First Look @ Verilog

Enjoy Yourself!

GNG not Gua

Overview

- Verilog结构
- Verilog数据类型
- Verilog运算符
- initial & always
- 赋值
- 流程控制
- 测试
- 讨论0
- 讨论1*

注：标题或副标题有“*”号的章节可以跳过，不影响后续内容。

请注意：所有有下划线的内容都超链接到某网页，页面右上角可能会有对应的习题，使用的平台是HDLBits，您可以选做，当然如果你不习惯阅读文档的话，也可以通过HDLBits进行快速的学习。

Verilog结构

Verilog是什么

Verilog是一种基本语法与C语言相近的硬件描述语言，其主要用于集成电路设计。

Verilog能够在晶体管级、逻辑门级以及寄存器传输级(register-transfer level, RTL)对数字逻辑系统进行描述。

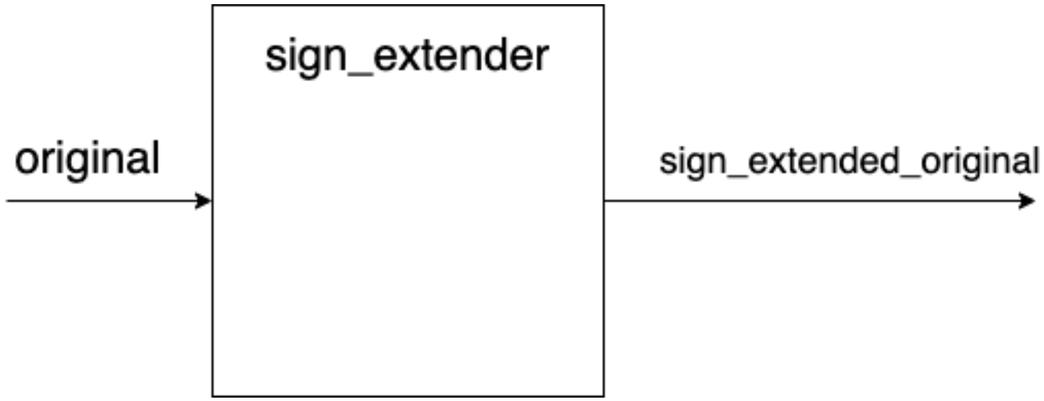
下边是一段简单的Verilog代码，它的输入是8位，输出为符号扩展为16位。

```
module sign_extender
    (
        input [7:0] original,
        output [15:0] sign_extended_original
    );

    assign sign_extended_original = {{8{original[7]}}, original[7:0]};

endmodule
```

Verilog结构

模块 module

- 模块是Verilog设计的基础，在设计过程中我们通常将复杂的功能破分为简单的功能，模块就是实现简单功能的基本结构。
- 一个模块由端口声明和功能代码组成。
- 通常情况下，我们将模块写到一个后缀名为 .v 的Verilog文件中，同时以模块名命名该文件。



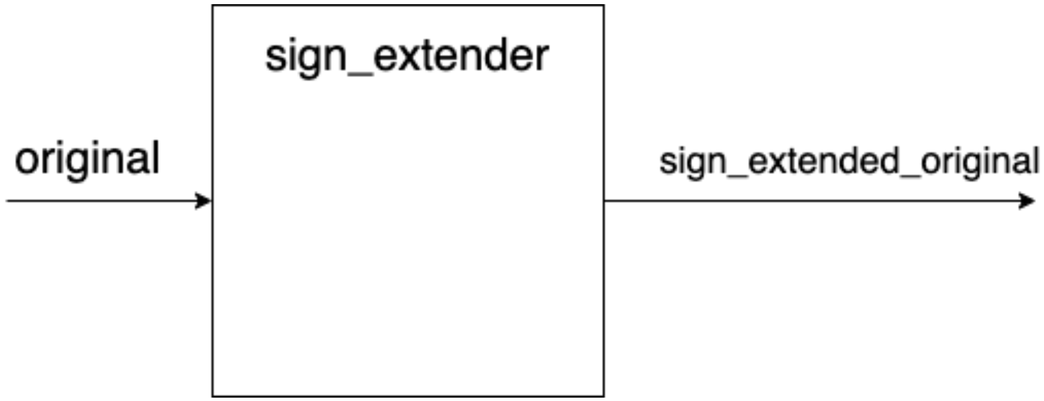
文件 sign_extender.v

```
module sign_extender ( input [7:0] original, output [15:0] sign_extended_original);
    // code here has access to inputs
    // code here can drive outputs
endmodule
```

Verilog结构

模块 module —— I/O端口

- 端口的存在用于一个模块与其他模块进行交流。
- 根据类型： input output inout
- （较常采用的写法）在设计时，模块被 module <module_name> (<IO Port 0>, <IO Port 1>, …) 以及endmodule包围，在模块声明的括号中对端口进行声明，在功能代码中可以直接使用。
- 需要注意的是，在不加额外类型说明的情况下，默认端口为 wire，如果希望端口类型为reg需要添加说明，比如 module this_module (input [1:0] in, output reg out)，就是输入2位，输出1位且输出的信号在功能代码中作为reg使用。reg与wire的区别请见下一节

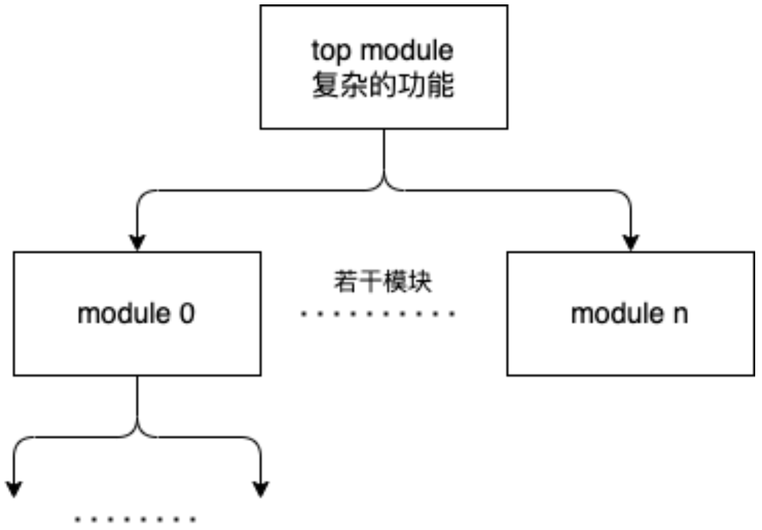


Verilog结构

Top Module

模块是实现功能的简单结构，我们需要一个“顶层模块”来实现期望的复杂功能，即在Top Module中调用其他模块。

每一个Verilog工程中有且只有一个模块作为Top Module，在Top Module中，我们规定了设计的系统的所有输入和输出。即调用顶层模块的模块并不能直接接触到内部模块的输出信号，比如在测试过程中，详见**测试**一节。



Verilog结构

模块的实例

实例化模块的方式如下：

<module name> <instance name> (<port list>)

其中<port list>有两种写法：给出端口名的(.port0(wire0), .port1(wire1), …)；不给出端口名的(wire0, wire1, …)。前者需要给出的端口名正确，不要求顺序；后者必须在对应端口的位置。

需要注意的是两种方式不能混用，推荐使用前者。

例子：输入一个8位向量，输出8位向量倒转后符号扩展到16位的结果。

```
module top
  (
    input [7:0] in,
    output [15:0] out
    );

  wire [7:0] reversed;

  // Module reverse be declaration somewhere else
  // "module reverse(input [7:0] in, output [7:0] out
);"
  reverse reverse_instance (.in(in), .out(reversed));

  // The module declare before
  sign_extender sign_extender_instance (.original(in)
, .sign_extended_original(out));

endmodule
```

Verilog结构

模块 module —— 模块参数*

- 在设计系统时，我们通常希望写少量的代码完成大量的工作，设计中经常会遇到代码结构类似但是因为数据的不同**位数**要重复的写功能类似的模块。比如8位符号拓展到16位，16位符号拓展到32位等等。
- 我们也希望模块有一些预置的参数，提高代码可读性。总的来说，我们希望写出能生成代码的代码。模块参数可以在某些方面(如仅有输入输出位宽不同的功能类似的模块)实现这一愿望。
- 其写法可参考下一页中的代码，如希望了解更多请点击这里。(如果看不懂这一段代码，不用慌张，这一slides的目的就是让大家初步了解Verilog，请先阅读后续内容)

Verilog结构

模块 module —— 模块参数举例*

```
module sign_extender_para
    #(
        parameter
        INPUT_WIDTH = 8,
        OUTPUT_WIDTH = 16
        )
    (
        input [INPUT_WIDTH-1:0] original,
        output reg [OUTPUT_WIDTH-1:0] sign_extended_origi
nal
        );

    wire [OUTPUT_WIDTH-INPUT_WIDTH-1:0] sign_extend;

    generate
        genvar i;
        for (i = 0; i < OUTPUT_WIDTH-
INPUT_WIDTH; i = i + 1) begin : gen_sign_extend
            assign sign_extend[i]  = (original[INPUT_WIDT
H-1]) ? 1'b1 : 1'b0;
        end
    endgenerate

    always @(*) begin
        sign_extended_original  = {sign_extend,original};
    end

endmodule
```

```
module top
    (
        input [7:0] input_8,
        input [15:0] input_16,
        output [15:0] output_8_to_16,
        output [31:0] output_16_to_32
        );

    sign_extender_para #(.INPUT_WIDTH(8), .OUTPUT_WIDTH(1
6))
        sign_extender_8_16 (.original(input_8), .sign_ext
ended_original(output_8_to_16));
    sign_extender_para #(.INPUT_WIDTH(16), .OUTPUT_WIDTH(
32))
        sign_extender_16_32 (.original(input_16), .sign_e
xtended_original(output_16_to_32));

endmodule
```

Verilog数据类型

四值逻辑

我们在数字逻辑课程中学到了二值逻辑(True or False)，在Verilog中则有四种不同的逻辑值，因此叫四值逻辑。

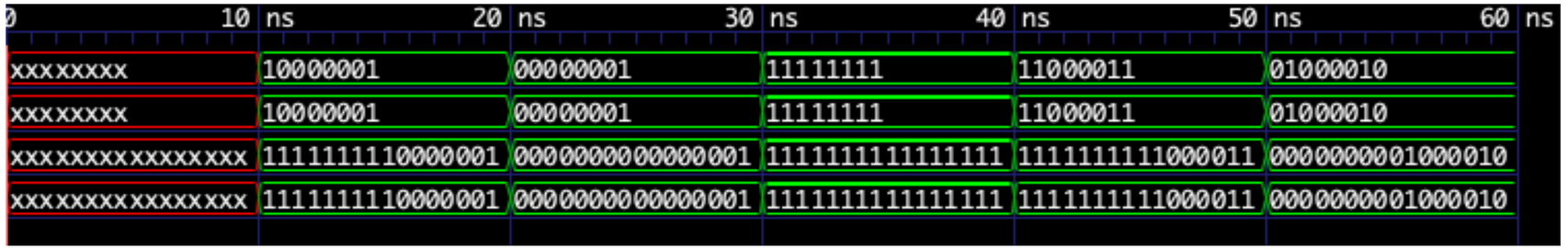
0: 逻辑低电平，假

1: 逻辑高电平，真

z: 高阻态，浮动

x: 未知逻辑电平

当发生竞争时，则会根据不同信号的信号强度产生结果，详见这里(数据类型-四值逻辑 节)。



Verilog数据类型

数字表示

在Verilog中，如果要表示一个具体的数值，格式如下

<bit width>’<base letter><number>

其中，常用的数制二进制(b)，八进制(o)，十进制(d)，十六进制(h)。

如果没有指明位宽，则默认的位宽和仿真器有关。如果数据不足位宽，则根据最高位扩展：一般填充0，如果最高位是x或z，则以x或z填充。

如果没有指明数制，则默认为十进制。

233: 十进制数233（未指明位宽）
12’h123: 十六进制数0x123（12位）
20’d77: 十进制数77（位宽20位，前补0）
4’b1001: 二进制数0b1001（位宽4位）
6’o77: 八进制数77（位宽6位）
-233: 十进制数-233（未指明位宽）
-32’d3: 十进制数-3（位宽32位）
32’hfffffffd: 十六进制数0xfffffffd（位宽32位）

注：举例来自维基百科。

Verilog数据类型

线网 （wire等）

- 线网是verilog中一种基本的数据类型，与实际的**电线**类似。有wire、tri、wor等，本课的实验中一般只需要使用wire，因此下面只给出wire的描述。
- 其值一般只能通过连续赋值得到（assign关键字），在初始化之前，其值为x。
- 连续赋值的含义是，assign右侧（驱动源）值变化后，wire变量的值随之变化，wire并不能在没有驱动源的情况下对当前值进行保存。
- 如果wire变量没有连接驱动源（没有进行assign），则其值为z。

```
wire [15:0] sign_extended_original;
assign sign_extended_original = {{8{original[7]}}, original[7:0]};
```

Verilog数据类型

寄存器（reg等）

- 寄存器与线网的区别在于，它可以保持目前的值，直到新的赋值操作修改它的值。
- 寄存器有reg、integer、time、real等，reg较常使用。
- 如果未对寄存器初始化值，其值为x。
- 在initial过程和alway过程中的赋值操作，一定是对寄存器的赋值操作（不能是wire）。

Verilog数据类型

向量 vectors

Verilog中标量指的是只有1位二进制位的变量，向量指的是有多个二进制位的变量。

其写法：

<nets type> [<Fitst> : <Last>] <var. name>

与C语言不同的是，向量分量的序号不一定从0开始；一般为了和数字电路中二进制高低位的表示方法一致，通常让最低位为0，即 [BIT_WIDTH-1:0]。

```
wire [3:0] input_add; //声明名为input_add的4位wire型向量
wire [4:1] input_add1; //也是4位wire型向量，但是分量序号从4到1
wire [0:3] input_add2; //也是4位wire型向量，但是分量序号从0到3

input_add [3] = 1'b1; //将1赋值给input_add向量的第三位（最高位）
input_add [1:0] = 2'b01; //将0和1分别赋值给input_add向量的第1、0位（最低两位）
```

注：代码来自维基百科。

Verilog数据类型

数组

Verilog中几种寄存器类型以及其向量形式都可以构成数组。

其表示方法为 <nets type/vectors type> <var. name> [<First> : <Last>]

请注意将数组和向量区分开，第一部分的<nets type/vectors type>可以为向量。

```
reg reg_arr [20:0]; // 一维数组共21个元素，每个元素为1位寄存器
reg_arr[1] = 1’b1;
reg_arr[1] = 3'b100; // 实际与预期不符，每一个元素（如reg_arr[1]）仅有1位

reg [7:0] reg_vectors_arr [255:0]; // 一维数组共256个元素，每个元素为8位寄存器向量
reg_vectors_arr[2] = 8'b0001_1000;

reg reg_arr_2 [255:0][31:0]; // 二维数组，两个维度分别有256和32，每个元素仅1位
reg_arr_2[233][23] = 1'b1;
```

Verilog运算符

按位

一元

按位取反 ~ 一个操作数按位取反。如：a=4’b0101，则 ~a=4’b1010

二元

按位与 & 两个操作数按位与。如：a=2’b01, b=2’b11，则a&b=2’b01

按位或 | 两个操作数按位或。如：a=2’b01, b=2’b11，则a|b=2’b11

按位异或 ^ 两个操作数按位异或。如：a=2’b01, b=2’b11，则a^b=2’b10

按位同或 ~^或^~ 两个操作数按位同或。如：a=2’b01, b=2’b11, 则a~^b=2’b01

Verilog运算符

逻辑

认为0为逻辑假，其他数为逻辑真。

一元

逻辑取反 !，a=4’b0100, b=32’b0，则 !a=0, !b=1

二元

逻辑与 &&, a=4’b0100, b=32’b0, a&&b=0; c=1’b1, 则a&&c=1

逻辑或 ||, a=4’b0100, b=32’b0, 则a||b=1

Verilog运算符

按位

一元

按位取反 ~ 一个操作数按位取反。如：a=4’b0101，则 ~a=4’b1010

二元

按位与 & 两个操作数按位与。如：a=2’b01, b=2’b11，则a&b=2’b01

按位或 | 两个操作数按位或。如：a=2’b01, b=2’b11，则a|b=2’b11

按位异或 ^ 两个操作数按位异或。如：a=2’b01, b=2’b11，则a^b=2’b10

按位同或 ~^或^~ 两个操作数按位同或。如：a=2’b01, b=2’b11, 则a~^b=2’b01

Verilog运算符

归约

归约运算符均为一元运算符，其意义为对一个多位变量从最高位和次高位操作获得一个结果，再与新的次高位进行操作，直到最低位。

归约与 & 如： &(4’b1001)=0；只要有一位为0结果就是0

归约与非 ~& 如：~&(4’b1011)=1；只要有一位为0结果就是1

归约或 | 如：|(4’b1011)=1；只要有一位为1结果就是1

归约或非 ~| 如： ~|(4’b1011)=0；只要有一位为1结果就是0

归约异或 ^ 如：^(4’b1011)=1；奇偶校验，有奇数个位为1结果为1，有偶数个位为1结果为0

归约同或 ~^或^~ 如：~^(4’b1011)=0；奇偶校验，有奇数个位为1结果为0，有偶数个位为1结果为1

Verilog运算符

算术 关系

算术

有加(+)、减(-)、乘(*)、除(/)、求幂(**)

关系

大于(等于)/小于(等于) > < >= <=

逻辑等(==), 如果各位均相等则结果为真；特别地，两个操作数中任何一个操作数中有x或z，结果为x

逻辑不等(!=), 如果有至少一位不相等结果为真；特别地，两个操作数中任何一个操作数中有x或z，结果为x

case相等(===), 两操作数各位相等(包括x, z)，结果为真

case不相等(!==), 两操作数至少一位不相等(包括x, z)，结果为真

Verilog运算符

移位

移位

逻辑左移(<<)，算术左移(<<<) 效果相同，左操作数左移位，产生的空位补0。如：a=32’b1100, b=3’b010, 则a<<b=32’b110000

逻辑右移(>>), 左操作数右移位，产生的空位补0。如：a=32’b1100, b=3’b010, 则a>>b=32’b11

算术右移(>>>)，左操作数右移位，无符号数与逻辑右移相同，产生的空位补0；如果是有符号数，产生的空位补原符号位。如：a=4’b1100, b=2’b01, c=4’b0110, 则a>>>b=4’b1110, a>>>c=4’b0011

Verilog运算符

拼接 重复 条件

拼接

将操作数分别置于高低位拼接，{(4’b1100), (3’b101)}结果为7’b1100101

重复

{n{m}} 将m重复n次，比如{3{2’b01}}结果为6’b010101

条件

?: 与C语言中的三目运算符类似，根据?前的条件是否为真执行后边的两个语句之一

initial & always

Verilog中initial和always用于定义过程。其区别在于initial只执行一次，即初始化；always则是满足条件的情况下不限次数的执行。

两个过程都从时间0开始执行，且每一次都执行到过程块的结束。always块执行结束后会准备再次执行（下一次满足always的条件时）。

需要注意的是，initial块并不一定比always块先执行，可以把initial看作always的一个特例，即只执行一次的always。

书写时，initial或always需要用begin和end关键词包围（类似于一组花括号），如果仅有一条语句，则不需要begin与end。

initial & always

initial

initial一般用于testbench，用来控制输入变量值的变化，其写法大致如下

```
// module demo (input in, output out);
initial begin
    in = 2'b10;  // 改变输入的值
    #10; // 等待10个单位时间
    in = 2'b01;
    #10;
    in = 2'b00;
    #10;
end
```

*一个比较别致的写法：在initial后添加forever关键词，使其行为与always无异，可以查看这里。

initial & always

always

always通过@来控制执行时机。

```
always @(*) // 在任何情况下都执行always块，常用于书写组合逻辑
always @(a) // 当a的值发生改变的时候，执行
always @(a or b) // 当a或b的值发生改变的时候，执行
always @(posedge a or negedge b) // 当遇到a的正边沿或b的负边沿时，执行
always // 不带@，执行完一次块内内容后，就再执行一次，常用于tesebench的clk信号改变
```

正边沿：信号从low到high

负边沿：信号从high到low

initial & always

always使用举例

always@(*)常用于组合逻辑的书写，比如

```
reg a;
reg a_rev;
always @(*) begin
  a_rev = ~a;
end
```

在测试中，控制时间信号clk

```
// 每过一个时间单位，clk倒置一次
always #1 clk = ~clk;
```

每一次时钟负边沿进行一次计数

```
// reg [255:0] counter;
always @(negedge clk) begin
  counter = counter+1;
end
```

赋值

连续赋值

线网不能像寄存器一样存储当前的值，它需要驱动源连续不断的进行“赋值”，这种赋值被称为连续赋值。

连续赋值的关键词是assign，左侧为被赋值的线网变量，右侧为驱动源，变量的值会随时随着驱动源值的变化而变化。

下面语句意义为，out变量的值为in0与in1按位与的结果。

```
wire [1:0] out;
wire [1:0] in0;
wire [1:0] in1;

assign out = in0 & in1;
```

*可以在assign后附加延迟信息，意义为驱动源变化多少个时间单位后，线网变量的值才发生改变。

```
assign #5 out = in0 & in1;
```

赋值

过程赋值（阻塞赋值）

在过程中（initial/always），可以使用“=”或“<=”对寄存器变量进行赋值，它们分别为阻塞赋值和非阻塞赋值。

阻塞赋值和C语言中的“=”作用类似，（在顺序代码块中）赋值结束之前，后边的语句不会被执行。

下边这段代码执行后，a和b的值都为b的初始值。

```
always @(posedge clk) begin
  // b初始值为0
  a = b;
  b = a;
  // 结束时，a=b=0
end
```

赋值

过程赋值（非阻塞赋值）

使用“<=”对寄存器进行非阻塞赋值，非阻塞赋值含义为执行赋值语句时不会阻碍下一条语句的执行。因为过程赋值是“同时”发生的，等号右侧都是原来的值，下面这个例子实现了寄存器a和b的值交换。

```
always @(posedge clk) begin
  // a初始值为1,b初始值为0
  a <= b;
  b <= a;
  // 结束时，a=0, b=1
end
```

一般来说，在条件为信号边缘时，使用非阻塞赋值。（见讨论）

流程控制

If-Else 条件结构

Verilog提供了条件分支结构(If-Else, case)和循环结构(for, while, forever)，这里仅介绍条件结构与分支结构。

条件分支结构可以对应到多路复用器。If-Else的结构与C语言相似，下面这段代码是一个计时器的作用（当然有更简单的写法），在counter到达0b1100时，对外输出0并重置计数器；其他情况下每一次clk信号的正边沿计数器自增一且保证输出为1。

```
reg [4:0] counter;
output reg out;

always @(posedge clk) begin
  if(counter == 4'b1100) begin
    counter = 4'b0; // 重置计数器
    out <= 1'b0;
  end else begin
    counter <= counter + 1; // 计数器加一
    out <= 1'b1;
  end
end
```

流程控制

case 分支结构

比起条件结构，分支结构更明显地对应了多路复用器的功能。

其基本结构如下(下列代码来自这里):

```
case(<expression>)
  case_item1: <single_statement>
  case_item2,
  case_item3: <single_statement>
  case_item4: begin
    <multiple_statement>
  end
  default: <statement>
endcase
```

请自行解释右侧代码。

```
input [1:0] op;
input [31:0] A;
input [31:0] B;
output reg [31:0] res;
output reg sig;

always @(posedge clk) begin
  case(op)
    2'b00,
    2'b01: res = 32'b0;
    2'b10: res = A & B;
    2'b11: begin
      res = 32'hFFFF_FFFF;
      sig = 1'b0;
    end
    default: sig = 1'b1;
  endcase
end
```

测试

testbench

在每一个模块（包括子模块和Top模块）完成后，我们需要检测其功能是否符合预期，此时我们需要进行完整有效的测试，本节的目的在于测试文件的书写，关于Windows/Linux/macOS下的测试方法以及测试的设计不在本文档的讨论范围内。

一个测试文件主要有以下几个部分组成：

- timescale
- testbench模块
  - 系统任务（可选）
  - reg（input）
  - wire（output）
  - 被测试的模块
  - initial块

测试

timescale

在测试过程中，我们需要规定时间单位以及精度，其写法如下

`timescale <time unit> <time precision>

- time unit: 时间单位，将决定延时（#10）的单位
- time precision: 精度，因为延时可以使用小数，time precision可以决定延时的精度（小数能精确到多少位）

规定：时间的书写中整数部分要求从1、10、100中选择，单位部分可以选择s/ms/us/ns/ps/fs。

`timescale 1ns/1ps 可以使用#10.5来表示延时10.5ns；而`timescale 1ns/1ns 中如果使用#10.5可能得到期望之外的结果，因为0.5ns已经超过了精度“ns”。

```
`timescale 1ps/1ps // #5表示延时5ps，#1.5的表现与预期不符
`timescale 1ns/1ps // #5表示延时5ns，#1.5可以正常延时1.5ns
`timescale 10ns/100ps // #5表示延时50ns，#1.5可以正常延时15ns
```

注：本页内容参考这个网页。

测试

testbench模块

每一个.v文件中都有至少一个module，测试文件的主体也是一个module，它通常的结构如下：

```
module module_name_to_test_tb; // "tb" is short for "testbench"
  // Code Here
endmodule
```

一般来说，这个模块没有input和output。

“// Code Here”部分的内容：首先是regs与nets。我们对某一个模块进行测试，需要控制其输入并获得其随输入改变的输出内容，联系前文的内容可以很轻易地产生一个想法：把所有inputs声明为寄存器变量(通常为reg)，把所有outputs声明为线网变量(通常为wire)。比如我们希望测试的模块声明为：

```
module sign_extender( input [7:0] original, output [15:0] sign_extended_original);
```

那么我们在testbench模块中会写下：

```
reg[7:0] in;
wire[15:0] out;
```

测试

testbench模块

刚刚得到了两组变量：寄存器变量作为输入，线网变量记录输出。下面将需要测试的模块引进testbench：

```
sign_extender m0(in, out);
```

前文提到，initial块通常用于测试过程中，因其只执行一遍，可以很好的控制测试波形的结束。需要注意的是，initial块从time=0时就开始执行。在仿真过程中，如果我们不对赋值操作做延时，所有的操作都会“瞬间完成”，因此一般测试内容都由延时分割开。

```
initial begin
    // $dumpfile("demo1_tb.vcd");
    // $dumpvars(0, demo1_tb);

    in = 8'b0000_0000;
    #10; in = 8'b1000_0001;
    #10; in = 8'b0000_0001;
    // Other tests
    #10;
end
```

注：仿真过程中资源是无限的，同时由于某些玄学原因，仿真验证通过并不代表着下板验证能够通过。这一点在数逻课程中体现不明显，但是在计算机组成等课程实验中有可能会遇到（板载资源利用率较高）。

测试

系统任务*

这里需要提到的是$dumpfile() 与 $dumpvars()。

引用文档*Verilog_Simulation(MacOS|Linux)*中的内容：

$dumpfile("filename.vcd");将nets与regs的变化写入filename.vcd中，VCD即value change dump。在一个模拟中，我们只能有至多一个$dumpfile。

$dumpfile指定了我们将dump得到的信息存储到哪里，而$dumpvars规定了我们希望存储什么信息。

具体内容可见这里。

```
$dumpfile("demo1_tb.vcd");
$dumpvars(0, demo1_tb);
```

测试

testbench举例

```
`timescale 1ns/1ps

module demo1_tb;
reg[7:0] in;
wire[15:0] out;

sign_extender m0(in, out);

initial begin
    $dumpfile("demo1_tb.vcd");
    $dumpvars(0, demo1_tb);

    in = 8'b0000_0000;
    #10; in = 8'b1000_0001;
    #10; in = 8'b0000_0001;
    #10; in = 8'b1111_1111;
    #10; in = 8'b1100_0011;
    #10; in = 8'b0100_0010;
    #10;
end
endmodule
```

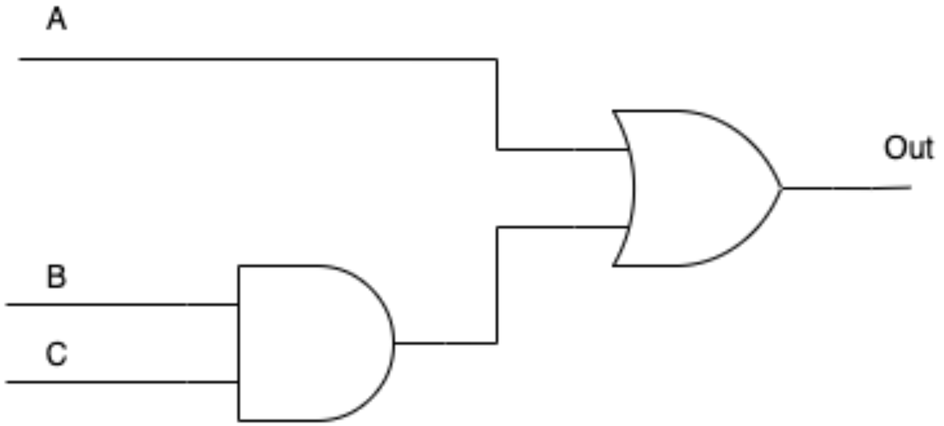
讨论

结构化描述

对于右侧电路，我们根据之前学到的内容可以表示为

```
// input A, B, C; output Out;
assign Out = A | (B & C);
```

以上的表示方法是行为描述，FPGA工具链会帮助转换为门级，一个更加明显的例子是全加器与“+”。



讨论

宏

讨论

时序逻辑

讨论

for-generate loops

讨论

localparam

讨论

数组与向量

讨论

非阻塞赋值

讨论

事件时序控制

参考

https://en.wikipedia.org/wiki/Verilog
https://inst.eecs.berkeley.edu/~eecs151/sp22/files/verilog/Verilog_Primer_Slides.pdf
https://github.com/seldridge/verilog
https://www.chipverify.com/verilog/verilog-case-statement
https://www.chipverify.com/verilog/verilog-timescale
https://verilogguide.readthedocs.io/en/latest/verilog/testbench.html