                  end                  RM:                  begin                      state <= cache\_data\_data\_ok  ? IDLE : RM;                      in\_RM <= 1'b1;                      in\_RWM <= 1'b0;                  end                  RWM:                  begin                      state <= read   & cache\_data\_data\_ok ? RM    : RWM;                      in\_RM <= 1'b0;                      in\_RWM <= 1'b1;                  end                  WWM:                  begin                      state <= write  & cache\_data\_data\_ok ? IDLE  : WWM;                      in\_RM <= 1'b0;                      in\_RWM <= 1'b0;                  end              endcase          end      end  *//读内存，为端口信号做准备*  *//变量read\_req, addr\_rcv, read\_finish用于构造类sram信号。*      wire read\_req;      *//一次完整的读事务，从发出读请求到结束*      reg addr\_rcv;       *//地址接收成功(addr\_ok)后到结束, 1表示握手成功（和MEMORY）*      wire read\_finish;   *//数据接收成功(data\_ok)，即读请求结束，1表示*      always @(posedge clk) begin          addr\_rcv <= rst ? 1'b0 :                      read & cache\_data\_req & cache\_data\_addr\_ok ? 1'b1 :                      read\_finish ? 1'b0 : addr\_rcv;      end      assign read\_req = state==RM;      assign read\_finish = read & cache\_data\_data\_ok;     *//1表示从Memory读完了*  *//写内存，为端口信号做准备*      wire write\_req;      reg waddr\_rcv;      wire write\_finish;      always @(posedge clk) begin          waddr\_rcv <= rst ? 1'b0 :                       write & cache\_data\_req & cache\_data\_addr\_ok ? 1'b1 :                       write\_finish ? 1'b0 : waddr\_rcv;      end      assign write\_req = (state == RWM) | (state == WWM);      assign write\_finish = write & cache\_data\_data\_ok;   *//1表示已经向Memory写完了*  *//output to mips core*      assign cpu\_data\_rdata   = hit ? c\_block[choose\_way] : cache\_data\_rdata;          *// hit命中就算读命中，否则读缺失，读取cache的data上来*      assign cpu\_data\_addr\_ok = cpu\_data\_req & hit | cache\_data\_req & cache\_data\_addr\_ok & ~in\_RWM | write & cpu\_data\_req & ~hit & ~dirty;      assign cpu\_data\_data\_ok = cpu\_data\_req & hit | cache\_data\_data\_ok & ~in\_RWM | write & cpu\_data\_req & ~hit & ~dirty;  *//output to axi interface*      assign cache\_data\_req   =   read\_req & ~addr\_rcv | write\_req & ~waddr\_rcv;      assign cache\_data\_wr    =   write\_req;      assign cache\_data\_size  =   cpu\_data\_size;      assign cache\_data\_addr  =   cache\_data\_wr ? {c\_tag[choose\_way], index, offset}:                                                  cpu\_data\_addr;      assign cache\_data\_wdata =   wc\_block;           *// 写回是从cache读出去，到内存中*  *//写入Cache*  *//保存地址中的tag, index，防止addr发生改变*      reg [TAG\_WIDTH-1:0] tag\_save;      reg [INDEX\_WIDTH-1:0] index\_save;      always @(posedge clk) begin          tag\_save   <= rst ? 0 :                        cpu\_data\_req ? tag : tag\_save;          index\_save <= rst ? 0 :                        cpu\_data\_req ? index : index\_save;      end      wire [31:0] write\_cache\_data;      wire [3:0] write\_mask;  *//根据地址低两位和size，生成写掩码（针对sb，sh等不是写完整一个字的指令），4位对应1个字（4字节）中每个字的写使能*  *// write-mask 1位对应对应的字节*      assign write\_mask = cpu\_data\_size==2'b00 ?                              (cpu\_data\_addr[1] ? (cpu\_data\_addr[0] ? 4'b1000 : 4'b0100):                                                  (cpu\_data\_addr[0] ? 4'b0010 : 4'b0001)) :                              (cpu\_data\_size==2'b01 ? (cpu\_data\_addr[1] ? 4'b1100 : 4'b0011) : 4'b1111);  *//掩码的使用：位为1的代表需要更新的。*  *//位拓展：{8{1'b1}} -> 8'b11111111*  *//new\_data = old\_data & ~mask | write\_data & mask*      assign write\_cache\_data = wc\_block & ~{{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}} |       *//这个写进Cache的原因预测与57条指令有关，先别动*                                cpu\_data\_wdata & {{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}};            *// 从cpu来的data*  *// 读，写缺失，需要选择一个Victim Block*      always @(\*) begin          if (rst) begin              victim\_index = 0;          end          else if (miss & cpu\_data\_req) begin              victim\_index = c\_tree == 3'b000 ? 2'b11 :                             c\_tree == 3'b001 ? 2'b10 :                             c\_tree == 3'b010 ? 2'b11 :                             c\_tree == 3'b011 ? 2'b10 :                             c\_tree == 3'b100 ? 2'b01 :                             c\_tree == 3'b101 ? 2'b01 :                             c\_tree == 3'b110 ? 2'b00 :                                                2'b00 ;          end          else begin              victim\_index = victim\_index;          end      end      integer t, s;      always @(posedge clk) begin     *// 修改cache\_valid的具体位置*          if(rst) begin              for(t = 0; t < CACHE\_DEEPTH; t = t + 1) begin   *//刚开始将Cache置为无效，以及dirty位置为0*                  for (s = 0; s < WAY\_NUM; s = s + 1) begin                      cache\_valid[s][t] <= 0;                      cache\_dirty[s][t] <= 0;                      cache\_block[s][t] <= 0;                      cache\_tag  [s][t] <= 0;                  end                  tree\_table [t]    <= 0;              end          end          else begin              if(read\_finish & in\_RM) begin *//读缺失，访存结束时，read隐含之意就是缺失了,read\_finish就是对于cache,Memory的数据读出来了*  *// 特别要注意！要从RM回来才写，不然可能发生意想不到的错误*                  cache\_valid[choose\_way][index] <= 1'b1;             *//将Cache line置为有效*                  cache\_dirty[choose\_way][index] <= 1'b0;                  cache\_tag  [choose\_way][index] <= tag\_save;                  cache\_block[choose\_way][index] <= cache\_data\_rdata; *//写入Cache line*              end              else if(write & cpu\_data\_req & hit) begin   *//写命中，dirty置为1即可，更新下来写来的数据*                  cache\_block[choose\_way][index] <= write\_cache\_data;      *//写入Cache line，使用index而不是index\_save*                  cache\_dirty[choose\_way][index] <= 1'b1;              end              else if(write & cpu\_data\_req & ~hit & ~dirty) begin *// 写缺失，但是~dirty*                  cache\_block[choose\_way][index] <= write\_cache\_data;      *//写入Cache line，使用index而不是index\_save*                  cache\_dirty[choose\_way][index] <= 1'b1;                  cache\_tag  [choose\_way][index] <= tag;              end              else if (write\_finish) begin    *// 写缺失，且dirty的情况*                  cache\_valid[choose\_way][index] <= 1'b1;             *//将Cache line置为有效*                  cache\_dirty[choose\_way][index] <= 1'b0;                  cache\_tag  [choose\_way][index] <= tag\_save;                  cache\_block[choose\_way][index] <= write\_cache\_data; *//写入Cache line*              end          end          if (cpu\_data\_req & hit) begin   *// 没有Hit的话就别更新了，找到victim\_block替换后，hit了再更*              tree\_table[index] <= choose\_way == 2'b00 ?                                      tree\_table[index][2] == 1'b0 ? 3'b000 : 3'b001 :                                   choose\_way == 2'b01 ?                                      tree\_table[index][2] == 1'b0 ? 3'b010 : 3'b011 :                                   choose\_way == 2'b10 ?                                      tree\_table[index][1] == 1'b0 ? 3'b100 : 3'b110 :                                      tree\_table[index][1] == 1'b0 ? 3'b101 : 3'b111 ;          end      end  endmodule | | | | | | |
| 四、实验结果及分析和（或）源程序调试过程  （一）结果  本次实验的结果（写回+写分配+四路组相联+PLRU）如下所示：      下面是标程代码（写直达+写不分配）的结果：      前后结果的对比，可以发现确实有所优化。  （二）碰到的问题  本次实验碰到的最大问题是：由于状态机的设计，在RWM和RM中要访问两次内存，加上对信号控制的不明确，导致Cache提前向CPU发送data\_ok应答信号，导致错误。信号的控制十分重要。 | | | | | | |
| 五、小组分工情况说明   |  |  | | --- | --- | | 姓名 | 分工 | | 黄昊 | 负责写回和写分配的实现，撰写实验报告 | | 陈秦 | 负责写回和写分配的实现 | | 蒋胜 | 改写为四路组相联 | | 邓晨程 | 增加PLRU逻辑 | | | | | | | |