Example: Serial Bit Sequence Detector in Verilog – Full Model

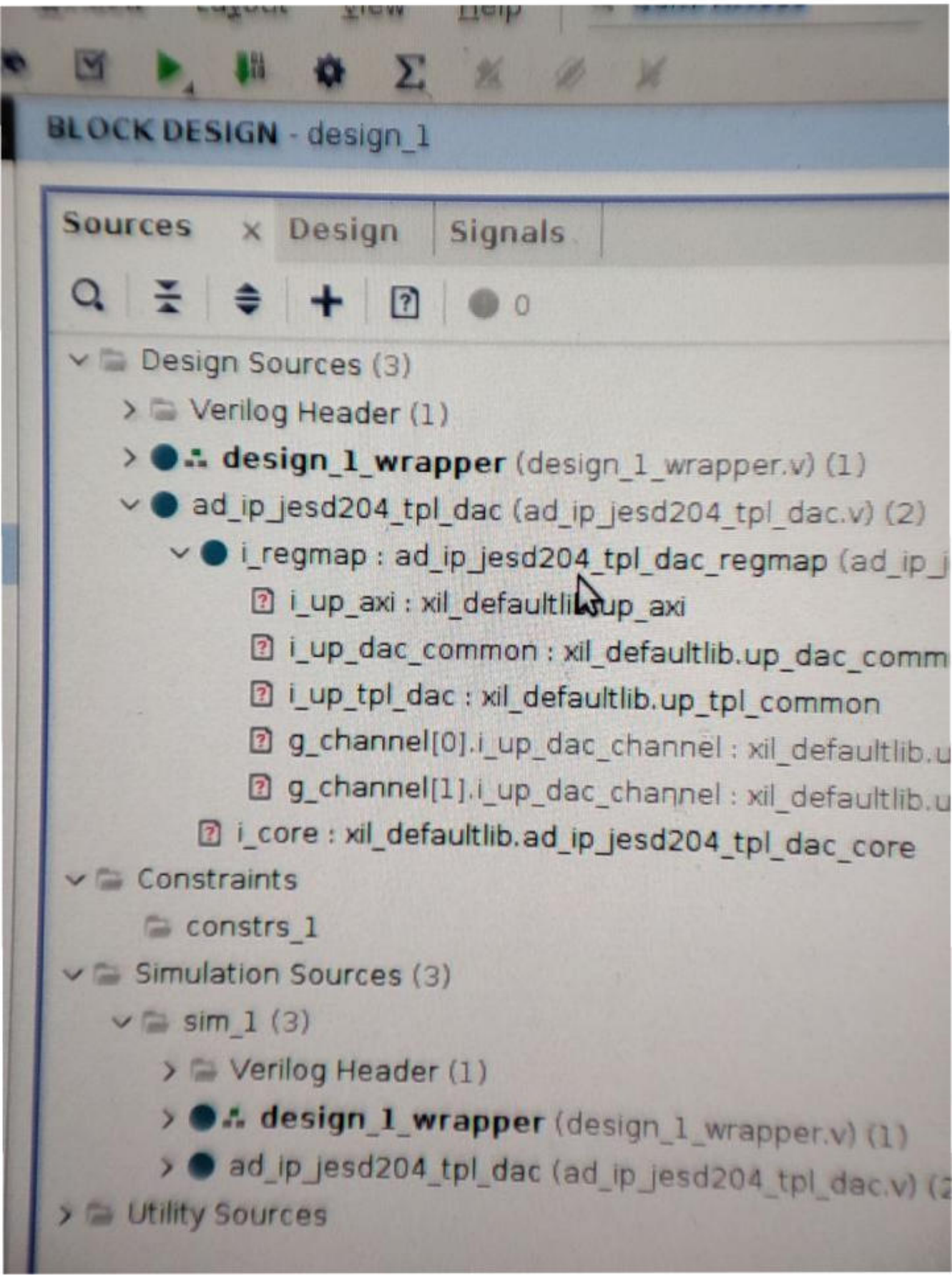
```
module Seq_Det (output reg  ERR,
                input  wire Clock, Reset, Din);

  reg [2:0] current_state, next_state;
  parameter Start    = 3'b000,
            D0_is_1  = 3'b001,
            D1_is_1  = 3'b010,
            D0_not_1 = 3'b011,
            D1_not_1 = 3'b100;

  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= Start;
      else
        current_state <= next_state;
    end

  always @ (current_state or Din)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        Start    : if (Din == 1'b1)
                     next_state = D0_is_1;
                   else
                     next_state = D0_not_1;
        D0_is_1  : if (Din == 1'b1)
                     next_state = D1_is_1;
                   else
                     next_state = D1_not_1;
        D1_is_1  : next_state = Start;
        D0_not_1 : next_state = D1_not_1;
        D1_not_1 : next_state = Start;
        default  : next_state = Start;
      endcase
    end

  always @ (current_state or Din)
    begin: OUTPUT_LOGIC
      case (current_state)
        D1_is_1  : if (Din == 1'b1)
                     ERR = 1'b1;
                   else
                     ERR = 1'b0;

        default  : ERR = 1'b0;
      endcase
  end

endmodule
```

Declaration of state variables and state encoding.

State memory block. This is sequential logic so non-blocking assignments are used. This block only makes assignments to the signal "current_state".

Next state logic block. This is combinational logic so blocking assignments are used. This block only makes assignments to the signal "next_state".

Output logic block. This is combinational logic so blocking assignments are used. Since there is only one condition where the output "ERR" is asserted, the default clause can be used for all other conditions.

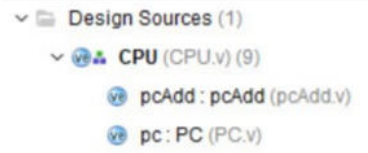

BLOCK DESIGN - design_1
Sources
Design
Signals
0
Design Sources (3)
Verilog Header (1)
design_1_wrapper (design_1_wrapper.v) (1)
ad_ip_jesd204_tpl_dac (ad_ip_jesd204_tpl_dac.v) (2)
i_regmap : ad_ip_jesd204_tpl_dac_regmap (ad_ip_j
i_up_axi : xil_defaultlib.up_axi
i_up_dac_common : xil_defaultlib.up_dac_comm
i_up_tpl_dac : xil_defaultlib.up_tpl_common
g_channel[0].i_up_dac_channel : xil_defaultlib.u
g_channel[1].i_up_dac_channel : xil_defaultlib.u
i_core : xil_defaultlib.ad_ip_jesd204_tpl_dac_core
Constraints
constrs_1
Simulation Sources (3)
sim_1 (3)
Verilog Header (1)
design_1_wrapper (design_1_wrapper.v) (1)
ad_ip_jesd204_tpl_dac (ad_ip_jesd204_tpl_dac.v) (2
Utility Sources

赛灵思中文社区论坛欢迎您 (Archived) — marxi (Customer) asked a question.
2020年12月17日 at 01:57

第一次用vivado，请问如何层次化添加子模块

在写一个CPU的项目，顶层设计要把子模块添加进来。但是在CPU顶层项目里添加子文件的话会搞到同一级里，就像这样

想问下怎么才能弄成下面这种树形的



以前用的quartus，第一次用vivado，不太懂这种层次级或者说调用子模块的情况下，子模块的文件怎么用这种树形的方式添加进来，希望有经验的同志给点拨一下，谢谢

开发工具

Like Answer Share 2 answers · 830 views

Top Rated Answers

graces (Employee)
3 years ago

BEST SOLUTION
顶层RTL文件里如果例化了子模块，例如CPU.v里面例化了ALU模块，添加文件后source窗口会自动更新成对应的树形结构。

Selected as Best · Like

All Answers

graces (Employee)
3 years ago

BEST SOLUTION
顶层RTL文件里如果例化了子模块，例如CPU.v里面例化了ALU模块，添加文件后source窗口会自动更新成对应的树形结构。

Selected as Best · Like · Reply

marxi (Customer)
3 years ago

谢谢，已经成功了

Like · Reply

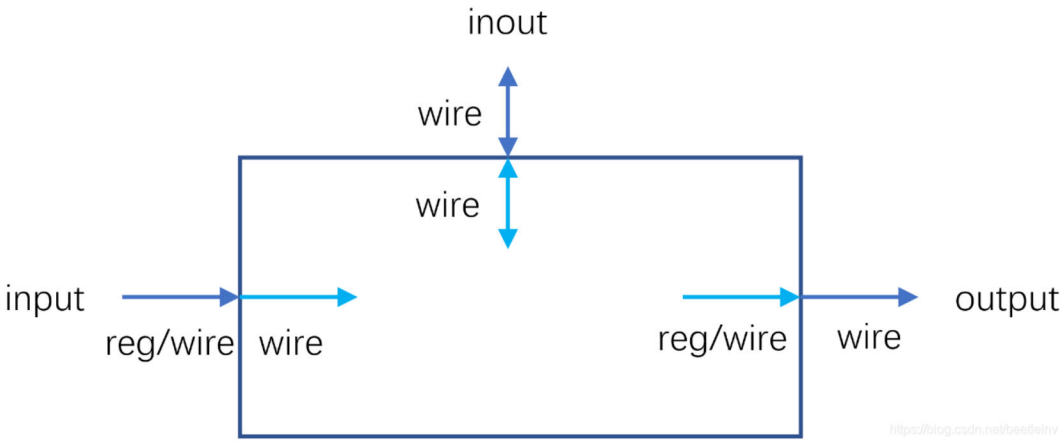
Log In to Answer

verilog实例化时端口信号传递规则


```
module hello_top(
input clk_t,
input rst_t,
input rxd_t,
output txd_t
);

uart_send u_uart_send(
.clk (clk_t),
.rst (rst_t),
.txd (txd_t)

);

endmodule
```

```
module uart_send(
input clk,
input rst,
output txd
);

// coding from here

endmodule
```

示图



表格

端口	从模块内部看	从模块外部看
input 输入端口	必须为线网类型	额可以线网类型或寄存器类型
output 输出端口	可以是线网类型或寄存器类型	必须为线网类型
inout 输入输出端口	必须为线网类型	必须为线网类型

说明

端口连接规则 将一个端口看成由相互链接的两个部分组成，一部分位于模块内部，另一部分位于模块外部。当在一个模块中调用（实例引用）另一个模块时，端口之间的连接必须遵守一些规则。

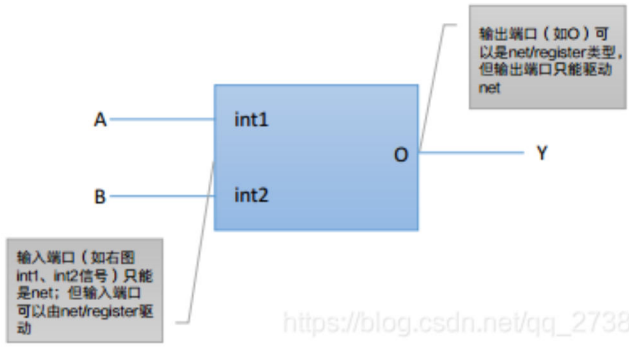
1、输入端口：从模块内部来讲，输入端口必须为线网数据类型，从模块外部来看，输入端口可以连接到线网或者reg数据类型的变量。

2、 输出端口：从模块内部来讲，输出端口可以是线网或者reg数据类型，从模块外部来看，输出必须连接到线网类型的变量（显式，隐式），而不能连接到reg类型的变量。

3、输入/输出端口（必须为wire） 从模块内部来讲，输入/输出端口必须为线网数据类型；从模块外部来看，输入/输出端口也必须连接到线网类型的变量。

在Verilog中如何给端口选择正确的数据类型

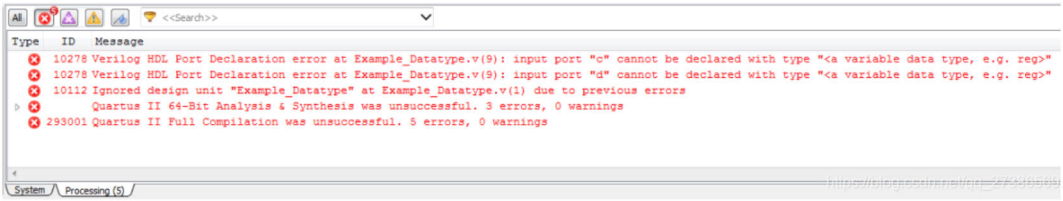
今天要介绍的知识点相信很多同学都已经很清楚了，在这里做介绍只是对我和一些新手做一下知识的巩固。
那么到底如何理解输入和输出端口到底是使用reg类型还是wire类型？



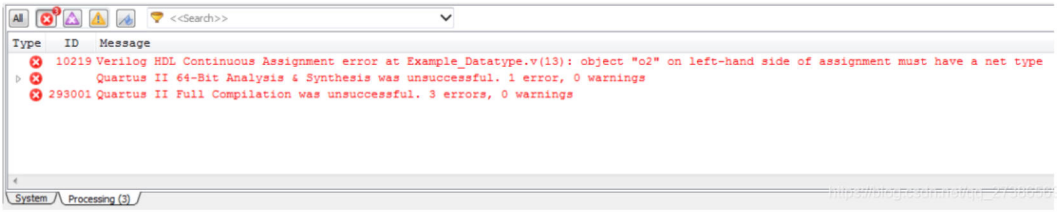
如图1 所示，对于输入A和输入B来说，我们只能使用线网类型，但是，输入A和输入B这两个端口可以由寄存器和线网所驱动。也就是说，寄存器和线网可以连接到这两个输入端口作为输入源。对于输出端口Y来讲，可以是线网，也可以是寄存器类型，但是，对于输出端口Y，它只能驱动线网类型。这样说比较抽象，下面我们结合具体的实例进行说明，代码如下：

```
module Datatype_eg(
a,b,c,d,o1,o2
);
input a,b,c,d;
ouput o1,o2;

reg c,d;
reg o1,o2;
assign o2=c&&d;

always@(a or b)
begin
     if(a)
         o1= b;
      else
         o1=1'b0;
end
endmodule
```

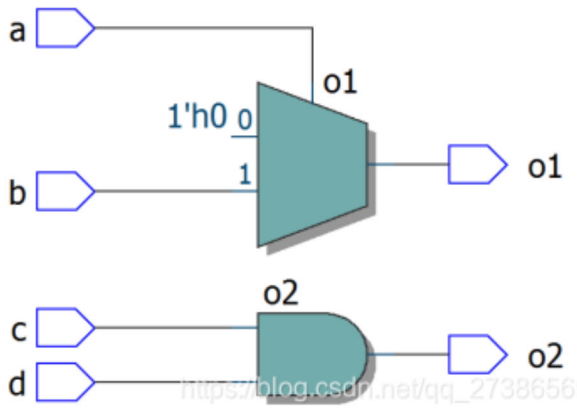
在Quartus II上对上述代码进行编译，会显示出以下错误，如图2所示：



从图2中可知，c不能被声明为reg类型，也就是错误提示中的variable data type,这里为什么会是variable,原因是在新的Verilog-2001标准中，已经没有寄存器类型（register）类型，它们把register改成了variable变量类型，但是一般来说，我们还是比较习惯称为寄存器类型。
言归正传，为什么c和d不能被定义为寄存器类型，是因为c和d是输入端口，前面已经说过了，输入端口不能定义为寄存器类型，因此c和d只能定义为wire类型。这里进行了代码修改，将reg c和reg d注释掉就可以了，默认的类型就是wire类型。修改完成后，代码仍有错误，如图3所示：



如图3所示，o2必须为线网类型，也许有同学会问，前面说对于输出端口来说，可以是线网，也可以是寄存器类型，为什么在这里输出端口o2必须为线网类型不能为寄存器类型呢？这是因为o2是由关键字assign（连续赋值语句）指定的，必须是wire类型，这一点大家可以记住。那么对于o1必须定义为reg类型的，是由于它是在always块里赋值的。因此，只需要将o2注释掉，编译就没有错误了。
这里再补充一点：



如图4所示，**尽管我们将o1声明成了reg类型，但是并没有综合成触发器。原因是对于寄存器reg类型，在always等过程中被赋值的信号，如果该always模块中描述的是时序逻辑电路，那么该**

信号常常被综合为D触发器，如果该always模块中描述的是组合逻辑电路，那么该信号会被综合成连线。

*verilog*基础教程

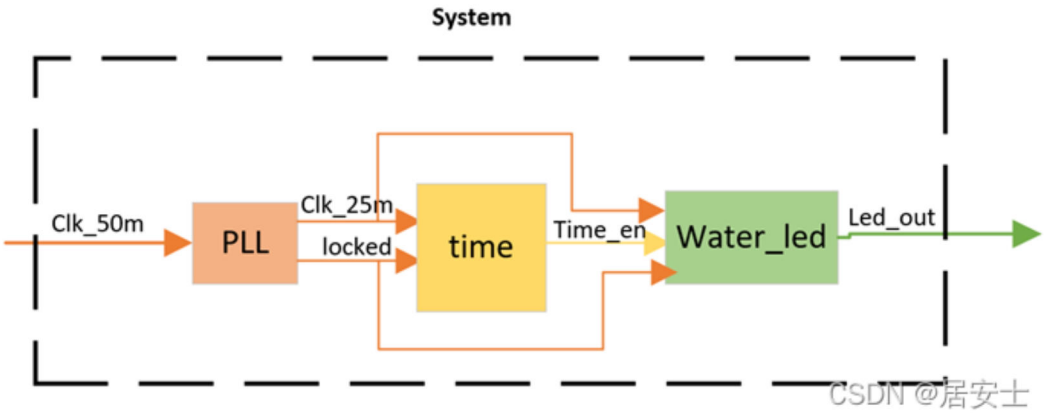
FPGA参数定义 reg&wire 详解_fpge定义参数_朴实妲己的博客-CSDN博客

在写FPGA代码进行参数类型定义时，对于写reg还是wire常常叫人迷惑，下面我将分为以下三种情况，详细解释如何定义参数类型

目录

一 、输入输出定义

输入输出定义是指，在写程序的时候，向各个程序模块里面输入和输出某些参数，例如下面的流水灯程序，PLL，time，water_led三个模块分别有不同的输入输出



针对这种情况应该这样定义

input 都为wire型；

output 若在always块下定义为reg型，写作output reg xxxxx

（不加reg默认wire，不用写output wire xxxx）

（原因：在设计中，输入信号一般来说你是不知道上一级是寄存器输出还是组合逻辑输出，那么对于本级来说就是一根导线，也就是wire型。而输出信号则由你自己来决定是寄存器输出还是组合逻辑输出，wire型、reg型都可以。always块下面进行赋值时，参数相当于被寄存器寄存，所以需要写reg型）

二 、例化定义

这种情况是指，top层汇总各个模块时进行的例化，如上面的图就是system上需要例化PLL，time，water_led三个模块

针对这种情况应该这样定义：

首先top层的输入和输出依然按照上面的定义方法写

把不包含在输入输出里的参数（如上面的图就是clk_25m,locked,time_en）挑出来，写wire型

（原因：例化时，可以把里面的参数都当做信号线，只有连接作用）

三 、 仿真定义

这种情况是指在写TB文件的时候进行参数定义

首先说一下写TB文件的过程，TB文件可以测试小模块也可以测试top层，比如我要测试time模块，那么我就需要把time模块的输入输出复制到TB文件里面，也就是module后面的（....）；

复制过去按照例化的步骤进行例化

先将模块里面为input的变量写reg型，为output的变量写wire型

（注意不要忘写位宽，reg[x:x] wire[x:x]）

但是有些时候会报错，所以按照上面的方法写完之后需要进行一些检测修改：

TB文件里面的参数在initial进行赋值的写reg型，其他的写wire型

然后就没有问题了

输入输出定义，例化定义，仿真定义已经几乎包含了FPGA里面所有的参数定义情况，按照我上面的方法定义不会有问题，下面来介绍一下原理：

Verilog 中变量的物理数据分为线型和寄存器型，线型数据包括wire,wand,wor等几种类型，其中wire最常用，对应于实际的数字电路，线型wire实际上就对应着硬件的连线，起到连接作用。寄存器是存储单元的抽象，寄存器数据类型的关键字是reg。常用来表示always模块内的指定信号，代表触发器。在always模块内被赋值的每一个信号都必须定义成reg型。

所以reg相当于存储单元，wire相当于物理连线

在需要赋值计算的时候就用reg，只是连接作用就用wire

将verilog中的wire和reg型数据相连接的方法

写过的典例：
数据说明：例化元件输出寄存器.dig；中间wire dig_temp;顶层程序输出寄存器dig。

目标：将例化元件的输出寄存器的值接到顶层程序输出寄存器中，因为两者都是寄存器，所以不能直接相连，需要借助中间一根线wire。

```
1. output reg [5:0] dig;

2. wire [5:0] dig_temp;

3. .dig(dig_temp);

4. always@(dig_temp)

5. begin dig=dig_temp; end
```

总结笔记：

reg：寄存器类型数据；wire：线类型数据

在使用例化模块进行编程时经常会遇到这两种类型的数据相连接的情况

wire型数据：在assign左侧被赋值；

reg型数据：在always@的左侧被赋值；

两者均能在assign和always@的右侧被使用。

Verilog中 reg和wire 用法和区别以及always和assign的区别

1、从仿真角度来说，HDL语言面对的是编译器，相当于使用软件思路，此时：

wire对应于连续赋值，如assign；

reg对应于过程赋值，如always，initial；

2、从综合角度，HDL语言面对的是综合器，相当于从电路角度来思考，此时：

wire型变量综合出来一般情况下是一根导线。

reg变量在always中有两种情况：

（1）always @（a or b or c）形式的，即不带时钟边沿的，综合出来还是组合逻辑；

（2）always @（posedge clk）形式的，即带有边沿的，综合出来一般是时序逻辑，会包含触发器（Flip-Flop）

3、设计中，输入信号一般来说不能判断出上一级是寄存器输出还是组合逻辑输出，对于本级来说，就当成一根导线，即wire型。而输出信号则由自己来决定是reg还是组合逻辑输出，wire和reg型都可以。但一般的，整个设计的外部输出（即最顶层模块的输出），要求是reg输出，这比较稳定、扇出能力好。

4、Verilog中何时要定义成wire型？

情况一：assign语句

例如：

```
reg   a,b;
wire  out;
......
assign   out = a & b;
```

如果把out定义成reg型，对不起，编译器报错！

情况二：元件实例化时必须用wire型

例如：

```
 wire   dout;

ram   u_ram

(

 ....

.out(dout);

)
```

wire为无逻辑连线，wire本身不带逻辑性，所以输入什么就的输出什么。所以如果用always语句对wire变量赋值，对不起，编译器报错。

那么你可能会问， assign c = a & b; 不是对wire的赋值吗？

并非如此，综合时是将 a & b综合成 a、b经过一个与门，而c是连接到与门输出线，真正综合出来的是与门&，不是c。

5、何时用reg、何时用wire？

大体来说，reg和wire类似于C、C++的变量，但若此变量要放在begin...end之内，则该变量只能是reg型；在begin...end之外，则用wire型；

使用wire型时，必须搭配assign；reg型可以不用。

input、output、inout预设值都是wire型。

在Verilog中使用reg型，并不表示综合出来就是暂存器register：在组合电路中使用reg，组合后只是net；在时序电路中使用reg，合成后才是以Flip-Flop形式表示的register触发器。

6、reg和wire的区别：

reg型数据保持最后一次的赋值，而wire型数据需要持续的驱动。wire用在连续赋值语句assign中；reg用于always过程赋值语句中。

在连续赋值语句assign中，表达式右侧的计算结果可以立即更新到表达式的左侧，可以理解为逻辑之后直接连接了一条线，这个逻辑对应于表达式的右侧，这条线对应于wire；

在过程赋值语句中，表达式右侧的计算结果在某种条件的触发下放到一个变量当中，这个变量可以声明成reg型，根据触发条件的不同，过程语句可以建模不同的硬件结构：

（1）如果这个条件是时钟上升沿或下降沿，那硬件模型就是一个触发器，只有是指定了always@（posedge or negedge）才是触发器。

（2）如果这个条件是某一信号的高低电平，那这个硬件模型就是一个锁存器。

（3）如果这个条件是赋值语句右侧任意操作数的变化，那这个硬件模型就是一个组合逻辑。

7、过程赋值语句always@和连续赋值语句assign的区别：

（1）wire型用于assign的赋值，always@块下的信号用reg型。这里的reg并不是真正的触发器，只有敏感列表内的为上升沿或下降沿触发时才综合为触发器。

（2）另一个区别，举例：

```
 wire     a;

 reg      b;

assign     a = 1'b0;

always@(*)

        b = 1'b0;
```

上面例子仿真时a将会是0，但是b的状态是不确定的。因为Verilog规定，always@（*）中的*指的是该always块内的所有输入信号的变化为敏感列表，就是说只有当always@（*）块内输入信号发生变化，该块内描述的信号才会发生变化。

像always@（*） b= 1'b0; 中由于1‘b0是个常数，一直没有变化，由于b的足组合逻辑输出，所有复位时没有明确的值--即不确定状态，又因为always@（*）块内没有敏感信号变化，此时b信号一直保持不变，即不确定是啥，取决于b的初始状态。

Verilog 对assign和always的一点理解

blog.csdn.net/iamoyjj/article/details/3478321

assign 用于描述组合逻辑

always@(敏感事件列表) 用于描述时序逻辑

敏感事件 上升沿 posedge，下降沿 negedge，或电平

敏感事件列表中可以包含多个敏感事件，但不可以同时包括电平敏感事件和边沿敏感事件，也不可以同时包括同一个信号的上升沿和下降沿，这两个事件可以合并为一个电平敏感事件。

在新的verilog2001中“，”和“or”都可以用来分割敏感事件了，可以用“*”代表所有输入信号，这可以防止遗漏。

合法的写法：

always@ *

always@ (posedge clk1,negedge clk2)

always@ (a or b)

`timescale 100ns/100ns //定义仿真基本周期为100ns

always #1 clk=~clk //#1代表一个仿真周期即100ns

所有的assign 和 always 块都是并行发生的！

并行块、顺序块

将要并行执行的语句写在

fork

//语句并行执行

join

将要顺序执行的语句写在

begin

//语句顺序执行

end

并行块和顺序块都可以写在

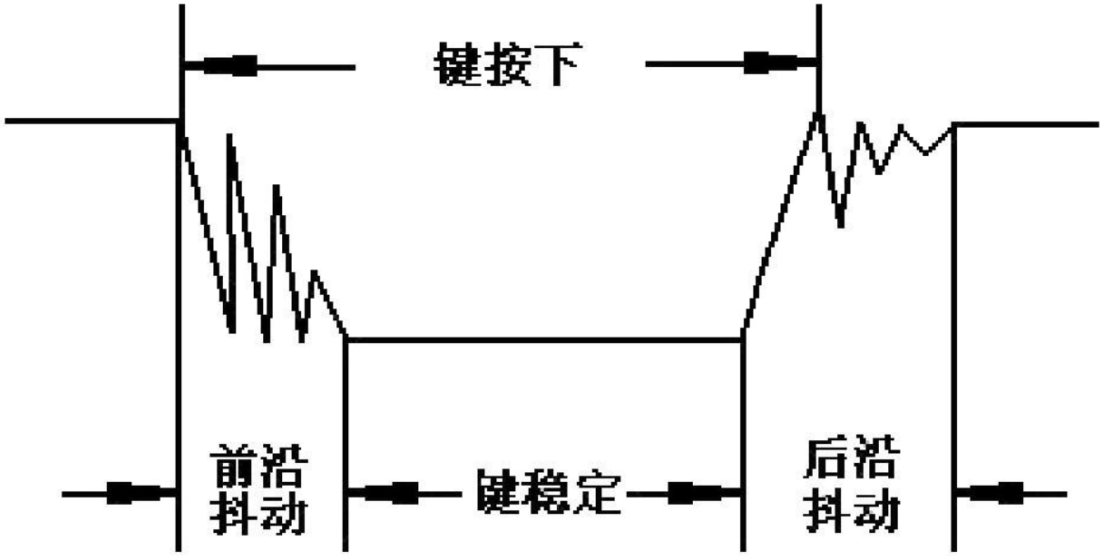
initial 或 always@ 之后，也就是说写在块中的语句是时序逻辑的

对assign之后不能加块，实现组合逻辑只能用逐句的使用

assign
组合逻辑如果不考虑门的延时的话当然可以理解为瞬时执行的，因此没有并行和顺序之分，并行和顺序是针对时序逻辑来说的。值得注意的是所有的时序块都是并行执行的。
initial块只在信号进入模块后执行1次而always块是由敏感事件作为中断来触发执行的。

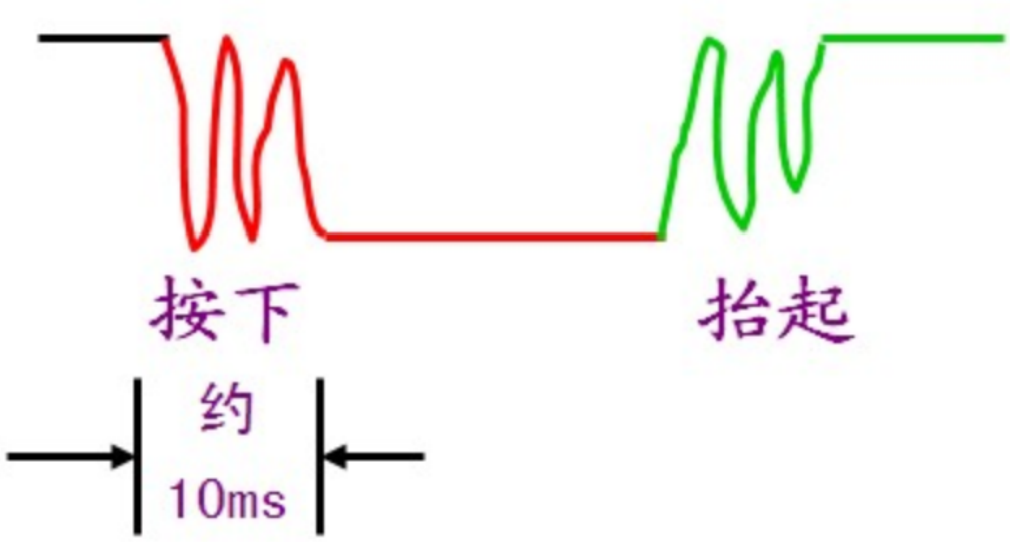
底层硬件底板按键抖动：关于按键抖动的时间、按键消抖

通常按键抖动所用的开关都是机械弹性开关，当机械触点断开、闭合时，由于机械触点的弹性作用，一个按键开关在闭合时不会马上就稳定的接通，在断开时也不会一下子彻底断开，而是在闭合和断开的瞬间伴随着一连串的抖动，如下图所示。



理想状态下的按键

按键稳定闭合时间长短是由操作人员决定的，通常都会在100ms以上，刻意快速按的话能达到40~50ms左右，很难在低了。抖动时间是由按键的机械特性决定的，一般都会在10ms以内，为了确保程序对按键的一次闭合或者一次断开只响应一次，必须进行按键的消抖处理。



在绝大多数情况下，是用软件即程序来实现消抖的。最简单的消抖原理，就是当检测到按键状态变化后，先等待一个10ms左右的延迟时间，让抖动消失后再进行一次按键状态检测，如果与刚才检测到的状态相同，就可以确认按键已经稳定的动作了。实际应用中，这种做法局限性大（实时性差）

软件去抖的几个思路：

1、延迟
先定义一个变量记录按键的状态：char key；
然后轮询检测按键状态，当按键状态改变的时候，判断为有按键动作，接下来进入延迟函数；
等到延迟时间过了之后，再读取按键状态，如果按键状态仍为按下状态，则说明确实是有按键动作，不是抖动。

2、中断加延迟方式
单片机总是轮询状态会浪费很多资源，尤其是单片机运行的时间，可以通过中断方式来解决。
中断模式下，单片机不需要轮询按键状态，当有中断产生的时候才会进入延时函数，进行按键去抖程序。

这种情况下，延时函数在哪里进行处理就有两种选择：
一种是放在中断函数中进行，这个时候中断函数占用的时间就比较长，影响响应速度；
另一种是放在中断函数之外进行，这个时候就缩短了中断处理时间，但是这个时候就需要一个标志位来表明是否有中断产生，而且单片机也需要不断查询，只是节约了查询时候读取IO状态的步骤。

3、持续采样
持续采样会大大提高采样的准确度，但是同时也会增加CPU的开销。
在使用中，需要根据需要选择不同的采样频率，一般每10ms采集一次就足够了。

关于延迟方法：
简单的延时，可以采用空循环来实现，这个方法比较消耗CPU的资源，CPU任务较重的时候不建议使用。延时函数可以考虑使用定时器替换，但这又消耗了定时器资源，不过只要够用的话还是尽量用，毕竟可以减轻CPU的负担。

关于一次按键过程中产生多次中断的处理：
一次按键过程，可能产生多次中断，可以设置一个标志位来显示是否处于按键识别处理阶段，如果处于按键识别处理阶段即便产生中断也不进行任何响应，这样就可以忽略多余的中断了。