

变量 SV中，多用 logic. → 是 verilog 中 reg, wire 的加强型。

```
logic a;  
logic [3:0] b;  
logic [31:0][31:0] c;
```

always_comb → SV中的过程块，是 verilog 中 always @ (*) 的增强。

always_comb 用于描述复杂电路

```
always_comb begin  
    a = 1'b1;  
    b = a;          // 性质①：内部覆盖性  
    a = 1'b0;        同一变量，再次赋值后，值可被覆盖。  
    c = a;  
end
```

性质② 块的原子性。

不可中断的执行

always_comb 块内的所有语句在一个仿真时间步内连续执行，不会被其他事件（如时钟边沿、其他进程的触发）打断。例如：

```
systemverilog ^  
  
always_comb begin  
    a = b + c;    // 语句1  
    d = a * e;    // 语句2（依赖语句1的结果）  
end
```

- 当 b、c 或 e 变化时，整个块会被触发。
- 语句 1 和语句 2 会按顺序执行，且执行过程中不会有其他事件插入。

对外界的瞬间可见性

块内所有赋值完成后，结果才会同时对外可见。例如：

```
systemverilog ^
```

```
logic a, b, c, d;  
  
always_comb begin  
    a = b + c;  
    d = a * 2;  
end
```

- 当 b 或 c 变化时，a 和 d 的值会同时更新，外界不会观察到 a 已更新但 d 仍使用旧值的中间状态。

在 always 块中，不仅可以进行赋值操作，还可以使用完整的过程性控制语句（if, else, case, for, while...）。

条件语句 (if-else)

```
systemverilog ^
```

```
always_comb begin  
    if (sel == 1'b0)  
        out = a;  
    else  
        out = b;  
end
```

多路分支 (case)

```
systemverilog ^
```

```
always_comb begin  
    case (opcode)  
        ADD: result = a + b;  
        SUB: result = a - b;  
        default: result = '0;  
    endcase  
end
```

循环语句 (for, while)

```
systemverilog ^
```

```
// 使用 for 循环计算数组和  
always_comb begin  
    sum = '0;  
    for (int i = 0; i < ARRAY_SIZE; i++)  
        sum += array[i];  
end
```

过程连续赋值 (assign/deassign)

```
systemverilog ^
```

```
always_comb begin  
    if (enable)  
        assign out = in; // 临时赋值  
    else  
        deassign out;  
end
```

always-comb 中的特殊 case.

(1). unique case.

按顺序检查分支，遇到第一个匹配就执行，忽略后续分支。

(优先级编码器)

特性	普通 case	后序分支
分支重叠检查	不检查	
覆盖率要求	不要求覆盖所有可能值 → 按顺序检测存在优先级，适合状态机	
综合工具行为	生成优先级编码器 (如第一个匹配优先)	
仿真错误条件	无	

unique case

→ 代码相对 case 更加安全。

比较重要。

(并行多路选择器)

unique0 case

强制检查分支是否互斥

允许部分重叠，但至少一个匹配

要求覆盖所有可能值 (隐式或显式)

不要求覆盖所有可能值

生成并行多路选择器 (无优先级)

生成带优先级的多路选择器

若存在重叠分支或未覆盖所有值报错

若所有分支都不匹配则报错

- 所有分支必须互斥 (无重叠)。
- 必须覆盖所有可能值 (显式列出或通过 default)。

• 示例：

```
systemverilog ^
unique case (sel)
  2'b00: out = a;
  2'b01: out = b;
  2'b10: out = c;
  2'b11: out = d;
  // 无需 default，因为所有 4 种情况都被显式覆盖
endcase
```

(如枚举类)

→ 应用场景。

分支条件互斥且覆盖所有可能性。

```
systemverilog ^
typedef enum {ADD, SUB, MUL, DIV} op_e;
op_e op;

unique case (op)
  ADD: result = a + b;
  SUB: result = a - b;
  MUL: result = a * b;
  DIV: result = a / b;
endcase // 无需 default，因为枚举值已全部覆盖
```

定义枚举类变量。

(2) priority case.

在 SystemVerilog 中, priority case 是一种特殊的 case 语句, 用于显式声明**优先级编码逻辑**。与普通 case 和 unique case 相比, 它强调分支的执行顺序和优先级关系。以下是详细解释:

1. priority case 的核心特性

- **显式优先级**: 按代码顺序检查分支, 第一个匹配的分支立即执行, 后续分支被忽略。
- **允许分支重叠**: 多个分支可能匹配同一个值, 但只有最高优先级 (即最靠前) 的分支生效。
- **不强制覆盖所有值**: 无需 default 分支, 但未覆盖的值会导致输出保持不变 (可能隐含锁存器)。
- **综合提示**: 向工具明确指示生成优先级编码器, 而非并行多路选择器。

基本语法

```
systemverilog ^  
  
priority case (expression)  
    value1: statement1; // 最高优先级  
    value2: statement2;  
    // ...  
    valueN: statementN; // 最低优先级  
    default: default_statement; // 可选  
endcase
```

典型应用场景

```
systemverilog ^  
  
// 中断处理 (高优先级中断优先响应)  
priority case (1'b1) // 常使用 1'b1 结合位向量进行优先级比较  
    int_req[3]: service_int3(); // 最高优先级  
    int_req[2]: service_int2();  
    int_req[1]: service_int1();  
    int_req[0]: service_int0(); // 最低优先级  
    default: idle();  
endcase
```

systemverilog ^ priority case (1'b1) 的意思是什么？

```
priority case (1'b1) // 寻找第一个为 1 的条件
    condition1: action1; // 若 condition1 为 1, 执行 action1
    condition2: action2; // 否则检查 condition2
    // ...
endcase
```

等价.

systemverilog ^

```
if (request[3] == 1'b1)
    $display("处理最高优先级请求");
else if (request[2] == 1'b1)
    $display("处理次高优先级请求");
else if (request[1] == 1'b1)
    $display("处理中等优先级请求");
else if (request[0] == 1'b1)
    $display("处理最低优先级请求");
else
    $display("无请求");
```

条件语句. if, else.

在SV中:

必须覆盖所有可能情况

always_comb 要求所有输出信号在所有条件下都被赋值, 否则会生成锁存器 (Latch) :

```
systemverilog ^

// 错误: 未覆盖 !condition 情况, 会生成锁存器
always_comb begin
    if (condition)
        out = value; // 错误: 缺少 else 分支
    end

// 正确: 明确覆盖所有情况
always_comb begin
    if (condition)
        out = value;
    else
        out = '0; // 或其他默认值
    end
```

SV中的for 循环. (用法和C语言中无异).

systemverilog ^

```
parameter SIZE = 4;
logic [7:0] array[SIZE];
logic [10:0] sum; // 确保足够位宽避免溢出
```

```
always_comb begin
    sum = '0; // 初始化累加器
    for (int i = 0; i < SIZE; i++) begin
        sum += array[i]; // 计算数组元素和
    end
end
```

sum = '0 是一种 0 值初始化的语法.

在 verilog 中, i 需在 always 外用.

Integer i; 来定义.

在 SV 中, i 需直接在 for () 里用
int i; 来定义.

补充. 附录与 genvar i

用于 always-comb.
always块中.

仅可用于 generate 块中.

二者不可互通.

genvar 的应用

systemverilog ^

```
// 生成多个 FIFO 实例
parameter DEPTH = 4;
generate
    genvar i;
    for (i = 0; i < DEPTH; i++) begin : fifo_array
        fifo #(._WIDTH(8)) u_fifo (
            .clk(clk),
            .rst_n(rst_n),
            .data_in(data_in[i]),
            .data_out(data_out[i])
        );
    end
endgenerate
```

引用常数

迭代.

需要注意的是：

避免锁存器 (Latch)

`always_comb` 会强制检查所有条件下的赋值完整性，若未覆盖所有分支，会报错：

```
systemverilog ^

always_comb begin
    if (sel)
        out = a; // 错误：未覆盖 !sel 情况，会生成锁存器
    // SystemVerilog 会报错，要求添加 else 分支
end
```

块内的信号依赖不能形成闭环，否则会导致仿真无限循环：

```
systemverilog ^
```

```
always_comb begin
    a = b + 1; // 错误：与下一行形成循环依赖
    b = a - 1;
end
```

always_ff 用于描述触发器。

基本等同于 always @ (posedge clk) 的用法。

logic 变量可一位多位，类似 wire reg。

typedef

在 SystemVerilog 中，typedef 是用于创建自定义数据类型的关键字，它允许你为已有的数据类型定义一个新名称，从而提高代码的可读性、可维护性和重用性。

基本语法

```
systemverilog ^
```



```
typedef 已存在的数据类型 新类型名;
```

1. 简化基本数据类型

为常用的位宽定义别名，避免重复书写冗长的位宽声明：

```
systemverilog ^
```

已有数据类型

```
// 定义8位无符号整数类型  
typedef logic [7:0] uint8_t;  
  
// 定义32位有符号整数类型  
typedef int signed int32_t;
```

新变量名

```
// 使用自定义类型  
uint8_t data; // 等价于 logic [7:0] data;  
int32_t counter; // 等价于 int signed counter;
```

2. 定义复杂数据结构

为数组、结构体等复杂类型创建别名：

systemverilog ^

// 定义一个16元素的8位数组类型

```
typedef logic [7:0] byte_array_t [0:15];
```

byte_array_t my_array.

则 → logic [7:0] my_array [0:15]

// 定义结构体类型

```
typedef struct {  
    logic [31:0] address;  
    logic [7:0] data;  
    logic write_en;  
} transaction_t;
```

// 使用自定义结构体类型

```
transaction_t tx; // 声明一个事务类型变量
```

// type definition

```
typedef struct packed {  
    logic [3:0] alufunc;  
    logic mem_read;  
    logic mem_write;  
    logic rewrite;  
} control_t;  
  
// variable declaration  
control_t control;  
  
logic rewrite;  
assign rewrite = control[0]; // control.rewrite;
```

control [7:0] → rewrite.
(拆出来用。)
主要用这个。

3. 定义枚举类型

配合 enum 创建有意义的状态类型：

systemverilog ^

// 定义状态机状态类型

```
typedef enum logic [2:0] {  
    IDLE,  
    READ,  
    WRITE,  
    CHECK,  
    DONE  
} fsm_state_t;
```

→ 有5个状态。

↓ 用 [3:0].

同时，[2:0]也表明
各个状态的位置

要注意枚举类赋值也只能用枚举类

```
fsm_state_t current_state, next_state; // 使用自定义枚举类型
```

4. 定义接口和端口类型

在模块接口中标准化信号类型：

```
systemverilog ^

// 定义总线接口类型
typedef logic [15:0] bus_data_t;
typedef logic [2:0] bus_cmd_t;

module controller (
    input bus_data_t data_in,      // 使用自定义类型
    output bus_data_t data_out,
    input bus_cmd_t cmd
);
    // 模块逻辑...
endmodule
```

5. 参数化类型（高级用法）

结合 parameter 创建可配置的类型：

```
systemverilog ^

// 定义参数化的数组类型
typedef logic [DATA_WIDTH-1:0] data_array_t [0:DEPTH-1];

// 使用时指定参数
parameter DATA_WIDTH = 32;
parameter DEPTH = 1024;
data_array_t #(.DATA_WIDTH(DATA_WIDTH), .DEPTH(DEPTH)) memory;
```

```
typedef logic[31:0] word_t;  
typedef logic[5:0] entry_t;  
typedef entry_t[31:0] table_t;
```

- 第一行 `typedef logic[31:0] word_t;` 定义了一个新的数据类型 `word_t`, 它表示一个 32 位的 `logic` 类型变量, 可以用来代表一个 32 位的字数据。
- 第二行 `typedef logic[5:0] entry_t;` 定义了新的数据类型 `entry_t`, 它表示一个 6 位的 `logic` 类型变量。
- 第三行 `typedef entry_t[31:0] table_t;` 基于前面定义的 `entry_t` 类型, 进一步定义了新的数据类型 `table_t`。
。

`table_t` 类型相当于一个由 32 个元素组成的数组, 其中每个元素都是 `entry_t` 类型, 也就是每个元素都是一个 6 位的 `logic` 变量。从数据结构角度看, `table_t` 类似于一个有 32 个存储单元的“表格”, 每个存储单元可以存放一个 6 位的数据。

在 SystemVerilog 中, 方括号 [] 的含义取决于它的**位置和上下文**, 这也是你观察到 `[31:0]` 和 `[5:0]` 在此处含义不同的原因。具体来说:

1. 两种方括号的本质区别

- 变量位宽声明:** 当 `[MSB:LSB]` 直接跟在数据类型后 (如 `logic`、`int` 等), 表示**变量的位宽** (即一个变量包含多少个二进制位)。
 - 例: `logic [5:0] entry_t;` 中, `[5:0]` 表示 `entry_t` 是一个 **6 位宽的变量** (从 bit5 到 bit0, 共 6 位)。
- 数组维度声明:** 当 `[start:end]` 跟在**类型名称后** (而非基础数据类型后), 表示**数组的维度和元素数量** (即数组包含多少个元素)。
 - 例: `typedef entry_t [31:0] table_t;` 中, `[31:0]` 表示 `table_t` 是一个 **包含 32 个元素的数组** (从索引 31 到 0, 共 32 个元素), 每个元素的类型是 `entry_t` (6 位宽)。

2. 语法规则的直观区分

记住一个简单规律:

- “类型 + [位宽]”** → 定义一个 “具有 N 位宽的变量类型”

```
systemverilog ^  
logic [5:0] a; // a是一个6位宽的变量 (位宽声明)
```

- “已定义类型 + [范围]”** → 定义一个 “包含 M 个该类型元素的数组”

```
systemverilog ^  
entry_t [31:0] b; // b是一个数组, 包含32个entry_t类型的元素 (数组声明)
```

parameter 的用法，在 SV、V 中并无太大差异。

数据类型支持

Verilog

Verilog 中，`parameter` 只能定义整数类型的常量，包括十进制、十六进制、八进制和二进制表示的整数。

SystemVerilog

SystemVerilog 对 `parameter` 进行了扩展，除了整数类型，还支持定义实数、字符串、枚举类型、数组类型等。

预编译命令