中国科学院大学计算机组成原理实验课

实 验 报 告

学号： 2020K8009926006 姓名： 游昆霖 专业： 计算机科学与技术

实验序号： 5.4 实验名称： 高速缓存（cache）设计

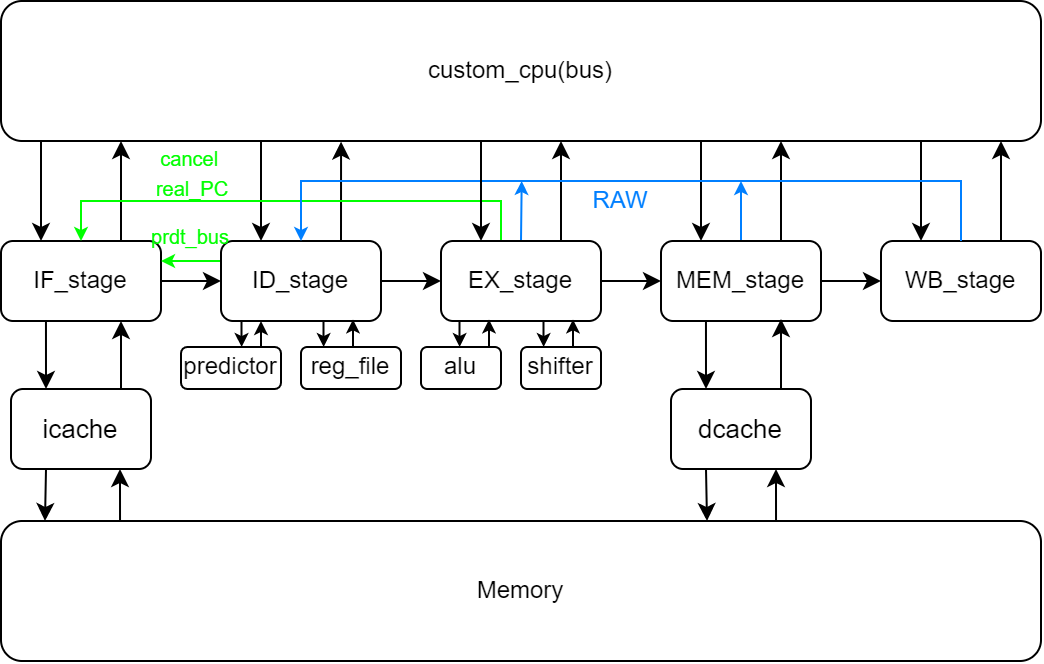
注1：撰写此Word格式实验报告后以PDF格式保存在~/COD-Lab/reports目录下。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为1至4的阿拉伯数字。例如：prj1.pdf。PDF文件大小应控制在5MB以内。此外，实验项目5包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：prj5-dma.pdf。具体要求详见实验项目5讲义。

注2：使用git add及git commit命令将实验报告PDF文件添加到本地仓库master分支，并通过git push推送到GitLab远程仓库master分支（具体命令详见实验报告）。

注3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

1. 逻辑电路结构与仿真波形的截图及说明（比如关键RTL代码段{包含注释}及其对应的逻辑电路结构图、相应信号的仿真波形和信号变化的说明等）

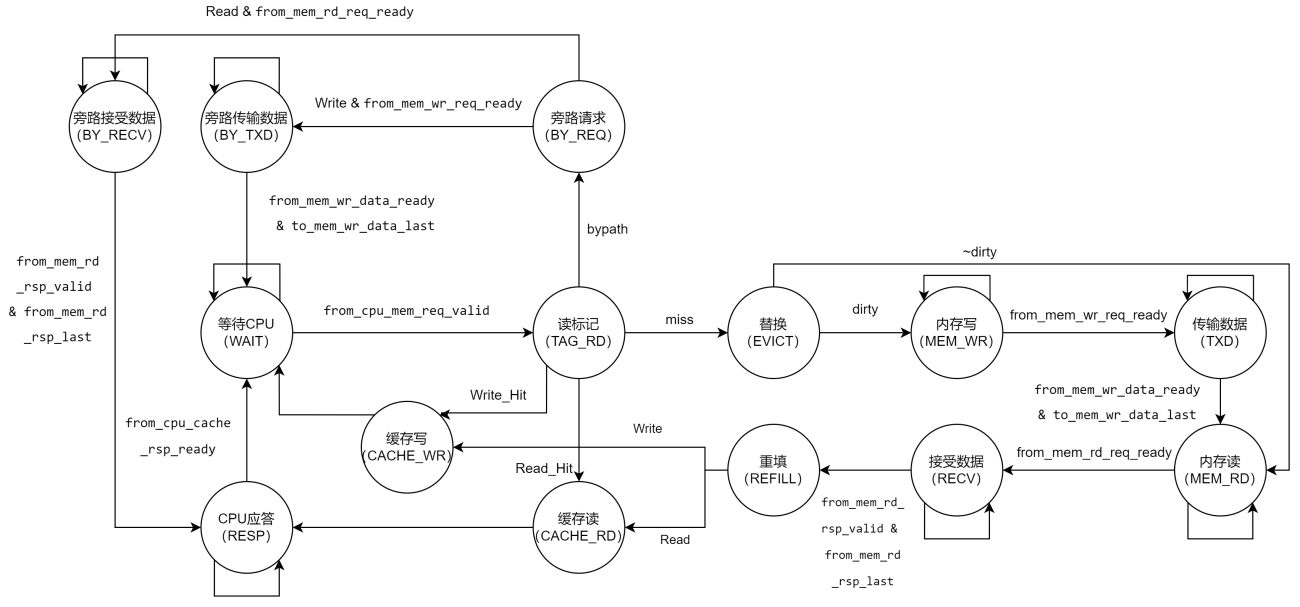
\*说明：由于icache和dcache大部分代码类似，且dcache还需要处理写缺失和写命中问题，复杂程度较高。因此将只介绍dcache代码，icache所有功能在dcache中均有实现，不再赘述。  
 整体数据流向示意图（基于turbo）：



1、dcache状态状态转移

|  |
| --- |
| localparam      WAIT    = 14'b00000000000001 ,                  TAG\_RD  = 14'b00000000000010 ,                  EVICT   = 14'b00000000000100 ,                  MEM\_WR  = 14'b00000000001000 ,                  TXD     = 14'b00000000010000 ,  //transmit data                  MEM\_RD  = 14'b00000000100000 ,                  RECV    = 14'b00000001000000 ,                  REFILL  = 14'b00000010000000 ,                  CACHE\_WR= 14'b00000100000000 ,                  CACHE\_RD= 14'b00001000000000 ,                  RESP    = 14'b00010000000000 ,                  BY\_REQ  = 14'b00100000000000 ,                  BY\_TXD  = 14'b01000000000000 ,                  BY\_RECV = 14'b10000000000000 ;  ...  always @(\*) begin          case (cur\_state)                  WAIT: begin                          if(from\_cpu\_mem\_req\_valid)                                  next\_state = TAG\_RD;                          else                                  next\_state = WAIT;                  end                    TAG\_RD: begin                          if(bypath)                                  next\_state = BY\_REQ;                          else if(cpu\_mem\_rw & Hit)  //write hit                                  next\_state = CACHE\_WR;                          else if(~cpu\_mem\_rw & Hit) //read hit                                  next\_state = CACHE\_RD;                          else                                  next\_state = EVICT;                  end                    EVICT: begin                          if(dirty)                                  next\_state = MEM\_WR;                          else                                  next\_state = MEM\_RD;                  end                    MEM\_WR: begin                          if(from\_mem\_wr\_req\_ready)                                  next\_state = TXD;                          else                                  next\_state = MEM\_WR;                  end                    TXD: begin                          if(from\_mem\_wr\_data\_ready & to\_mem\_wr\_data\_last)                                  next\_state = MEM\_RD;                          else                                  next\_state = TXD;                  end                    MEM\_RD: begin                          if(from\_mem\_rd\_req\_ready)                                  next\_state = RECV;                          else                                  next\_state = MEM\_RD;                  end                    RECV: begin                          if(from\_mem\_rd\_rsp\_valid & from\_mem\_rd\_rsp\_last)                                  next\_state = REFILL;                          else                                  next\_state = RECV;                  end                    REFILL: begin                          if(cpu\_mem\_rw) //write                                  next\_state = CACHE\_WR;                          else                                  next\_state = CACHE\_RD;  //save a cycle than CACHE\_RD                  end                    CACHE\_WR: begin                          next\_state = WAIT;                  end                    CACHE\_RD: begin                          next\_state = RESP;                  end                    RESP: begin                          if(from\_cpu\_cache\_rsp\_ready)                                  next\_state = WAIT;                          else                                  next\_state = RESP;                  end                    BY\_REQ: begin                          if(cpu\_mem\_rw & from\_mem\_wr\_req\_ready)  //write                                  next\_state = BY\_TXD;                          else if(~cpu\_mem\_rw & from\_mem\_rd\_req\_ready) //read                                  next\_state = BY\_RECV;                          else                                  next\_state = BY\_REQ;                  end                    BY\_TXD: begin                          if(from\_mem\_wr\_data\_ready & to\_mem\_wr\_data\_last)                                  next\_state = WAIT;                          else                                  next\_state = BY\_TXD;                  end                    BY\_RECV: begin                          if(from\_mem\_rd\_rsp\_valid & from\_mem\_rd\_rsp\_last)                                  next\_state = RESP;                          else                                  next\_state = BY\_RECV;                  end                    default:                          next\_state = WAIT;          endcase  end |

状态转移图：



Cache在接受到CPU的请求后，先进入TAG\_RD状态，此处将分辨是否旁路请求并进入不同路径。

对于非旁路请求，如果命中将根据读或写分为CACHE\_RD和CACHE\_WR状态，分别进行CACHE中数据的读和写。如果未命中，则需进入EVICT替换状态。此时需要进一步根据dirty信号判断该替换块是否已被写过，若是，则需先后经历MEM\_WR请求态和TXD传输数据态。然后，与替换块未写过的情形相同，先后进入MEM\_RD请求态和RECV接受态，接受来自内存的替换数据，并进入REFILL状态进行数据重填。然后和命中情形一致，再进入CACHE\_RD和CACHE\_WR状态进行读写。

对于IO旁路请求，将从TAG\_RD状态直接进入BY\_REQ请求态，然后根据读写类型分别进入BY\_RECV旁路接收状态和BY\_TXD旁路传输状态。

对于CACHE\_WR和BY\_TXD，当写缓存或是写内存完毕后可以直接回到初始态等待下一次使用。对于CACHE\_RD和BY\_RECV，当读缓存或者读内存完毕后，需要额外经过RESP响应态，等待CPU准备好接收数据。

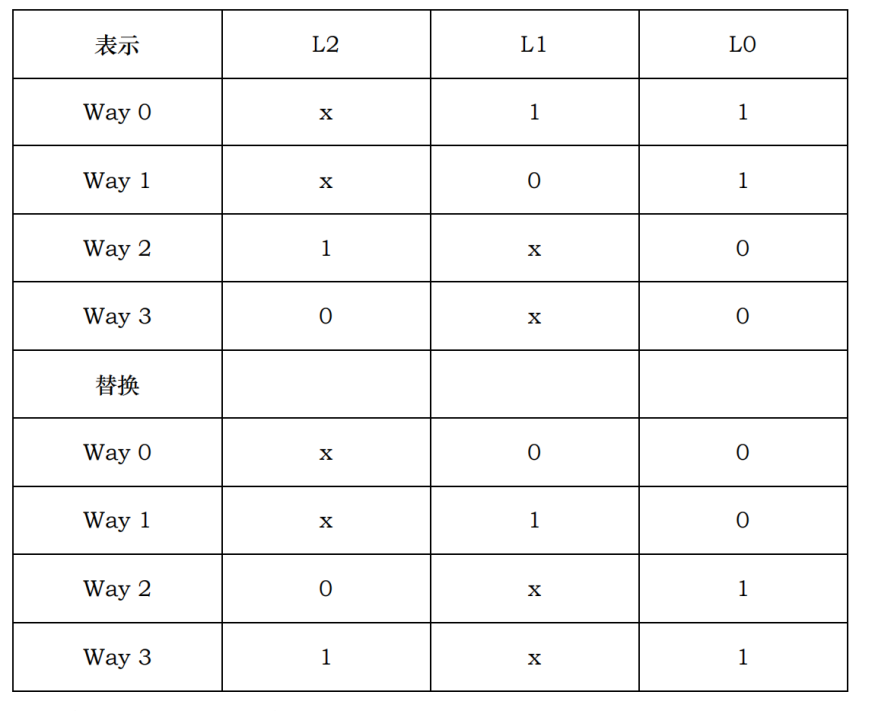
1. 地址分析与路径确定

|  |
| --- |
| //Analyse mem addr, compare and judge path          ...//相应数据初始化          assign tag     = cpu\_mem\_addr[31:32-`TAG\_LEN];          assign set      = cpu\_mem\_addr[4 + `CACHE\_SET\_WD:5];          assign offset   = cpu\_mem\_addr[4:0];          //0x00 ~ 0x1F OR above 0x40000000          assign bypath   = (~|cpu\_mem\_addr[31:5]) | (|cpu\_mem\_addr[31:30]);            assign hit0     = valid0[set] & Tag0 == tag;          assign hit1     = valid1[set] & Tag1 == tag;          assign hit2     = valid2[set] & Tag2 == tag;          assign hit3     = valid3[set] & Tag3 == tag;            assign Hit      = hit0 | hit1 | hit2 | hit3; |

根据不可缓存地址确定bypath旁路标志。当某组路的某块数据有效且tag相等时表示该路命中。为适配后续修改cache容量，针对可能的修改使用宏定义定义位宽。

1. 替换策略

|  |
| --- |
| //EVICT : use PLRU algorithm          //when some way are invalid, replace them first by order          //only when all 4 way are valid, use PLRU to choose          assign valid =  valid0[set] & valid1[set] & valid2[set] & valid3[set];          assign way0 =   (valid & ~PLRU[set][1] & ~PLRU[set][0]) |                          (~valid0[set]);          assign way1 =   (valid &  PLRU[set][1] & ~PLRU[set][0]) |                          (valid0[set] & ~valid1[set]);          assign way2 =   (valid & ~PLRU[set][2] &  PLRU[set][0]) |                          (valid0[set] & valid1[set] & ~valid2[set]);          assign way3 =   (valid &  PLRU[set][2] &  PLRU[set][0]) |                          (valid0[set] & valid1[set] & valid2[set] & ~valid3[set]);            //refresh PLRU          assign choose01 = choose0 | choose1;          always @(posedge clk) begin                  if(rst) begin                          PLRU[0] <= 3'b0; PLRU[1] <= 3'b0; PLRU[2] <= 3'b0; PLRU[3] <= 3'b0;                          PLRU[4] <= 3'b0; PLRU[5] <= 3'b0; PLRU[6] <= 3'b0; PLRU[7] <= 3'b0;                  end                  else if (cur\_state == CACHE\_RD | cur\_state == CACHE\_WR) begin //after read or write visit                          PLRU[set][0] <= choose01;                          if(choose01)                                  PLRU[set][1] <= choose0;                          else                                  PLRU[set][2] <= choose2;                  end          end |



当有某些组路数据无效时，先按顺序选择无效组路进行更新，如果4组路均有效时，根据PLRU算法进行更新，具体规则如上图所示，其中下半部分为替换条件，上半部分为替换或是读写命中时的更新表示。通过PLRU算法，可以使得替换倾向于选择较少被访问的组路，从而增加cache读写命中率，减少访存次数。

1. 读写路组选择

|  |
| --- |
| //choose logic :          //hit : get cache\_data from which way          //miss: get refresh which way          //note the init is all 0 because of the valid init          always @(posedge clk) begin                  if(cur\_state == TAG\_RD) // if hit, get cache data way                  begin                          Hit\_tmp <= Hit;                          choose0 <= hit0;                          choose1 <= hit1;                          choose2 <= hit2;                          choose3 <= hit3;                  end                  if(cur\_state == EVICT) //if miss, refresh which way, op before will be ignore                  begin                          choose0 <= way0;                          choose1 <= way1;                          choose2 <= way2;                          choose3 <= way3;                  end          end |

当命中时，根据命中路组进行选择；当缺失时，根据替换数据进行选择。

1. 各路组相关数组

|  |
| --- |
| //valid / tag / data / dirty array: initialization or refresh          //valid array          always @(posedge clk) begin                  if(rst) begin                          valid0[`CACHE\_SET-1:0] <= {`CACHE\_SET{1'b0}};                          valid1[`CACHE\_SET-1:0] <= {`CACHE\_SET{1'b0}};                          valid2[`CACHE\_SET-1:0] <= {`CACHE\_SET{1'b0}};                          valid3[`CACHE\_SET-1:0] <= {`CACHE\_SET{1'b0}};                  end                  if(cur\_state == REFILL) begin                          if(choose0)             valid0[set] <= 1'b1;                          else if(choose1)        valid1[set] <= 1'b1;                          else if(choose2)        valid2[set] <= 1'b1;                          else if(choose3)        valid3[set] <= 1'b1;                  end          end          //tag array : when rst, no need to refresh because valid = 0                  //wdata is tag          assign TagWen0 = cur\_state[7] & choose0; //REFILL          assign TagWen1 = cur\_state[7] & choose1;          assign TagWen2 = cur\_state[7] & choose2;          assign TagWen3 = cur\_state[7] & choose3;          //data array          //读写REFILL均需更新，因为CACHE写只会进行部分更新          assign DataWen0 = (cur\_state[7] | cur\_state[8]) & choose0; //REFILL or CACHE\_WR          assign DataWen1 = (cur\_state[7] | cur\_state[8]) & choose1;          assign DataWen2 = (cur\_state[7] | cur\_state[8]) & choose2;          assign DataWen3 = (cur\_state[7] | cur\_state[8]) & choose3;  assign Array\_Wdata =    {`LINE\_LEN{cur\_state[7]}} & mem\_block\_data | //REFILL                                  {`LINE\_LEN{cur\_state[8]}} & cache\_modified\_block ;          //dirty array          always @(posedge clk) begin                  if (rst) begin                          dirty0[`CACHE\_SET-1:0] <= {`CACHE\_SET{1'b0}};                          dirty1[`CACHE\_SET-1:0] <= {`CACHE\_SET{1'b0}};                          dirty2[`CACHE\_SET-1:0] <= {`CACHE\_SET{1'b0}};                          dirty3[`CACHE\_SET-1:0] <= {`CACHE\_SET{1'b0}};                  end                  else if (cur\_state == CACHE\_WR) begin                          if (choose0)            dirty0[set] <= 1'b1;                          else if (choose1)       dirty1[set] <= 1'b1;                          else if (choose2)       dirty2[set] <= 1'b1;                          else if (choose3)       dirty3[set] <= 1'b1;                  end                  else if (cur\_state == REFILL) begin //refill cache with mem data, reset dirty                          if (choose0)            dirty0[set] <= 1'b0;                          else if (choose1)       dirty1[set] <= 1'b0;                          else if (choose2)       dirty2[set] <= 1'b0;                          else if (choose3)       dirty3[set] <= 1'b0;                  end          end |

初始化时只需初始化valid和dirty数据即可让其余两个数组无效化。在REFILL重填阶段需要根据替换选择对4个数组都进行更新。CACHE\_WR在向CACHE写数据阶段需要根据选择更新对应的dirty和data数组，data数组需要将cache块拿出，根据offset和strb修改对应位置，在重新将修改后的块填入data数组。

6、CPU读数据相关

|  |
| --- |
| //READ : final data to cpu  -- source : cache(hit)/mem(miss)/bypath          //data from cache          assign cache\_block\_data =       ( {`LINE\_LEN{choose0}} & Data0 ) |                                          ( {`LINE\_LEN{choose1}} & Data1 ) |                                          ( {`LINE\_LEN{choose2}} & Data2 ) |                                          ( {`LINE\_LEN{choose3}} & Data3 ) ;          assign cache\_final\_data = cache\_block\_data[ {offset,3'b0} +: 32 ];            //data from mem / bypath          assign to\_mem\_rd\_req\_addr[31:5] = cpu\_mem\_addr[31:5];          assign to\_mem\_rd\_req\_addr[4:0]  = bypath ? cpu\_mem\_addr[4:0] : 5'b0;            assign to\_mem\_rd\_req\_len = {5'b0 , {3{~bypath}} }; //bypath 0 other 7          always @(posedge clk) begin                  if((cur\_state == RECV | cur\_state == BY\_RECV) & from\_mem\_rd\_rsp\_valid)                          mem\_block\_data <= {from\_mem\_rd\_rsp\_data , mem\_block\_data[255:32]};          end     //result : {data7,...,data0} or {data,x,..,x}            assign mem\_final\_data = mem\_block\_data[ {offset,3'b0} +: 32 ];          assign bypath\_read\_data = mem\_block\_data [255:224];            //choose source          assign to\_cpu\_cache\_rsp\_data =  ( {32{ bypath}} & bypath\_read\_data)     |                                          ( {32{~bypath}} & cache\_final\_data)     ; |

根据路组选择可以从data数组中取得数据块，然后根据offset可以确定cache所读的4byte数据。读写缺失时，需要根据替换算法从内存取得数据块。根据bypath旁路标志确定请求地址和长度，然后在对应状态通过mem\_block\_data寄存器移位接收内存传递的数据，此时由于旁路只传输一拍，对应的数据即最高32位。最后根据旁路标志选择CPU读响应数据的来源。

7、CPU写数据相关

|  |
| --- |
| //WRITE : final data from cpu --target : cache(hit or miss) / mem(bypath)                  //note write back to MEM is dirty block or bypath          //cpu write          assign cache\_modified\_final = {                                          { ({8{cpu\_mem\_wstrb[3]}} & cpu\_mem\_wdata[31:24]) | ({8{~cpu\_mem\_wstrb[3]}} & cache\_final\_data[31:24]) },                                          { ({8{cpu\_mem\_wstrb[2]}} & cpu\_mem\_wdata[23:16]) | ({8{~cpu\_mem\_wstrb[2]}} & cache\_final\_data[23:16]) },                                          { ({8{cpu\_mem\_wstrb[1]}} & cpu\_mem\_wdata[15: 8]) | ({8{~cpu\_mem\_wstrb[1]}} & cache\_final\_data[15: 8]) },                                          { ({8{cpu\_mem\_wstrb[0]}} & cpu\_mem\_wdata[ 7: 0]) | ({8{~cpu\_mem\_wstrb[0]}} & cache\_final\_data[ 7: 0]) }                                           };            //finally refresh to data array          assign cache\_modified\_block =          {`LINE\_LEN{offset[4:2]==3'b000 }} & {cache\_block\_data[`LINE\_LEN-1 : 32],cache\_modified\_final} |          {`LINE\_LEN{offset[4:2]==3'b001 }} & {cache\_block\_data[`LINE\_LEN-1 : 64],cache\_modified\_final,cache\_block\_data[31:0]} |          {`LINE\_LEN{offset[4:2]==3'b010 }} & {cache\_block\_data[`LINE\_LEN-1 : 96],cache\_modified\_final,cache\_block\_data[63:0]} |          {`LINE\_LEN{offset[4:2]==3'b011 }} & {cache\_block\_data[`LINE\_LEN-1 : 128],cache\_modified\_final,cache\_block\_data[95:0]} |          {`LINE\_LEN{offset[4:2]==3'b100 }} & {cache\_block\_data[`LINE\_LEN-1 : 160],cache\_modified\_final,cache\_block\_data[127:0]} |          {`LINE\_LEN{offset[4:2]==3'b101 }} & {cache\_block\_data[`LINE\_LEN-1 : 192],cache\_modified\_final,cache\_block\_data[159:0]} |          {`LINE\_LEN{offset[4:2]==3'b110 }} & {cache\_block\_data[`LINE\_LEN-1 : 224],cache\_modified\_final,cache\_block\_data[191:0]} |          {`LINE\_LEN{offset[4:2]==3'b111 }} & {cache\_modified\_final,cache\_block\_data[223:0]} ;          assign bypath\_write\_data = cpu\_mem\_wdata;          //write back to MEM          assign to\_mem\_wr\_req\_addr = ( {32{bypath}} & cpu\_mem\_addr )                     |                                      ( {32{~bypath & choose0}} & {Tag0,set,5'b0} )       |                                      ( {32{~bypath & choose1}} & {Tag1,set,5'b0} )       |                                      ( {32{~bypath & choose2}} & {Tag2,set,5'b0} )       |                                      ( {32{~bypath & choose3}} & {Tag3,set,5'b0} )       ;          assign to\_mem\_wr\_req\_len = {5'b0,{3{~bypath}}};            //use last shifter to get last signal          always @(posedge clk) begin                  if(cur\_state == MEM\_WR | cur\_state == BY\_REQ) begin                          if(bypath)                                  last\_shifter <= 8'b1;                          else                                  last\_shifter <= {1'b1,7'b0};                  end                  else if(cur\_state == TXD & from\_mem\_wr\_data\_ready) begin                          last\_shifter <= {1'b0,last\_shifter[7:1]};                  end          end            //use dirty block shifter to transmit it 4byte each time          always @(posedge clk) begin                  if(cur\_state == MEM\_WR) begin                          dirty\_block\_data <= cache\_block\_data;                  end                  else if(cur\_state == TXD & from\_mem\_wr\_data\_ready) begin                          dirty\_block\_data <= {32'b0 , dirty\_block\_data[255:32]};                  end          end            assign to\_mem\_wr\_data\_last = last\_shifter[0] & from\_mem\_wr\_data\_ready & to\_mem\_wr\_data\_valid;          assign to\_mem\_wr\_data = ( {32{ bypath}} & bypath\_write\_data)      |                                  ( {32{~bypath}} & dirty\_block\_data[31:0]) ;          //for dirty block, strb are always 4'b1111 to ensure complete data transmit          //for bypath , depends on input port          assign to\_mem\_wr\_data\_strb = {4{~bypath}} | cpu\_mem\_wstrb; |

在写命中或写缺失后数据重填完毕后，需要将cache中对应位置的32位数据取出并根据wstrb将相应位置修改为cpu写入的数据，然后在根据offset修改cache块的对应位置，将修改后的块重填至data数组。  
 旁路写内存或读写缺失时，需要根据旁路标志或替换算法的选择确定写内存的地址和长度。对于读写缺失的情形，写内存地址由被替换块的tag等相关信息确定。

向内存burst传输数据时，预先在MEM\_WR请求阶段即将cache上的数据块搬移到一个寄存器中，通过每次传输低32位并右移，以从低向高往内存传输数据。同时设置last\_shifter寄存器，以最低位表示last信号。该寄存器将1放置在高位并不断右移，从而使得最后一个数据传输时last信号拉高。由于last最好在传输结束释放，额外考虑了向内存的写数据握手信号。

最后，需要根据旁路标志确定向内存写数据的来源，及相应strb信号。

8、控制信号

|  |
| --- |
| //handshake signals related to state machine          assign to\_cpu\_mem\_req\_ready   = cur\_state == WAIT;          assign to\_cpu\_cache\_rsp\_valid = cur\_state == RESP;            assign to\_mem\_rd\_req\_valid  = (cur\_state == MEM\_RD) | (cur\_state == BY\_REQ & ~cpu\_mem\_rw);          assign to\_mem\_rd\_rsp\_ready  = (cur\_state == RECV) | (cur\_state == BY\_RECV) | (cur\_state == WAIT);            assign to\_mem\_wr\_req\_valid  = (cur\_state == MEM\_WR) | (cur\_state == BY\_REQ & cpu\_mem\_rw);          assign to\_mem\_wr\_data\_valid = (cur\_state == TXD) | (cur\_state == BY\_TXD); |

根据相应状态含义确定CPU和内存的请求和响应相关信号。由于旁路读写请求合并为一个状态，而读写请求不能同时有效，此处应当考虑读写类型。特别的，由于内存接口设计，在初始态应当将to\_mem\_rd\_rsp\_ready拉高。

1. 模块例化

\*说明：由于对于4个路组的例化较为重复，仅分别展示0路组的例化。

|  |
| --- |
| //inst of tag\_array and data\_array          tag\_array tag\_way0(                  .clk    (clk),                  .waddr  (set),                  .raddr  (set),                  .wen    (TagWen0),                  .wdata  (tag),                  .rdata  (Tag0)          );  ...          data\_array data\_way0(                  .clk    (clk),                  .waddr  (set),                  .raddr  (set),                  .wen    (DataWen0),                  .wdata  (Array\_Wdata),                  .rdata  (Data0)          ); |

其中set对于不同路组均一致，且写数据同样一致，根据不同路的写使能信号选择写入哪个路组。

10、性能比较(100hz)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 处理器(RISCV) | 时钟周期数 | 指令总数 | 访存延迟周期数 | CPI |
| 15pz | 多周期 | 324238832 | 5224477 | 90004486 | 62.06 |
| 流水线(无cache) | 236498270 | 5224477 | 94278671 | 45.27 |
| 流水线(有cache) | 35492152 | 5224477 | 249547 | 6.79 |
| bf | 多周期 | 23775610 | 452850 | 3956925 | 52.50 |
| 流水线(无cache) | 19655008 | 452850 | 3947242 | 43.40 |
| 流水线(有cache) | 2847937 | 452850 | 630209 | 6.29 |
| dinic | 多周期 | 906059 | 16687 | 171982 | 54.30 |
| 流水线(无cache) | 723066 | 16687 | 172866 | 43.33 |
| 流水线(有cache) | 148090 | 16687 | 9475 | 8.87 |
| fib | 多周期 | 110676382 | 2549521 | 183886 | 43.41 |
| 流水线(无cache) | 110479201 | 2549521 | 185795 | 43.33 |
| 流水线(有cache) | 15598973 | 2549521 | 1853649 | 6.12 |
| md5 | 多周期 | 236203 | 4911 | 22927 | 48.10 |
| 流水线(无cache) | 212774 | 4911 | 22781 | 43.33 |
| 流水线(有cache) | 32305 | 4911 | 774 | 6.58 |
| qsort | 多周期 | 483070 | 9476 | 65405 | 50.98 |
| 流水线(无cache) | 410526 | 9476 | 66011 | 43.32 |
| 流水线(有cache) | 58659 | 9476 | 6992 | 6.19 |
| queen | 多周期 | 4284106 | 81486 | 637557 | 52.57 |
| 流水线(无cache) | 3532577 | 81486 | 640402 | 43.35 |
| 流水线(有cache) | 492974 | 81486 | 43914 | 6.05 |
| sieve | 多周期 | 456038 | 10191 | 14335 | 44.75 |
| 流水线(无cache) | 441542 | 10191 | 14452 | 43.33 |
| 流水线(有cache) | 63947 | 10191 | 4836 | 6.27 |
| ssort | 多周期 | 27431823 | 619041 | 514622 | 44.31 |
| 流水线(无cache) | 26922713 | 619041 | 554778 | 43.49 |
| 流水线(有cache) | 3808486 | 619041 | 413803 | 6.15 |
| 平均结果 | 多周期 | 54720902.56 | 996515.56 | 10619125 | 54.91 |
| 流水线(无cache) | 44319519.67 | 996515.56 | 11098110.89 | 44.47 |
| 流水线(有cache) | 6504835.89 | 996515.56 | 357022.11 | 6.59 |

由上表可见，添加cache后大幅度降低了流水线的访存延迟，增加了指令并行度。添加cache的流水线性能约达到了未加cache流水线的7倍，多周期处理器的9倍。但是加cache流水线的CPI仍只达到6左右，未能降至理想的1，一方面是因为存在一定读写缺失的概率，当读写缺失时，由于需要进行缓存块替换，比直接从内存获得32位数据消耗更多的时钟周期。另一方面，由于icache和dcache仍采用多周期实现，即使读写命中，也需经过多个状态，一定程度上降低了运行效率。如果使用流水线实现cache，有望进一步降低CPI。

1. 结构设计优化  
    由于同时实现流水线和cache，对二者的结构优化同时进行，并使用Dhrystone 和 Coremark进行性能评测。

初始版的icache和dcache均未例化data\_array和tag\_array，而是在模块内部直接定义寄存器，一方面不便于修改cache容量大小，另一方面，由于寄存器摆放较不对齐，会消耗额外的布线资源，因此当cache修改为4x32（4路组，每个路组中32个cache块）之后，bit-gen资源就已经不够了。

在通过例化tag\_array和data\_array后，分别尝试了增大路组数和增大路组内cache块数的尝试。发现使用8路组时，每个路组只能放置16个cache块，且通过性能评测，发现和4x16相比几乎无性能优化，查阅资料得知，路组数达到4以上后，增大路组数提高的效率已经微乎其微，因此后续容量在4路组基础上增大每路组数据块数。

经过尝试，cache容量最大可开至4x128。在验收过程中，通过刘士祺助教的讲解得知vivado会对成块的cache块组进行一定的布局布线优化，且可节省布局布线资源，将cache容量增大到4x128后，根据Dhrystone的性能评测结果，性能约提高了6%，根据Coremark也有了一定的性能提升。

然后，通过增大时钟频率加快处理器的运行。经过测试发现，在280hz时可以保证所有bit-gen任务WNS均大于0，在300hz时也只有部分任务WNS小于0，且处理器通过fpga上板任务的最高频率（不稳定）可以达到400hz以上。对于WNS小于0的任务，查看bit-gen阶段报告可见，关键路径主要为框架dcache相关框架代码部分，优化空间较小，扇出也均符合规范，不需特别优化。

综上，最后得到稳定的处理器为280hz时钟频率下，icache和dcache容量均为4x128的流水线处理器，和最开始100hz下cache容量4x8的流水席处理器性能对比如下：

|  |  |  |
| --- | --- | --- |
| 流水线(有cache) | Dhrystones per second | Comark Iterations/Sec |
| 100hz 4x8 | 5411 | 0.676 |
| 100hz 4x128 | 5774 | 0.676 |
| 280hz 4x128 | 16168 | 1.894 |
| 稳定优化倍数 | 2.988 | 2.802 |

1. 实验过程中遇到的问题、对问题的思考过程及解决方法（比如RTL代码中出现的逻辑bug，逻辑仿真和FPGA调试过程中的难点等）

问题1：仿真通过但上板和硬件仿真无法结束  
 通过手动确定硬件仿真时间范围，查看波形可见最终PC可以到达hit good trap位置，且通过旁路向内存0x0c位置写0。分析与内存交互信号发现，读写有效请求同时拉高，内存优先获取读请求有效信号，而忽略写请求。进一步查看代码相关逻辑，发现旁路读写请求用同一个状态表示，应当进一步根据读写类型进行分类，保证读写请求不同时有效。

问题2：上板每组任务只有第1个可以通过，其余任务可以通过硬件仿真

检查cache内部功能实现无误情况下，将问题定位至与内存交互信号。与多周期处理器和未加cache的流水线处理器对比发现，应当在初始状态拉高向内存的读数据准备信号。添加后即可正确通过。  
问题3：框架代码tag\_array和data\_array问题

在行为仿真过程中，发现部分任务出现了wdata=xxxx的不定态，查看波形发现set改变后，data\_array反馈的数据并未变化，进一步查看发现，问题在于框架代码中声明数组时优先级错误。由于减法优先级高于移位优先级，应当在移位运算外加上括号。

1. 对讲义中思考题（如有）的理解和回答

本次实验无思考题。

1. 在课后，你花费了大约\_\_\_\_\_15\_\_\_\_\_\_小时完成此次实验。
2. 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

心得感受：

本次实验难度适中，较好的加深了对于组相联缓存结构，和缺失替换策略的理解。同时讲义对组路中数据结构的讲解也较为清楚明晰，整体实验实现过程较为顺畅。同时在进行结构设计优化的过程中，通过增大路组数的比较和增大路组内容量的比较，更清晰的了解了组相联缓存策略的替换算法和vivado对于布局布线的优化。

建议：

实验过程中可以强调和内存接口的交互信号问题，鼓励同学们通过查看框架代码掌握内存接口逻辑，从而定位和修改自身代码错误。

致谢：

感谢常老师和陈欲晓助教在硬件仿真工具上的帮助，感谢刘士祺助教在结构设计优化及vivado优化布局布线上的讲解。