Lab 7：高速缓存 Cache

> 注：本次实验在原有代码框架上实现，读写的时序逻辑不变

一、参数化组相联 cache 实现

```
    // 参数化产生cache的各个分路，每个分路有一个Tag Bram和一个Data Bram
    genvar i;
    generate
        for (i = 0; i < WAY_NUM; i = i + 1) begin : way_gen
            // Tag Bram
            bram #(
                .ADDR_WIDTH(INDEX_WIDTH),
                .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_way[i]),
                .dout({valid[i], dirty_way[i], r_tag_way[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_way[i]),
                .dout(r_line_way[i])
            );

            // 分路命中判断：该分路的有效位为1且标签匹配
            assign hit_way[i] = valid[i] && (r_tag_way[i] == tag);
            // 分路写使能：总写使能信号有效时，如果命中，选择命中路；如果未命中，则选择替换路
            assign tag_we_way[i] = tag_we && (hit ? hit_way[i] : (replace_way == i));
            assign data_we_way[i] = data_we && (hit ? hit_way[i] : (replace_way == i));
        end
    endgenerate

    // 总命中信号，只在读或写状态下判断命中
    assign hit = (|hit_way) && (CS == READ || CS == WRITE);
```

```
    // 命中路索引：选择hit_way中为1的分路索引
    reg [WAY_WIDTH-1:0] hit_way_idx;
    always @(*) begin
        hit_way_idx = 0;
        for (integer i = 0; i < WAY_NUM; i = i + 1) begin
            if (hit_way[i]) begin
                hit_way_idx = i;
            end
        end
    end

    assign dirty = dirty_way[replace_way]; // 如果替换路是脏的，则dirty为1

    // 选择命中路数据
    assign r_line = hit ? r_line_way[hit_way_idx] : // 命中时选择对应路的数据
                          r_line_way[replace_way];  // 未命中时选择替换路的数据

    // 选择命中路标签
    assign r_tag = hit ? r_tag_way[hit_way_idx] : // 命中时选择对应路的标签
                         r_tag_way[replace_way];  // 未命中时选择替换路的标签
```

- generate 部分实现__参数化__组相联 cache，每个分路有一个 Tag Bram 和一个 Data Bram
- hit 的逻辑为：只要有一条分路在 READ 和 WRITE 期间命中即命中
- 通过只让目标路的写使能信号为 1，来写入相应分路
- dirty 只需看替换路是否脏
- 读标签 r_tag 和读数据 r_line 在命中时选择命中路，在未命中时选择替换路

二、不同替换策略的实现

1. LRU

```
    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            // 重置LRU状态
            for (integer j = 0; j < SET_NUM; j = j + 1) begin
                for (integer k = 0; k < WAY_NUM; k = k + 1) begin
                    lru_state[j][k] <= 0;
                end
            end
        end else if (hit) begin
            for (integer i = 0; i < WAY_NUM; i = i + 1) begin
                if (i == hit_way_idx) begin
                    lru_state[w_index][i] <= WAY_NUM - 1;
                end else begin
                    if (lru_state[w_index][i] > lru_state[w_index][hit_way_idx])
begin
                        lru_state[w_index][i] <= lru_state[w_index][i] - 1; // LRU
状态减1
                    end
                end
```

```
            end
        end else if (refill) begin
            for (integer i = 0; i < WAY_NUM; i = i + 1) begin
                if (i == replace_way) begin
                    lru_state[w_index][i] <= WAY_NUM - 1; // 替换路状态置为最大值
                end else begin
                    if (lru_state[w_index][i] > lru_state[w_index][replace_way])
begin
                        lru_state[w_index][i] <= lru_state[w_index][i] - 1; // LRU
状态减1
                    end
                end
            end
        end
    end

    // 替换路索引
    always @(*) begin
        replace_way = 0;
        for (integer i = 1; i < WAY_NUM; i = i + 1) begin
            if (lru_state[w_index][i] == 0) begin
                replace_way = i;
            end
        end
    end
```

- LUR 状态数组 `lru_state [SET_NUM-1:0][WAY_NUM-1:0]` 记录每组每条分路的最近访问顺序，越高代表最近访问时间越短
- 该数组初始化为 0
- 当某一条分路被访问（`hit` 或 `refill`）时，将该条分路对应的数据置最大值，并把状态数据大于该分路旧值的分路数据减一（可以理解为把在他前面的挤到后面了，而后面的本来就在后面，所以不变）
- 状态数据为 0 的分路，即为最长时间未访问分路

2. FIFO

```
    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            for (integer j = 0; j < SET_NUM; j = j + 1) begin
                fifo_state[j] <= 0; // FIFO状态初始化为0
            end
        end else if (refill) begin
            if (fifo_state[w_index] == WAY_NUM - 1) begin
                fifo_state[w_index] <= 0; // 如果FIFO状态已满，则重置为0
            end else begin
                fifo_state[w_index] <= fifo_state[w_index] + 1; // FIFO状态递增
            end
        end
    end

    // 替换路索引
    always @(*) begin
```

```
        replace_way = fifo_state[w_index];
    end
```

- 分路内访问顺序按从小到大循环替换，即实现了 FIFO 替换策略

3. 随机

```
    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            replace_way <= 0;
        end else if (refill) begin
            if (replace_way < WAY_NUM - 1) begin
                replace_way <= replace_way + 1; // 替换路索引递增
            end else begin
                replace_way <= 0; // 循环替换
            end
        end
    end
```

- 类似于一个计时器，让替换路一直循环
- 与 FIFO 不同的是，前者所有组共享 `replace_way`，而后者是在组内循环