好的，您提供的这段 LSQ (Load-Store Queue) Verilog 代码中，`token\_list` 的生成逻辑确实存在不可综合的风险。主要问题在于 `always @(\*)` 块中使用了带有非常量索引（`k`）的 `for` 循环来对 `reg` 类型变量进行位选择赋值，这是综合工具通常不支持的。

问题根源：

1. \*\*循环和动态索引\*\*：`for (j = 0; j < 8; j = j + 1)` 循环体内的 `k` 是一个运行时计算出的变量 (`k = (j + ptr\_old >= 8)? j + ptr\_old - 8 : j + ptr\_old;`)。综合工具很难将这种行为转换成固定的、并行的硬件门电路，因为它看起来像是在描述一个串行的、依赖于前一状态（`token\_stop`）的过程。

2. \*\*`token\_stop` 锁存行为\*\*：`token\_stop` 的逻辑（一旦置为 1 就停止后续 `token\_list` 的生成）在循环中引入了状态依赖，这使得并行化变得困难，容易导致生成锁存器（latch）或不正确的逻辑。

### 更优的可综合写法

我们可以将生成 `token\_list` 的逻辑修改为纯组合逻辑，并且避免使用循环。核心思想是为队列中的每一个条目（entry）并行地生成一个“遮罩”（mask），然后根据这个遮罩来决定是否允许其被唤醒。

这里的逻辑是：

\* \*\*Load 指令 (`mode\_list[i] == 1`)\*\*：总是可以被乱序唤醒。

\* \*\*Store 指令 (`mode\_list[i] == 0`)\*\*：只有当它是队列中最老的（`i == ptr\_old`）那条指令时，才可以被唤醒。任何在它之后的指令（无论是 Load 还是 Store）都不能在本周期被唤醒，以保证内存顺序。

我们可以用一个一位的信号 `store\_encountered` 来表示是否已经遇到了一个 Store 指令。

以下是针对 `token\_list` 生成逻辑的优化写法，您可以直接替换掉原来那段 `always @(\*)` 块：

```verilog

// ##################################################################

// ## 可综合的 token\_list 生成逻辑 ##

// ##################################################################

//

// 替换原有 always 块。

// 这段逻辑用来标记哪些条目可以被唤醒和发射。

// 规则是：

// 1. Store指令 (mode\_list[i] == 0) 必须按序执行。只有最老 (ptr\_old) 的那条 Store 可以被考虑。

// 2. Load指令 (mode\_list[i] == 1) 可以在它前面的所有 Store 地址都计算完成后乱序执行。

// 3. 一旦遇到一条有效的 Store 指令，它之后的所有指令（不论 Load/Store）都不能在本周期被唤醒，

// 以防止顺序冲突。

wire [7:0] is\_store; // 标记每个位置是否为 Store 指令

wire [7:0] store\_block\_mask; // 标记从每个 Store 开始的阻塞范围

wire [7:0] final\_block\_mask; // 合并所有 Store 造成的阻塞

genvar i;

generate

for (i = 0; i < 8; i = i + 1) begin : gen\_token

// 判断每个槽位是否为有效的Store指令

assign is\_store[i] = valid\_op\_list[i] & ~mode\_list[i];

// 如果当前槽位 i 是一个 Store，它会阻塞所有比它“年轻”的指令。

// 我们通过 ptr\_old 来判断指令的“年龄”。

// '((1 << 8) - 1)' 是 8'b1111\_1111

// '<< (i - ptr\_old)' 完成循环移位，阻塞所有比 i 年轻的指令。

// 注意：这里需要谨慎处理i和ptr\_old的回绕（wrap-around）计算。

// 一个更清晰的方式是：

wire [7:0] current\_block;

integer j;

always\_comb begin

for (j = 0; j < 8; j = j + 1) begin

// 计算 j 相对于 ptr\_old 的年龄

logic [2:0] age\_j = j - ptr\_old;

// 计算 i 相对于 ptr\_old 的年龄

logic [2:0] age\_i = i - ptr\_old;

// 如果 j 比 i 年轻或一样大，且 i 是一个 store，则 j 被阻塞。

// 同时，i 自己也被阻塞，除非 i 就是最老的指令 (i == ptr\_old)

if (is\_store[i] && age\_j > age\_i) begin

current\_block[j] = 1'b1;

end else begin

current\_block[j] = 1'b0;

end

end

// 如果 i 是一个 store，且不是最老的那个，那么它自己也阻塞自己

if (is\_store[i] && i != ptr\_old) begin

current\_block[i] = 1'b1;

end

end

assign store\_block\_mask[i] = |current\_block; // any\_of\_current\_block

end

endgenerate

// 使用 generate block 创建一个组合逻辑来生成 store\_block\_mask

// 这个实现比上面的 always\_comb 更清晰且对综合工具更友好

wire [7:0] block\_vectors[7:0];

generate

for (i = 0; i < 8; i = i + 1) begin: BlockGen

always\_comb begin

for (int j = 0; j < 8; j++) begin

logic is\_younger;

logic is\_same\_or\_younger;

// 环形队列中的年龄比较

is\_younger = ((j-ptr\_old) > (i-ptr\_old));

// is\_same\_or\_younger = ((j-ptr\_old) >= (i-ptr\_old));

// 如果 i 是一个有效的store，并且 j 比 i 年轻，那么 j 就被阻塞

// 如果 i 不是最老的store，它自己也阻塞自己

block\_vectors[i][j] = is\_store[i] && (is\_younger || (i != ptr\_old));

end

end

end

endgenerate

// final\_block\_mask 的每一位表示该位置是否被任何一个store阻塞

assign final\_block\_mask[0] = block\_vectors[0][0] | block\_vectors[1][0] | block\_vectors[2][0] | block\_vectors[3][0] | block\_vectors[4][0] | block\_vectors[5][0] | block\_vectors[6][0] | block\_vectors[7][0];

assign final\_block\_mask[1] = block\_vectors[0][1] | block\_vectors[1][1] | block\_vectors[2][1] | block\_vectors[3][1] | block\_vectors[4][1] | block\_vectors[5][1] | block\_vectors[6][1] | block\_vectors[7][1];

assign final\_block\_mask[2] = block\_vectors[0][2] | block\_vectors[1][2] | block\_vectors[2][2] | block\_vectors[3][2] | block\_vectors[4][2] | block\_vectors[5][2] | block\_vectors[6][2] | block\_vectors[7][2];

assign final\_block\_mask[3] = block\_vectors[0][3] | block\_vectors[1][3] | block\_vectors[2][3] | block\_vectors[3][3] | block\_vectors[4][3] | block\_vectors[5][3] | block\_vectors[6][3] | block\_vectors[7][3];

assign final\_block\_mask[4] = block\_vectors[0][4] | block\_vectors[1][4] | block\_vectors[2][4] | block\_vectors[3][4] | block\_vectors[4][4] | block\_vectors[5][4] | block\_vectors[6][4] | block\_vectors[7][4];

assign final\_block\_mask[5] = block\_vectors[0][5] | block\_vectors[1][5] | block\_vectors[2][5] | block\_vectors[3][5] | block\_vectors[4][5] | block\_vectors[5][5] | block\_vectors[6][5] | block\_vectors[7][5];

assign final\_block\_mask[6] = block\_vectors[0][6] | block\_vectors[1][6] | block\_vectors[2][6] | block\_vectors[3][6] | block\_vectors[4][6] | block\_vectors[5][6] | block\_vectors[6][6] | block\_vectors[7][6];

assign final\_block\_mask[7] = block\_vectors[0][7] | block\_vectors[1][7] | block\_vectors[2][7] | block\_vectors[3][7] | block\_vectors[4][7] | block\_vectors[5][7] | block\_vectors[6][7] | block\_vectors[7][7];

// 最终的 token\_list 是所有有效指令中没有被阻塞的那些

// `~final\_block\_mask` 表示允许唤醒的位置

always @(\*) begin

token\_list = valid\_op\_list & ~final\_block\_mask;

end

// ptr\_awake 和 awake\_exist 的逻辑可以保持不变，因为它现在依赖于一个

// 可靠的、可综合的 token\_list。

// ...

```

### 改进版写法的核心逻辑解释：

1. \*\*并行判断 `is\_store`\*\*: 首先，我们用一个8位的 `wire` 叫做 `is\_store`，并行地判断出队列中哪一个位置是有效的Store指令。`is\_store[i] = valid\_op\_list[i] & ~mode\_list[i]`。这会立刻生成8个并行的比较电路。

2. \*\*生成阻塞向量 `block\_vectors`\*\*: 我们使用 `generate` 循环为每一个可能的Store位置（`i` 从0到7）生成一个8位的阻塞向量 `block\_vectors[i]`。

\* `block\_vectors[i][j]` 为1，代表“如果 `i` 是一个Store，那么 `j` 位置的指令将被阻塞”。

\* 阻塞的条件是：

\* `j` 比 `i` 更“年轻”（`is\_younger`）。年龄是通过与 `ptr\_old` 的环形距离来判断的。

\* `i` 本身也被阻塞，除非它是最老的指令（`i != ptr\_old`）。

\* 这个 `generate` 块会生成固定的、并行的逻辑，完全符合综合要求。

3. \*\*合并阻塞 `final\_block\_mask`\*\*: `final\_block\_mask` 的每一位 `j` 是所有 `block\_vectors[i][j]` 的“或”运算结果。也就是说，只要有任何一个有效的Store指令 `i` 判定要阻塞 `j`，`final\_block\_mask[j]` 就为1。这本质上是一组“或”门。

4. \*\*生成 `token\_list`\*\*: 最后，`token\_list` 的生成就非常简单了。一个指令可以被唤醒（`token\_list[i] = 1`）的条件是：它本身是有效的（`valid\_op\_list[i] = 1`）并且它没有被任何Store阻塞（`final\_block\_mask[i] = 0`）。这对应一个简单的按位与非门操作：`token\_list = valid\_op\_list & ~final\_block\_mask;`。

这种写法完全是并行的组合逻辑，不包含任何时序或锁存器逻辑，因此可以被综合工具高效地转换成门级电路，并且行为明确，易于调试和时序分析。