

Verilog硬件描述语言

高翠芸

School of Computer Science

gaocuiyun@hit.edu.cn

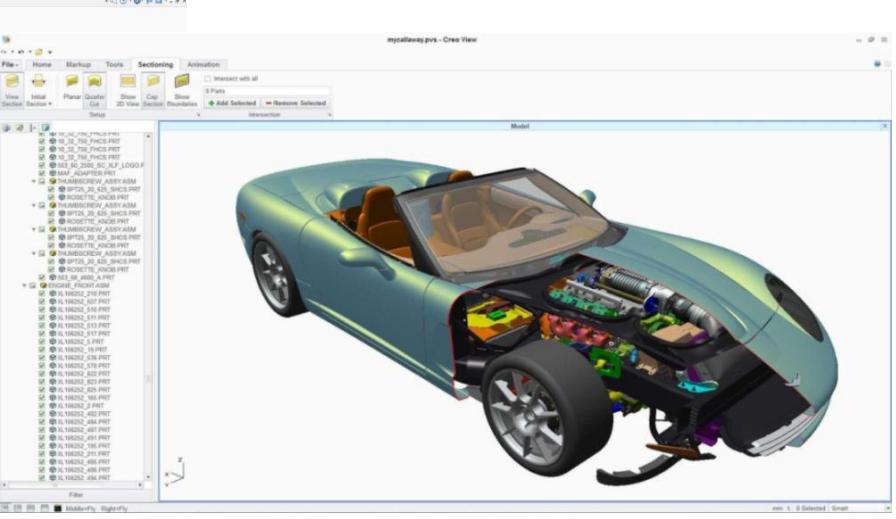
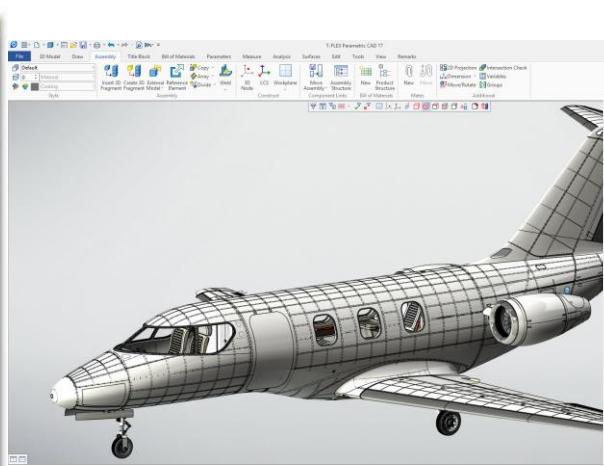
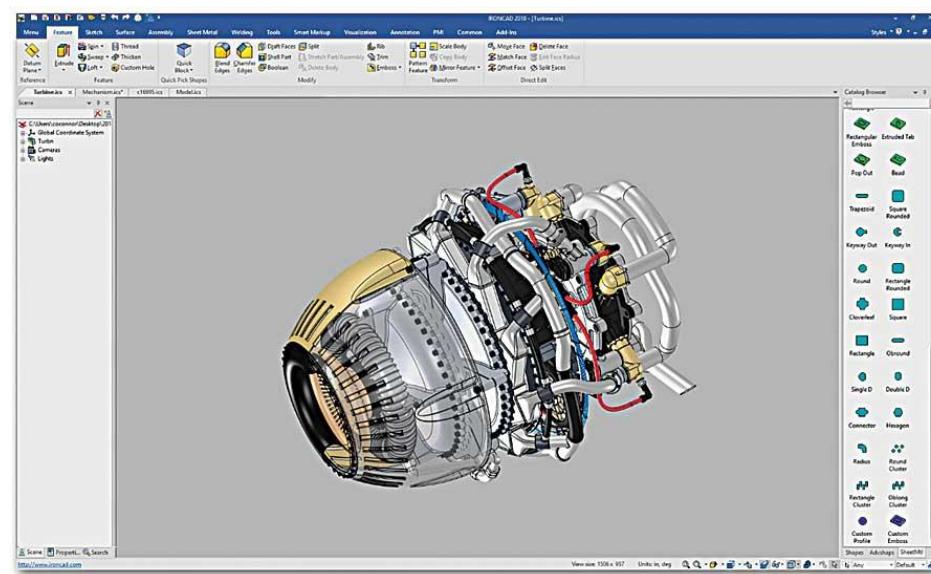
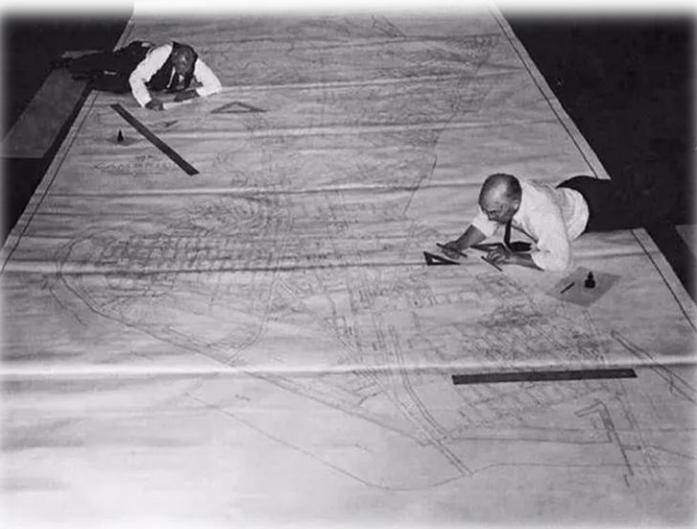
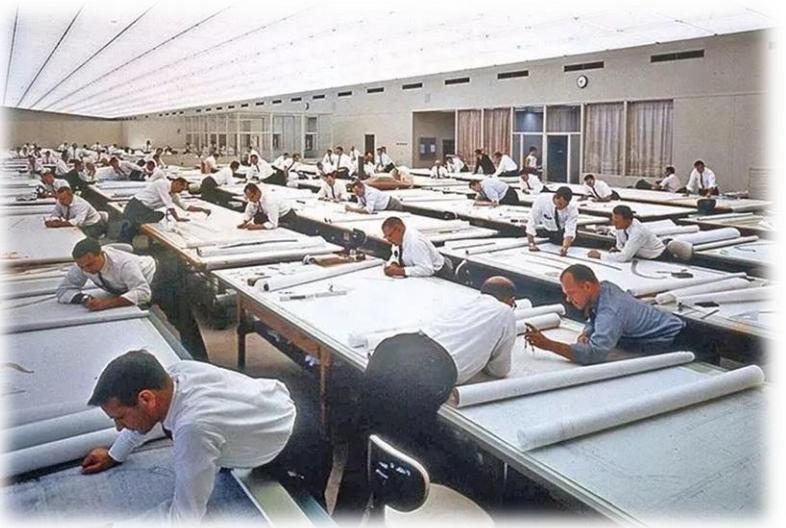
主要内容

- 数字系统设计
 - Verilog HDL基本知识
 - Verilog HDL语法
-
- 方法：
 - 讲义 → 自学 → 实践

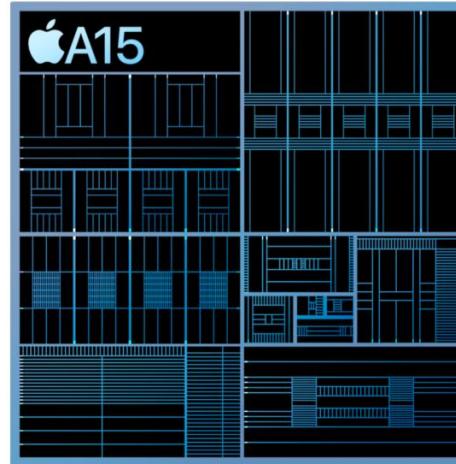
Verilog硬件描述语言

- Verilog HDL概述
- 结构化设计与Verilog模块
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和过程块（组合逻辑电路描述）
- 条件语句和条件表达式
- 模块的测试

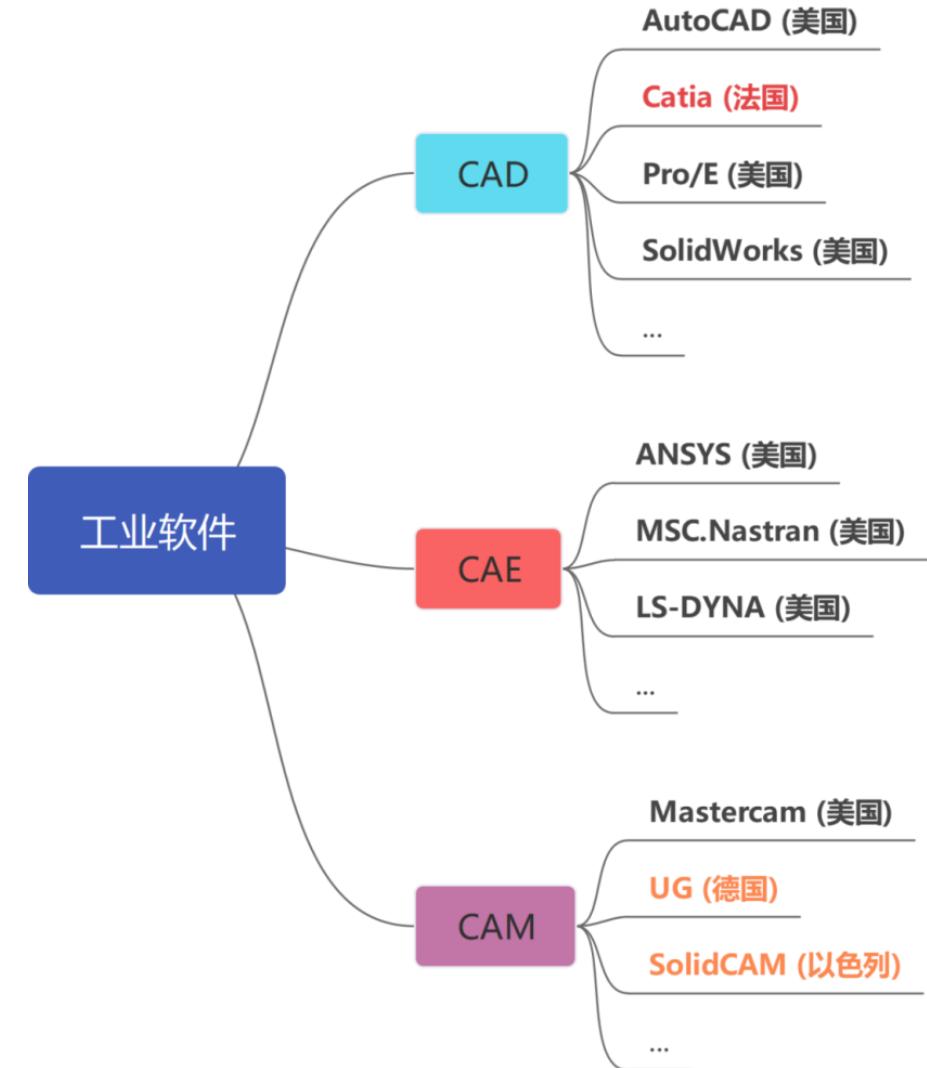
计算机辅助设计 (CAD)



计算机辅助设计



- iPhone 13系列采用最新的A15仿生芯片，指甲盖大小的面积上集成了150亿个晶体管！再说一遍，是150亿，这样的数量让设计人员用手工来画是不可能的。



EDA技术

- EDA (Electronic Design Automation) 电子设计自动化
- 以计算机为工具，在EDA软件平台上，用硬件描述语言完成设计文件，然后由计算机自动地完成逻辑编译、化简、分割、综合、优化、布局、布线和仿真，直至对于特定目标芯片的适配编译、逻辑映射和编程下载等工作。
- 设计者：从概念、算法、协议等设计电子系统
- 计算机：电路设计、性能分析到设计出IC版图或PCB版图

数字设计的发展

电子系统设计自动化**EDA**
(Electronic Design Automation)

计算机辅助工程设计**CAE**
(computer Assist Engineering)

计算机辅助设计**CAD**
(computer Assist Design)



20世纪90年代
电子设计自动化

20世纪80年代
计算机辅助工程

20世纪70年代
计算机辅助设计

行为

行为描述

功能

逻辑综合

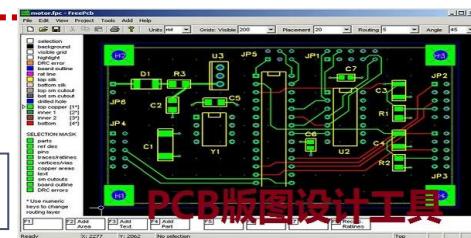
逻辑

布局布线

版图

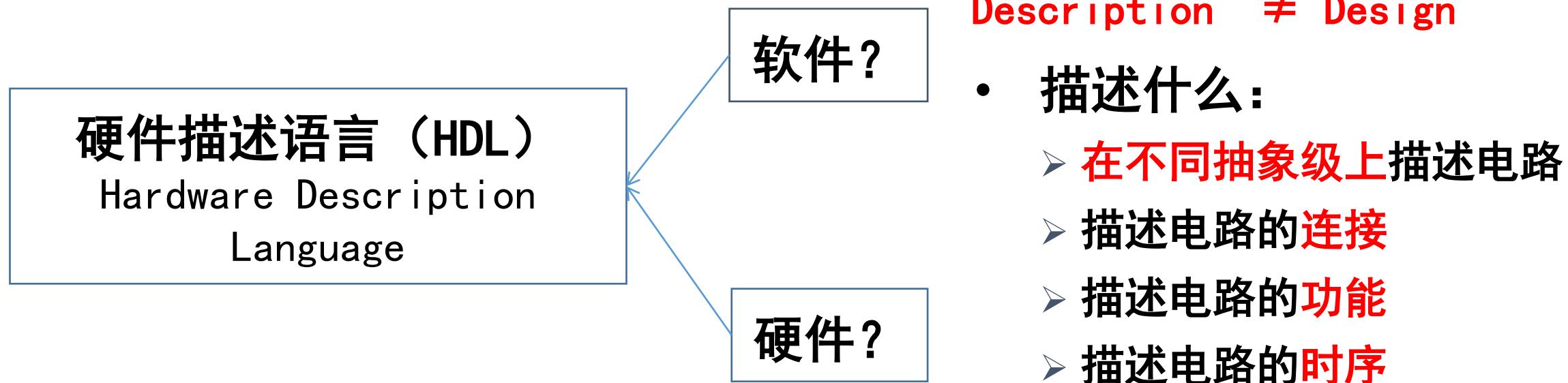
图形生成

掩膜



什么是硬件描述语言

- 具有特殊结构能够对**硬件逻辑电路的功能进行描述**的一种高级编程语言



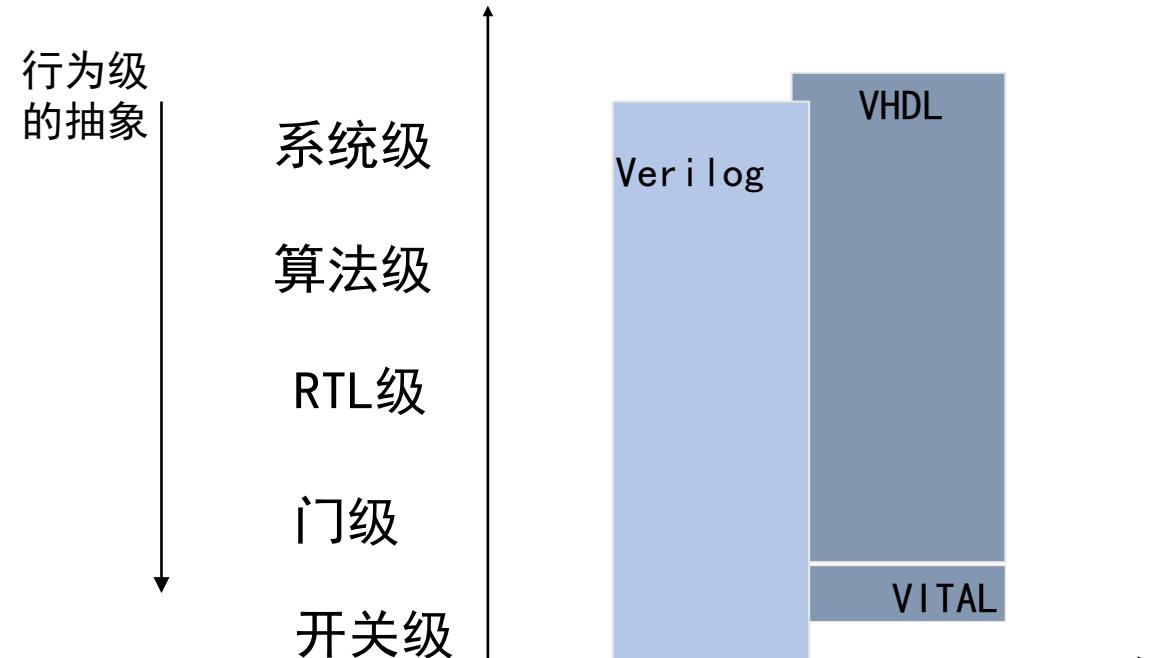
- Verilog HDL**: 一种**硬件描述语言**，以文本形式来描述数字系统硬件的结构和行为的语言，可以表示**逻辑电路图、逻辑表达式**，也可以表示数字逻辑系统所完成的**逻辑功能**。

为什么要使用硬件描述语言

- 电路的逻辑功能容易理解；
- 便于计算机对逻辑进行分析处理；
- 把逻辑设计与具体电路的实现分成两个独立的阶段来操作；
- 逻辑设计与实现的工艺无关；
- 逻辑设计的资源积累可以重复使用；
- 可以由多人共同更好更快地设计非常复杂的逻辑电路（几十万门以上的逻辑系统。）

VHDL vs. Verilog

- VHDL:
 - 起源于ADA语言
 - 侧重于**系统级**描述
 - 含有大量的内置数据类型和用户自定义类型
- Verilog:
 - 起源于C语言
 - 侧重于**电路级**描述
 - 数据类型由语言本身定义，含有专门描述连线等的类型



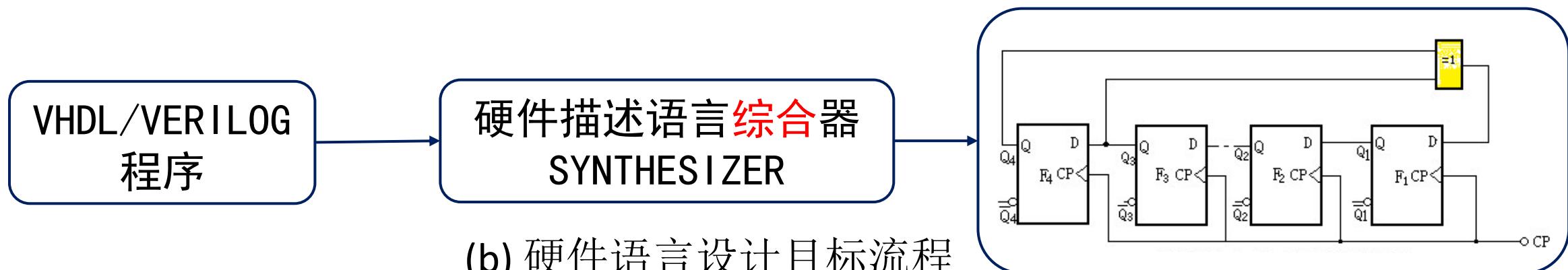
Verilog与VHDL建模能力的比较

Verilog 应用较广泛、起步更容易！

软件描述语言和硬件描述语言的区别



(a) 软件语言设计目标流程



(b) 硬件语言设计目标流程



Verilog语言的层次

• 行为描述语言&结构描述语言

系统级：

用高级语言结构实现设计模块的外部性能的模型；

算法级：

用高级语言结构实现设计算法的模型；

RTL级（寄存器传输级）：

描述数据在寄存器之间流动和如何处理这些数据的模型；

门级：

描述逻辑门以及逻辑门之间的连接的模型；

开关级：

描述器件中三极管和存储节点以及它们之间连接的模型。

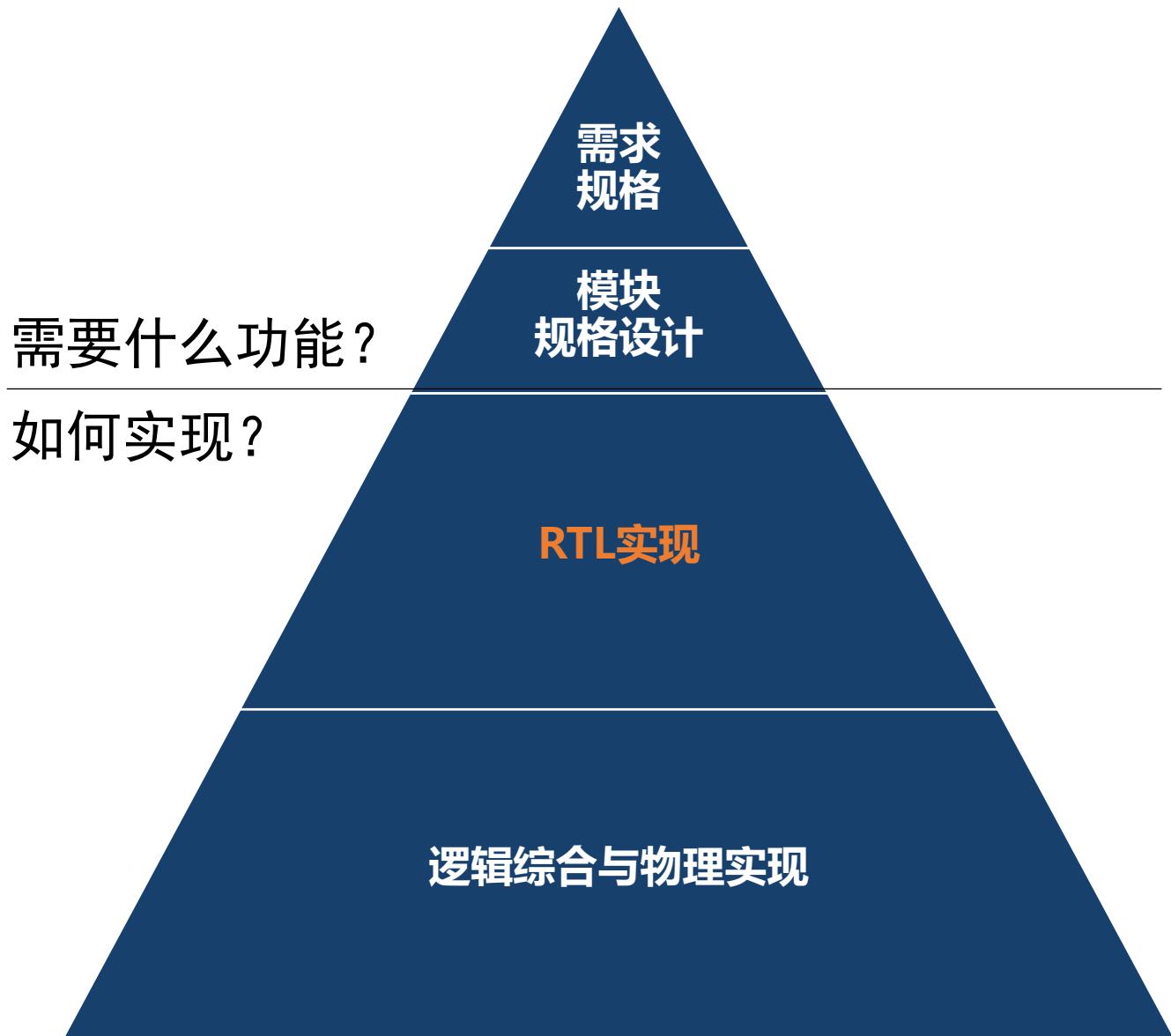
行为级描述

侧重对模块**行为功能**的抽象描述

结构级描述

侧重对模块**内部结构实现**的具体描述

设计层次



系统架构师:

系统级或算法级，描述系统的规格。

逻辑设计工程师:

使用Verilog RTL级代码，可综合，精确到时钟周期。

物理设计工程师:

对门级网表进行布局布线，将其做成实际的芯片。

设计层次



验证工程师：
对设计好的电路进行验证。

描述方式

```
module decoder_38_struct(
    input [2:0] data_in,
    input [2:0] en,
    output reg[7:0] data_out
);

    // 内部连线定义
    wire en_out,en0_out,en1_not;

    // 调用基本库元件
    not not0(en0_out, en[0]);
    not not1(en1_out, en[1]);
    and and0(en_out,en0_out,en1_not,en[2]);
endmodule
```

结构化描述

```
module decoder_38_dataflow(
    input [2:0] data_in,
    input [2:0] en,
    output [7:0] data_out
);

    assign data_out[5]=data_in[2]&&~data_in[1]
        &&data_in[0];
    // 或 assign data_out[5]=(data_in == 3'd5);

    // 其他赋值语句

endmodule
```

数据流描述

```
module decoder_38(
    input [2:0] data_in,
    input [2:0] en,
    output reg[7:0] data_out
);

    always @(*) begin //非完整实现，需要自行补全其他信号处理
        case( data_in )
            3'b000: data_out = 8'b0000_0001;
            3'b001: data_out = 8'b0000_0010;
            3'b010: data_out = 8'b0000_0100;
            3'b011: data_out = 8'b0000_1000;
            3'b100: data_out = 8'b0001_0000;
            3'b101: data_out = 8'b0010_0000;
            3'b110: data_out = 8'b0100_0000;
            3'b111: data_out = 8'b1000_0000;
        endcase
    endmodule
```

行为描述

结构化描述：通过调用库中的元件或已设计好的模块来完成设计。

数据流描述：主要使用assign连续赋值语句，多用于组合逻辑电路。

行为描述：从电路的功能出发，关注逻辑电路输入、输出的因果关系。

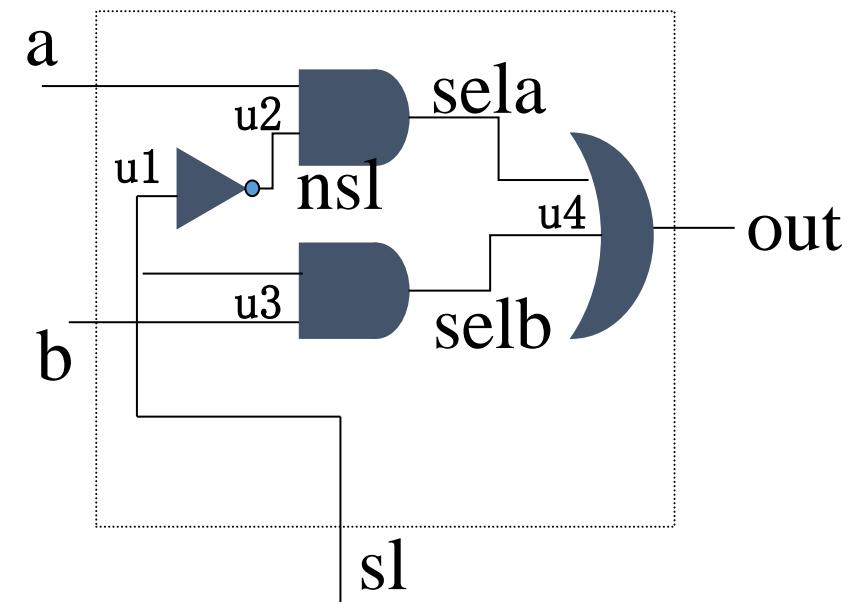
即在何种输入条件下产生何种输出，描述的是一种行为特性。

在较复杂的电路设计中，三种描述方法往往混合使用。

结构级Verilog HDL

- Verilog内部带有描述基本逻辑功能的基本单元(primitive)，如and门。
- 综合产生的结果网表通常是结构级的。
- 结构级Verilog适合开发小规模元件

```
module muxtwo (out, a, b, sl); //二选一多路选择器
  input a, b, sl;
  output out;
  not u1 (nsl, sl); //nsl=~sl
  and #1 u2 (sela, a, nsl); //sela=a&nsl
  and #1 u3 (selb, b, sl); //selb=b&sl
  or #2 u4 (out, sela, selb); //out=sela|selb
endmodule
```

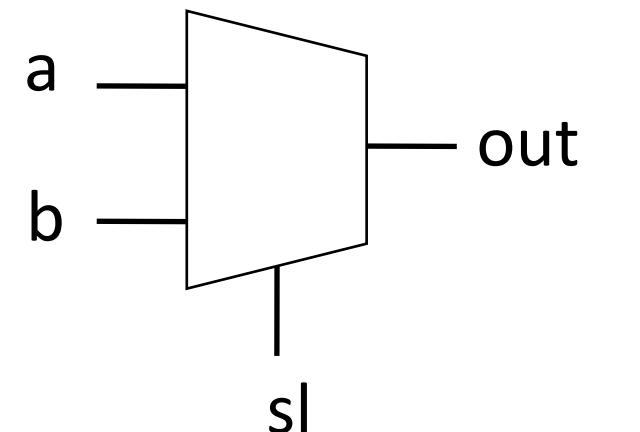


二选一多路选择器

行为级Verilog HDL

- 在行为级模型中，逻辑功能描述采用高级语言结构，如@、while、wait、if、case。
- RTL模块是可综合的，它是行为模块的一个子集合。
- 行为级Verilog

```
module muxtwo (out, a, b, sl); //二选一多路选择器
  input a, b, sl; //输入信号名
  output out; //输出信号名
  reg out;
  always @( sl or a or b)
    if (! sl) out = a; //控制信号sl为非，输出与输入信号a一致
    else out = b; //控制信号sl为非，输出与输入信号b一致
endmodule
```



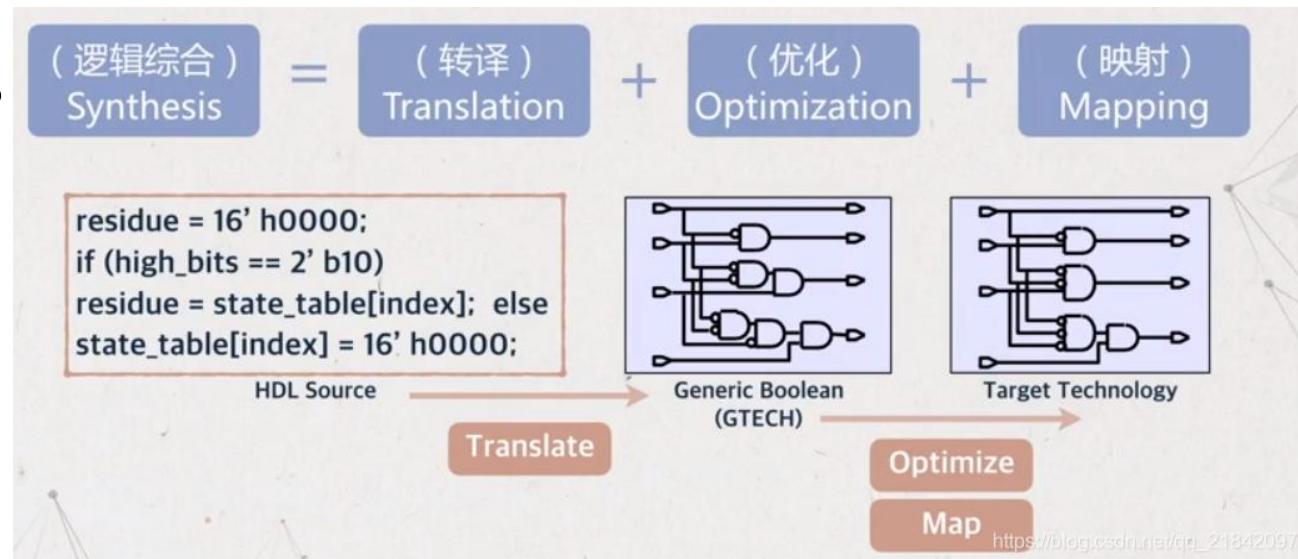
二选一多路选择器

RTL与综合

- **综合：**

将**HDL语言、原理图**等设计输入翻译成由与、或、非等基本逻辑单元组成的门级连接，并根据设计目标和要求优化所生成的逻辑链接，输出**门级网表文件**。

- **RTL级语言最重要的特性就是可综合。**



- **典型的RTL级设计包括：**

组合逻辑描述、时序逻辑描述、时钟域描述

学生的总结

- **Verilog是硬件描述语言**，用语言描述的方式进行电路设计，最终要实现出硬件电路。`verilog`只是简化了电路设计的工作量，本质上就是设计数字电路，永远绕不开电路这点！
- 评价一个`verilog`代码的好坏是看**最终实现的功能和性能**。合理的设计方法是首先理解要设计的电路，也就是把需求转化为数字电路，对此电路的结构和连接十分清晰，然后再用`verilog`表达出这段电路。
- 要理解硬件的并行性，**搞清楚时序关系**，一定要摒弃顺序执行的思维。
- 建议`Verilog`语言采用“学、用、查”的方式，“用”着“用”着就会了。

初学Verilog的几点提醒

要知道 verilog 是硬件描述语言，它应该是用于描述电路的。但是它的语法较为宽松，以至于将其用于描述算法时也能符合语法规则。但是如果这么做 vivado 会不开心，因为这样做出来的电路往往是不可综合的。综合，简单的说就是按照代码中你对你所设计的电路的描述，生成你所期望的电路。但是如果你一开始脑海里就没有过任何电路，你用 verilog 描述了一个算法，然后扔给 vivado：“给爷综！”那你也太难为人家了。

有的读者可能会疑问：电路是什么呢？怎样才叫设计了一个电路呢？这就回归到数字逻辑设计课程理论课部分的内容了。这么说吧，电路就是由逻辑门，锁存器，触发器，导线等组成的一个。。。呃。。。东西。设计电路，就是画出一张电路图，可以用软件画，也可以在纸上画，或者更随便一点，在脑海里画。在哪里画并不重要，重要的是你设计出来的真的是一个电路，而不是算法。

Verilog语言中只有很少一部分是用于设计电路的。

Verilog HDL与C语言的最大区别

- Verilog HDL语言是**并行的**，即具有在同一时刻执行多任务的能力，因为在实际硬件中许多操作都是在同一时刻发生的。一般来讲，**计算机编程语言是非并行的**。
- Verilog HDL语言**有时序**的概念，因为在硬件电路中从输入到输出**总是有延迟存在的**。
- 阻塞与非阻塞赋值！

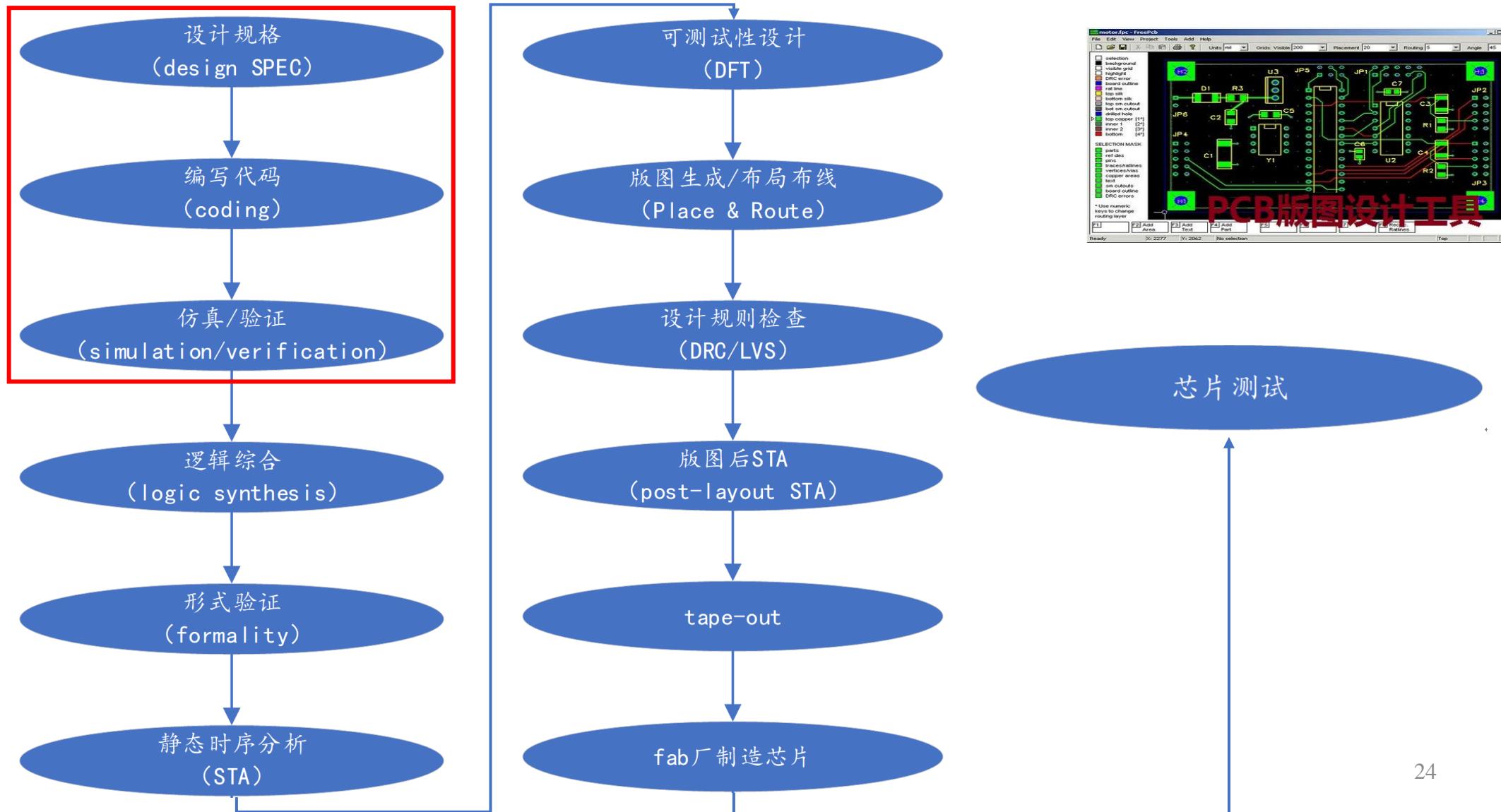


注意

Verilog硬件描述语言

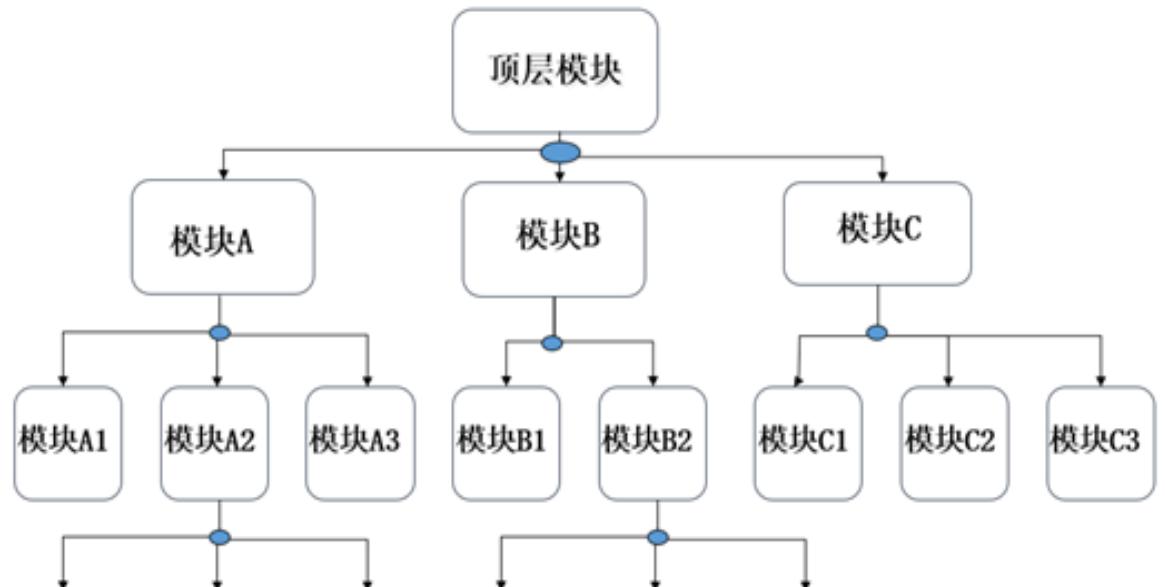
- 概述
- 结构化设计与Verilog模块
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和过程块（组合逻辑电路描述）
- 条件语句和条件表达式
- 模块的测试

数字集成电路开发流程



Verilog的设计方法-自顶向下的结构化设计

- 从顶层模块开始，先确定好顶层模块的输入输出和内部的逻辑功能，然后逐层分解成更小的功能模块，总体的设计步骤就是“**自顶向下、模块划分、逐层细化**”，直到子模块的功能较为纯粹、单一。最终采用Verilog语言直接描述硬件行为，由逻辑综合工具自动完成从HDL到门级电路的转换。

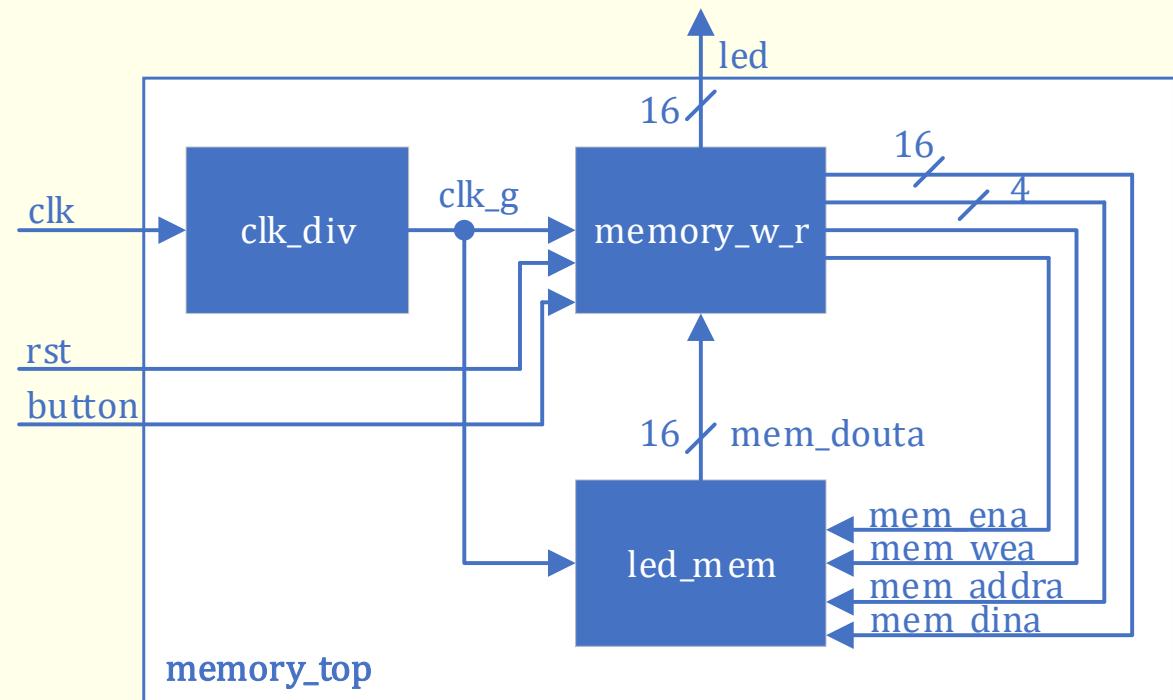


- **模块划分**
- **功能说明**

描述实现的功能，框图展示系统的输入、输出、功能模块、内部数据通路和重要的控制信号（包括必要的说明）。
- **设计代码**

设计框图

- 展示系统的输入、输出、功能模块及说明、内部数据通路及重要的控制信号。



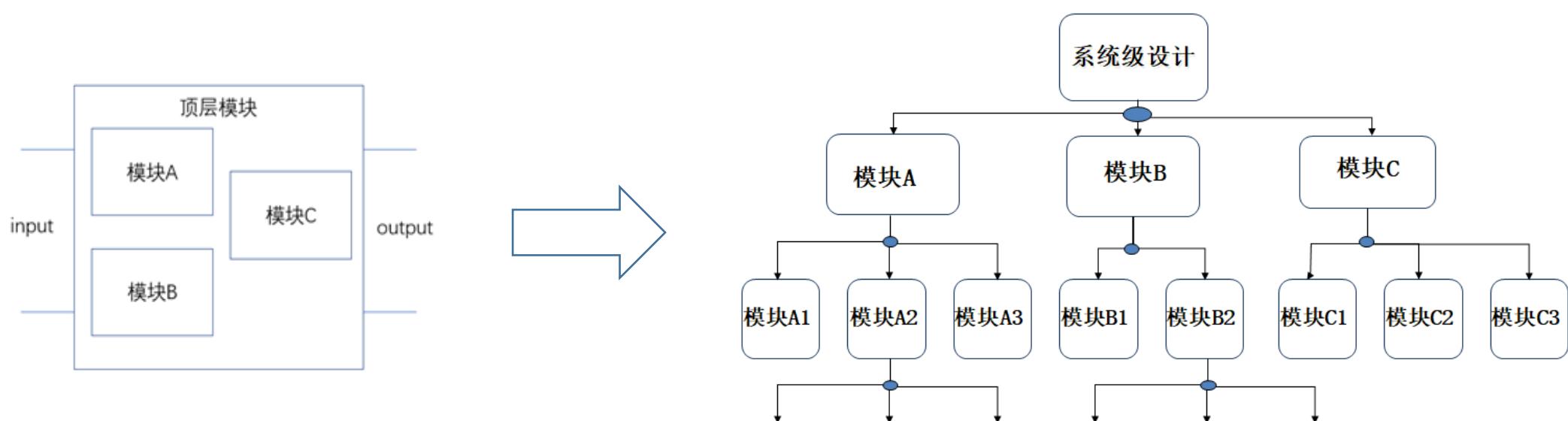
- 系统分为3个模块：
- `clk_div`: 用于时钟分频；
- `led_mem`: 用于存储LED灯依次显示的序列；
- `memory_w_r`: 用于控制存储器的读写，以及将读取回来的序列，显示到LED上；

流水灯框图

从“模块”开始搭电路-模块定义

- Verilog的基本设计单元是模块（module），模块的实例化建立了模块描述的层次，通过模块端口的连接，把下层模块连接到了上层模块中。
- 模块包括**接口描述**和**逻辑功能描述**两部分。（**电路外特性**）
- 模块定义包括4个主要部分：

端口定义、I/O说明、内部信号声明和功能定义，在module**和**endmodule** 之间。**



从“模块”开始搭电路-模块定义

- **module module_name(**

- input wire [width-1:0] port_name1,**

- input wire [width-1:0] port_name2,**

- output wire/reg [width-1:0] port_name3**

-); //在模块名后面的()中做端口定义和I/O说明**

//内部信号声明

wire [width-1:0] name1;

reg [width-1:0] name2;

//功能定义

//用assign语句和always块描述

endmodule

```
module some_module (
    input wire clk ,
    input wire rst_n,
    input wire sel ,
    input wire [3:0] addr ,
    output reg data
);
    //模块代码
endmodule
```

模块基本结构

模块声明：

module module_name (port_list) ; //模块名（端口声明列表）

端口定义：

input[信号位宽]; //输入声明

output [信号位宽]； //输出声明

...

数据类型说明：

reg [信号位宽]； //寄存器类型声明

wire [信号位宽]； //线网类型声明

parameter; //参数声明

...

功能描述： //主程序代码

assign a=b+c

always@(posedge clk or negedge reset)

function

task

...

endmodule

模块声明

module module_name (port_list); //模块名（端口声明列表）

- “模块名”是模块唯一的标识符，区分大小写。
- “端口列表”是由模块各个输入、输出和双向端口组成的列表。
(input,output,inout)
- 端口用来与其它模块进行连接，括号中的列表以“,”来区分，列表的顺序没有规定，先后自由。

module muxtwo (out, a, b, sl);

```
module muxtwo (out, a, b, sl); //二选一多路选择器
    input a, b, sl;
    output out;
    reg out;
    always @( sl or a or b)
        if (! sl) out = a;
        else out = b;
endmodule
```

端口定义

input[信号位宽]; //输入声明
output [信号位宽]; //输出声明
inout[信号位宽]; //输入/输出端口

input a, b, sl; //输入信号名
output out; //输出信号名

```
module muxtwo (out, a, b, sl); //二选一多路选择器
    input a, b, sl; //输入信号名
    output out; //输出信号名
    reg out;
    always @( sl or a or b )
        if (! sl) out = a;
            //控制信号sl为非，输出与输入信号a一致
        else out = b;
            //控制信号sl为非，输出与输入信号b一致
    endmodule
```

- **输入端口：** 模块从外界读取数据的接口，是连线类型
- **输出端口：** 模块向外界传输数据的接口，是连线或寄存器型
- **输入输出端口：** 可读取数据也可接收数据的端口，数据是双向的，是连线型
- 端口定义也可以写在端口声明的位置：

```
module module_name(input port1,input port2,...output port1,...);
```

数据类型说明

```
reg [信号位宽]; //寄存器类型声明  
wire [信号位宽]; //线网类型声明  
parameter; //参数声明
```

reg out; //输出信号**reg**类型

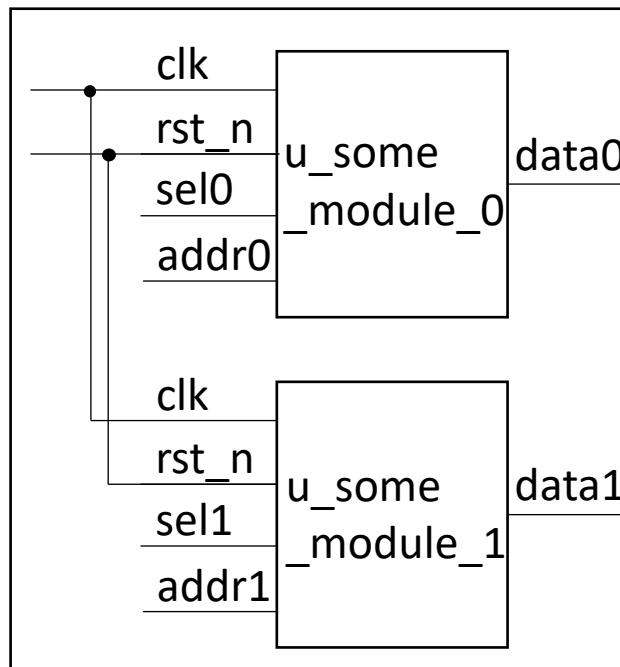
```
module muxtwo (out, a, b, sl); //二选一多路选择器  
input a, b, sl; //输入信号名  
output out; //输出信号名  
reg out;  
always @( sl or a or b)  
  if (! sl) out = a;  
    //控制信号sl为非，输出与输入信号a一致  
  else out = b;  
    //控制信号sl为非，输出与输入信号b一致  
endmodule
```

- 模块中用到的所有信号都必须进行数据类型的定义。
- 声明变量的数据类型后，不能再进行更改
- 在VerilogHDL中只要在使用前声明即可
- 声明后的变量、参数不能再次重新声明
- 声明后的数据使用时的配对数据必须和声明的数据类型一致

从“模块”开始搭电路-模块实例化

- 一个模块在另外一个模块中实例化，等效于实际电路中加入了被实例化的电路。
- 模块实例应用u_x_x表示（多次例化用序号0、1、2等表示）；

```
module_name instance_name(  
    .port_name1(data_name1),  
    .port_name2(data_name2),  
    .....);
```



```
wire      sel0 ;  
wire      sel1 ;  
wire [3:0] addr ;  
wire      data0;  
wire      data1;  
  
some_module u_some_module_0 (  
    .clk      (clk ),  
    .rst_n   (rst_n),  
    .sel     (sel0 ),  
    .addr    (addr ),  
    .data    (data0)  
);  
  
some_module u_some_module_1 (  
    .clk      (clk ),  
    .rst_n   (rst_n),  
    .sel     (sel1 ),  
    .addr    (addr ),  
    .data    (data1)  
);
```

Verilog硬件描述语言

- 概述
- 结构化设计与Verilog模块
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和过程块（组合逻辑电路描述）
- 条件语句和条件表达式
- 模块的测试

标识符

- 赋给对象的唯一名称，可以是字母、数字、下划线和符号“\$”的组合，且首字符只能是字母或者下划线。
- 大小写敏感。
- 注释有两种：
 - 以 “/*” 开头，以 “*/” 结束。 /*这是注释*/
 - 以 “//” 开头到本行结束。 //这也是注释

数据类型

- 共有19种数据类型，分为物理数据类型和抽象数据类型
- 物理数据类型：与实际硬件电路有明显的映射关系
 - `wire`(连线型)、`reg`(寄存器型)、`memory`(存储器型)等
- 抽象数据类型：用于进行辅助设计和验证的数据类型
 - 整型`integer`、时间型`time`、实型`real`、参数型`parameter`等
- 数据类型还可分为常量和变量
 - 常量：数字、参数型`parameter`
 - 变量：`wire`(连线型)、`reg`(寄存器型)、`(memory)`存储器型等

常量—数字

- 整数：
 - 二进制(b或B)、十进制(d或D)、十六进制(h或H)、八进制(o或O)
 - 表达方式：
 - <位宽><进制><数字>
 - `8'b10101100` //位宽为8的数的**二进制**表示，'b表示**二进制**
 - `8'ha2` //位宽为8的数的**十六进制**表示，'h表示**十六进制**
 - 没有数位宽采用**默认位宽**（这由具体的机器系统决定，但至少32位）
 - 在<数字>这种描述方式中，采用默认进制（十进制）

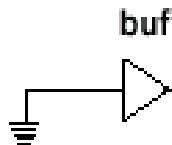
数字

- 下列表达式的位模式是什么？写出其具体的二进制表示值
 - 7'o44,
 - 'Bx0,
 - 5'bx110,
 - 'hA0,
 - 10'd2,
 - 'hzF

Verilog描述	实际的二进制值
7'o44	100100
'Bx0	x0
5'bx110	xx110
'hA0	10100000
10'd2	0000000010
'hzF	zzzz1111

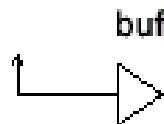
四种逻辑值

- Verilog语言规定了4种基本的逻辑值



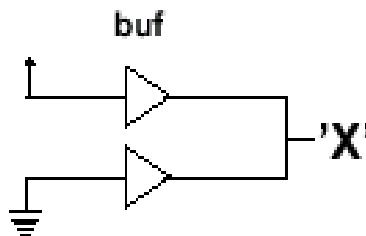
'0'

低电平、逻辑0、“假”、接地

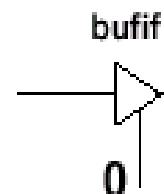


'1'

高电平、逻辑1、“真”



不确定或未知的逻辑状态

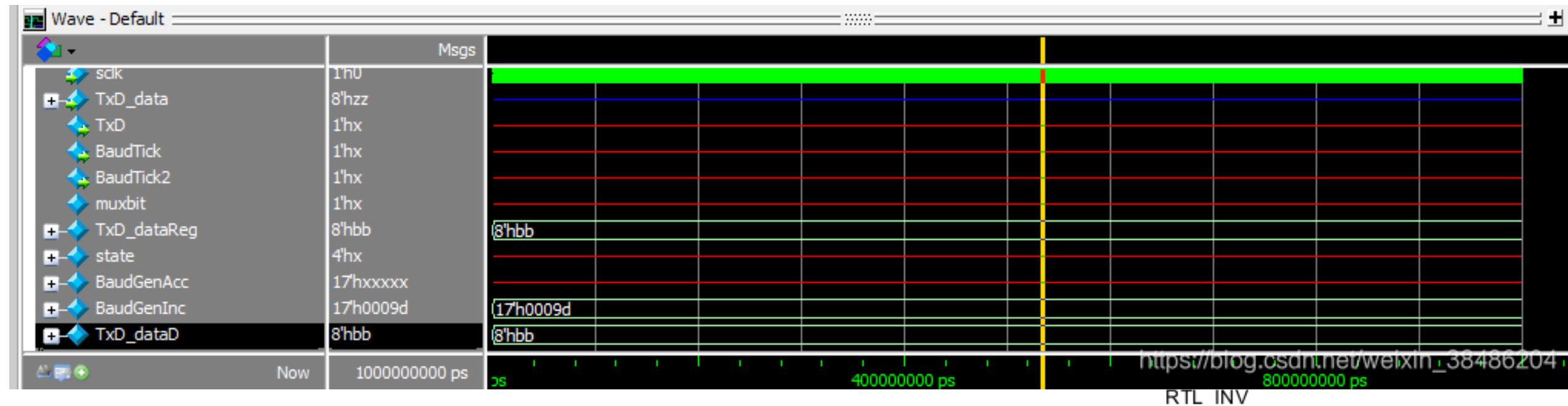


'z'

高阻态（一般情况，电路分析时可做开路理解。）

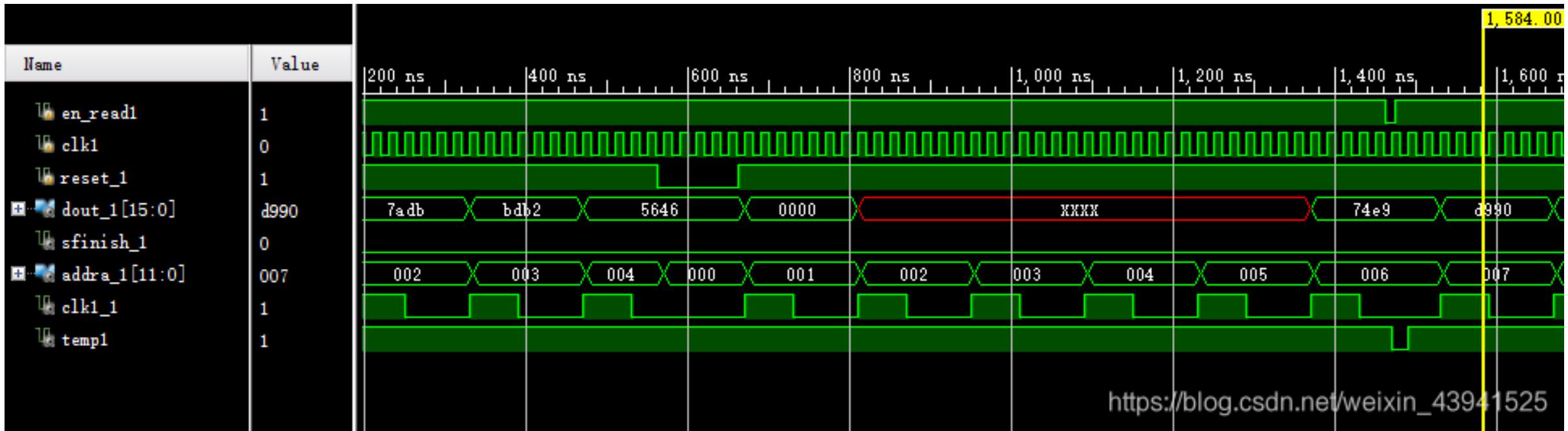
wire型

- 硬件电路中元件之间实际连线的抽象



- 当一个wire 类型的信号没有被驱动时，缺省值为Z(高阻)。
- 信号没有定义数据类型时，缺省为 wire 类型。

reg型



- reg类型在reset的时候赋初值，没有赋值的情况下默认为不关心。
- reg型和wire型的区别：
 - reg型保持最后一次赋值
 - wire型需要持续的驱动

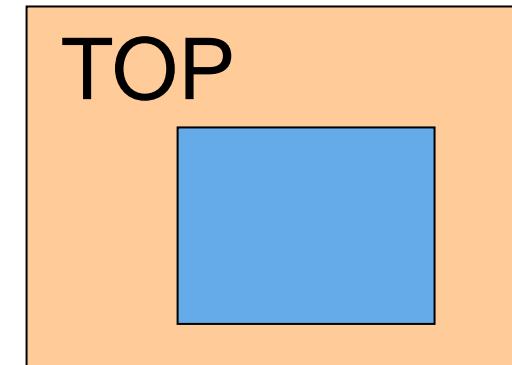
参数型 (parameter)

- 定义格式：
 - parameter 参数名1=表达式1, ..., 参数名n=表达式n;
 - parameter length=32, weight=16; //定义了两个参数
 - 其中:表达式既可以是常数，也可以是表达式。
 - 参数定义完后，程序中所有的参数名将被替换为相应的表达式。
- 属于常量，常用来定义**延迟时间**和**变量的位宽**。
- 在模块或实例引用时，可通过参数传递改变在被引用模块或实例中已定义的参数。

参数型 (parameter)

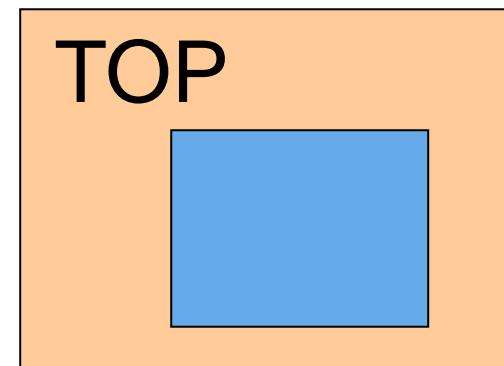
```
module TOP (NewA, NewB, NewS, NewC);
    input NewA, NewB;
    output NewS, NewC;
    defparam Ha1.OR_DELAY=5, //实例Ha1中的参数OR_DELAY。
        Ha1.AND_DELAY=2; //实例Ha1中的参数AND_DELAY。
    HA Ha1(NewA, NewB, NewS, NewC);
endmodule
```

```
module HA(A,B,S,C);
    input A,B;
    output S,C;
    parameter AND_DELAY=1,
        OR_DELAY=2;
    assign #OR_DELAY S=A+B;
    assign #AND_DELAY C=A&B;
endmodule
```



参数型 (parameter)

```
module TOP3(NewA,NewB,NewS,NewC);
    input NewA, NewB;
    output NewS,NewC;
    HA #(5,2) Ha1 (NewA, NewB, NewS, NewC);
    //第1个值5赋给参数AND_DELAY，该参数在模块HA中说明。
    //第2个值2赋给参数OR_DELAY，该参数在模块HA中说明。
endmodule
```



数据类型声明

```
reg [width-1:0] 变量名1, 变量名2;      //reg型声明
```

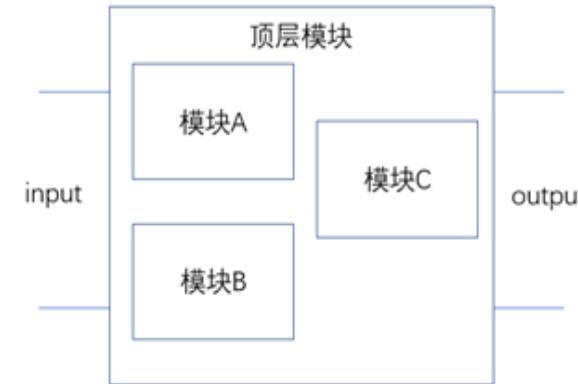
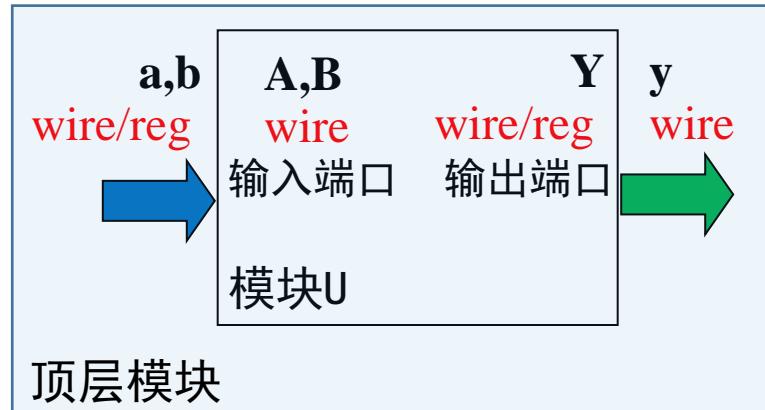
```
wire [width-1:0] 变量名1, 变量名2;      //wire型声明
```

```
parameter 变量名1=表达式1, 变量名2=表达式2 ;      //参数声明
```

- 模块中用到的所有信号都必须进行数据类型的声明。
- 在Verilog HDL中变量和参数只要在使用前声明即可。
- 声明变量的数据类型后，不能再更改（不能再次重新声明）。
- 声明后的数据使用时的配对数据必须和声明的类型一致。

如何选择正确的数据类型？

- 端口：(input、output)
- 可以由reg或wire连接驱动，但它本身只能驱动wire连接。



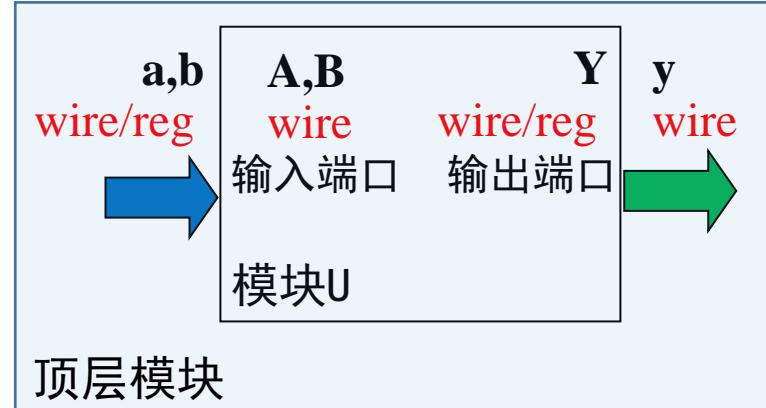
- 变量：
 - assign赋值语句左侧数据类型，必须是wire类型；
 - 过程块（always块 或 initial块）中被赋值的左侧数据类型，必须把它声明为reg类型变量。

如何选择正确的数据类型？

模块U：

```
module U(  
output wire Y,  
input wire A, B);
```

//功能描述
endmodule



顶层模块： module top(

端口定义
);

```
wire y;  
reg a, b;
```

```
U u1(  
.Y(y),  
.A(a),  
.B(b)  
);
```

//其他逻辑
endmodule

模块实例化不是参数传递

模块端口与外部信号按照其名字进行连接

如何选择正确的数据类型？

- 在过程块 (always initial) 中对变量赋值时，忘了把它定义为寄存器类型 (reg) 或已把它定义为连接类型了 (wire) 
- 把实例的输出连接出去时，把它定义为寄存器类型 
- 把模块的输入信号定义为寄存器类型。 
 - 这是经常犯的三个错误！！！

Verilog硬件描述语言

- 概述
- 结构化设计与Verilog模块
- 标识符和数据类型
- **运算符及表达式**
- 赋值语句和过程块（组合逻辑电路描述）
- 条件语句和循环语句
- 模块的测试

运算符

- 按功能分为以下几类：

① 算术运算符： +, -, *, /, %

② 逻辑运算符： && , || , !

③ 位运算符： ~, |, ^, &, ^~

④ 赋值运算符： =, <=

⑤ 关系运算符： >, <, >=, <=

⑥ 条件运算符： ? :

⑦ 移位运算符： <<, >>

⑧ 拼接运算符： { }

⑨ 等式运算符： ==, !=, ===, !==

运算符—续

- **算术运算符**： +, -, *, /, %

- 在进行整数的除法运算时，结果要略去小数部分，只取整数部分；
- 进行取模运算时（%，亦称求余运算符），结果的符号位采用模运算符中第一个操作数的符号。
- 在进行算术运算时，如有一个操作数为不确定值x，则结果也为不确定值x。

- **逻辑运算符**： && , || , !

- &&和||是双目运算符，优先级别低于关系运算符，而！高于算术运算符。

- **位运算符**： ~, |, ^, &, ^~

- 在不同长度的数据进行位运算时，系统会自动的将两个数右端对齐，位数少的操作数会在相应的高位补0，两个操作数按位进行操作。

运算符—续

- **关系运算符:** > , < , >= , <=

- 如果关系运算是假的，则返回值是0，如果关系是真的，则返回值是1。
- 关系运算符的优先级别低于算数运算符。如： $a < \text{size}-1$ 等同于 $a < (\text{size}-1)$
- 如果某个操作数值不定，则关系是模糊的，返回值是不定值。

- **移位运算符:** << , >>

- $a >> n$ 其中a代表要进行移位的操作数，n代表要移几位。这两种移位运算都用0来填补移出的空位。如果操作数已经定义了位宽，则进行移位后操作数改变，但是其位宽不变。

- **拼接运算符 :** { }

- {信号1的某几位, 信号2的某几位, 信号n的某几位} 将某些信号的某些位列出来，中间用逗号分开，最后用大括号括起来表示一个整体的信号。在位拼接的表达式中不允许存在没有指明位数的信号。

- {a, b[3:0], w} //等同于 {a, b[3], b[2], b[1], b[0], w}

运算符—续

拼接运算符 : { }

实例

```
A = 4'b1010 ;
B = 1'b1 ;
Y1 = {B, A[3:2], A[0], 4'h3 } ; //结果为Y1='b1100_0011
Y2 = {4{B}, 3'd4} ; //结果为 Y2=7'b111_1100
Y3 = {32{1'b0}}; //结果为 Y3=32h0, 常用作寄存器初始化时匹配位宽的赋初值
```

条件运算符 : ? :

实例

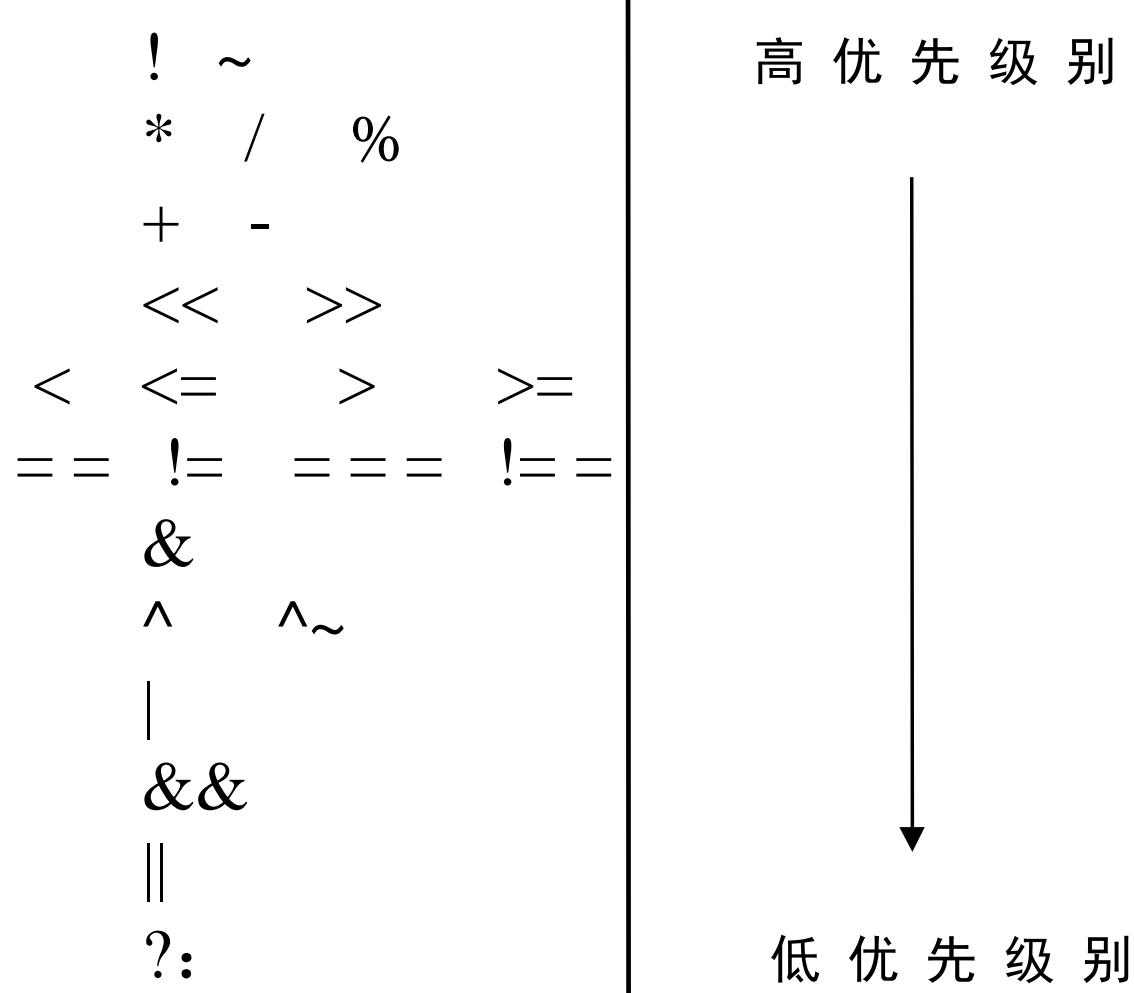
```
assign hsel = (addr[9:8] == 2'b00) ? hsel_p1 :
              (addr[9:8] == 2'b01) ? hsel_p2 :
              (addr[9:8] == 2'b10) ? hsel_p3 :
              (addr[9:8] == 2'b11) ? hsel_p4 ;
```

运算符—续

- 等式运算符: ==, !=, ===, !==
 - ==, !=: X和Z进行比较时为X
 - ===, !==: 操作数相同结果为1, 常用于case表达式的判别。

====	0	1	X	Z	=====	==	0	1	X	Z
0	1	0	0	0	=====	0	1	0	X	X
1	0	1	0	0	=====	1	0	1	X	X
X	0	0	1	0	=====	X	X	X	X	X
Z	0	0	0	1	=====	Z	X	X	X	X

运算符优先级

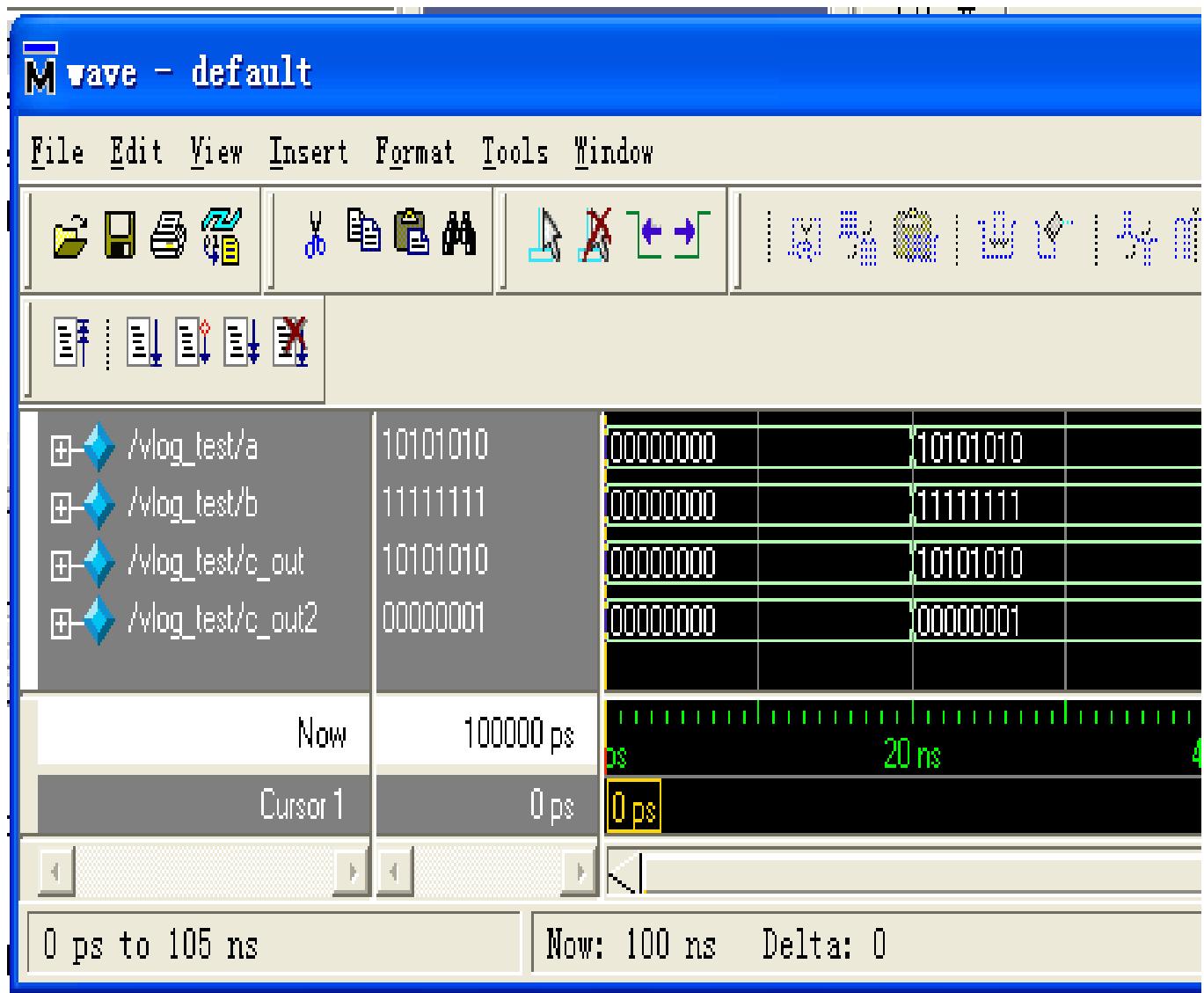


逻辑门的描述

	Verilog描述	逻辑表达式
与门	$F = A \& B;$	$F=AB$
或门	$F = A B;$	$F=A+B$
非门	$F = \sim A;$	$F=A'$
与非门	$F = \sim (A \& B);$	$F=(AB)'$
或非门	$F = \sim (A B);$	$F=(A+B)'$
与或非门	$F = \sim((A \& B) (C \& D));$	$F=(AB+CD)'$
异或门	$F = (A \wedge B);$ $F = (\sim A \& B) (A \& \sim B);$	$F=A \oplus B$ $F=A'B+AB'$
同或门	$F = (A \wedge \sim B);$ $F = \sim (A \wedge B);$ $F = (\sim A \& \sim B) (A \& B);$	$F=(A \oplus B)'$ $F=A'B'+AB$

逻辑门的描述

```
assign a=8'haa;  
assign b=8'hff;  
assign  
c_out=a&b;  
  
assign  
c_out2=a&&b;
```



Verilog硬件描述语言

- 概述
- 结构化设计与Verilog模块
- 标识符和数据类型
- 运算符及表达式
- 赋值语句 和过程块（组合逻辑电路描述）
- 条件语句和循环语句
- 模块的测试

赋值语句

- 在Verilog中，变量是不能随意赋值的，需要使用**连续赋值语句**或**过程赋值语句**。
- assign称为**连续赋值**；
- 过程块initial或always中的赋值称为**过程赋值**。

连续赋值语句

- 语法格式： assign 变量名 (wire类型) =赋值表达式；

例：

```
    wire [7:0] a , b , c, d;  
    assign a = 8' b00001111;  
    assign b = a;  
    assign c[3:0] = d[3:0];  
    assign d = ^a&b;
```

- 对应到电路中 → 导线；
- 左侧数据类型必须是**wire型数据**；
- 描述组合电路，每条**assign赋值语句相当于一个逻辑单元**；
- 右侧任何信号的变化都会激活该语句，使其被立即执行一次，所有右值都是敏感信号；
- 各个**assign赋值语句之间是并发的关系**；
- 在过程块（initial/always）外面。

连续赋值语句

● 变量（标量）

```
wire a , b ;
```

```
assign a = b ;
```

● 向量中某一位

```
wire [7:0] a , b ;
```

```
assign a[3] = b[3] ;
```

● 向量

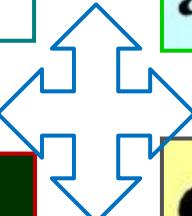
```
wire [7:0] a , b ;
```

```
assign a = b ;
```

● 向量中某几位

```
wire [7:0] a , b ;
```

```
assign a[3:2] = b[3:2] ;
```

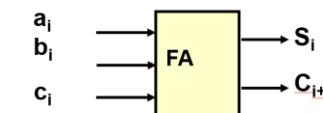
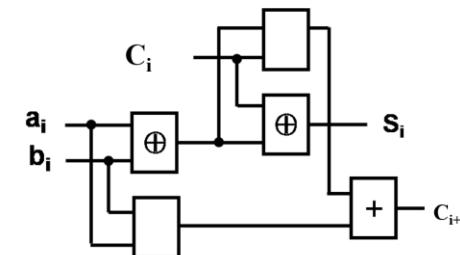


连续赋值语句举例

```
module FA_Df(A, B, Cin, Sum, Cout) ;  
input A, B, Cin;  
output Sum, Cout ;  
  
assign Sum = A ^ B ^ Cin;  
assign Cout = (A & Cin) | (B & Cin) | (A & B) ;  
endmodule
```

solution 1:

$$\begin{cases} S_i = a_i \oplus b_i \oplus C_i \\ C_{i+1} = (a_i \oplus b_i) C_i + a_i b_i \end{cases}$$



在本例中，有两个连续赋值语句。这些赋值语句是并发的，与其书写的顺序无关

连续赋值语句

```
wire [7:0] wire_a;  
wire [7:0] wire_b;  
reg  [7:0] reg_a;  
wire [7:0] wire_y1;  
wire [7:0] wire_y2;  
wire [7:0] wire_y3;  
wire [7:0] wire_y4;  
wire [7:0] wire_y5;  
wire [7:0] wire_y6;
```

一些常用的基础逻辑门

```
assign wire_b = wire_a;  
assign wire_a = reg_a;  
assign wire_y1 = ~wire_a;    //取反  
assign wire_y2 = wire_a & wire_b;    //与  
assign wire_y3 = wire_a | wire_b; //或  
assign wire_y4 = wire_a ^ wire_b; //异或  
assign wire_y5 = ~(wire_a & wire_b); //与非  
assign wire_y6 = ~(wire_a | wire_b); //或非
```

过程赋值语句

- 只能出现在过程块中 (initial/always) ;
- 过程赋值语句中：**没有关键词“assign”**；
- 左侧数据类型**必须是reg类型的变量**；
- 每条过程赋值语句之间是**顺序执行**的关系。
- 包括**阻塞赋值**（运算符=）和**非阻塞赋值**（运算符<=）

阻塞赋值与非阻塞赋值

- 阻塞 (Blocking) 赋值方式

- 符号 “`=`”， 如`b = a ;`
- 在同一个`always` 中，一条阻塞赋值语句如果没有执行结束，那么该语句后面的语句就不能被执行，即被“阻塞”。
- 使用在触发事件为电平敏感信号的`always`块中，描述组合逻辑电路

- 非阻塞赋值 (Non-Blocking) 赋值方式

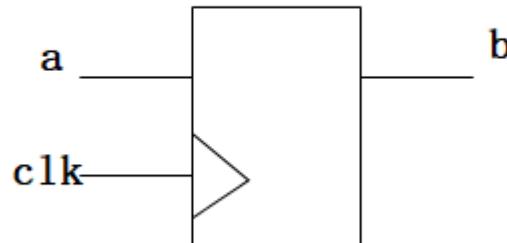
- 符号 “`<=`”， 如`b <= a ;`
- 使用在触发事件为时钟边沿的`always`块中，描述时序逻辑电路

阻塞赋值与非阻塞赋值

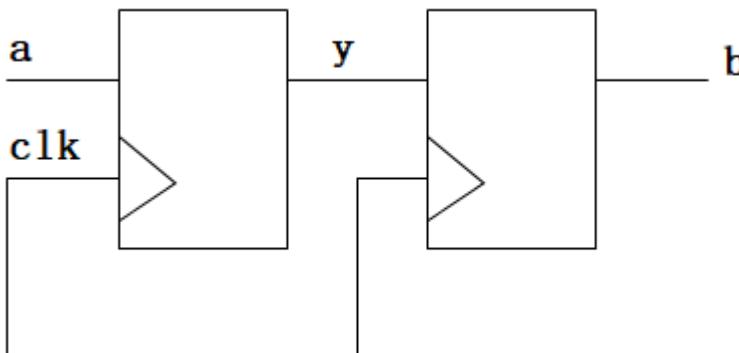
- 阻塞（Blocking）赋值方式（如 $b = a ;$ ）
 - 赋值语句执行完后，块才结束；
 - b的值在赋值语句执行完后**立刻就改变**。
- 非阻塞（Non_Blocking）赋值方式（如 $b <= a;$ ）
 - 块结束后才完成赋值操作；
 - b的值**并不是立刻就改变**。

阻塞赋值与非阻塞赋值

```
module bloc(clk, a, b);  
input clk, a;  
output b; reg b;  
reg y;  
always @(posedge clk)  
begin  
    y=a;  
    b=y;  
end  
endmodule
```



```
module nonbloc(clk, a, b);  
input clk, a;  
output b; reg b;  
reg y;  
always @(posedge clk)  
begin  
    y<=a;  
    b<=y;  
end  
endmodule
```



如何区分阻塞赋值与非阻塞赋值？

- 时序逻辑
 - 一定用非阻塞赋值 “`<=`”, 只要看到敏感列表有posedge就用 “`<=`”。
- 组合逻辑
 - 一定用 “`=`”， 只要敏感列表没有posedge就用 “`=`”。
- 时序逻辑和组合逻辑分成不同的模块
 - 即一个always模块里面只能出现非阻塞赋值 “`<=`”或者 “`=`”。

连续赋值与过程赋值的比较

	过程赋值	连续赋值
assign 符号	无assign (过程性连续赋值除外) 使用“=”, “<=”	有assign 只使用“=”
位置	在always语句或initial语句中 均可出现	不可出现于always语句和initial语句
执行条件	与周围其他语句有关	等号右端操作数的值发生变化时
用途	驱动寄存器	驱动线网

Verilog硬件描述语言

- 概述
- 结构化设计与Verilog模块
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和过程块（组合逻辑电路描述）
- 条件语句和循环语句
- 模块的测试

过程块

- 过程块有两种：
 - always块
 - 循环执行。
 - 满足触发条件则执行。
 - 一个模块中有多个always块时，可以**并行进行**。
 - initial块
 - 只能执行一次。
 - 不带触发条件。
 - 通常**用于仿真模块**。

```
always @(*) begin  
    sum = a + b;  
    //其他代码  
end
```

```
initial begin  
    c=1' b0;  
    //其他代码  
end
```

always块

- 格式如下：

```
always @ (<敏感信号列表>)
begin
    //过程赋值
    //if-else、case选择语句
    //for、while等循环块
end
```

```
module reg_adder (
    input wire [2: 0] a, b,
    output wire [3: 0] out
);
    always @ (a or b) //a或b发生变化，执行
                    sum = a + b;
endmodule
```

- always语句通常带触发条件，触发条件被写在**敏感信号**列表中，只有当触发条件满足条件或发生变化时，其后的”begin-end”块语句才能执行。
- **敏感信号**可分为两种：电平敏感、**边沿敏感**，敏感信号列表中可以有多个信号，用关键字or连接。
- 用关键字**posedge**和**negedge**限定信号敏感边沿。
- 既可以描述**组合逻辑电路**也可以描述**时序逻辑电路**。

组合逻辑的Verilog描述—always块

- 使用触发事件为**电平敏感信号**的 always块
- 使用always块描述组合逻辑电路时，需用**阻塞赋值**
- 将always模块中使用到的**所有**输入信号和条件判断信号都列在敏感信号列表中（建议使用*）

```
//3-8译码器
always @[*] begin
    if (enable) begin
        case (switch)
            3'h0 : decoder_38 = 8'hfe;
            3'h1 : decoder_38 = 8'hfd;
            3'h2 : decoder_38 = 8'hfb;
            3'h3 : decoder_38 = 8'hf7;
            3'h4 : decoder_38 = 8'hef;
            3'h5 : decoder_38 = 8'hdf;
            3'h6 : decoder_38 = 8'hbf;
            3'h7 : decoder_38 = 8'h7f;
            default : decoder_38 = 8'hff;
        endcase
    end
    else decoder_38 = 8'hff;
end
```

组合逻辑的Verilog描述

- 组合逻辑电路是任意时刻的输出仅仅取决于该时刻的输入，与电路原来的状态无关，电路中不包含记忆元件。
- 组合逻辑的电路行为，通常采用两种常用的RTL 级描述方式。

使用**assign**关键字描述的赋值语句

使用触发事件为电平敏感信号的**always**块

initial块

- 格式如下：

```
Initial  
begin  
    语句1;  
    语句2;  
    .....  
    语句n;  
end
```

```
initial  
begin  
#20 begin a = 0;b = 0; cin= 1;end  
#20 begin a = 0;b = 1; cin= 0;end  
#20 begin a = 0;b = 1; cin= 1;end  
#20 begin a = 1;b = 0; cin= 0;end  
#20 begin a = 1;b = 0; cin= 1;end  
#20 begin a = 1;b = 1; cin= 0;end  
end
```

- **begin_end**为顺序块，用来标识顺序执行的语句。

Verilog硬件描述语言

- 概述
- 结构化设计与Verilog模块
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和过程块（组合逻辑电路描述）
- 条件语句和条件运算符
- 模块的测试

条件语句

- 必须在**过程块**中使用
- 条件语句：

if语句：

```
if (条件表达式1)
    语句块1;
else if (条件表达式2)
    语句块2;
.....
else
    语句块n;
```

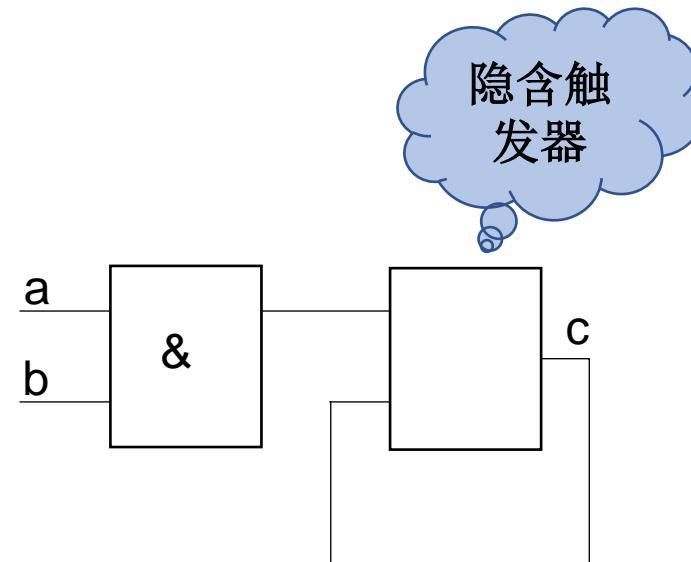
语句块超过1条语句使用begin…end

case语句：

```
case (条件表达式)
    分支1：语句块1;
    分支2：语句块2;
    .....
    default：语句块n;
endcase
```

条件语句-if语句

```
module buried_ff(c,b,a);  
input a,b;  
output c;  
reg c;  
  
always @(a or b)  
begin  
    if ((b=1)&&(a=1)) c=1;  
end  
endmodule
```



如何改正错误？

条件语句-case语句

- case (条件表达式)

分支1：语句块1；

分支2：语句块2；

.....

default: 语句块n；

endcase

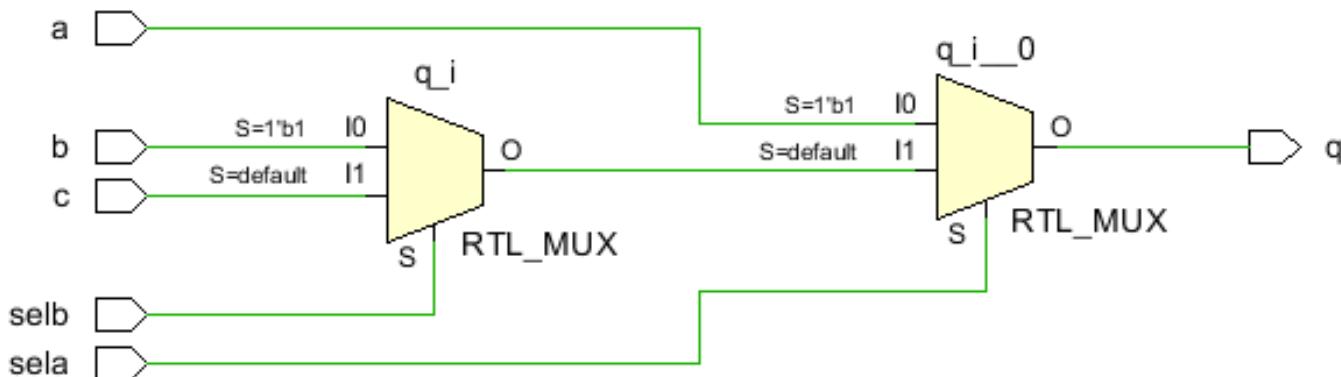
例： reg [2:0] cnt;
case(cnt)
 3'b000:q=q+1;
 3'b001:q=q+2;
 default:q=q;
endcase

- case语句的所有表达式值的位宽必须相等
- 语句中**default**一般不要缺省。在"always"块内，如果给定条件下变量没有赋值，这个变量将保持原值（生成一个锁存器）
- 分支表达式中可以存在**不定值x**和**高阻值z**
 - 如2'b0x, 或2'b0z

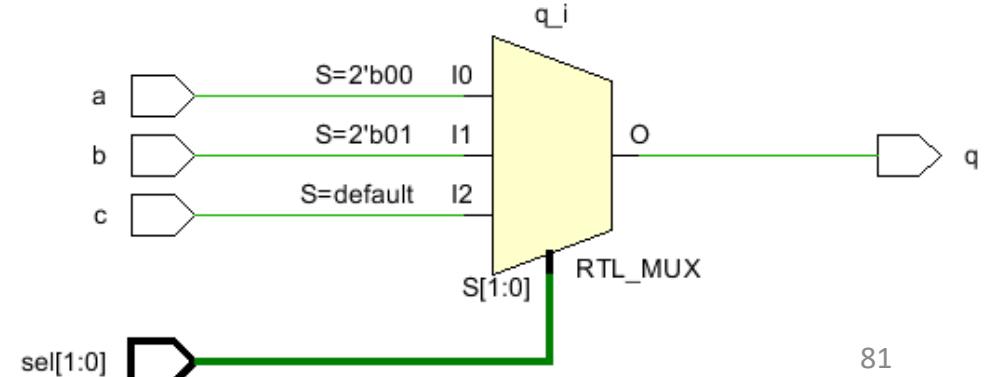
条件语句 if...else 与 case 的区别

- if 生成的电路是串行，是有优先级的
编码逻辑；
- case 生成的电路是并行的，各种判定
情况的优先级相同。

```
always @ * begin
    if(sel_a)
        q = a;
    else if(sel_b)
        q = b;
    else
        q = c;
end
```



```
always @ * begin
    case(sel)
        2'b00: q = a;
        2'b01: q = b;
        default: q = c;
    endcase
end
```



条件语句 if与case的区别

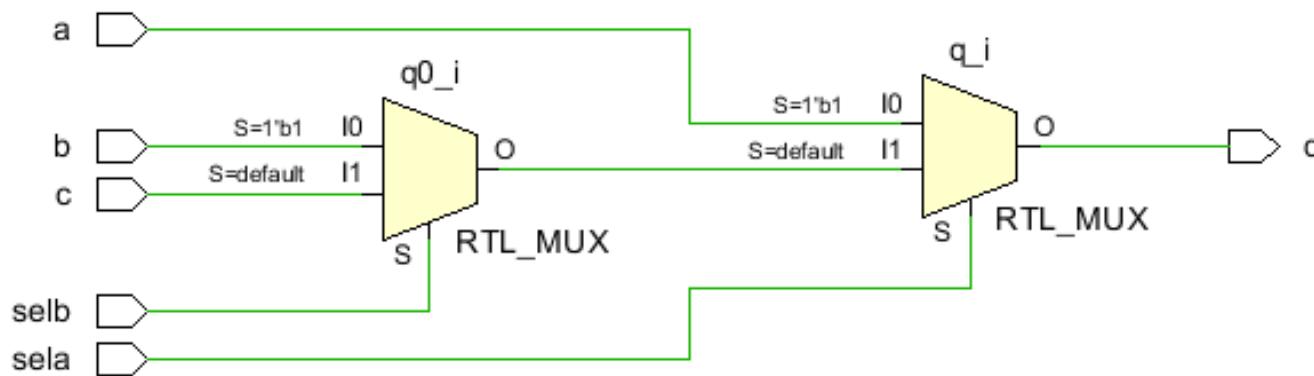
- if 生成的电路是**串行**的，是有优先级的编码逻辑；
 - case生成的电路是**并行**的，各种判定情况的优先级相同。
-
- if 生成的电路**延时较大**，占用硬件资源少；
 - case生成的电路**延时较短**，占用硬件资源多。

条件运算符

- <条件表达式>?<条件为真>:<条件为假>;

```
assign q = (sel_a==1'b1)?a:((sel_b==1'b1)?b:c);
```

```
always @ * begin
    if(sel_a)
        q = a;
    else if(sel_b)
        q = b;
    else
        q = c;
end
```



什么样的Verilog描述会生成锁存器

- 只发生在组合逻辑电路中

1、if...else...语句没有else

```
always @ (*) begin
    if (d_en) q = d;
end
```

2、case语句没有default

- 锁存器的危害：

- 使静态时序分析变得非常复杂

- 对毛刺敏感，不能异步复位，所以上电以后处于不确定的状态；

```
reg q;
always @ (*) begin
    if (d_en) q2 = d;
    else         q2 = 1'h0;
end

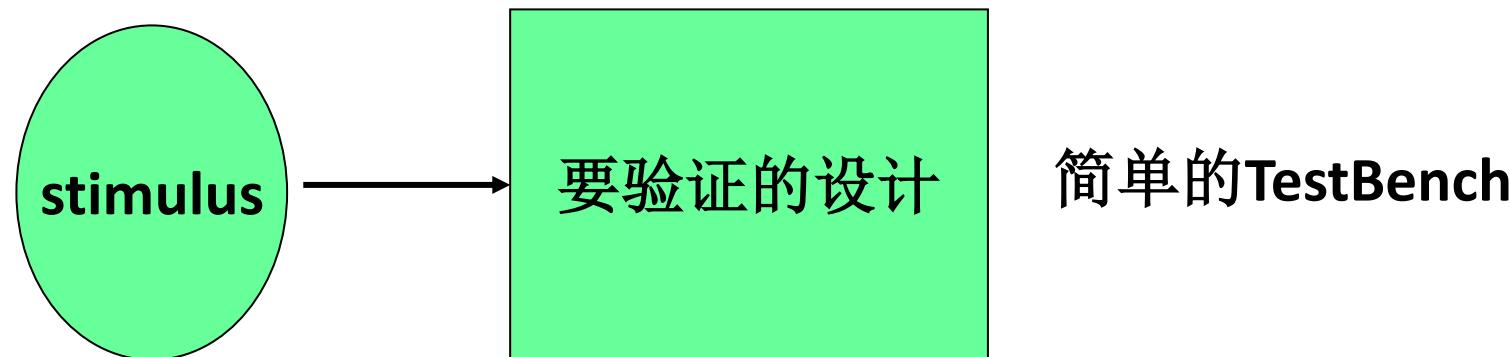
always @ (*) begin
    case (cnt[1:0])
        2'b00 : q = d1;
        2'b01 : q = d2;
        2'b10 : q = d3;
    endcase
end
```

Verilog硬件描述语言

- 概述
- 结构化设计与Verilog模块
- 标识符和数据类型
- 运算符及表达式
- 赋值语句和过程块（组合逻辑电路描述）
- 条件语句和条件运算符
- **模块的测试**

Testbench

- 测试平台（test bench）是一个无输入，有输出的顶层调用模块。
- 一个简单的测试平台包括：
 - 产生激励信号；
 - 实例化待测模块，并将激励信号加入到待测模块中。



关于timescale的用法

- timescale是Verilog HDL 中的一种**时间尺度预编译指令**，它用来定义模块的**仿真时的时间单位和时间精度**。
- 格式如下：`timescale 仿真时间单位/时间精度
- **注意：**用于说明仿真时间单位和时间精度的 数字只能是**1、10、100**，不能为其它的数字。而且，**时间精度不能比时间单位还要大**。最多两则一样大。
- 比如`timescale 1ns/1ps
`timescale 100ns/100ns
- 错误的定义：`timescale 1ps/1ns

Testbench—组合逻辑

```
`timescale 1ns/1ps //1ns表示延时单位，1ps表示时间精度
module decoder_38_sim();
    reg [2:0] data_in;      // 3位二进制输入
    reg [2:0] en;          // 3位控制信号输入
    wire [7:0] data_out;   // 8位输出

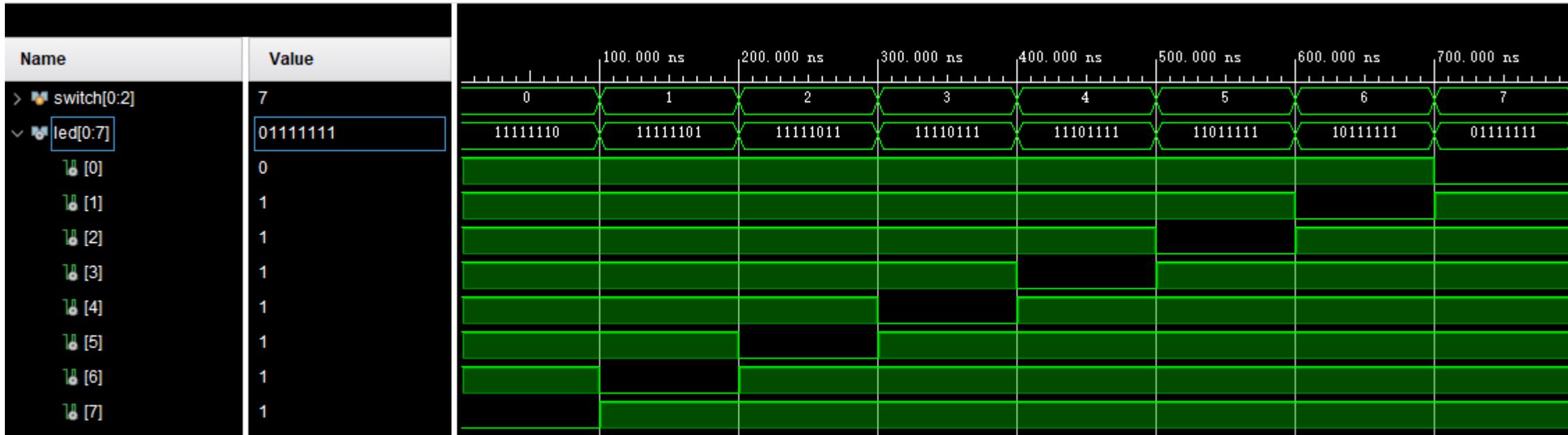
    decoder_38 d_38(
        .data_in(data_in), // 结合自己的实现完成实例化
        .en(en),
        .data_out(data_out)
    );

initial
begin
    #5 begin en = 3'b100;data_in = 3'b000; end // 构造输入激励信号
    #5 begin en = 3'b100;data_in = 3'b001; end
    #5 begin en = 3'b100;data_in = 3'b010; end
    #5 begin en = 3'b100;data_in = 3'b011; end
    #5 begin en = 3'b100;data_in = 3'b100; end
    #5 begin en = 3'b100;data_in = 3'b101; end
    #5 begin en = 3'b100;data_in = 3'b110; end
    #5 begin en = 3'b100;data_in = 3'b111; end
    #5 begin en = 3'b101;data_in = 3'b000; end // 使能端无效
    #5 $stop ; // 立即结束仿真
end
endmodule
```

- 仿真模块没有输入、输出端口
- 激励信号数据类型要求为**reg**, 以便保持激励值不变, 直至执行到下一跳激励语句为止
- 输出信号数据类型要求为**wire**, 以便能随时跟踪激励信号的变化
- **initial**块只能执行一次, 不带触发条件, 通常用于仿真模块。

仿真分析

■ 3-8译码器仿真波形



- 100ns 时， switch 信号由1' b000 变为了 1' b001， led输出为8' b11111101（低电平有效）；
- 200ns 时， switch 信号由1' b001 变为了 1' b010， led输出为8' b11111011（低电平有效）；
-

本章要点

- Verilog语言概述：

- 行为级描述、结构级描述

- RTL与可综合

- 模块定义与模块实例化

- 组合逻辑的Verilog描述

- Testbench模块