SystemVerilog: A Short Introduction

让你开心的 HDL，开了又开

陈晟祺

2020 年 10 月 14 日

为什么需要 SystemVerilog?

Verilog 是个好语言，除了……

为什么需要 SystemVerilog?

Verilog 是个好语言，除了……

- `reg` 到底是不是寄存器?
- `always` 到底是组合逻辑还是时序逻辑?
- AXI 接口的连线这么多，好麻烦!
- 枚举量只能用 `localparam` 定义常量吗?
- `case` 究竟应该怎么写?

为什么需要 SystemVerilog?

Verilog 是个好语言，除了……

- `reg` 到底是不是寄存器?
- `always` 到底是组合逻辑还是时序逻辑?
- AXI 接口的连线这么多，好麻烦!
- 枚举量只能用 `localparam` 定义常量吗?
- `case` 究竟应该怎么写?

是时候升级到 SystemVerilog 了!

如何升级?

SystemVerilog 是 Verilog 的超集，因此只要：

- `.v` → `.sv`
- `.vh` → `.svh`

~~你就学会了 SystemVerilog!~~

组合逻辑与时序逻辑

SystemVerilog 中，一切都可以是 `logic`：

- 不必显式区分 `wire` 和 `reg`
- 根据用法（见下）自动推断为寄存器或组合逻辑
- 可以（几乎在）所有场合代替原有的 `wire` 和 `reg`

组合逻辑与时序逻辑

SystemVerilog 中，一切都可以是 `logic`：

- 不必显式区分 `wire` 和 `reg`
- 根据用法（见下）自动推断为寄存器或组合逻辑
- 可以（几乎在）所有场合代替原有的 `wire` 和 `reg`

并且有功能细分的 `always` 语句：

- `always_comb`：纯组合逻辑，无需“敏感信号”列表
- `always_ff`：时序逻辑，按照原有写法即可：`always_ff @(posedge clk)`
- `always_latch`：显式指出需要锁存器（本课程中没有此需求，因此**禁止**使用）

在代码无法满足功能描述时，综合器将报错退出，避免非预期的行为。

组合逻辑与时序逻辑（例子）

```
logic [3:0] counter, counter_next;
logic counter_wrap;

// combinational logic
assign counter_wrap = counter == 4'b1111;

// also combinational logic
always_comb begin
    counter_next = counter + 1;
end

// sequential logic
always_ff @(posedge clk) begin
    if (reset) begin
        counter <= 'b0;
    end else begin
        counter <= counter + 1;
    end
end
```

组合逻辑与时序逻辑（错误的例子）

```
logic not_latch, must_be_seq;

always_comb begin
    if (some_cond) begin
        not_latch = 1'b1;
    end
    // latch (no else) in always_comb -- synthesis error
end

always_ff @(clk) begin
    must_be_seq <= 1'b1;
    // no trigger edge in always_ff -- synthesis error
end
```

unique 与 priority

Verilog 的 `case` 语句语义比较复杂，需要注解提示综合器（参见：`full case`，`parallel case`），并且容易产生 latch。`if` 也可能会忘记书写分支，导致 latch，或者产生意料之外的多个命中。

SystemVerilog 中，新增了三个关键词 `unique`，`unique0`，`priority`，可以搭配 `case` 和 `if` 使用：

- `unique`：只可能命中一项。仿真时如果命中多个或者没有命中将产生**警告**。
- `unique0`：只可能命中至多一项（危险!）。仿真时如果命中多个将产生**警告**。
- `priority`：可能命中多项，此时语句书写顺序作为匹配优先级。仿真时如果没有命中将产生**警告**。

注意：上述几个关键词并**不会避免**产生 latch，推荐总是将组合逻辑放置在 `always_comb` 中让综合器进行检查。

unique 与 priority（例子）

注意：如果需要通配符 ?，请使用 casez，**不要使用** casex。此外，可综合的代码的 case 中**不应当**出现 X 和 Z。否则，可能出现非预期的结果。

```
// 4-bit priority decoder
// p: 4 bits input, d: 2 bits output
always_comb begin
    priority casez (p) begin
        4'b1???: {valid, d} = 3'b111;
        4'b01??: {valid, d} = 3'b110;
        4'b001?: {valid, d} = 3'b101;
        4'b0001: {valid, d} = 3'b100;
        default: {valid, d} = 3'b000;
    end
end
```

思考：上面的例子中能不能用 unique case? 能不能没有 default?

更多的数据类型

整数类型

`byte`, `shortint`, `int`, `longint`：不同宽度的整数（相当于 `logic[WID-1:0]`）

更多的数据类型

整数类型

byte, shortint, int, longint：不同宽度的整数（相当于 logic[WID-1:0]）

枚举量

```
typedef enum logic [2:0] {
    STATE_RESET, STATE_INIT, STATE_WORK, STATE_DONE
} state_t;

state_t state, next_state;
```

可以直接**在仿真中显示为名称**，妈妈再也不用担心我看不懂状态机了！

更多的数据类型（cont'd）

struct 和 union

```
typedef struct packed {
    union packed {
        ip4_hdr ip4; // defined as struct elsewhere
        arp_hdr arp; // defined as struct elsewhere
    } payload;
    logic [15:0] ether_type;
    logic [47:0] src_mac;
    logic [47:0] dst_mac;
} ether_hdr;
```

它们不香吗?

注意：必须带上 packed，表明不需要任何额外（填充）空间。

流操作符（Streaming Operators）

两个操作符：{>>{}} 和 {<<{}}，分别用于从两个方向（从左到右，从右到左）对某个比特流进行遍历和拼接，并支持指定块大小。

```
byte array[4] = '{ 8'h01, 8'h02, 8'h03, 8'h04 };
// pack every bit in array from left to right
int big_endian = {>>{array}}; // 0x01020304
// reverse byte order (pack every 8 bits from right to left)
int little_endian = {<<8{big_endian}}; // 0x04030201, or
int little_endian = {<<8{array}}; // 0x04030201
// reverse bit order (pack every bit from right to left)
int big_reverse = {<<{little_endian}}; // 0x8040C020
```

本实验中的最大用途：交换字节序，而无需手动赋值。

更多高级操作：https://www.amiq.com/consulting/2017/05/29/how-to-pack-data-using-systemverilog-streaming-operators/。

更方便的模块连线

同名信号无需写两遍名字，甚至可以一次性全部连接：

```
arp_cache #(
    .ITEM_COUNT(16)
) arp_cache_inst(
    .clk, // equals to .clk(clk)
    .rst,
    .* // connect all other ports to signals with the same name
)
```

更方便的模块连线

同名信号无需写两遍名字，甚至可以一次性全部连接：

```
arp_cache #(
    .ITEM_COUNT(16)
) arp_cache_inst(
    .clk, // equals to .clk(clk)
    .rst,
    .* // connect all other ports to signals with the same name
)
```

老师，能不能再给力一点？

进一步简化：结合 struct

```
typedef struct packed { // all from master to slave
    logic [31:0] araddr;
    ...
} axi_req_t;

typedef struct packed { // all from salve to master
    logic        arready;
    ...
} axi_resp_t;
```

进一步简化：结合 struct

```
typedef struct packed { // all from master to slave
    logic [31:0] araddr;
    ...
} axi_req_t;

typedef struct packed { // all from salve to master
    logic        arready;
    ...
} axi_resp_t;

dcache_pass #(
    .BUS_WIDTH(BUS_WIDTH)
) uncached_inst(
    .clk,
    .rst,
    .axi_req(uncached_axi_req),
    .axi_resp(uncached_axi_resp)
);
```

进一步简化：结合 struct

```
typedef struct packed { // all from master to slave
    logic [31:0] araddr;
    ...
} axi_req_t;

typedef struct packed { // all from salve to master
    logic        arready;
    ...
} axi_resp_t;

dcache_pass #(
    .BUS_WIDTH(BUS_WIDTH)
) uncached_inst(
    .clk,
    .rst,
    .axi_req(uncached_axi_req),
    .axi_resp(uncached_axi_resp)
);
```

但是好像还有点麻烦?

interface 的结构

一些 interface = 一些有方向的连线（modport）+ 其他辅助信号!

```
interface arp_cache_if(input clock_t clk); // clock_t
 ↪  encapsulates some different clocks
    logic read, write, done;
    mac_t phy_addr_req, phy_addr_resp;
    uint32_t ip_addr;

    modport master ( // used by master
        output read, write, phy_addr_req, ip_addr,
        input  done, phy_addr_resp
    );

    modport slave ( // used by slave
        input  read, write, phy_addr_req, ip_addr,
        output done, phy_addr_resp
    );
endinterface
```

interface 的使用

interface 定义:

```
module arp_cache(arp_cache_if.slave pipeline);
    assign pipeline.done = 1'b1;
    always_ff @(posedge pipeline.clk.clk_125M or negedge
    ↪ pipeline.clk.rst) begin
        if (pipeline.read) begin
            pipeline.phy_addr_resp <= 'b0;
            ...
        end
    end
endmodule
```

interface 的使用

interface 定义:

```
module arp_cache(arp_cache_if.slave pipeline);
    assign pipeline.done = 1'b1;
    always_ff @(posedge pipeline.clk.clk_125M or negedge
     ↪  pipeline.clk.rst) begin
        if (pipeline.read) begin
            pipeline.phy_addr_resp <= 'b0;
            ...
        end
    end
endmodule
```

interface 实例化与连线:

```
clock_t clk(.clk_125M, .clk_50M, .rst);
arp_cache_if arp_cache_if_inst(.clk);
pipeline pipeline_inst(.arp_cache(arp_cache_if_inst.master));
arp_cache arp_cache_inst(.pipeline(arp_cache_if_inst.slave));
```

其他

关于仿真的新功能

SystemVerilog 真正强大之处在于**验证**！（然而我也不懂）

- `string` 类型和增强的 `$display` 函数（回忆 `printf`）
- 动态数组 `int da[]`、关联数组（map）`int as[string]` 和队列 `int qa[$]`
- 基于类的面向对象编程
- 含约束的随机化 `rand`（可以在写 testbench 时尝试这一特性）

注意：上面所有内容都不应当出现在可综合代码中

其他

关于仿真的新功能

SystemVerilog 真正强大之处在于**验证**!（然而我也不懂）

- `string` 类型和增强的 `$display` 函数（回忆 `printf`）
- 动态数组 `int da[]`、关联数组（map）`int as[string]` 和队列 `int qa[$]`
- 基于类的面向对象编程
- 含约束的随机化 `rand`（可以在写 testbench 时尝试这一特性）

注意：上面所有内容都不应当出现在可综合代码中

其他值得一试的（非新）功能

- 宏和 `const` 常量
- 函数（`function`）用于复用小块组合逻辑（而无需写成独立模块）
- `generate if` 和 `generate for` 控制代码生成（不要复制粘贴!）

或许可以节约编写代码的时间

还有更多吗?

推荐使用 Chisel / SpinalHDL 生成 RTL，早日逃离苦海。

(详见第四周第七组报告)