Episode 3 Practical verilog format

——《你可能不知道的verilog提高班指南》By TA-马子睿

前言

近来，想必大家已经接触到了**"veribug"**这门神奇的语言，或许对于刚刚经历C语言洗礼的大家来说，用起来并不顺手，在格式规范方面特为尤甚。verilog作为一门硬件描述语言，一个模块动辄几百行代码，很容易由于代码风格导致严重的可读性问题。因此，非常希望大家能够通过本辑，将自己的verilog代码书写得更加规范。

> 注：如不作标注，本辑中代码来自2022第六届"龙芯杯"大赛中国科大"小步快跑"战队LoongarchCPU设计

目录

Vivado中的使用规范

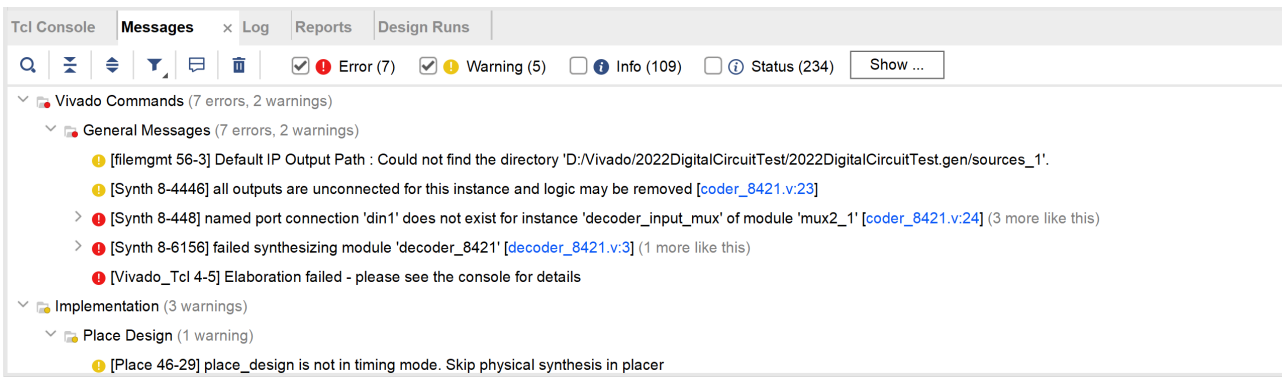
1. **一个.v文件只写一个module**

   > 这是一个很无理的要求，因为我写的一个模块只有不到10行！

   事实上，这个规范并不是针对大家本学期，而是针对大家下学期组成原理的实验。一个module虽然小，但如果我们把它单独写一个文件，我们就可以在vivado点击它的时候直接进入这个文件，这样可以使得观感非常好。另外，如果我们在一个文件中写多个模块，很容易导致找bug时在不同模块之间切换时容易晕。所以，建议大家还是分文件写模块吧！

2. **遇事不决RTL**

   看电路图是一个解决大部分问题的很好办法：仿真出错看RTL，综合出错看RTL……大家要看的不只是电路图，更是要结合出错步骤的warnings来分析，并看RTL下面Message中的warnings，这通常可以帮助大家解决很多难以发现的问题：



另外，对于warnings，大家每次使用分析综合时，记得按下上图中“垃圾桶”符号来清空错误信息（vivado不会自动清空）。

承继C语言的衣钵

verilog作为一种C语言风格的代码，自然也承继了C语言的一些风格要求：

1. **begin...end、括号中的代码要作一个Tab的缩进，end与开启begin...end的字符左对齐**，如：

```
//括号缩进
cache_memory way0(
    .addra  (w_index),//Tab 缩进
    .clka   (clk),
    .dina   (mem_din),
    .ena    (mem_en[0]),
    .wea    (mem_we),
    .addrb  (r_index),
    .doutb  (mem_dout0)
);      //分号不要忘记

// begin...end缩进
always @(*) begin
    case(wrt_data_sel)  // Tab缩进
    1'b0: mem_din = w_data_AXI;
    1'b1: begin
        case(wrt_type)  // Tab缩进
        BYTE: mem_din = {64{w_data_CPU[7:0]}};
        HALF: mem_din = {32{w_data_CPU[15:0]}};
        WORD: mem_din = {16{w_data_CPU}};
        default: mem_din = 0;
        endcase
    end // 与1'b1对齐
    endcase
end     // 与always对齐
```

2. **双目运算符两侧加空格**，如：

```
assign hit[3] = (tag == tag_3) && vld_3; // 赋值运算符
```

3. **除了要求的最外层的接口名称，内部名称应该具有意义，不要怕名字长**，如：

```
assign index = r_addr[11:6];
```

（索引号等于读操作地址的11位到6位）

例化风格

例化是一种很常见的操作，特别是对于优先状态机，例化的接口可能多达几十个，这时例化的风格就显得极为重要，我们通过例子来详细阐述例化的格式要求（**再次重申：不要使用顺序例化！！！！！！**）：

```
main_FSM_i main_FSM(
    .clk            (clk),
    .rstn           (rstn),
    // 把clk，rstn等功能简单的通用接口在这里用一个空行隔开
```

```
    .valid          (valid),
    .cache_hit      (cache_hit),
    .r_rdy_AXI      (r_rdy),
    ……
    .data_valid     (data_valid_oIzprAXodb8T),
    .cache_ready    (cache_ready), // 括号与下方最长接口名的括号对齐，可以用Tab实现
    // 来自不同模块组或实现功能不同的信号应当用空行分开
    // 上面是主要操作，下面是处理特殊指令和例外的接口

    .cacop_ready    (cacop_ready),
    .cacop_complete (cacop_complete), // 这是被例化模块最长的接口名，其他接口括号位置不应比它靠左
    ……
    .exp_sel        (exp_sel)
);
```

这个模块一共有42个接口，如果不采用这种方法，整个模块代码将会杂乱不堪。

位操作使用规范

verilog与C语言风格不同的是，verilog可以非常轻松地使用[n:m]来使用m-n位的数据，但是这里是非常容易出错的：

位的引用

- **声明变量时，索引域在前，索引域中大数在前：**

```
wire [5:0] r_index, w_index;
```

- **使用变量时，索引域在后，索引域中大数在前：**

```
assign w_index = w_addr[11:6];
```

(这种大数在前的方法，很符合“低位在右”的计算机设计理解)

以上所有的引用方法，都不考虑“寄存器堆”这种结构，这种结构将会在第五次实验时涉及，大家感兴趣可以自行了解。

模块声明和case的书写规范

之所以介绍位引用，是为了规范模块声明部分的书写规范：

- **关键字部分要和最长的output reg 对齐，短于它的要在右边补空格**
- **位索引域的左右括号要对齐**

```
module TLB_out(
    input       [ 1:0] ad_mode,
    input              s0_dmw_hit,
    input              s1_dmw_hit,
    input              s0_dmw_hit_obuf,
    input              s1_dmw_hit_obuf,
    input       [31:0] s0_addr,
    input       [31:0] s1_addr,
    input       [ 1:0] s0_tlb_mat,
    input       [ 1:0] s1_tlb_mat,
    input       [ 1:0] s0_dmw_mat,
```

```
    input       [ 1:0] s1_dmw_mat,
    input       [31:0] s0_dmw_paddr,
    input       [31:0] s1_dmw_paddr,
    input       [19:0] s0_pfn,
    input       [19:0] s1_pfn,
    input       [ 5:0] found_ps0,
    input       [ 5:0] found_ps1,
    output reg  [31:0] s0_paddr,
    output reg  [31:0] s1_paddr,
    output reg  [ 1:0] s0_mat,
    output reg  [ 1:0] s1_mat
    );
```

这样写的好处是，使用多点编辑操作同一列时非常方便（直接Alt+Shift）。同理，对于case语句，我们也应该对每种case的值进行对齐(当然，这里只要求将冒号前的部分对齐即可)

```
always @(*) begin
    case(addr_rbuf[5:2])
    4'd0:  r_data_mem = way_data[31:0];
    4'd1:  r_data_mem = way_data[63:32];
    4'd2:  r_data_mem = way_data[95:64];
    4'd3:  r_data_mem = way_data[127:96];
    4'd4:  r_data_mem = way_data[159:128];
    4'd5:  r_data_mem = way_data[191:160];
    4'd6:  r_data_mem = way_data[223:192];
    4'd7:  r_data_mem = way_data[255:224];
    4'd8:  r_data_mem = way_data[287:256];
    4'd9:  r_data_mem = way_data[319:288];
    4'd10: r_data_mem = way_data[351:320];
    4'd11: r_data_mem = way_data[383:352];
    4'd12: r_data_mem = way_data[415:384];
    4'd13: r_data_mem = way_data[447:416];
    4'd14: r_data_mem = way_data[479:448];
    4'd15: r_data_mem = way_data[511:480];
    endcase
end
```

位的拼接

verilog支持位拼接操作，但位拼接非常容易出现问题！

- **使用大括号拼接，如果希望把某一部分重复几次，则在大括号前写需要重复的数字：**

  ```
  assign mem_dout = {
      mem_dout3, mem_dout2, mem_dout1, mem_dout0
  };
  assign mem_we = {64{1'b1}};//把1位宽的1重复64次，拼成由1组成的64位数
  ```

- 再复习一下verilog中的数值描述方式

  > **二进制位宽 ' 数制 数字**

```
32'h10; //32二进制位宽的16进制数字0x10，即十进制下的16
32'd10; //32二进制位宽的10进制数字10，正是十进制下的10
32'b10; //32二进制位宽的2进制数字10，即十进制下的2
```

为什么前面一定是二进制位宽呢？是因为这样可以方便verilog编译器快速识别真实的位宽。

另外，**verilog中所有数字全部是“零拓展”而不是“位拓展”**：如32'b10描述的最高位（即第二位）是1，零拓展会将更高位全部补0，位拓展会将更高位补位你给出的最高位值（即第二位的1）

判断真假，不需要 == 0 和 == 1

在if条件判断中，不需要这种 == 1来判断真假

```
if(cacop_en_rbuf) AXI_we_temp = wrt_type;
```

是大家熟悉的C语言，哪怕这个信号位宽不是1，只要它的所有位不同时为0，那么就为真。

声明变量，注意“垃圾分类”

在最顶层的模块中，往往需要例化其他所有模块，这时会产生大量连线所用的wire型变量。这些变量杂乱无章，这里给大家一个建议；

- **位宽相同的放在一堆**
- **功能类似的可以放在同一行**

```
    wire op_rbuf, r_data_sel, wrt_data_sel, cache_hit, data_valid_temp, cache_ready_temp;
    wire fill_finish, way_sel_en, mbuf_we, dirty_data, dirty_data_mbuf;
    wire w_dirty_data, rbuf_we, wbuf_AXI_we, wbuf_AXI_reset, wrt_AXI_finish;
    wire pbuf_we, cacop_en_rbuf, is_atom_rbuf, llbit_rbuf, exp_sel;
    wire [3:0] mem_en, hit, way_replace, way_replace_mbuf, tagv_we, dirty_we,
write_type_rbuf, way_visit;
    wire [1:0] cacop_code_rbuf;
    wire [6:0] exception_cache, exception_temp, exception_obuf, exception_mbuf;
    wire [19:0] replace_tag, store_data;
    wire [31:0] r_data_CPU_temp, addr_rbuf, w_data_CPU_rbuf, addr_pbuf, w_addr_mbuf;
    wire [63:0] mem_we, mem_we_normal;
    wire [511:0] w_line_AXI, miss_sel_data, mem_din;
    wire [2047:0] mem_dout;
```

结语

本辑中我们介绍了verilog基础语法的一些规范，这些规范看起来十分冗余，却是大家读懂代码、改动代码的制胜法宝。希望大家能够在今后的实验中尽量使用这些规则，来更有效率地完成实验哦！