杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

《计算机组成原理课程设计》

第 3 章　Verilog HDL 基础

主讲教师：章复嘉

第 3 章 Verilog HDL 基础

3.1 Verilog HDL 概述

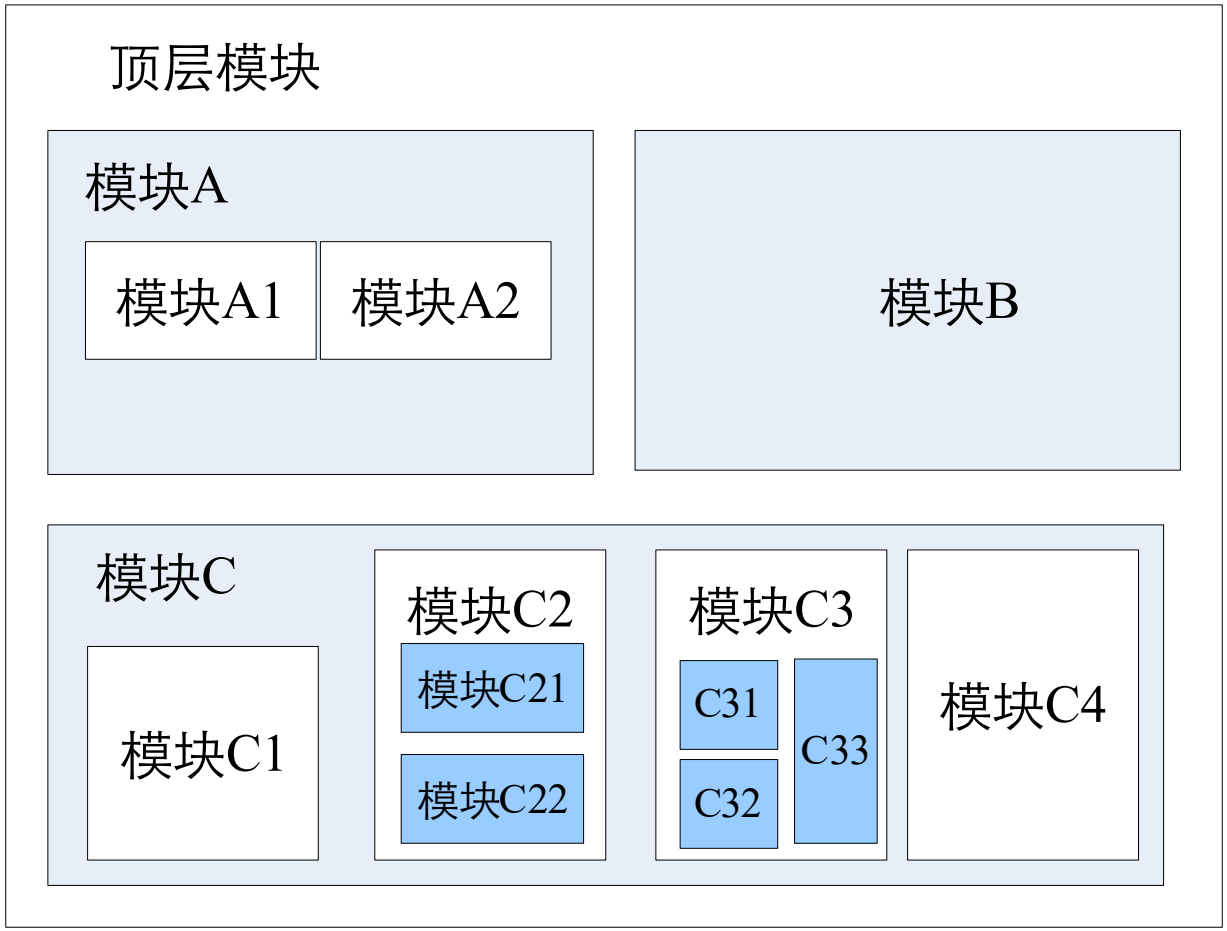
- 1、数字电路的设计方法
- 2、 Verilog HDL 程序结构

1、数字电路的设计方法

- （1） 自下而上的设计方法
  - 从基本单元出发，对设计进行逐层划分的过程
- （2）自上而下的设计方法
  - 从系统级开始，把系统划分为基本单元，然后再把基本单元划分为下一层次的基本单元，直到可用 EDA 元件实现为止。
- （3）混合的设计方法
  - 复杂数字逻辑电路和系统设计过程，通常是以上两种设计方法的结合。
  - 在高层系统用自上而下的设计方法实现，而使用自下而上的方法从库元件或设计库中调用已有的设计单元。

2、 Verilog HDL 程序结构

❖Verilog HDL 程序就是模块的集合：



2、 Verilog HDL 程序结构

- 模块：代表一个基本的功能块，一个模块可以是一个元件，也可以是低层次模块的组合。
  - 模块通过接口（输入和输出端口）被高层的模块调用，但隐藏了内部的实现细节。
  - 使得设计者可以方便地对某个模块内部进行修改，而不影响设计的其他部分。
- 使用 Verilog HDL 完成某个数字电路设计的过程，其实就是一个个模块的程序设计过程。

3.2 Verilog HDL 的模块

1、模块的结构

2、模块的声明与内容

3、模块实例与调用

4、时间单位与时延

1、模块的结构

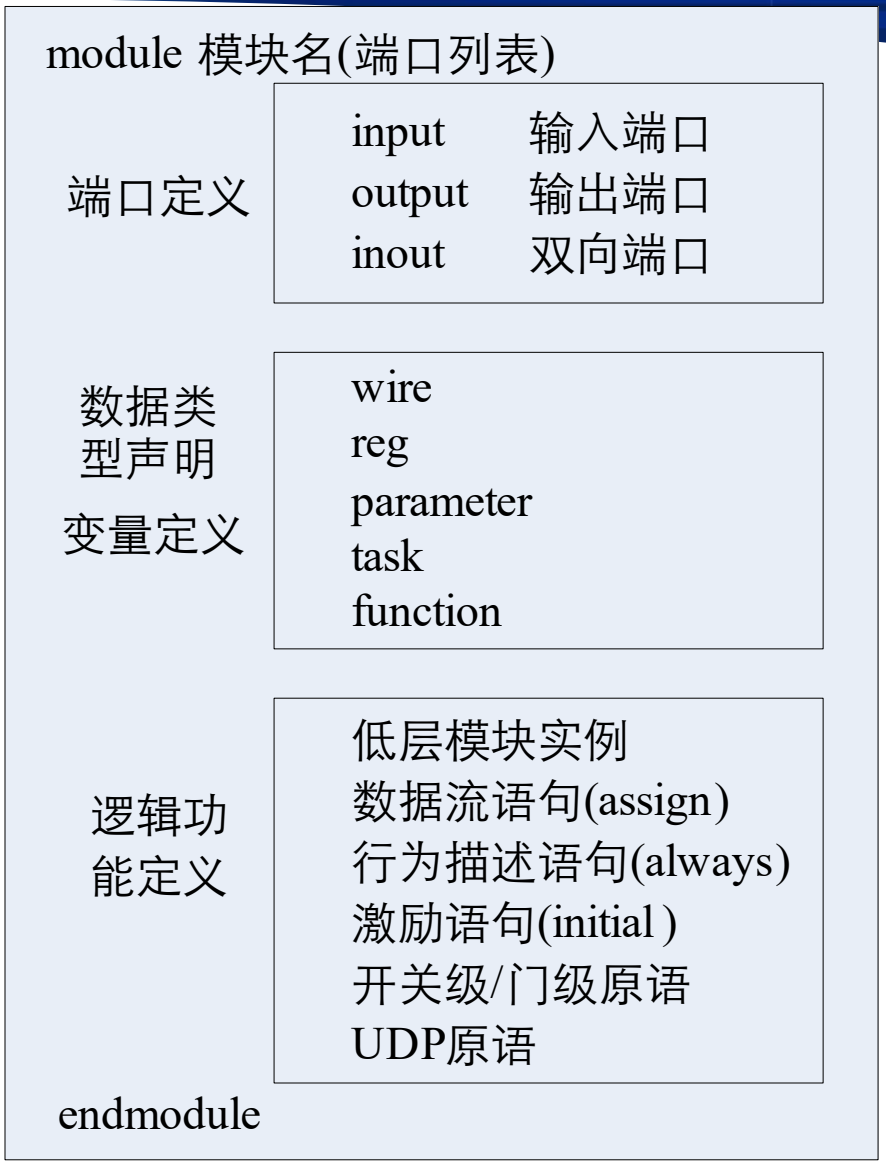
- 模块是 Verilog 的基本描述单位，用于描述某个设计的**功能或结构**及其与其他模块通信的**外部端口**。
- 一个模块可以在另一个模块中**调用**。
- 模块是**并行运行**的，通常需要一个**高层模块**通过**调用其他模块的实例**来定义一个封闭的系统，包括测试数据和硬件描述。

1、模块的结构

❖模块的结构：

❖关键字：

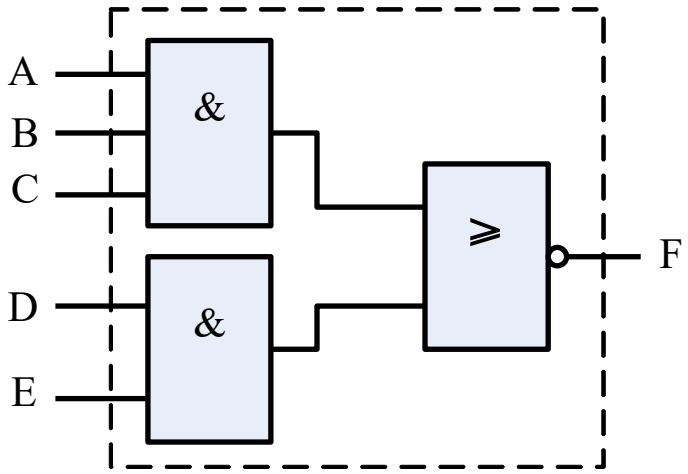
- module
- endmodule



1、模块的结构

❖举例：

$$F = \overline{ABC + DE}$$




```
module  First_M (A, B, C, D, E, F ); // 端口名和端口列表
  input          A, B, C, D, E;   // 声明输入端口；
  output         F;                          // 声明输出端口；
  wire           A, B, C, D, E, F;           // 声明端口的数据类型
       assign  F = ~ (( A & B &C) | ( D &E ));
                 // 逻辑功能描述
endmodule
```

2、模块的声明与内容

❖完整的 Verilog 模块由 4 个部分组成：

- （1）模块声明
- （2）端口类型定义
- （3）数据类型声明和变量定义
- （4）逻辑功能描述

2、模块的声明与内容

❖ （1）模块声明

```
module 模块名 (端口名 1, 端口名 2,…… 端口名 n);
  ……                    // 其他语句
endmodule               // 模块结束关键字
```

- 每个模块**必须有一个模块名**，惟一地标识这个模块。
- **端口列表**用于描述这个模块的输入和输出端口，它是可选的——可以没有端口列表。
- 对模块进行调用时，按端口列表定义分别连接到模块实例的端口列表信号。

❖ 在 Verilog 中，**不允许**在模块声明中**嵌套模块**。

2、模块的声明与内容

❖（2）端口类型定义

- 端口是指模块与外界或其他模块进行连接、通信的信号线，它是**模块与外界环境交互的接口**，就好像 IC 芯片的输入、输出引脚。
- 对于外界来说，**模块内部是不可见的**，**对模块的调用只能通过模块实例的端口进行**。
- 端口定义用于指明端口列表中每一个**端口的类型**：
  - 输入端口：输入引脚
  - 输出端口：输出引脚
  - 双向端口，双向引脚

2、模块的声明与内容

❖ （2）端口类型定义

```
input  [位宽] 输入端口名 1, 输入端口名 2,..... 输入端口名 n;
output [位宽] 输出端口名 1, 输出端口名 2,..... 输出端口名 n;
inout  [位宽] 双向端口名 1, 双向端口名 2,..... 双向端口名 n;
```

❖ 位宽的格式是 [high:low]，high 指明最高位的序号，low 指明最低位的序号，它们都是常量，low 一般是 1 或者 0 。

❖ 位宽省略时，表明是默认位宽为 1 ；

❖ 不同位宽的端口要分别定义。

2、模块的声明与内容

❖（2）端口类型定义

- 举例：

```
input[3:0]        a, b;
        //a、 b 是位宽为 4 位的输入端口 ,a[3] 是高位 ,a[0] 为低位
input             cin;      // cin 是输入端口，默认位宽为 1 位
output[4:1]       sum;
        // sum 是 4 位的输出端口 ,sum[4] 是高位 ,sum[1] 为低位
inout [31:0]      bus;      //bus 是 31 位的双向端口
```

2、模块的声明与内容

❖（3）数据类型声明和变量定义

- 模块描述中使用的所有信号（包括端口、寄存器和中间变量等）都**必须要事先定义，声明其数据类型**后方可使用。
- Verilog HDL 有 19 种数据类型，最常用的是：
  - 线网型（wire）
  - 寄存器型（reg）
  - 参数型（parameter）
- 变量、寄存器、信号和参数等的**声明部分必须在使用前出现**。
- 端口的**数据类型声明缺省**时，EDA 综合器将其**默认为 wire 型**。

2、模块的声明与内容

❖（3）数据类型声明和变量定义

- 举例：

```
reg [3:0]  a,b;   // 定义 a 和 b 的数据类型为 4 位的寄存器型 reg
wire     cin, cout;          //cin 、 cout 的数据类型为 1 位线网型
     wire
parameter x=8, y=x*2; // 定义常量 x 为 8 ，常量 y 为 16
```

2、模块的声明与内容

❖（4）逻辑功能描述

- 它是一个模块中最重要的部分，用于描述模块的行为和功能、子模块的调用和连接、逻辑门的调用、用户自定义部件的调用、初始态赋值 initial、 always 块、连续赋值语句 assign 等等。
- 在 Verilog 模块中，可用 4 种方式描述其逻辑功能：
  - **结构描述方式：** 可使用开关级原语、门级原语和用户定义的原语方式描述；
  - **数据流描述方式：** 使用连续赋值语句 assign 进行描述；
  - **行为描述方式：** 使用过程结构描述时序行为。
  - 3 种描述方式的混合

3、模块实例与调用

- 模块的声明类似于一个模板，使用这个模板就可以创建实际的对象。
- 创建对象方法：**调用模块**
- 从模板创建对象的过程，称为**实例化**；创建的对象称为**模块实例**。
- 模块间的相互调用就是通过**引用实例**来完成的。

```
// 顺序端口连接
模块名称 实例名称 ( 实例端口 1, 实例端口 2,..... 实例端口 n);
// 命名端口连接
模块名称 实例名称 ( . 模块端口 1( 实例端口 1), . 模块端口 2( 实例端口 2),
.....,. 模块端口 n( 实例端口 n));
```

3、模块实例与调用

❖ 【例 3.2 】：使用【例 3.1 】的模块 First_M 创建一个实例 MyFirst_UUT 。

```
module      First_M _Inst;          // 创建实例的模块
    wire    a, b, c, d, e, f        // 声明 6 个线网信号
    First_M  MyFirst_UUT (a, b, c, d, e, f);
en
```

顺序端口连接，默认 a 连接到 A ， b 连接到 B ，依次类推， f 连接到 F

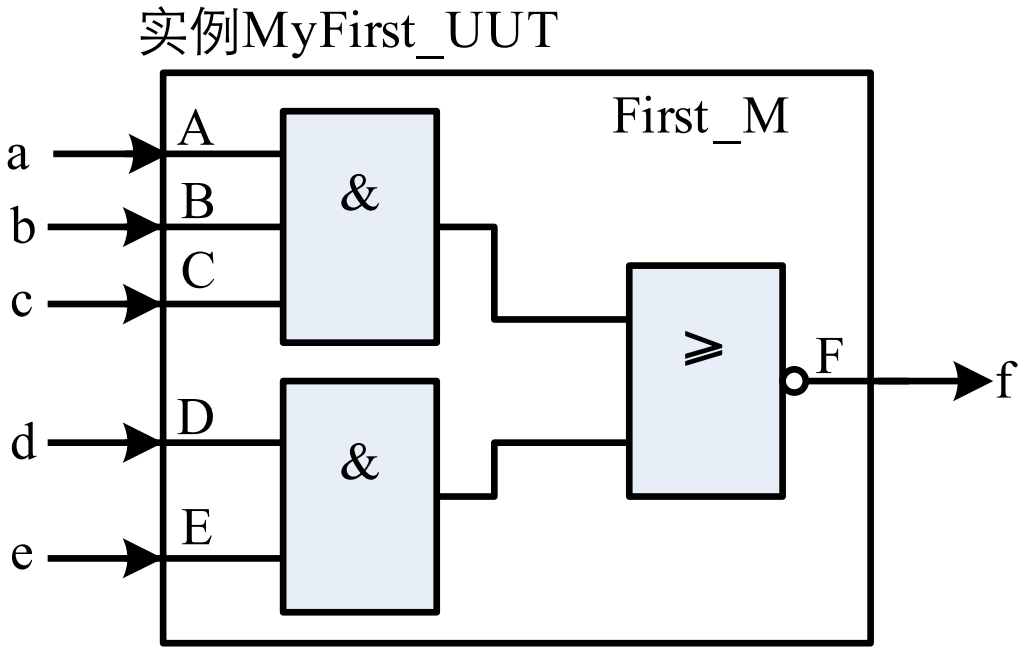
```
module First_M _Inst;     // 创建实例的模块
  reg    a, b, c, d, e;   // 声明 5 个 reg 信号
  wire   f;               // 声明 1 个 wire 信号
  First_M          MyFirst_UUT (
           .F(f) ,
           .A(a),
           .B(b),.C(c), .D(d),.E(e),
           );
endmodule
```

命名端口连接，可以不按声明的顺序

连接实例的输入端口 A 到 reg 变量 a

3、模块实例与调用

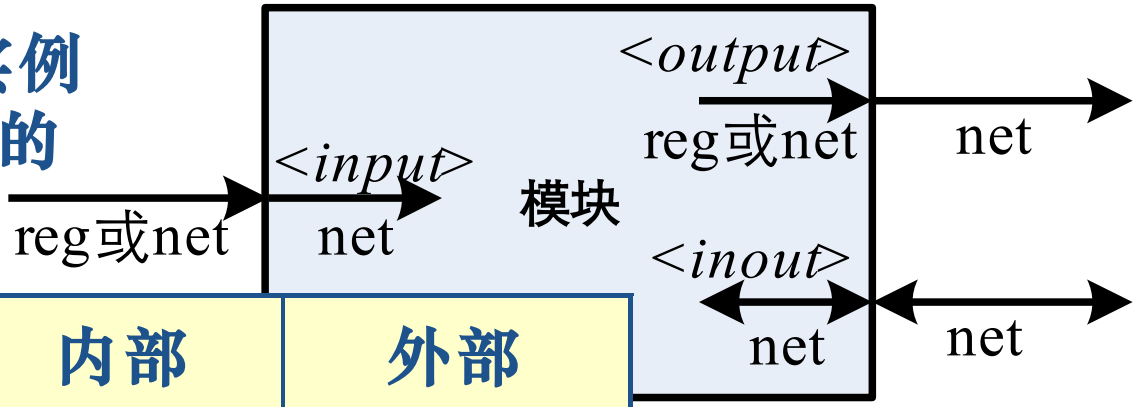
- 模块声明相当于定义了一个电路，而**实例引用**（即调用模块）就相当于又**复制**了一个和模块相同的电路，然后将其端口信号与外部**信号连接**（实例端口）。
- 模块实例 MyFirst_UUT 和模块 First_M 的关系



3、模块实例与调用

❖当一个模块中调用（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，否则会报错。

Verilog 允许模块实例的端口保持未连接的状态 。



	内部	外部
输入端口 input	net	net、 reg
输出端口 output	net、 reg	net
输入输出端口 inout	net	net

4、时间单位与时延

- Verilog HDL 模型中，所有时延都根据单位时间来定义。
  - 编译器指令 `timescale：用于定义时延的单位和时延精度，将时间单位与实际时间相关联。
  - 指令格式为：

`timescale 时延单位 / 时间精度

    - 时延单位和时间精度：由值 1、10、和 100 以及单位 s、ms、us、ns、ps 和 fs 组成。
  - 举例：
    - `timescale 1ns/100ps /* 时延单位为 1ns，时延精度为 100ps，即所有的时延必须被限定在 0.1ns 内 */
    - `timescale 10ns/1ns // 时延单位为 10ns，时延精度为 1ns

4、时间单位与时延

- `timescale 编译器指令需在模块描述前定义，并且影响后面所有的时延值。

```
`timescale 1ns/100ps
assign #2 Sum = A ^ B;
// #2 指 2 个时间单位，即 2ns
```

A、B 发生变化后延时 2 时间单位执行赋值

```
`timescale 10ns/1ns
assign #2 Sum = A ^ B;
// #2 指 2 个时间单位，即 20ns
```

3.3 词法约定

- 与 C 语言类似，语言要素包括以下内容：
  - （1）标识符
  - （2）关键字
  - （3）注释
  - （4）格式

3.3 词法约定

❖ （1）标识符

- 以字母或者下划线“_”开头的，由字母、数字、$符号和下划线“_”组成的字符串。
- 标识符是区分大小写的。
- 以“$”为开头的标识符是为系统函数保留的，不能做普通标识符的起始字符。

合法标识符：

```
Sum
SUM    // 与 Sum 不同，区分大小写
_Row_1
C79_F$
```

非法标识符：

```
$C79_F          // 不能以 $ 开头
356_Day         // 不能以数字开头
and             //and 是关键字
S-T*    // 不能含有 - 和 * 等其他字符
```

❖ （2）关键字

- 即保留字
- 只有小写的关键字才是保留字。

分分	分分分分分
分分	module endmodule input output inout function endfunction task endtask
分分分分	wire reg parameter integer signed
分分	assign always if else begin end case casex casez endcase default for posedge negedge
分分分分	and or not nand nor xor xnor notif0 notif1 bufif0 bufif1

❖ （3）注释

- 单行注释：以“//”开始，忽略从“//”到行尾的内容。
- 多行注释：以“/*”开始，结束于“*/”，允许注释有多行。
- 多行注释不允许嵌套，但是允许单行注释可以嵌套在多行注释中。

❖ （4）格式

- 区分大小写，即大小写不同的标识符是不同的；但是关键字都是小写的。
- Verilog HDL 是自由格式的，即结构可以跨越多行编写，也可以在一行内编写。

```
initial begin Top = 3'b001; #2 Top = 3' b011; end   // 一行书写完毕
```

等价于

```
initial
  begin
     Top = 3'b001;
     #2 Top = 3' b011;
  end
```

3.4 数据类型

1、常量

2、变量

3、标量与向量

4、参数

1、常量

- Verilog HDL 的信号值集合：
  - 1） 0：逻辑 0 或“假”，低电平；
  - 2） 1：逻辑 1 或“真”，高电平；
  - 3） x 或者 X：未知，不确定；
  - 4） z 或者 Z：高阻抗。
- 在门的输入或一个表达式中的为“z”的值通常解释成“x”。
- x 值和 z 值都是不分大小写的，也就是说，值 0x1z 与值 0X1Z 相同。
- Verilog HDL 中的常量是由以上这四类基本值组成的。

1、常量

- Verilog HDL 中有三类常量：
  - （1）整型
  - （2）实数型
  - （3）字符串型

1、常量

（1）整型常量

<位数>'<进制><数字序列>

- 位数：指明数字的二进制位数，只能用十进制数表示；
- 进制：表明数字序列的进制，有 4 种进制：
  - b 或 B：二进制
  - o 或 O：八进制
  - d 或 D：十进制
  - h 或 H：十六进制
- 数字序列：用连续的 0~9、 a~f 或者 A~F 来表示，不区分大小写，不同进制只能使用规定的部分数字。

1、常量

（1）整型常量

- 举例：

```
4'b1x01          //4 位（二进制）的二进制数 1x01
5'O37            //5 位（二进制）的八进制数 37
9'D256           //9 位（二进制）的十进制数 256
12'habc          //12 位（二进制）的十六进制数 abc
```

1、常量

（1）整型常量

- **默认位宽：** 与仿真器和使用的计算机有关（最小为 32 位）；
- **默认进制：** 默认为十进制数；
- **负数：** 在 <位数> 前加负号“-”；负数数值用二进制补码表示
- **符号说明“s”：** 用于表示参加算术运算是带符号数；
- **下划线符号“_”：** 可出现在数字序列的任何位置，用来提高易读性，在编译阶段将被忽略；
- **“？”：** 用来代替高阻“z”，以增强 casez 和 casex 语句的可读性；
- **x（或 z）：** 在十六进制值中代表 4 位 x（或 z），在八进制中代表 3 位 x（或 z），在二进制中代表 1 位 x（或 z）；
- **高位自动扩展：** <位数> 比 <数字序列> 的长度长，Verilog 约定：
  - 最高位是 0、x 或 z：则用 0、x 或 z 自动扩展，填补最高位；

1、常量

❖（2）实数常量

- 实数常量定义：
  - 十进制计数法
  - 科学计数法

```
2.0              // 小数点两边都必须要有数字，不能为 2 或者 2.
1125.678
0.1
23_5.1e2         //e 后面的数字为指数；忽略下划线；真值为
                  //23510.0
3.6E2            // E 与 e 相同；其值为 360.0
5E-4             // 其值为 0.0005
```

1、常量

❖（3）字符串常量

- 字符串是用双引号括起来的一个字符序列。
- 对于字符串的限制是：必须在一行中书写完，不能分成多行书写，也即不能包含回车符。
- 用 8 位的 ASCII 码表示字符串中的字符。
- 字符串常量定义：

```
"Hi,Verilog HDL!"          // 是字符串
“CLICK->HERE”              // 是字符串
reg [8*15:1] String_1;
    initial
     String_1 = "Hi,Verilog HDL!"
```

❖（3）字符串常量

- **特殊字符：**在显示字符串时具有特定的意义，例如换行符、制表符和显示参数的值等。
- 如果要在字符串中显示这些特殊字符，则必须在前面添加前缀**转义字符**，主要为反斜线“\”和“%”

□□□□	□□□□□□□
\n	□□□
\t	tab □□□□□□
\\	\
\"	"
%%	%
\ooo	□□□□ ooo □□□□□

2、变量

- Verilog HDL 有 19 种数据类型，依据其信号的物理特性，分为两大类：
  - （1）线网类型（net type）
  - （2）寄存器类型（register type）

2、变量

（1）线网类型

- 表示 Verilog 结构化元件间的**物理连线**，它由其连接器件的输出来**连续驱动**，例如连续赋值或门的输出。
- 如果**没有驱动**元件连接到线网，**线网的缺省值为 z**。

- **wire**： 普通线网
- **tri**： 三态线网（多驱动源线网）
- **wand**： 线与线网
- **wor**： 线或线网
- **triand**： 线与三态线网
- **trior**： 线或三态线网

- **trireg**： 三态寄存器
- **tri0**： 三态线网 0 （无驱动源时为 0）
- **tri1**： 三态线网 1 （无驱动源时为 1）
- **supply0**： 电源地
- **supply1**： 电源

2、变量

（1）线网类型

- 线网类型说明语法为：

  net_type [msb:lsb] net1, net2, . . . , netN;

  - net_type：是上述线网类型的一种；
  - msb 和 lsb：用于定义线网位宽的常量表达式；
  - 位宽定义是可选的；如果没有定义位宽，缺省的线网位宽为 1 位。

- 在所有网线型数据类型中，最常用的是 wire 型：

  - wire 型变量：用来表示以 assisn 语句生成的组合逻辑信号，输入 / 输出信号在默认情况下自动定义为 wire 型。
  - wire 型信号：可作为任何语句中的输入，也可作为 assign 语句和实例化元件的输出。

2、变量

❖（1）线网类型

❖声明举例：

```
wire Ready, Ack;          // 定义了 2 个 1 位的连线
wire [31:0] data;         //data 是 32 位的线网变量
wand [2:0] Addr;          //Addr 是 3 位的线与变量
```

- 如下的定义是非法的：

```
wire [4:0] x, [3:1]y;     // 不同的位宽要分开定义
wire z[7:0];              // 定义格式不对
```

2、变量

❖（2）寄存器类型

- 表示一个抽象的**数据存储单元**，它**保持原有的数值，直到被改写**。
- 寄存器类型变量**只能在 always 和 initial 语句中被赋值**，并且它的值从一个赋值到另一个赋值之间**被保存**下来。
- 寄存器类型的变量**具有 x 的缺省值**。

❖有 5 种不同的寄存器类型：

- reg： 寄存器类型
- integer： 整型
- time： 时间寄存器
- real： 实数类型
- realtime： 实数时间类型（与 real 类型完全相同）

2、变量

❖（2）寄存器类型

▪ 1） reg 型

- 最常用的寄存器类型，对应触发器、锁存器等具有状态保持功能的电路元件 。
- reg 型变量与 wire 型变量的区别是： wire 型变量需要持续地驱动，而 reg 型变量保持最后一次的赋值。
- reg 型变量的定义格式如下：

  reg[msb： lsb] 变量 l, 变量 2, …, 变量 n;

  - reg 是类型关键字；
  - [msb： lsb] 用于定义 reg 型变量的位宽， msb 是最高位序号， lsb 是最低位序号；
  - 如果不指定位宽，则自动默认为 l 。

2、变量

❖（2）寄存器类型

▪ 1） reg 型

▪ 定义举例：

```
reg [3:0] Sum;                  //Sum 为 4 位寄存器
reg Cnt;                        //1 位寄存器
reg [1:32] MAR, MDR;            // 定义了 2 个 32 位寄存器
```

2、变量

❖（2）寄存器类型

- 2）存储器类型
  - 存储器类型通常用来为 RAM 和 ROM 建模，它实际上就是寄存器的一维数组。
  - 数组的每个元素称为一个元素或者字（word），由索引来指定；字的位宽（字长）可以为 1 位或者多位。
  - 定义格式：

```
reg [msb: lsb] memory1[lower1:upper1], memory2[lower2:
    upper2],. . . ;
```

  - [msb: lsb] 定义存储器字的位宽
  - [lower1: upper1] 定义存储器的字数（即数组的元素个数）。

2、变量

❖（2）寄存器类型

- 2）存储器类型
  - 定义举例：

```
reg [0:3] MyMem [0:63];   //MyMem 为 64×4bits 的存储器
reg membit [0:1023];      // membit 为 1024×1bit 的存储器
```

  - 存储器属于寄存器数组类型。
  - 数组的维数不能大于 2。
  - 注意线网数据类型没有相应的存储器类型。

2、变量

❖（2）寄存器类型

- 2）存储器类型
  - 定义举例：

```
parameter ADDR_SIZE = 1024, WORD_SIZE = 8;
reg [0: WORD_SIZE-1] Ram1 [ ADDR_SIZE-1 : 0], DataReg;
```

  - Ram1 是 1K×8 位寄存器数组（存储器），
  - DataReg 是一个 8 位寄存器；
  - Ram1[345]：访问存储器 Ram1 的地址为 345 的那个单元的字节数据。

2、变量

❖（2）寄存器类型

- 3） integer 寄存器类型
  - 整数寄存器包含整数值，整数寄存器可以作为普通寄存器使用，典型应用为高层次行为建模。
  - 整数型变量定义：

integer integer1, integer2,. . . intergerN;

  - 整数声明中容许无位界限，默认是宿主机的位宽，最少容纳 32 位。
  - 整数变量是有符号数，而 reg 型变量是无符号数。

2、变量

❖（2）寄存器类型

■ 3） integer 寄存器类型

■ 定义举例：

```
integer A, B, C;          // 三个整数型寄存器。
integer List1 [3:6];      // 一组四个寄存器。
```

3、标量与向量

- 向量：位宽大于 1
- 标量：位宽为 1
- 线网和寄存器类型的数据均可声明为向量，如果在声明中未指定位宽，则默认为标量。

```
wire a;                          // 标量线网变量，默认位宽 =1
wire [15:0]   ab,db;             // 定义了 2 个 16 位的向量线网
reg [1:16]    AR,DR;             // 定义了 2 个 16 位的寄存器（向量）
reg clk;                         // 标量寄存器，默认
```

3、标量与向量

- 访问 wire 线网向量或者 reg 寄存器向量时，可以对向量整体访问（全选）或者部分访问（位选或域选）。

```
wire [7:0] in,out;                    //in 和 out 为 8 位 wire 线网向量
wire [4:1] temp1;                     // temp1 为 4 位 wire 线网向量
reg [3:0] temp2;                      // temp2 为 4 位 reg 寄存器向量
assign out=in;                        //8 位全选，整体访问
assign out[7:4]=in[3:0];              // 域选，将 in 的低 4 位赋值给 out 的
    高 4 位
assign out[6:3]=temp1;
          //out 域选， temp1 全选，将 temp1 赋值给 out 的 3~6 位
assign in[7:4]=temp2; //reg 变量 temp2 对 wire 变量 in 的高 4 位持
    续驱动
```

4、参数

- 参数是一个常量，常用于定义时延和变量的宽度。
- 使用 parameter 说明的参数只被赋值一次。
- 参数说明形式如下：

```
parameter param1=const_expr1, param2 = const_expr2 , . . . ,
    paramN = const_exprN;
```

- 定义举例：

```
parameter LINELENGTH = 132, ALL_X_S = 16'bx;    // 整型常量参数
parameter BIT = 1, BYTE = 8, PI = 3.14;      //PI 是实数型常量参数
parameter STROBE_DELAY = ( BYTE + BIT) / 2;
                                  // 由表达式计算结果赋值参数
parameter TQ_FILE = " /home/bhasker/TEST/add.tq";
                                  // 字符串型常量参数
```

3.5 表达式与操作符

- **表达式由操作数和操作符构成**，其作用是根据操作符的意义对操作数计算出一个结果。
- **操作数**：常数、参数、线网、寄存器、线网和寄存器的位选与域选、时间、存储器、函数调用。
- **常量表达式**：在编译时就计算出常数值的表达式，常量表达式由**常量和参数**构成。
- **标量表达式**：计算结果为 1 位的表达式。

3.5 表达式与操作符

- 注意区分无符号数和有符号数：
- 下列情况被当做无符号数：
  - 线网型
  - 一般寄存器型（ reg ）
  - 基数格式表示形式的整数，譬如 5’d12 、 -5’d12
- 下列情况被当做有符号数：
  - 整数寄存器（ integer ）
  - 十进制形式的整数，譬如 12 、 -12

```
-12 / 4          // 有符号数，结果是 -3
-'d12 / 4        /* 无符号数，结果是 -12 的 32 位二进制补码除以 4
    ，即（ 2^32-12 ） ÷4=1073741821 */
```

3.5 表达式与操作符

```
reg[3:0] A,B;
interger C,D,E;
   A=4'b0110;
B=4'b0100;
   C=7;        D=4;
E=2;              □□
```

- 操作符： 9 类
  - （1）算术操作符

□□□□□	□□□□	□□□□□	
+	□□□	1 □ 2	A+B // □□□ 4'b 1010
-	□□□	1 □ 2	A-B // □□□ 4'b 0010
*	□□	2	A*B // □□□ 5'b 11000
/	□□	2	C/D // □□□ 1 □□□
%	□□	2	C%D // □□□ 3 □□□□
**	□□	2	C**E // □□□ 7^2=49

3.5 表达式与操作符

a = 0 ; b = 1;
C =3 ; D =0 ;
E=4'b0x10;

- 操作符： 9 类
  - （2）逻辑操作符

□□□□□	□□□□	□□□□□	□□
&&	□□□	2	a && b // □□□ 0 C&&D // □□□ 1&&0=0
||	□□□	2	a || b // □□□ 1 C || D // □□□ 1 || 0=1
!	□□□	1	!a □□□ 1 □ !b □□□ 0 !C □□□ 0 □ !D □□□ 1

3.5 表达式与操作符

- 操作符： 9 类
  - （3）关系操作符

A= 6 ； B = 8;
C=4'b0x10;

□□□□□	□□□□	□□□□□	□□
>	□□	2	A > B // □□□ 0
<	□□	2	A < B // □□□ 1
>=	□□□□□□□□□	2	A >= B // □□□ 0
<=	□□□□□□□□□	2	A <= C // □□□ x

3.5 表达式与操作符

- 操作符： 9 类
  - （4）等价操作符

```
A= 6 ;          B = 8;
C=4'b1001;
D=4'b1011;
E=4'b1xxz;
F=4'b1xxx;
G=4'b1xxz;
```

□□	□□□□	□□□□□	□□	□□
==	□□	2	□□□□□□ x □ z □□□□□ x	A == B // □□□ 0 E == G // □□□ x
!=	□□□	2	□□□□□□ x □ z □□□□□ x	A!= B // □□□ 1 D!= E // □□□ x
===	□□	2	□□□□ x □ z	E === G // □□□ 1 E === F // □□□ 0
!==	□□□	2	□□□□ x □ z	D !== E // □□□ 1 G !== E // □□□ 0

3.5 表达式与操作符

- 操作符： 9 类
  - （5）位运算操作符

A=4’b1011; B=4’b1101;
C=4’b0000; D=4’b1xxx;

□□	□□□□	□□□□□	□□
&	□□□	2	A & B // □□□ 4’b1001 A & D // □□□ 4’b10xx
|	□□□	2	A | B // □□□ 4’b1111 A | D // □□□ 4’b1x11
^	□□□□	2	A^B // □□□ 4’b0110 A^D // □□□ 4’b0xxx
~	□□□□	1	~A // □□□ 4’b0100 ~D // □□□ 4’b0xxx
~^ □ ^~	□□□□	2	A~^B // □□□ 4’b1001 B~^C // □□□ 4’b0010

3.5 表达式与操作符

❖操作符： 9 类

- （6）归约操作符

```
A=4'b1011;
B=4'b1101;
C=4'b0000;
D=4'b1x0x;
```

□□	□□□□	□□□□□	□□
&	□□□	1	&A // □□□ 0 &D // □□□ 0
~&	□□□□	1	~&B // □□□ 1 ~&D // □□□ x
|	□□□	1	| A // □□□ 1 | D // □□□ 1
~|	□□□□	1	~| C // □□□ 1 ~| D // □□□ 0
^	□□□□	1	^C // □□□ 0 ^D // □□□ x
~^ □ ^~	□□□□	1	~^B // □□□ 0 ~^D // □□□ x

3.5 表达式与操作符

- 操作符： 9 类
  - （7）移位操作符

```
A=4'b1011;  B=4'b1101;
integer C=-10;
// 二进制为
1111_1111_1111_1111
_1111_1111_1111_0110
```

□□	□□□□	□□□□□	□□
>>	□□□□	2	A>>1 // □□□ 4'b0101 B>>2 // □□□ 4'b0011
<<	□□□□	2	A<<1 // □□□ 4'b0110 B<<2 // □□□ 4'b0100
>>>	□□□□	2	C>>>2 /* □□□ 32'b1111_ 1111_1111_ 1111_1111_1111_1111_1101 □□ -3*/
<<<	□□□□	2	C<<<3 /* □□□ 32'b1111_ 1111_1111_ 1111_1111_1111_1011_0000 □□ -80*/

3.5 表达式与操作符

- 操作符： 9 类
  - （8）条件操作符
    - 格式如下：

    条件表达式 ？ 真表达式 ： 假表达式

    - 如果条件表达式为真（即值为 1），则运算返回真表达式；如果条件表达式为假（即值为 0），则运算返回假表达式。
    - 如果条件表达式为 x 或 z，结果将真表达式和假表达式按位操作，运算逻辑如下： 0 与 0 得 0， 1 与 1 得 1，其余情况为 x。
    - 举例：变量 c 要取 a 和 b 中值大的那个数，则可以：

```
c = (a>b) ? a : b        // 即若 a>b，则 c=a，否则
```

3.5 表达式与操作符

- 操作符： 9 类
  - （9）拼接和复制操作符
    - 拼接：将小表达式合并形成大表达式，格式如下：

      {表达式 1，表达式 2，……，表达式 n}
    - 复制操作符：用于指定拼接时的重复次数，格式如下：

      {重复次数 {表达式 1，表达式 2，……，表达式 n}}
    - 举例：

```
Abus = {3{4'b1011}};              // 位向量 12'b1011_1011_1011
Abus = {{4{Dbus[7]}}, Dbus}; // 可用于符号扩展
{3{1'b1}}    // 结果为 111
```

3.5 表达式与操作符

❖操作符优先级：

- 除条件操作符从右向左关联外，其余所有操作符自左向右关联。
- 从最高优先级（顶行）到最低优先级（底行）排列，同一行中的操作符优先级相同。

□□	□□	□□□
□□□□	+ - ! ~	□□
□□□□□□	* / %	
□□□	+ -	
□□	>> <<	
□□	< <= > >=	
□□	== != === !==	
□□ □□ □□□	& ~&	
	^ ~^	
	| ~|	
	&&	
	||	
□□	?:	□□

3.7 Verilog HDL 建模方式

1、建模方式概述

2、结构建模方式

3、数据流建模方式

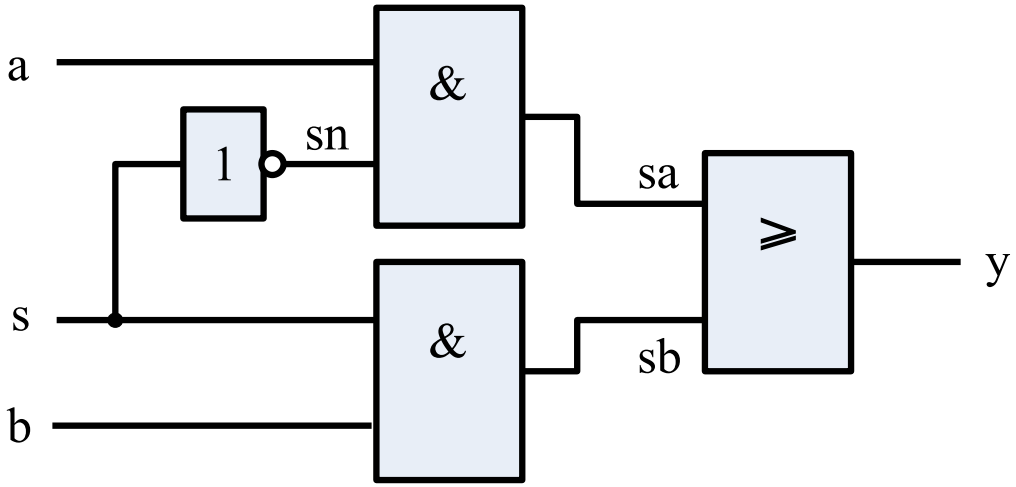
4、行为建模方式

1、建模方式概述

- 在 Verilog 模块中，可用下述方式描述一个设计：
  - （1）结构描述方式：使用门级和开关级内置器件来设计与描述逻辑电路；
  - （2）数据流描述方式：通过说明数据的流程对模块的逻辑功能进行描述；
  - （3）行为描述方式：只注重电路实现的算法，对电路行为进行描述；
  - （4）混合描述方式：上述描述方式的混合

1、建模方式概述

- 【例 3.3 】使用三种基本建模方式，描述模块 mux2to1 ，实现图 3.6 所示的 1 位二选一电路。



1 位二选一电路图

1、建模方式概述

- （1）结构描述方式：根据**电路结构**建模

```
module          mux2to1_1(y,a,b,s);
        output  y;
        input   a,b,s;
        wire    a,b,s,y;
        wire    sn,sa,sb
        not     u1(sn,s);
        and     u2(sa,a,sn);
        and     u3(sb,b,s);
        or      u4(y,sa,sb);
endmodule
```

建模方法：使用基本的内置逻辑门，将电路图翻译成 Verilog 的模块语句。

1、建模方式概述

- （2）数据流描述方式：根据**逻辑表达式**建模

```
module      mux2to1_2(y,a,b,s);
    output  y;
    input   a,b,s;
    wire    a,b,s,y;
    assign  y = ((~s) & a) | (s & b);
            // 等价于      assign y
    =(~s) ? a : b;
endmodule
```

建模方法：根据最简**逻辑表达式**，采用**连续赋值语句**（assign）对输出变量进行赋值。

特点：只关心逻辑表达式，而不关心具体的门电路

1、建模方式概述

- （3）行为描述方式：根据**电路行为**建模

```
module mux2to1_3(y,a,b,s);
        output  y;
        input   a,b,s;
        wire    a,b,s;
        reg     y;
        always @(s or a or b)
                if (!s) y = a;
                else y = b;
endmodule
```

建模方法：基于电路行为的因果关系来描述模块的逻辑功能，它看不到电路的内部结构，只看到电路表现的行为。

1、建模方式概述

- 总结：设计者可以根据需要，在每个模块内部，对逻辑功能进行不同方式的描述，这与建模的抽象层次有关。
- 模块对外显示的功能都是一样的，仅与外部环境有关，与内部的抽象层次无关；而模块的内部结构对外部环境来说是透明的。
- Verilog 有 4 个抽象层次：从低到高
  - 开关级（几乎不用）
  - 门级
  - 数据流级
  - 行为级（或称算法级）
- 寄存器传输级 RTL：能够被逻辑[illegible]工具综合的行为级建模和数据流级建模的结合。

Back

2、结构建模方式

- **结构建模方式**：通过**调用逻辑元件**，描述**信号之间的连接**，建立逻辑电路的模型。
- 可使用的逻辑元件：
  - 1）内置开关（晶体管）；
  - 2）内置逻辑门；
  - 3）用户定义的门级原语；
  - 4）模块实例；
- 逻辑元件之间，通过使用**线网信号**来相互连接

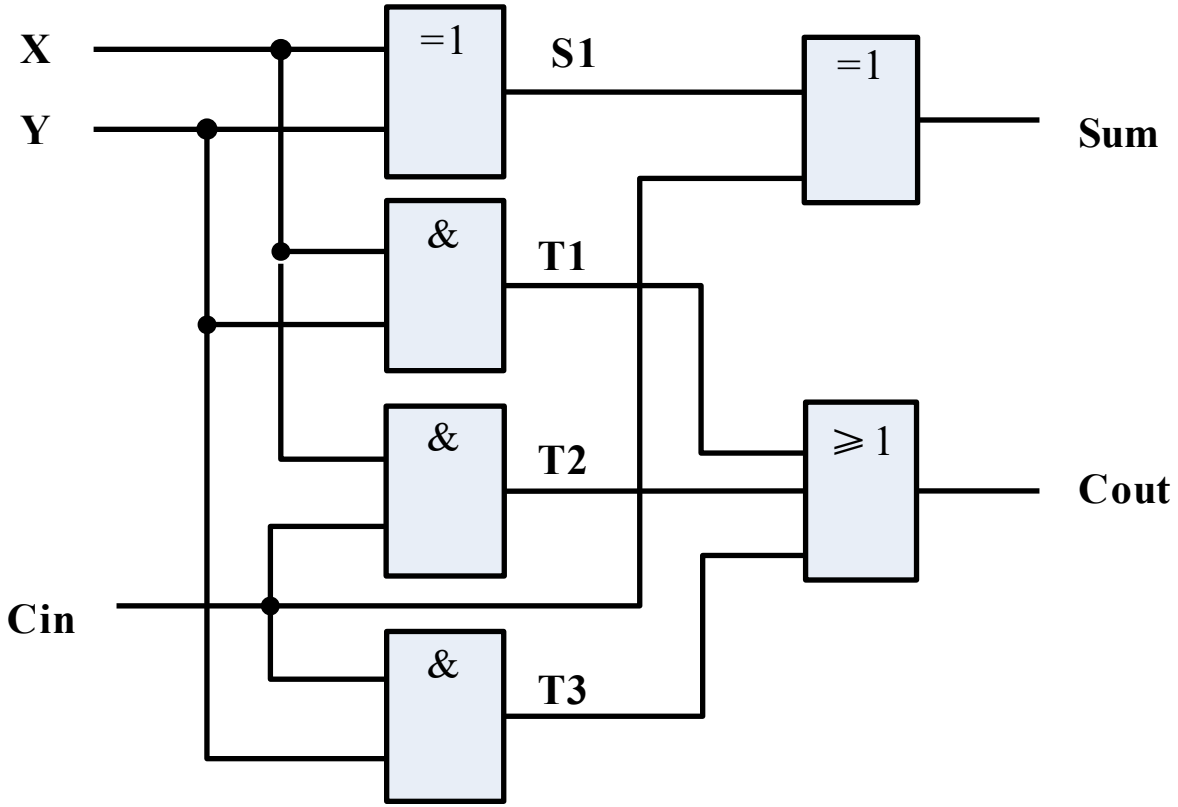
2、结构建模方式

❖ 内置门级元件：

□□	□□□	□□□□	元件调用
□□□□	and	□□	<元件名 ><实例名 >(<输出 >,<输入 1>,…,<输入 n>);
	nand	□□□	
	or	□□	
	nor	□□□	
	xor	□□□	
	xnor	□□□□□□□□	
□□□□	buf	□□□	<元件名 ><实例名 >(<输入 1>,……,<输入 n>, <输出 >);
	not	□□	
□□□	bufif0	□□□□□□	<元件名 ><实例名 >(<数据输出 >, <数据输入 >, <控制输入);
	bufif1	□□□□□□	
	notif0	□□□□□□□	
	notif1	□□□□□□□	

2、结构建模方式

❖ 【例 3.4 】使用结构建模方式，描述 1 位全加器模块 Full_Adder ，电路如图 3.7 所示。



2、结构建模方式

❖ 全加器模块 Full_Adder：

```
module Full_Adder (Sum, Cout, X, Y, Cin);
    output      Sum, Cout;
    input       X, Y, Cin;
    wire        S1, T1, T2, T3;
    xor  XU1 (S1, X, Y),              //S1 = X ⊕ Y
         XU2 (Sum, S1, Cin);          //Sum = S1 ⊕ Cin
    and  AU1 (T1, X, Y),              //T1 = X·Y
         AU2 (T2, X, Cin),            //T2 = X·Cin
         AU3 (T3, Y, Cin);            //T3 = Y·Cin
    or   OU1 (Cout, T1, T2, T3);      //Cout = T1+T2+T3
endmodule
```

2、结构建模方式

- 门级描述与结构建模的模型：

```
module 模块名(端口列表)
    端口定义
    input     输入端口
    output    输出端口

    数据类型声明
    wire

    门级建模与描述
    and     U1(输出,输入1,…输入n);
    not     U2(输出1,…输出n,输入);
    bufif0  U3(输出,输入,使能控制);

endmodule
```

3、数据流建模方式

- **数据流建模**：是通过信号变量之间的**逻辑关系**，采用**连续赋值语句（assign）**来描述逻辑电路的功能。
  - 实际上，数据流描述就是将传统意义上的**逻辑表达式**，用 Verilog HDL 的 assign 语句来表达。
- **逻辑综合**：借助于**计算机辅助设计工具**，自动将电路的**数据流设计直接转换为门级结构**。
- 数据流建模方式已经成为主流的设计方法。

3、数据流建模方式

- 【例 3.5 】使用数据流建模方式，描述 1 位全加器模块 Full_Adder 。

$$Sum = X \oplus Y \oplus Cin$$

$$Cout = X \bullet Y + (X \oplus Y) \bullet Cin$$

```
module Full_Adder (Sum, Cout, X, Y, Cin);
    output      Sum, Cout;
    input       X, Y, Cin;
    assign      Sum = X ^ Y ^ Cin;
    assign      Cout = ( X & Y ) | ((X ^ Y) & Cin);
endmodule
```

3、数据流建模方式

❖连续赋值语句 assign

- 用来驱动 wire 型变量，因其不具备数据保持能力，只有被连续驱动后，才能取得确定值。
- 若一个 wire 型信号变量没有得到任何持续驱动，它的取值将是不确定的“ x”。

❖连续赋值语句格式如下：

assign #延时量 wire 型变量名 = 赋值表达式；

- 只要右边赋值表达式中使用的操作数发生变化，就重新计算表达式的值，新结果在指定的时延后赋值给左边的 wire 型变量。
- 时延：定义了右边表达式操作数变化与赋值给左边表达式之间的延迟时间。如果没有定义时延值，缺省时延为 0，即立即赋值。

3、数据流建模方式

❖连续赋值语句 assign

- 赋值目标：必须为线网型变量，不能是寄存器型变量；
- 连续赋值语句是并发执行的，也就是说各语句的执行顺序与其在描述中出现的顺序无关。
- 体现了组合逻辑电路的特征：任何输入发生变化，输出立即随之而变。
- 多用来描述组合理辑电路。

3、数据流建模方式

- 数据流描述与建模的模型

```
module 模块名(端口列表)
    端口定义
    input     输入端口
    output    输出端口

    数据类型声明
    wire

    数据流级描述
    assign   线网变量名1 = 表达式1;
    ……
    assign   线网变量名n = 表达式n;

endmodule
```

4、行为建模方式

- 行为描述：关注逻辑电路的**外部行为**、输入输出变量的因果关系，而不关心电路的内部结构。
- 行为级建模是**从算法的角度**，即**从电路外部行为的角度**对其进行描述。
- 行为描述关心电路在何种输入下，产生何种输出，而不关心电路具体的实现。
- 行为级建模是**最高的抽象层次**，在这个层次上设计数字电路更类似于使用 C 语言编程。
- 行为级建模语法结构： always、 initial、 if-else、 case 语句等

4、行为建模方式

可综合的行为描述模型

```
module 模块名(端口列表)
    端口定义
    input      输入端口
    output     输出端口
    数据类型声明
    reg
    parameter
    行为或算法级描述
    always @(敏感事件列表)
    begin
        阻塞/非阻塞过程赋值语句
        if-else、case、for语句
    end
endmodule
```

不可综合的行为描述模型

```
module 模块名(端口列表)
    端口定义
    input      输入端口
    output     输出端口
    数据类型声明
    reg
    parameter
    行为或算法级描述
    initial
    begin
        阻塞/非阻塞过程赋值语句
        if-else、case、for语句
    end
endmodule
```

4、行为建模方式

- Verilog 中的行为描述有**两种结构化的过程语句**：
  - **always 语句**：该语句总是循环执行，可以被逻辑综合工具接受，综合为门级描述；
  - **initial 语句**：该语句只执行一次，不能被逻辑综合工具接受，用于初始化变量的值。
- 只有**寄存器类型数据**能够在这两种语句中被赋值，且寄存器类型数据在被赋新值前保持原有值不变；
- **所有的 initial 语句和 always 语句在 0 时刻并发执行。**

4、行为建模方式

- （1） initial 语句
- （2） always 语句
- （3）过程赋值语句
- （4） if-else 语句
- （5） case 语句
- （6）循环语句
  - forever 循环
  - repeat 循环
  - while 循环
  - for 循环

4、行为建模方式

- （1） initial 语句：只执行一次，不能被逻辑综合工具接受，用于初始化变量的值，常用于仿真。

```
module          ini_demo;
    reg a,b,m,n,k
    Initial     k = 1'b0;
    initial
        begin
            #5     a = 1'b1;
            #20    b = 1'b0;
        end;
    initial
    begin
            #10    m = 1'b1;
            #25    n = 1'b0;
    end;
    Initial    #50
        $finish;
endmodule
```

各条语句的执行顺序：

时间	所执行的语句
0	k = 1'b0;
5	a = 1'b1;
10	m = 1'b1;
25	b = 1'b0;
35	n = 1'b0;
50	$finish;

4、行为建模方式

（2） always 结构语句 ：

- 一个程序中可以有多个 always 语句，每个 always 语句内的所有行为语句构成了 always 语句块；
- 所有的 always 块在仿真 0 时刻开始顺序执行其中的行为语句；
- always 语句不断地重复运行，即最后一条 always 块语句执行完后，又开始执行 always 块的第一条指令。
- 敏感事件列表：敏感事件列表中的事件一旦发生变化，就执行一次 always 语句块。

4、行为建模方式

（2） always 结构语句 ：

- always 语句格式如下：

```
always   @( 敏感事件列表 )        语句；
    //always 语句块只有单条语句
always   @( 敏感事件列表 ) // 有多条语句
begin
    语句；
    ……
end
```

4、行为建模方式

（2） always 结构语句 ：

- 敏感事件列表：是激活 always 语句执行的条件
  - 单个事件或多个事件，多个事件之间用“ or”关键字或者“ ,”连接；
  - 电平触发或者边沿触发，电平触发的 always 块常用于描述组合逻辑的行为，而边沿触发的 always 块常用于描述时序行为；
  - 上升沿触发的信号前加关键字 posedge ，下降沿触发的信号前加关键字 negedge ；
  - @* 或者 @(*) ：表示对 always 语句块中的所有输入变量的变化是敏感的。

4、行为建模方式

（2） always 结构语句 ：

- 用 always 语句描述组合逻辑和时序逻辑电路区别：

	组合逻辑电路	时序逻辑电路
敏感事件列表	不应包含 posedge 和 negedge 关键字	用 posedge 和 negedge 关键字描述同步信号的有效跳变沿
	应包含所有输入信号	不一定要包含所有输入信号
所有被赋值的信号	声明为 reg 型	声明为 reg 型
所有的赋值语句	一律采用阻塞赋值语句	一律采用非阻塞赋值语句

4、行为建模方式

❖ （2） always 结构语句 ：

- 举例：组合逻辑电路

```
moduleFirst_M (A, B, C, D, E, F );
    input       A, B, C, D, E;
    output      reg   F; // 有赋值的输出变量必须是 reg 型
    always @( A or B or C or D or E)
        // 等价于 always @( A, B, C, D, E) ，也等价于 always @(*)
        F = ~ (( A & B &C) | ( D &E ));   // 逻辑功能描述
endmodule
```

4、行为建模方式

（2） always 结构语句 ：

- 举例：时序逻辑电路

```
module          UP_DFF (
    Input       CLK,
    input       CLR,
    input       D,
    output      reg   Q);
    always @( negedge CLR or posedge CLK)
    begin
        if (!CLR)       Q <= 1'b0;      // 异步清零
        else            Q <= D ;        // 触发
    end
endmodule
```

4、行为建模方式

- （3）过程赋值语句 ：
  - 用于对寄存器型变量赋值，这些变量在下一次过程赋值之前保持原来的值。
  - 过程赋值语句分为两类：
    - 阻塞赋值：串行执行

      变量 = 赋值表达式;
    - 非阻塞赋值：并行执行

      变量 <= 赋值表达式;
    - 主要区别：一条阻塞赋值语句执行时，下一条语句被阻塞，即只有当一条语句执行结束，下条语句才能执行；而非阻塞赋值的各条语句是同时执行的。

4、行为建模方式

❖（3）过程赋值语句：

程序 1：阻塞赋值

```
always @(posedge clk)
begin
    reg1 = in1;
    reg2 = in2 ^ in3;
    reg3 = reg1; //reg1 的新值
end
```

程序 2：非阻塞赋值

```
always @(posedge clk)
begin
    reg1 <= in1;
    reg2 <= in2 ^ in3;
    reg3 <= reg1; //reg1 的旧值
end
```

4、行为建模方式

- （4）其他语句 ：
  - if-else 语句和 case 语句：自学
    - 注意对于没有完全枚举的情况，需要用 else 和 default 给出默认值。
  - 循环语句：尽量少用，比较耗费资源。

The End!