实验 7：高速缓存 Cache

张子康 PB22020660

2024 年 5 月 27 日

1 实验目的与内容

1.1 实验目的

在本次实验中，我们将学习高速缓存 Cache 的组织结构和工作机理。

1.2 实验内容

1.2.1 任务 1：二路组相连 Cache

根据给出的直接映射 Cache 的代码，实现二路组相连 Cache，支持 LRU 替换策略，通过助教提供的读写测试。

1.2.2 任务 2：多样化的替换策略

让你的 Cache 支持伪随机替换策略，FIFO 替换策略等除了 LRU 之外至少两种替换策略，通过助教提供的读写测试。

1.2.3 任务 3：N 路组相连 Cache

实现参数化的 N 路组相连 Cache，支持 LRU 替换策略，通过助教提供的读写测试。

2 逻辑设计

本次实验中实现的高速缓存 Cache 均为 N 路组相连，故任务 1 与任务 3 实现代码相同。

2.1 N 路组相连 Cache(LRU)

```
1 module cache_lru #(
2     parameter INDEX_WIDTH       = 3,    // Cache索引位宽
        2^3=8行
3     parameter LINE_OFFSET_WIDTH = 2,    // 行偏移位宽，决定
        了一行的宽度 2^2=4字
```

```
    parameter SPACE_OFFSET      = 2,    // 一个地址空间占1个字节，因此一个字需要4个地址空间，由于假设为整字读取，处理地址的时候可以默认后两位为0
    parameter WAY_NUM           = 2,    // Cache N路组相联(N=1的时候是直接映射),
    parameter N = 1                     // log_2(WAY_NUM),即2的N次方
)(
    input                       clk,
    input                       rstn,
    /* CPU接口 */
    input [31:0]                addr,   // CPU地址
    input                       r_req,  // CPU读请求
    input                       w_req,  // CPU写请求
    input [31:0]                w_data,  // CPU写数据
    output [31:0]               r_data,  // CPU读数据
    output reg                  miss,   // 缓存未命中
    /* 内存接口 */
    output reg                      mem_r,  // 内存读请求
    output reg                      mem_w,  // 内存写请求
    output reg [31:0]               mem_addr,  // 内存地址
    output reg [127:0] mem_w_data,  // 内存写数据 一次写一行
    input      [127:0] mem_r_data,  // 内存读数据 一次读一行
    input                           mem_ready  // 内存就绪信号
);

    // Cache参数
    localparam
        // Cache行宽度
        LINE_WIDTH = 32 << LINE_OFFSET_WIDTH,
```

```
30          // 标记位宽度
31          TAG_WIDTH = 32 - INDEX_WIDTH - LINE_OFFSET_WIDTH -
                SPACE_OFFSET,
32          // Cache行数
33          SET_NUM    = 1 << INDEX_WIDTH;
34
35      // Cache相关寄存器
36      reg [31:0]            addr_buf;     // 请求地址缓存-用于
            保留CPU请求地址
37      reg [31:0]            w_data_buf;   // 写数据缓存
38      reg op_buf;  // 读写操作缓存，用于在MISS状态下判断是读
            还是写，如果是写则需要将数据写回内存 0:读 1:写
39      reg [LINE_WIDTH-1:0] ret_buf;      // 返回数据缓存-用于
            保留内存返回数据
40
41      // Cache导线
42      wire [INDEX_WIDTH-1:0] r_index;  // 索引读地址
43      wire [INDEX_WIDTH-1:0] w_index;  // 索引写地址
44      wire [LINE_WIDTH-1:0]  r_line;   // Data Bram读数据
45      wire [LINE_WIDTH-1:0]  w_line;   // Data Bram写数据
46      wire [LINE_WIDTH-1:0]  w_line_mask;  // Data Bram写数据
            掩码
47      wire [LINE_WIDTH-1:0]  w_data_line;  // 输入写数据移位
            后的数据
48      wire [TAG_WIDTH-1:0]   tag;       // CPU请求地址中分离的
            标记 用于比较 也可用于写入
49      wire [TAG_WIDTH-1:0]   r_tag;     // Tag Bram读数据 用于
            比较
50      wire [LINE_OFFSET_WIDTH-1:0] word_offset;  // 字偏移
51      reg  [31:0]            cache_data;  // Cache数据
52      reg  [31:0]            mem_data;    // 内存数据
53      wire [31:0]            dirty_mem_addr; // 通过读出的tag
            和对应的index，偏移等得到脏块对应的内存地址并写回到
```

```
        正确的位置
54      wire valid;  // Cache有效位
55      wire dirty;  // Cache脏位.
56      reg  w_valid;  // Cache写有效位
57      reg  w_dirty;  // Cache写脏位
58      wire hit;    // Cache命中
59
60      // Cache相关控制信号
61      reg addr_buf_we;  // 请求地址缓存写使能
62      reg ret_buf_we;   // 返回数据缓存写使能
63      reg [WAY_NUM-1:0]data_we;       // Cache写使能
64      reg [WAY_NUM-1:0]tag_we;        // Cache标记写使能
65      reg data_from_mem;  // 从内存读取数据
66      reg refill;        // 标记需要重新填充, 在MISS状态下接受
        到内存数据后置1,在IDLE状态下进行填充后置0
67
68      // 状态机信号
69      localparam
70          IDLE       = 3'd0,  // 空闲状态
71          READ       = 3'd1,  // 读状态
72          MISS       = 3'd2,  // 缺失时等待主存读出新块
73          WRITE      = 3'd3,  // 写状态
74          W_DIRTY    = 3'd4;  // 写缺失时等待主存写入脏块
75      reg [2:0] CS;  // 状态机当前状态
76      reg [2:0] NS;  // 状态机下一状态
77
78      // 状态机
79      always @(posedge clk or negedge rstn) begin
80          if (!rstn) begin
81              CS <= IDLE;
82          end else begin
83              CS <= NS;
84          end
```

```
85      end
86
87
88      // 中间寄存器保留初始的请求地址和写数据，可以理解为
            addr_buf中的地址为当前Cache正在处理的请求地址，而
            addr中的地址为新的请求地址
89      always @(posedge clk or negedge rstn) begin
90          if (!rstn) begin
91              addr_buf <= 0;
92              ret_buf <= 0;
93              w_data_buf <= 0;
94              op_buf <= 0;
95              refill <= 0;
96          end else begin
97              if (addr_buf_we) begin
98                  addr_buf <= addr;
99                  w_data_buf <= w_data;
100                 op_buf <= w_req;
101             end
102             if (ret_buf_we) begin
103                 ret_buf <= mem_r_data;
104             end
105             if (CS == MISS && mem_ready) begin
106                 refill <= 1;
107             end
108             if (CS == IDLE) begin
109                 refill <= 0;
110             end
111         end
112     end
113
114     // 对输入地址进行解码
115     assign r_index = addr[INDEX_WIDTH+LINE_OFFSET_WIDTH+
```

```
    SPACE_OFFSET - 1: LINE_OFFSET_WIDTH+SPACE_OFFSET];
116 assign w_index = addr_buf[INDEX_WIDTH+LINE_OFFSET_WIDTH
    +SPACE_OFFSET - 1: LINE_OFFSET_WIDTH+SPACE_OFFSET];
117 assign tag = addr_buf[31:INDEX_WIDTH+LINE_OFFSET_WIDTH+
    SPACE_OFFSET];
118 assign word_offset = addr_buf[LINE_OFFSET_WIDTH+
    SPACE_OFFSET-1:SPACE_OFFSET];
119
120 // 脏块地址计算
121 assign dirty_mem_addr = {r_tag, w_index}<<(
    LINE_OFFSET_WIDTH+SPACE_OFFSET);
122
123 // 写回地址、数据寄存器
124 reg [31:0] dirty_mem_addr_buf;
125 reg [127:0] dirty_mem_data_buf;
126 always @(posedge clk or negedge rstn) begin
127     if (!rstn) begin
128         dirty_mem_addr_buf <= 0;
129         dirty_mem_data_buf <= 0;
130     end else begin
131         if (CS == READ || CS == WRITE) begin
132             dirty_mem_addr_buf <= dirty_mem_addr;
133             dirty_mem_data_buf <= r_line;
134         end
135     end
136 end
137
138 // 伪LRU
139 reg [N-1:0] p;
140 wire [N-1:0] r_p;
141 reg p_we;
142
143 wire valid_[WAY_NUM-1:0];  // Cache有效位
```

```
144     wire dirty_[WAY_NUM-1:0];  // Cache脏位.
145     reg  w_valid_[WAY_NUM-1:0];  // Cache写有效位
146     reg  w_dirty_[WAY_NUM-1:0];  // Cache写脏位
147     wire hit_[WAY_NUM-1:0];    // Cache命中
148     wire [TAG_WIDTH-1:0] r_tag_[WAY_NUM-1:0];
149     wire [LINE_WIDTH-1:0] w_line_[WAY_NUM-1:0];
150     wire [LINE_WIDTH-1:0] r_line_[WAY_NUM-1:0];
151
152
153     // 生成相关块
154     // 伪LRU
155     reg [(WAY_NUM-2>0?WAY_NUM-2:0):0] age;
156     wire [(WAY_NUM-2>0?WAY_NUM-2:0):0] r_age;
157     reg age_we;
158     bram #(
159             .ADDR_WIDTH(INDEX_WIDTH),
160             .DATA_WIDTH((WAY_NUM-2>0?WAY_NUM-2:0)+1)
161     ) bram_age(
162             .clk(clk),
163             .raddr(r_index),
164             .waddr(w_index),
165             .din(age),
166             .we(age_we),
167             .dout(r_age)
168     );
169
170     generate
171         genvar i;
172         for (i = 0; i<WAY_NUM; i=i+1) begin:Bram
173             // Tag Bram
174             bram #(
175                 .ADDR_WIDTH(INDEX_WIDTH),
176                 .DATA_WIDTH(TAG_WIDTH + 2) // 最高位为有效
```

```
                    位，次高位为脏位，低位为标记位
            ) tag_bram(
                .clk(clk),
                .raddr(r_index),
                .waddr(w_index),
                .din({w_valid,w_dirty, tag}),
                .we(tag_we[i]),
                .dout({valid_[i],dirty_[i], r_tag_[i]})
            );
            // Data Bram
            bram #(
                .ADDR_WIDTH(INDEX_WIDTH),
                .DATA_WIDTH(LINE_WIDTH)
            ) data_bram(
                .clk(clk),
                .raddr(r_index),
                .waddr(w_index),
                .din(w_line),
                .we(data_we[i]),
                .dout(r_line_[i])
            );
        end
    endgenerate


    // 判定Cache是否命中
    reg [N-1:0] j,tem_j,tem_j2,tem_j3,i_,tem_i;
    reg flag,tem;
    integer i1;
    initial begin
        flag=0;
        tem=0;
        j=0;
```

```
209         tem_j=0;
210         i_=0;
211         tem_i=0;
212     end
213     always @(*) begin
214         if(CS==READ || CS==WRITE)begin
215             flag=0;
216             tem=0;
217             j=0;
218             tem_j=0;
219             i_=0;
220             tem_i=0;
221             // 若命中j中为命中的块的序号
222             for (i1 = 0;i1<WAY_NUM ; i1=i1+1) begin
223                 tem=flag;
224                 flag=tem || (r_tag_[i1] == tag && valid_[i1
                        ]);
225                 if(r_tag_[i1] == tag && valid_[i1])begin
226                     j=i1;
227                 end
228             end
229             // 若未命中，则计算需要替换的块的序号，存在j中
230             if(!flag)begin
231                 for(i1=0;i1<N;i1=i1+1)begin
232                     tem_j=j;
233                     j=(tem_j<<1)+r_age[i_];
234                     tem_i=i_;
235                     i_=((tem_i+1)<<1)-r_age[tem_i];
236                 end
237             end
238         end
239     end
240     assign r_tag=r_tag_[j];
```

```
241     assign dirty=dirty_[j];
242     assign r_line=r_line_[j];
243     assign hit = flag;
244 
245     // 写入Cache 这里要判断是命中后写入还是未命中后写入
246     assign w_line_mask = 32'hFFFFFFFF << (word_offset*32);
            // 写入数据掩码
247     assign w_data_line = w_data_buf << (word_offset*32);
            // 写入数据移位
248     assign w_line = (CS == IDLE && op_buf) ? ret_buf & ~
        w_line_mask | w_data_line : // 写入未命中, 需要将内
        存数据与写入数据合并
249                     (CS == IDLE) ? ret_buf : // 读取未命中
250                     r_line & ~w_line_mask | w_data_line; //
                            写入命中,需要对读取的数据与写入的数
                        据进行合并
251 
252     // 选择输出数据 从Cache或者从内存 这里的选择与行大小有
        关, 因此如果你调整了行偏移位宽, 这里也需要调整
253     always @(*) begin
254         case (word_offset)
255             0: begin
256                 cache_data = r_line[31:0];
257                 mem_data = ret_buf[31:0];
258             end
259             1: begin
260                 cache_data = r_line[63:32];
261                 mem_data = ret_buf[63:32];
262             end
263             2: begin
264                 cache_data = r_line[95:64];
265                 mem_data = ret_buf[95:64];
266             end
```

```
267             3: begin
268                 cache_data = r_line[127:96];
269                 mem_data = ret_buf[127:96];
270             end
271             default: begin
272                 cache_data = 0;
273                 mem_data = 0;
274             end
275         endcase
276     end
277 
278     assign r_data = data_from_mem ? mem_data : hit ?
        cache_data : 0;
279 
280     // 状态机更新逻辑
281     always @(*) begin
282         case(CS)
283             IDLE: begin
284                 if (r_req) begin
285                     NS = READ;
286                 end else if (w_req) begin
287                     NS = WRITE;
288                 end else begin
289                     NS = IDLE;
290                 end
291             end
292             READ: begin
293                 if (miss&& !dirty) begin
294                     NS = MISS;
295                 end else if (miss && dirty) begin
296                     NS = W_DIRTY;
297                 end else if (r_req) begin
298                     NS = READ;
```

```
299                 end else if (w_req) begin
300                     NS = WRITE;
301                 end else begin
302                     NS = IDLE;
303                 end
304             end
305             MISS: begin
306                 if (mem_ready) begin // 这里回到IDLE的原因
                        是为了延迟一周期，等待主存读出的新块写入
                        Cache中的对应位置
307                     NS = IDLE;
308                 end else begin
309                     NS = MISS;
310                 end
311             end
312             WRITE: begin
313                 if (miss && !dirty) begin
314                     NS = MISS;
315                 end else if (miss && dirty) begin
316                     NS = W_DIRTY;
317                 end else if (r_req) begin
318                     NS = READ;
319                 end else if (w_req) begin
320                     NS = WRITE;
321                 end else begin
322                     NS = IDLE;
323                 end
324             end
325             W_DIRTY: begin
326                 if (mem_ready) begin  // 写完脏块后回到MISS
                        状态等待主存读出新块
327                     NS = MISS;
328                 end else begin
```

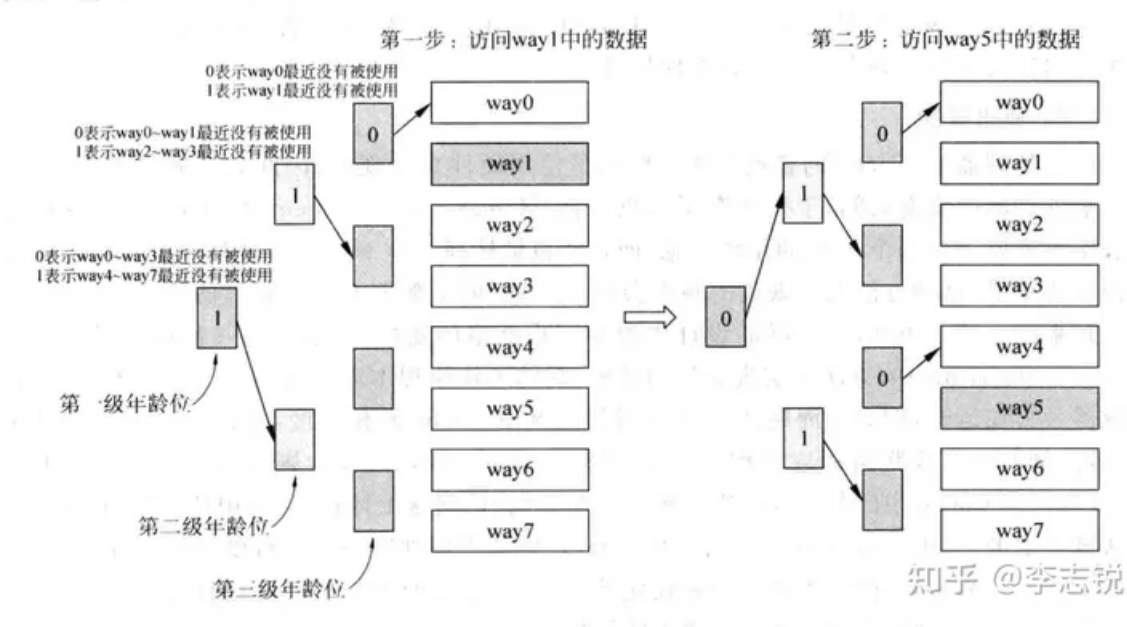
```
329                     NS = W_DIRTY;
330                 end
331             end
332             default: begin
333                 NS = IDLE;
334             end
335         endcase
336     end
337     initial begin
338         p=0;
339     end
340     // 状态机控制信号
341     always @(*) begin
342         addr_buf_we   = 1'b0;
343         ret_buf_we    = 1'b0;
344         data_we       = 0;
345         tag_we        = 0;
346         w_valid       = 1'b0;
347         w_dirty       = 1'b0;
348         data_from_mem = 1'b0;
349         miss          = 1'b0;
350         mem_r         = 1'b0;
351         mem_w         = 1'b0;
352         mem_addr      = 32'b0;
353         mem_w_data    = 0;
354         tem_j2=0;
355         tem_j3=0;
356         age_we=0;
357         case(CS)
358             IDLE: begin
359                 addr_buf_we = 1'b1; // 请求地址缓存写使能
360                 miss = 1'b0;
361                 ret_buf_we = 1'b0;
```

```
362                 if(refill) begin
363                     data_from_mem = 1'b1;
364                     w_valid = 1'b1;
365                     w_dirty = 1'b0;
366                     data_we = 1'b1<<j;
367                     tag_we = 1'b1<<j;
368                     if (op_buf) begin // 写
369                         w_dirty = 1'b1;
370                     end
371                     // 更新age
372                     age_we=1;
373                     age=r_age;
374                     for (i1 = 0;i1< N;i1=i1+1) begin
375                         age[tem_j3]=~j[i1];
376                         tem_j3=tem_j2;
377                         tem_j2=((tem_j3+1)<<1)-j[i1];
378                     end
379                 end
380             end
381             READ: begin
382                 data_from_mem = 1'b0;
383                 if (hit) begin // 命中
384                     miss = 1'b0;
385                     addr_buf_we = 1'b1; // 请求地址缓存写使
                        能
386                     // 更新age
387                     age_we=1;
388                     age=r_age;
389                     for (i1 = 0;i1< N;i1=i1+1) begin
390                         age[tem_j3]=~j[i1];
391                         tem_j3=tem_j2;
392                         tem_j2=((tem_j3+1)<<1)-j[i1];
393                     end
```

```
394                 end else begin // 未命中
395                     miss = 1'b1;
396                     addr_buf_we = 1'b0;
397                     if (dirty) begin // 脏数据需要写回
398                         mem_w = 1'b1;
399                         mem_addr = dirty_mem_addr;
400                         mem_w_data = r_line; // 写回数据
401                     end
402                 end
403             end
404             MISS: begin
405                 miss = 1'b1;
406                 mem_r = 1'b1;
407                 mem_addr = addr_buf;
408                 if (mem_ready) begin
409                     mem_r = 1'b0;
410                     ret_buf_we = 1'b1;
411                 end
412             end
413             WRITE: begin
414                 data_from_mem = 1'b0;
415                 if (hit) begin // 命中
416                     miss = 1'b0;
417                     addr_buf_we = 1'b1; // 请求地址缓存写使
                        能
418                     w_valid = 1'b1;
419                     w_dirty = 1'b1;
420                     // 通过控制写使能信号控制写入块的序号
421                     data_we = 1'b1<<j;
422                     tag_we = 1'b1<<j;
423                     // 更新age
424                     age_we=1;
425                     age=r_age;
```

```
426                     for (i1 = 0;i1< N;i1=i1+1) begin
427                         age[tem_j3]=~j[i1];
428                         tem_j3=tem_j2;
429                         tem_j2=((tem_j3+1)<<1)-j[i1];
430                     end
431                 end else begin // 未命中
432                     miss = 1'b1;
433                     addr_buf_we = 1'b0;
434                     if (dirty) begin // 脏数据需要写回
435                         mem_w = 1'b1;
436                         mem_addr = dirty_mem_addr;
437                         mem_w_data = r_line; // 写回数据
438                     end
439                 end
440             end
441             W_DIRTY: begin
442                 miss = 1'b1;
443                 mem_w = 1'b1;
444                 mem_addr = dirty_mem_addr_buf;
445                 mem_w_data = dirty_mem_data_buf;
446                 if (mem_ready) begin
447                     mem_w = 1'b0;
448                 end
449             end
450             default:;
451         endcase
452     end
453
454 endmodule
```

采用伪 LRU 实现，具体原理如下：



2.2 多样化的替换策略

主要通过更改 j 的计算方法实现。

2.2.1 FIFO

```
/*
直接映射Cache
- Cache行数：8行
- 块大小：4字（16字节 128位）
- 采用写回写分配策略
*/
module cache_fifo #(
    parameter INDEX_WIDTH       = 3,    // Cache索引位宽 2^3=8行
    parameter LINE_OFFSET_WIDTH = 2,    // 行偏移位宽，决定了一行的宽度 2^2=4字
    parameter SPACE_OFFSET      = 2,    // 一个地址空间占1个字节，因此一个字需要4个地址空间，由于假设为整字读
```

```
        取，处理地址的时候可以默认后两位为0
11      parameter WAY_NUM           = 2,    // Cache N路组相联(
            N=1的时候是直接映射),
12      parameter N = 1                     // log_2(WAY_NUM),
            即2的N次方
13  )(
14      input                       clk,
15      input                       rstn,
16      /* CPU接口 */
17      input [31:0]                addr,   // CPU地址
18      input                       r_req,  // CPU读请求
19      input                       w_req,  // CPU写请求
20      input [31:0]                w_data,  // CPU写数据
21      output [31:0]               r_data,  // CPU读数据
22      output reg                  miss,   // 缓存未命中
23      /* 内存接口 */
24      output reg                      mem_r,  // 内存读请求
25      output reg                      mem_w,  // 内存写请求
26      output reg [31:0]               mem_addr,  // 内存地址
27      output reg [127:0] mem_w_data,  // 内存写数据 一次写一
            行
28      input      [127:0] mem_r_data,  // 内存读数据 一次读一
            行
29      input                           mem_ready  // 内存就绪信
            号
30  );
31
32      // Cache参数
33      localparam
34          // Cache行宽度
35          LINE_WIDTH = 32 << LINE_OFFSET_WIDTH,
36          // 标记位宽度
37          TAG_WIDTH = 32 - INDEX_WIDTH - LINE_OFFSET_WIDTH -
```

```
        SPACE_OFFSET,
38      // Cache行数
39      SET_NUM    = 1 << INDEX_WIDTH;
40
41  // Cache相关寄存器
42  reg [31:0]            addr_buf;     // 请求地址缓存-用于
        保留CPU请求地址
43  reg [31:0]            w_data_buf;   // 写数据缓存
44  reg op_buf;   // 读写操作缓存，用于在MISS状态下判断是读
        还是写，如果是写则需要将数据写回内存 0:读 1:写
45  reg [LINE_WIDTH-1:0] ret_buf;      // 返回数据缓存-用于
        保留内存返回数据
46
47  // Cache导线
48  wire [INDEX_WIDTH-1:0] r_index;   // 索引读地址
49  wire [INDEX_WIDTH-1:0] w_index;   // 索引写地址
50  wire [LINE_WIDTH-1:0]  r_line;    // Data Bram读数据
51  wire [LINE_WIDTH-1:0]  w_line;    // Data Bram写数据
52  wire [LINE_WIDTH-1:0]  w_line_mask;  // Data Bram写数据
        掩码
53  wire [LINE_WIDTH-1:0]  w_data_line;  // 输入写数据移位
        后的数据
54  wire [TAG_WIDTH-1:0]   tag;       // CPU请求地址中分离的
        标记 用于比较 也可用于写入
55  wire [TAG_WIDTH-1:0]   r_tag;     // Tag Bram读数据 用于
        比较
56  wire [LINE_OFFSET_WIDTH-1:0] word_offset;  // 字偏移
57  reg  [31:0]            cache_data;  // Cache数据
58  reg  [31:0]            mem_data;    // 内存数据
59  wire [31:0]            dirty_mem_addr; // 通过读出的tag
        和对应的index，偏移等得到脏块对应的内存地址并写回到
        正确的位置
60  wire valid;  // Cache有效位
```

```
61      wire dirty;  // Cache脏位.
62      reg  w_valid;  // Cache写有效位
63      reg  w_dirty;  // Cache写脏位
64      wire hit;    // Cache命中
65
66      // Cache相关控制信号
67      reg addr_buf_we;  // 请求地址缓存写使能
68      reg ret_buf_we;   // 返回数据缓存写使能
69      reg [WAY_NUM-1:0]data_we;       // Cache写使能
70      reg [WAY_NUM-1:0]tag_we;        // Cache标记写使能
71      reg data_from_mem;  // 从内存读取数据
72      reg refill;        // 标记需要重新填充, 在MISS状态下接受
            到内存数据后置1,在IDLE状态下进行填充后置0
73
74      // 状态机信号
75      localparam
76          IDLE      = 3'd0,  // 空闲状态
77          READ      = 3'd1,  // 读状态
78          MISS      = 3'd2,  // 缺失时等待主存读出新块
79          WRITE     = 3'd3,  // 写状态
80          W_DIRTY   = 3'd4;  // 写缺失时等待主存写入脏块
81      reg [2:0] CS;  // 状态机当前状态
82      reg [2:0] NS;  // 状态机下一状态
83
84      // 状态机
85      always @(posedge clk or negedge rstn) begin
86          if (!rstn) begin
87              CS <= IDLE;
88          end else begin
89              CS <= NS;
90          end
91      end
92
```

```
93      // 中间寄存器保留初始的请求地址和写数据，可以理解为
           addr_buf中的地址为当前Cache正在处理的请求地址，而
           addr中的地址为新的请求地址
94      always @(posedge clk or negedge rstn) begin
95          if (!rstn) begin
96              addr_buf <= 0;
97              ret_buf <= 0;
98              w_data_buf <= 0;
99              op_buf <= 0;
100             refill <= 0;
101         end else begin
102             if (addr_buf_we) begin
103                 addr_buf <= addr;
104                 w_data_buf <= w_data;
105                 op_buf <= w_req;
106             end
107             if (ret_buf_we) begin
108                 ret_buf <= mem_r_data;
109             end
110             if (CS == MISS && mem_ready) begin
111                 refill <= 1;
112             end
113             if (CS == IDLE) begin
114                 refill <= 0;
115             end
116         end
117     end
118
119     // 对输入地址进行解码
120     assign r_index = addr[INDEX_WIDTH+LINE_OFFSET_WIDTH+
           SPACE_OFFSET - 1: LINE_OFFSET_WIDTH+SPACE_OFFSET];
121     assign w_index = addr_buf[INDEX_WIDTH+LINE_OFFSET_WIDTH
           +SPACE_OFFSET - 1: LINE_OFFSET_WIDTH+SPACE_OFFSET];
```

```
122     assign tag = addr_buf[31:INDEX_WIDTH+LINE_OFFSET_WIDTH+
            SPACE_OFFSET];
123     assign word_offset = addr_buf[LINE_OFFSET_WIDTH+
            SPACE_OFFSET-1:SPACE_OFFSET];
124
125     // 脏块地址计算
126     assign dirty_mem_addr = {r_tag, w_index}<<(
            LINE_OFFSET_WIDTH+SPACE_OFFSET);
127
128     // 写回地址、数据寄存器
129     reg [31:0] dirty_mem_addr_buf;
130     reg [127:0] dirty_mem_data_buf;
131     always @(posedge clk or negedge rstn) begin
132         if (!rstn) begin
133             dirty_mem_addr_buf <= 0;
134             dirty_mem_data_buf <= 0;
135         end else begin
136             if (CS == READ || CS == WRITE) begin
137                 dirty_mem_addr_buf <= dirty_mem_addr;
138                 dirty_mem_data_buf <= r_line;
139             end
140         end
141     end
142
143     // FIFO
144     reg [N-1:0] p;
145     wire [N-1:0] r_p;
146     reg p_we;
147
148     wire valid_[WAY_NUM-1:0];  // Cache有效位
149     wire dirty_[WAY_NUM-1:0];  // Cache脏位.
150     reg  w_valid_[WAY_NUM-1:0];  // Cache写有效位
151     reg  w_dirty_[WAY_NUM-1:0];  // Cache写脏位
```

```
152     wire hit_[WAY_NUM-1:0];     // Cache命中
153     wire [TAG_WIDTH-1:0] r_tag_[WAY_NUM-1:0];
154     wire [LINE_WIDTH-1:0] w_line_[WAY_NUM-1:0];
155     wire [LINE_WIDTH-1:0] r_line_[WAY_NUM-1:0];
156
157     // 该块用于存储要写入的块序号
158     bram #(
159             .ADDR_WIDTH(INDEX_WIDTH),
160             .DATA_WIDTH(N)
161     ) bram_p(
162             .clk(clk),
163             .raddr(r_index),
164             .waddr(w_index),
165             .din(p),
166             .we(p_we),
167             .dout(r_p)
168     );
169
170     generate
171         genvar i;
172         for (i = 0; i<WAY_NUM; i=i+1) begin:Bram
173             // Tag Bram
174             bram #(
175                 .ADDR_WIDTH(INDEX_WIDTH),
176                 .DATA_WIDTH(TAG_WIDTH + 2) // 最高位为有效
                        位，次高位为脏位，低位为标记位
177             ) tag_bram(
178                 .clk(clk),
179                 .raddr(r_index),
180                 .waddr(w_index),
181                 .din({w_valid,w_dirty, tag}),
182                 .we(tag_we[i]),
183                 .dout({valid_[i],dirty_[i], r_tag_[i]})
```

```
184             );
185             // Data Bram
186             bram #(
187                 .ADDR_WIDTH(INDEX_WIDTH),
188                 .DATA_WIDTH(LINE_WIDTH)
189             ) data_bram(
190                 .clk(clk),
191                 .raddr(r_index),
192                 .waddr(w_index),
193                 .din(w_line),
194                 .we(data_we[i]),
195                 .dout(r_line_[i])
196             );
197         end
198     endgenerate
199
200
201
202     // 判定Cache是否命中
203     reg [N-1:0] j,tem_j;
204     reg flag,tem;
205     integer i1;
206     initial begin
207         flag=0;
208         tem=0;
209         j=6'b000000;
210         tem_j=6'b000000;
211     end
212     always @(*) begin
213         if(CS==READ || CS==WRITE)begin
214             flag=0;
215             tem=0;
216             j=6'b000000;
```

```
217             tem_j=6'b000000;
218             // 若命中则j为命中的块的序号
219             for (i1 = 0;i1<WAY_NUM ; i1=i1+1) begin
220                 tem=flag;
221                 flag=tem || (r_tag_[i1] == tag && valid_[i1
                        ]);
222                 if(r_tag_[i1] == tag && valid_[i1])begin
223                     j=i1;
224                 end
225             end
226             // 若未命中则j为需要写入的块的序号
227             if(!flag)begin
228                 j=r_p;
229             end
230         end
231
232     end
233     assign r_tag=r_tag_[j];
234     assign dirty=dirty_[j];
235     assign r_line=r_line_[j];
236     assign hit = flag;
237
238     // 写入Cache 这里要判断是命中后写入还是未命中后写入
239     assign w_line_mask = 32'hFFFFFFFF << (word_offset*32);
            // 写入数据掩码
240     assign w_data_line = w_data_buf << (word_offset*32);
            // 写入数据移位
241     assign w_line = (CS == IDLE && op_buf) ? ret_buf & ~
        w_line_mask | w_data_line : // 写入未命中, 需要将内
        存数据与写入数据合并
242                     (CS == IDLE) ? ret_buf : // 读取未命中
243                     r_line & ~w_line_mask | w_data_line; //
                            写入命中,需要对读取的数据与写入的数
```

```
                据进行合并
244
245     // 选择输出数据 从Cache或者从内存 这里的选择与行大小有
            关，因此如果你调整了行偏移位宽，这里也需要调整
246     always @(*) begin
247         case (word_offset)
248             0: begin
249                 cache_data = r_line[31:0];
250                 mem_data = ret_buf[31:0];
251             end
252             1: begin
253                 cache_data = r_line[63:32];
254                 mem_data = ret_buf[63:32];
255             end
256             2: begin
257                 cache_data = r_line[95:64];
258                 mem_data = ret_buf[95:64];
259             end
260             3: begin
261                 cache_data = r_line[127:96];
262                 mem_data = ret_buf[127:96];
263             end
264             default: begin
265                 cache_data = 0;
266                 mem_data = 0;
267             end
268         endcase
269     end
270
271     assign r_data = data_from_mem ? mem_data : hit ?
            cache_data : 0;
272
273     // 状态机更新逻辑
```

```
274     always @(*) begin
275         case(CS)
276             IDLE: begin
277                 if (r_req) begin
278                     NS = READ;
279                 end else if (w_req) begin
280                     NS = WRITE;
281                 end else begin
282                     NS = IDLE;
283                 end
284             end
285             READ: begin
286                 if (miss&& !dirty) begin
287                     NS = MISS;
288                 end else if (miss && dirty) begin
289                     NS = W_DIRTY;
290                 end else if (r_req) begin
291                     NS = READ;
292                 end else if (w_req) begin
293                     NS = WRITE;
294                 end else begin
295                     NS = IDLE;
296                 end
297             end
298             MISS: begin
299                 if (mem_ready) begin // 这里回到IDLE的原因
                        是为了延迟一周期，等待主存读出的新块写入
                        Cache中的对应位置
300                     NS = IDLE;
301                 end else begin
302                     NS = MISS;
303                 end
304             end
```

```
305             WRITE: begin
306                 if (miss && !dirty) begin
307                     NS = MISS;
308                 end else if (miss && dirty) begin
309                     NS = W_DIRTY;
310                 end else if (r_req) begin
311                     NS = READ;
312                 end else if (w_req) begin
313                     NS = WRITE;
314                 end else begin
315                     NS = IDLE;
316                 end
317             end
318             W_DIRTY: begin
319                 if (mem_ready) begin  // 写完脏块后回到MISS
                        状态等待主存读出新块
320                     NS = MISS;
321                 end else begin
322                     NS = W_DIRTY;
323                 end
324             end
325             default: begin
326                 NS = IDLE;
327             end
328         endcase
329     end
330     initial begin
331         p=0;
332     end
333     // 状态机控制信号
334     always @(*) begin
335         addr_buf_we   = 1'b0;
336         ret_buf_we    = 1'b0;
```

```
337         data_we        = 0;
338         tag_we         = 0;
339         w_valid        = 1'b0;
340         w_dirty        = 1'b0;
341         data_from_mem  = 1'b0;
342         miss           = 1'b0;
343         mem_r          = 1'b0;
344         mem_w          = 1'b0;
345         mem_addr       = 32'b0;
346         mem_w_data     = 0;
347         p=r_p;
348         p_we=0;
349         case(CS)
350             IDLE: begin
351                 addr_buf_we = 1'b1; // 请求地址缓存写使能
352                 miss = 1'b0;
353                 ret_buf_we = 1'b0;
354                 if(refill) begin
355                     data_from_mem = 1'b1;
356                     w_valid = 1'b1;
357                     w_dirty = 1'b0;
358                     data_we = 1'b1<<j;
359                     tag_we = 1'b1<<j;
360                     // 块序号指针向后移动一位
361                     p=r_p+1;
362                     p_we=1;
363                     if (op_buf) begin // 写
364                         w_dirty = 1'b1;
365                     end
366                 end
367             end
368             READ: begin
369                 data_from_mem = 1'b0;
```

```
370                 if (hit) begin // 命中
371                     miss = 1'b0;
372                     addr_buf_we = 1'b1; // 请求地址缓存写使
                            能
373                 end else begin // 未命中
374                     miss = 1'b1;
375                     addr_buf_we = 1'b0;
376                     if (dirty) begin // 脏数据需要写回
377                         mem_w = 1'b1;
378                         mem_addr = dirty_mem_addr;
379                         mem_w_data = r_line; // 写回数据
380                     end
381                 end
382             end
383             MISS: begin
384                 miss = 1'b1;
385                 mem_r = 1'b1;
386                 mem_addr = addr_buf;
387                 if (mem_ready) begin
388                     mem_r = 1'b0;
389                     ret_buf_we = 1'b1;
390                 end
391             end
392             WRITE: begin
393                 data_from_mem = 1'b0;
394                 if (hit) begin // 命中
395                     miss = 1'b0;
396                     addr_buf_we = 1'b1; // 请求地址缓存写使
                            能
397                     w_valid = 1'b1;
398                     w_dirty = 1'b1;
399                     data_we = 1'b1<<j;
400                     tag_we = 1'b1<<j;
```

```
                end else begin // 未命中
                    miss = 1'b1;
                    addr_buf_we = 1'b0;
                    if (dirty) begin // 脏数据需要写回
                        mem_w = 1'b1;
                        mem_addr = dirty_mem_addr;
                        mem_w_data = r_line; // 写回数据
                    end
                end
            end
            W_DIRTY: begin
                miss = 1'b1;
                mem_w = 1'b1;
                mem_addr = dirty_mem_addr_buf;
                mem_w_data = dirty_mem_data_buf;
                if (mem_ready) begin
                    mem_w = 1'b0;
                end
            end
            default:;
        endcase
    end

endmodule
```

2.2.2 RANDOM

采用伪随机数实现。

```
/*
直接映射Cache
- Cache行数：8行
- 块大小：4字（16字节 128位）
- 采用写回写分配策略
```

```
6  */
7  module cache_random #(
8      parameter INDEX_WIDTH       = 3,    // Cache索引位宽
           2^3=8行
9      parameter LINE_OFFSET_WIDTH = 2,    // 行偏移位宽，决定
           了一行的宽度 2^2=4字
10     parameter SPACE_OFFSET      = 2,    // 一个地址空间占1
           个字节，因此一个字需要4个地址空间，由于假设为整字读
           取，处理地址的时候可以默认后两位为0
11     parameter WAY_NUM           = 2,    // Cache N路组相联(
           N=1的时候是直接映射),
12     parameter N = 0
13 )(
14     input                       clk,
15     input                       rstn,
16     /* CPU接口 */
17     input [31:0]                addr,   // CPU地址
18     input                       r_req,  // CPU读请求
19     input                       w_req,  // CPU写请求
20     input [31:0]                w_data, // CPU写数据
21     output [31:0]               r_data, // CPU读数据
22     output reg                  miss,   // 缓存未命中
23     /* 内存接口 */
24     output reg                      mem_r,  // 内存读请求
25     output reg                      mem_w,  // 内存写请求
26     output reg [31:0]               mem_addr,  // 内存地址
27     output reg [127:0] mem_w_data,  // 内存写数据 一次写一
           行
28     input      [127:0] mem_r_data,  // 内存读数据 一次读一
           行
29     input                           mem_ready  // 内存就绪信
           号
30 );
```

```
31
32      // Cache参数
33      localparam
34          // Cache行宽度
35          LINE_WIDTH = 32 << LINE_OFFSET_WIDTH,
36          // 标记位宽度
37          TAG_WIDTH = 32 - INDEX_WIDTH - LINE_OFFSET_WIDTH -
                SPACE_OFFSET,
38          // Cache行数
39          SET_NUM    = 1 << INDEX_WIDTH;
40
41      // Cache相关寄存器
42      reg [31:0]            addr_buf;     // 请求地址缓存-用于
            保留CPU请求地址
43      reg [31:0]            w_data_buf;   // 写数据缓存
44      reg op_buf;   // 读写操作缓存，用于在MISS状态下判断是读
            还是写，如果是写则需要将数据写回内存 0:读 1:写
45      reg [LINE_WIDTH-1:0] ret_buf;      // 返回数据缓存-用于
            保留内存返回数据
46
47      // Cache导线
48      wire [INDEX_WIDTH-1:0] r_index;   // 索引读地址
49      wire [INDEX_WIDTH-1:0] w_index;   // 索引写地址
50      wire [LINE_WIDTH-1:0]  r_line;    // Data Bram读数据
51      wire [LINE_WIDTH-1:0]  w_line;    // Data Bram写数据
52      wire [LINE_WIDTH-1:0]  w_line_mask;  // Data Bram写数据
            掩码
53      wire [LINE_WIDTH-1:0]  w_data_line;  // 输入写数据移位
            后的数据
54      wire [TAG_WIDTH-1:0]   tag;       // CPU请求地址中分离的
            标记 用于比较 也可用于写入
55      wire [TAG_WIDTH-1:0]   r_tag;     // Tag Bram读数据 用于
            比较
```

```
56      wire [LINE_OFFSET_WIDTH-1:0] word_offset;  // 字偏移
57      reg  [31:0]             cache_data;  // Cache数据
58      reg  [31:0]             mem_data;    // 内存数据
59      wire [31:0]             dirty_mem_addr; // 通过读出的tag
            和对应的index，偏移等得到脏块对应的内存地址并写回到
            正确的位置
60      wire valid;  // Cache有效位
61      wire dirty;  // Cache脏位.
62      reg  w_valid;  // Cache写有效位
63      reg  w_dirty;  // Cache写脏位
64      wire hit;    // Cache命中
65
66      // Cache相关控制信号
67      reg addr_buf_we;  // 请求地址缓存写使能
68      reg ret_buf_we;   // 返回数据缓存写使能
69      reg [WAY_NUM-1:0]data_we;       // Cache写使能
70      reg [WAY_NUM-1:0]tag_we;        // Cache标记写使能
71      reg data_from_mem;  // 从内存读取数据
72      reg refill;        // 标记需要重新填充，在MISS状态下接受
            到内存数据后置1,在IDLE状态下进行填充后置0
73
74      // 状态机信号
75      localparam
76          IDLE      = 3'd0,  // 空闲状态
77          READ      = 3'd1,  // 读状态
78          MISS      = 3'd2,  // 缺失时等待主存读出新块
79          WRITE     = 3'd3,  // 写状态
80          W_DIRTY   = 3'd4;  // 写缺失时等待主存写入脏块
81      reg [2:0] CS;  // 状态机当前状态
82      reg [2:0] NS;  // 状态机下一状态
83
84      // 状态机
85      always @(posedge clk or negedge rstn) begin
```

```
86          if (!rstn) begin
87              CS <= IDLE;
88          end else begin
89              CS <= NS;
90          end
91      end
92
93      // 伪随机数
94      reg [N-1:0] random;
95      initial begin
96          random=0;
97      end
98      always @(posedge clk) begin
99          random<=random+1;
100     end
101
102     // 中间寄存器保留初始的请求地址和写数据，可以理解为
            addr_buf中的地址为当前Cache正在处理的请求地址，而
            addr中的地址为新的请求地址
103     always @(posedge clk or negedge rstn) begin
104         if (!rstn) begin
105             addr_buf <= 0;
106             ret_buf <= 0;
107             w_data_buf <= 0;
108             op_buf <= 0;
109             refill <= 0;
110         end else begin
111             if (addr_buf_we) begin
112                 addr_buf <= addr;
113                 w_data_buf <= w_data;
114                 op_buf <= w_req;
115             end
116             if (ret_buf_we) begin
```

```
117                 ret_buf <= mem_r_data;
118             end
119             if (CS == MISS && mem_ready) begin
120                 refill <= 1;
121             end
122             if (CS == IDLE) begin
123                 refill <= 0;
124             end
125         end
126     end
127
128     // 对输入地址进行解码
129     assign r_index = addr[INDEX_WIDTH+LINE_OFFSET_WIDTH+
            SPACE_OFFSET - 1: LINE_OFFSET_WIDTH+SPACE_OFFSET];
130     assign w_index = addr_buf[INDEX_WIDTH+LINE_OFFSET_WIDTH
            +SPACE_OFFSET - 1: LINE_OFFSET_WIDTH+SPACE_OFFSET];
131     assign tag = addr_buf[31:INDEX_WIDTH+LINE_OFFSET_WIDTH+
            SPACE_OFFSET];
132     assign word_offset = addr_buf[LINE_OFFSET_WIDTH+
            SPACE_OFFSET-1:SPACE_OFFSET];
133
134     // 脏块地址计算
135     assign dirty_mem_addr = {r_tag, w_index}<<(
            LINE_OFFSET_WIDTH+SPACE_OFFSET);
136
137     // 写回地址、数据寄存器
138     reg [31:0] dirty_mem_addr_buf;
139     reg [127:0] dirty_mem_data_buf;
140     always @(posedge clk or negedge rstn) begin
141         if (!rstn) begin
142             dirty_mem_addr_buf <= 0;
143             dirty_mem_data_buf <= 0;
144         end else begin
```

```
145             if (CS == READ || CS == WRITE) begin
146                 dirty_mem_addr_buf <= dirty_mem_addr;
147                 dirty_mem_data_buf <= r_line;
148             end
149         end
150     end
151 
152 
153     wire valid_[WAY_NUM-1:0];  // Cache有效位
154     wire dirty_[WAY_NUM-1:0];  // Cache脏位.
155     reg  w_valid_[WAY_NUM-1:0];  // Cache写有效位
156     reg  w_dirty_[WAY_NUM-1:0];  // Cache写脏位
157     wire hit_[WAY_NUM-1:0];    // Cache命中
158     wire [TAG_WIDTH-1:0] r_tag_[WAY_NUM-1:0];
159     wire [LINE_WIDTH-1:0] w_line_[WAY_NUM-1:0];
160     wire [LINE_WIDTH-1:0] r_line_[WAY_NUM-1:0];
161 
162     generate
163         genvar i;
164         for (i = 0; i<WAY_NUM; i=i+1) begin:Bram
165             // Tag Bram
166             bram #(
167                 .ADDR_WIDTH(INDEX_WIDTH),
168                 .DATA_WIDTH(TAG_WIDTH + 2) // 最高位为有效
                        位, 次高位为脏位, 低位为标记位
169             ) tag_bram(
170                 .clk(clk),
171                 .raddr(r_index),
172                 .waddr(w_index),
173                 .din({w_valid,w_dirty, tag}),
174                 .we(tag_we[i]),
175                 .dout({valid_[i],dirty_[i], r_tag_[i]})
176             );
```

```
177             // Data Bram
178             bram #(
179                 .ADDR_WIDTH(INDEX_WIDTH),
180                 .DATA_WIDTH(LINE_WIDTH)
181             ) data_bram(
182                 .clk(clk),
183                 .raddr(r_index),
184                 .waddr(w_index),
185                 .din(w_line),
186                 .we(data_we[i]),
187                 .dout(r_line_[i])
188             );
189         end
190     endgenerate
191
192
193
194     // 判定Cache是否命中
195     reg [N-1:0] j,tem_j;
196     reg flag,tem;
197     integer i1;
198     initial begin
199         flag=0;
200         tem=0;
201         j=6'b000000;
202         tem_j=6'b000000;
203     end
204     always @(*) begin
205         if(CS==READ || CS==WRITE)begin
206             flag=0;
207             tem=0;
208             j=6'b000000;
209             tem_j=6'b000000;
```

```
210             for (i1 = 0;i1<WAY_NUM ; i1=i1+1) begin
211                 tem=flag;
212                 flag=tem || (r_tag_[i1] == tag && valid_[i1
                        ]);
213                 if(r_tag_[i1] == tag && valid_[i1])begin
214                     j=i1;
215                 end
216             end
217             if(!flag)begin
218                 j=random;
219             end
220         end
221 
222     end
223     assign r_tag=r_tag_[j];
224     assign dirty=dirty_[j];
225     assign r_line=r_line_[j];
226     assign hit = flag;
227 
228     // 写入Cache 这里要判断是命中后写入还是未命中后写入
229     assign w_line_mask = 32'hFFFFFFFF << (word_offset*32);
            // 写入数据掩码
230     assign w_data_line = w_data_buf << (word_offset*32);
            // 写入数据移位
231     assign w_line = (CS == IDLE && op_buf) ? ret_buf & ~
        w_line_mask | w_data_line : // 写入未命中, 需要将内
        存数据与写入数据合并
232                     (CS == IDLE) ? ret_buf : // 读取未命中
233                     r_line & ~w_line_mask | w_data_line; //
                            写入命中,需要对读取的数据与写入的数
                        据进行合并
234 
235     // 选择输出数据 从Cache或者从内存 这里的选择与行大小有
```

```
    关，因此如果你调整了行偏移位宽，这里也需要调整
always @(*) begin
    case (word_offset)
        0: begin
            cache_data = r_line[31:0];
            mem_data = ret_buf[31:0];
        end
        1: begin
            cache_data = r_line[63:32];
            mem_data = ret_buf[63:32];
        end
        2: begin
            cache_data = r_line[95:64];
            mem_data = ret_buf[95:64];
        end
        3: begin
            cache_data = r_line[127:96];
            mem_data = ret_buf[127:96];
        end
        default: begin
            cache_data = 0;
            mem_data = 0;
        end
    endcase
end

assign r_data = data_from_mem ? mem_data : hit ?
   cache_data : 0;

// 状态机更新逻辑
always @(*) begin
    case(CS)
        IDLE: begin
```

```
267                 if (r_req) begin
268                     NS = READ;
269                 end else if (w_req) begin
270                     NS = WRITE;
271                 end else begin
272                     NS = IDLE;
273                 end
274             end
275             READ: begin
276                 if (miss&& !dirty) begin
277                     NS = MISS;
278                 end else if (miss && dirty) begin
279                     NS = W_DIRTY;
280                 end else if (r_req) begin
281                     NS = READ;
282                 end else if (w_req) begin
283                     NS = WRITE;
284                 end else begin
285                     NS = IDLE;
286                 end
287             end
288             MISS: begin
289                 if (mem_ready) begin // 这里回到IDLE的原因
                        是为了延迟一周期，等待主存读出的新块写入
                        Cache中的对应位置
290                     NS = IDLE;
291                 end else begin
292                     NS = MISS;
293                 end
294             end
295             WRITE: begin
296                 if (miss && !dirty) begin
297                     NS = MISS;
```

```
298                 end else if (miss && dirty) begin
299                     NS = W_DIRTY;
300                 end else if (r_req) begin
301                     NS = READ;
302                 end else if (w_req) begin
303                     NS = WRITE;
304                 end else begin
305                     NS = IDLE;
306                 end
307             end
308             W_DIRTY: begin
309                 if (mem_ready) begin  // 写完脏块后回到MISS
                        状态等待主存读出新块
310                     NS = MISS;
311                 end else begin
312                     NS = W_DIRTY;
313                 end
314             end
315             default: begin
316                 NS = IDLE;
317             end
318         endcase
319     end
320     initial begin
321         p=0;
322     end
323     // 状态机控制信号
324     always @(*) begin
325         addr_buf_we   = 1'b0;
326         ret_buf_we    = 1'b0;
327         data_we       = 0;
328         tag_we        = 0;
329         w_valid       = 1'b0;
```

```
330         w_dirty       = 1'b0;
331         data_from_mem = 1'b0;
332         miss          = 1'b0;
333         mem_r         = 1'b0;
334         mem_w         = 1'b0;
335         mem_addr      = 32'b0;
336         mem_w_data    = 0;
337         case(CS)
338             IDLE: begin
339                 addr_buf_we = 1'b1; // 请求地址缓存写使能
340                 miss = 1'b0;
341                 ret_buf_we = 1'b0;
342                 if(refill) begin
343                     data_from_mem = 1'b1;
344                     w_valid = 1'b1;
345                     w_dirty = 1'b0;
346                     data_we = 1'b1<<j;
347                     tag_we = 1'b1<<j;
348                     if (op_buf) begin // 写
349                         w_dirty = 1'b1;
350                     end
351                 end
352             end
353             READ: begin
354                 data_from_mem = 1'b0;
355                 if (hit) begin // 命中
356                     miss = 1'b0;
357                     addr_buf_we = 1'b1; // 请求地址缓存写使
                        能
358                 end else begin // 未命中
359                     miss = 1'b1;
360                     addr_buf_we = 1'b0;
361                     if (dirty) begin // 脏数据需要写回
```

```
362                     mem_w = 1'b1;
363                     mem_addr = dirty_mem_addr;
364                     mem_w_data = r_line; // 写回数据
365                 end
366             end
367         end
368         MISS: begin
369             miss = 1'b1;
370             mem_r = 1'b1;
371             mem_addr = addr_buf;
372             if (mem_ready) begin
373                 mem_r = 1'b0;
374                 ret_buf_we = 1'b1;
375             end
376         end
377         WRITE: begin
378             data_from_mem = 1'b0;
379             if (hit) begin // 命中
380                 miss = 1'b0;
381                 addr_buf_we = 1'b1; // 请求地址缓存写使
                        能
382                 w_valid = 1'b1;
383                 w_dirty = 1'b1;
384                 data_we = 1'b1<<j;
385                 tag_we = 1'b1<<j;
386             end else begin // 未命中
387                 miss = 1'b1;
388                 addr_buf_we = 1'b0;
389                 if (dirty) begin // 脏数据需要写回
390                     mem_w = 1'b1;
391                     mem_addr = dirty_mem_addr;
392                     mem_w_data = r_line; // 写回数据
393                 end
```

```
394                 end
395             end
396             W_DIRTY: begin
397                 miss = 1'b1;
398                 mem_w = 1'b1;
399                 mem_addr = dirty_mem_addr_buf;
400                 mem_w_data = dirty_mem_data_buf;
401                 if (mem_ready) begin
402                     mem_w = 1'b0;
403                 end
404             end
405             default:;
406         endcase
407     end
408 
409 endmodule
```

3 仿真结果与分析

3.1 N 路组相连 Cache(LRU)

图 1: 2 路组相连，采用伪 LRU 替换策略

图 2: 4 路组相连，采用伪 LRU 替换策略

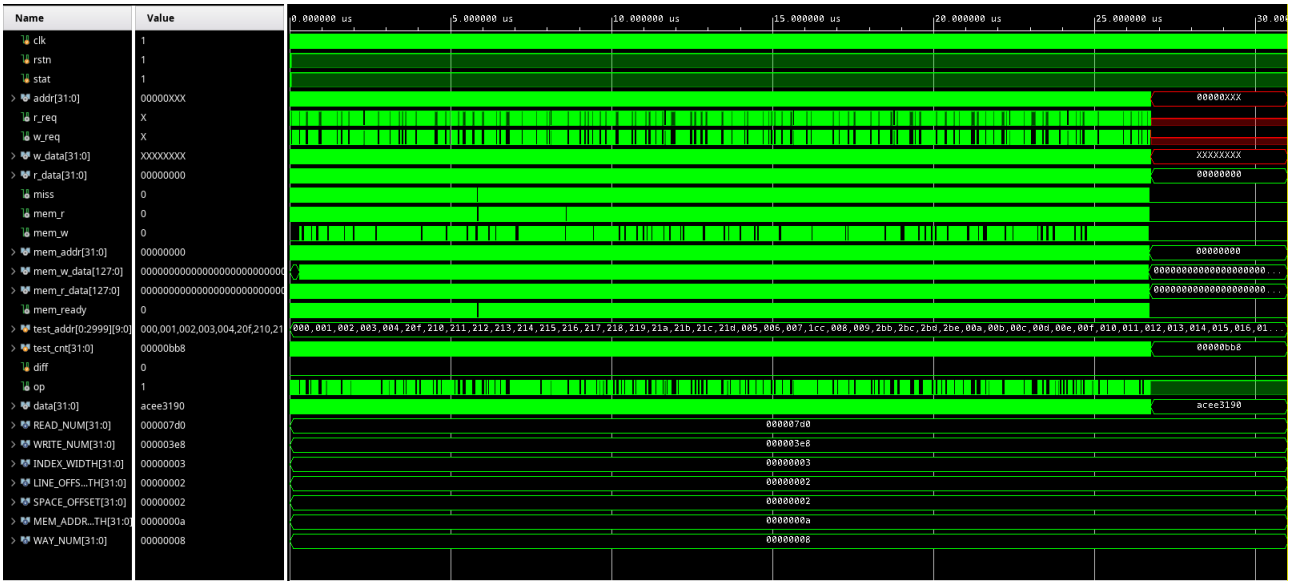


图 3: 8 路组相连，采用伪 LRU 替换策略

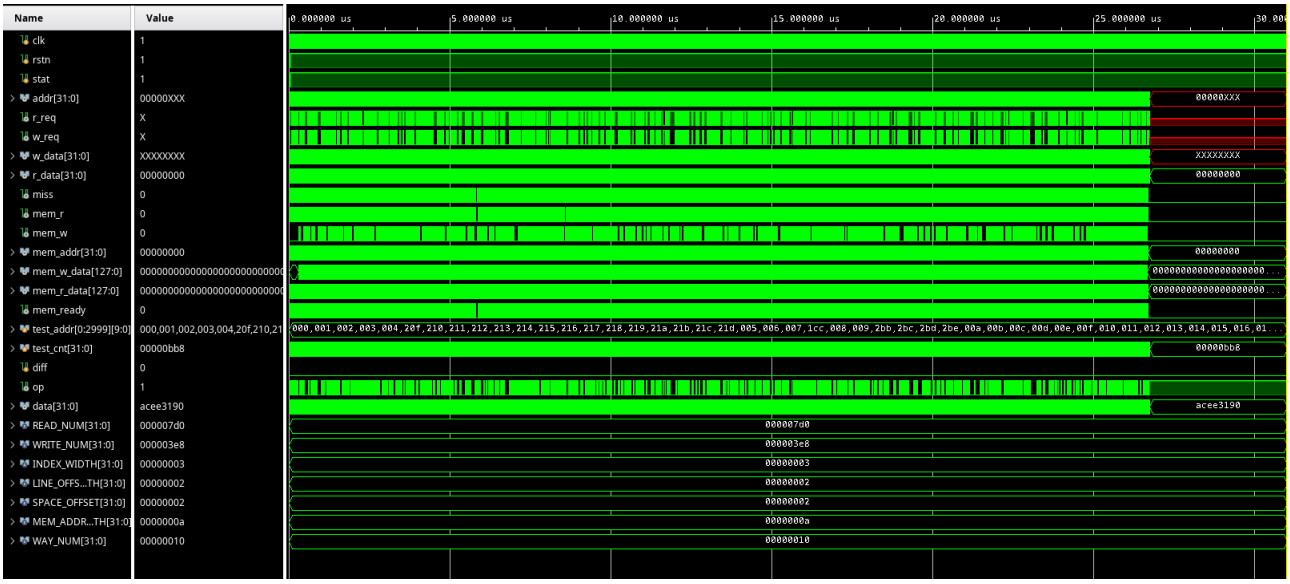


图 4: 16 路组相连，采用伪 LRU 替换策略

3.2 其他替换策略 (FIFO, 伪随机)

3.2.1 FIFO

图 5: 2 路组相连，采用 FIFO 替换策略

3.2.2 伪随机

图 6: 2 路组相连，采用伪随机替换策略

4 测试结果与分析

从仿真中可以看到测试结果均正确 (diff 信号始终为 0)

5 总结

本次实验实现了高速缓存 Cache，采用的替换测略包括伪 LRU(N 路组相联) ，FIFO(2 路组相连) 和伪随机 (2 路组相连)。