第8章　初识 Verilog HDL

8.1 章节导读

在本章节中，笔者会带领大家熟悉了解一下硬件描述语言 —— Verilog HDL，并介绍一些基础语法，目的是让初学者在正式学习 FPGA 之前不至于对 Verilog HDL 一无所知。Verilog HDL 语法的讲解之所以会放在基础篇的开始，目的是为了方便大家查阅，如果对本章的语法讲解存在尚未理解的地方可以先行跳过，在后面的章节中笔者会通过具体的工程实例对本章提到的语法应用进行更为透彻的讲解。

8.2 为什么选择 Verilog HDL 开发 FPGA

8.2.1 Verilog HDL 和 VHDL 的比较

硬件描述语言（Hardware Description Lagnuage，HDL）通过描述硬件的实现方法，来产生与之对应的真实的硬件电路，最终实现所设计的预期功能。设计方式与软件不同，因此也就意味着其描述的各个功能之间，可以像硬件一样实现真正的并行执行，这也是硬件描述语言和软件的不同。之所以不直接叫硬件语言却叫硬件描述语言的原因是：这是通过一种语言来“描述”我们设计的硬件所要实现的功能，而不是直接对硬件进行设计。硬件描述语言描述完设计的功能后，还需要通过“综合”这一过程才能最终生成所设计功能的硬件电路。

目前常用的硬件描述语言主要有两种，一种是 Verilog HDL（以下简称 Verilog），另一种是 VHDL，它们之间有什么不同呢？下面让我们简单对比一下。

VHDL 与 Verilog 相比，有以下优势：语法比 Verilog 严谨，通过 EDA 工具自动语法检查，易排除许多设计中的疏忽。有很好的行为级描述能力和一定的系统级描述能力，而 Verilog 建模时，行为与系统级抽象及相关描述能力不及 VHDL。

VHDL 与 Verilog 相比，有以下不足之处：VHDL 代码较冗长，在相同逻辑功能描述时，Verilog 的代码比 VHDL 少许多。VHDL 对数据类型匹配要求过于严格，初学时会感到不是很方便，编程耗时也较多；而 Verilog 支持自动类型转换，初学者容易入门。VHDL 对版图级、管子级这些较为底层的描述级别几乎不支持，无法直接作集成电路底层建模。

综上所述，我们推荐大家在初学时选择语法更简单、更容易接受的 Verilog 来作为 FPGA 的开发语言，这样能够使我们更快速的上手 FPGA 的开发，把省去学习复杂语法的时间用来专攻 FPGA 设计方法，但是无论对于哪种语言我们都希望大家能够做到的是精通一个、熟练一个，这样就能应付更多的问题。

8.2.2 Verilog HDL 和 C 语言的比较

如果学习者有过学习 C 语言的经历，在学习 Verilog 的时候就会发现，Verilog 在很多语法上都和 C 语言极其相似，甚至有些语法是通用的，这也是 Verilog 语言容易上手的一个很重要的原因。Verilog 语言本身就是从 C 语言继承并发展而来的，但是它主要用于描述硬件，和 C 语言这种软件语言思想完全不同。C 语言所描述的代码功能在执行时都是一行一行顺序执行的，而 Verilog 语言在设计完成后执行时则是并行执行的，C 语言所描述的代码功能并不会真实的映射成最后的硬件，只是对内存的操作和进行数据的搬移，而用 Verilog 语言所描述的代码功能则会真真正正的生成所对应的硬件电路，所以这也是 Verilog 语言被称为“硬件描述语言”的原因，C 语言和 Verilog 语言之间的关系就是软件和硬件之间的关系，所以大家不要混为一谈，可以通过 C 语言的语法基础来辅助学习 Verilog 语法，但是切不可生搬硬套，特别在代码的风格和理解上一定要区别对待。

8.3 Verilog HDL 语言基础语法

Verilog 的语法有很多，我们不能够面面俱到，这里我们只简单介绍常用的一些语法，因为有很多语法是几乎用不到的，所以我们要避免因讲太多语法使初学者陷入语法的泥潭而丧失对 FPGA 的学习兴趣，一些高级的、不常用的语法不需要大家记住，万一用到时会在后面的具体实例中特别介绍或者查阅前面推荐的参考书，也可以直接看 IEEE 官方提供的《IEEE Standard Verilog Hardware Description Language》手册。

所有的 Verilog 代码都以 module（模块）的方式存在，一个简单的逻辑可以由一个 module 组成，复杂的逻辑可以包含多个 modules，每个 module 有独立的功能，并可通过输入、输出端口被其它 module 调用（实例化）。通过 module 的方式可以将一些比较独立、可以复用的功能进行模块化，代码阅读起来也比较直观。

Verilog 语法是有很多的，而且分为可综合（综合后可以生成对应的硬件电路）的语法和不可综合（综合后不可以生成对应的硬件电路）的语法，可综合的代码是非常少的，大多数的代码是不可综合的，但是可以在仿真用于验证逻辑的正确性，十分方便，我们都在后面的应用中进行详细的介绍。

8.3.1 标识符

标识符用于定义常数、变量、信号、端口、子模块或参数名称。Verilog 语言是区分大小写的，也就是说同一个名称，用大写和用小写就代表了两个不同的符号，这一点与 VHDL 不同，因此书写的时候要格外注意。

在 Verilog 语言中，所有的关键字（又叫保留字）都为小写。完整的 Verilog 关键字在编辑器会以高亮的形式突出出来。Verilog 的内部信号名（又称标识符）使用大写和小写都可以。标识符可以是字母、数字、$（美元符号）和下划线的任意组合，只要第一个字符是字母或者下划线即可。

8.3.2 逻辑值

在二进制计数中，单比特逻辑值只有“0”和“1”两种状态，而在 Verilog 语言中，为了对电路了进行精确的建模，又增加了两种逻辑状态，即“X”和“Z”。

当“X”用作信号状态时表示未知，当用作条件判断时（在 casex 或 casez）表示不关心；“Z”表示高阻状态，也就是没有任何驱动，通常用来对三态总线进行建模。在综合工具眼中，或者说在实际实现的电路中，并没有什么 X 值，只存在 0、1 和 Z 三种状态。在实际电路中还可能出现亚稳态，它既不是 0，也不是 1，而是一种不稳定的状态。

Verilog 语言中的所有数据都是由以上描述的 4 种基本逻辑值“0”、“1”、“X”和“Z”构成的，同时，“X”和“Z”是不区分大小写的，例如 0z1x 和 0Z1X 表示同一个数据。

8.3.3 常量

1. 常量是 Verilog 中不变的数值，Verilog 中的常量有三种类型

(1) 整数型；

(2) 实数型；

(3) 字符串型。

2. 用户可以使用简单的十进制表示一个整数型常量，例如：

(1) 直接写 16 表示位宽为 32bit 的十进制数 16；

(2) -15 表示十进制的-15，用二进制补码表示至少需要 5bit，即 1_0001，最高一位为符号位；如果用 6bit 表示，则为 11_0001，同样最高一位为符号位。

3. 整数型常量也可以采用基数表示法表示，这种写法清晰明了，所以更推荐这种表示方法，例如：

(1) `8'hab` 表示 8bit 的十六进制数，换算成二进制是 1010_1011；

(2) `8'd171` 表示 8bit 的十进制数，换算成二进制是 1010_1011；

(3) `8'o253` 表示 8bit 的八进制数，换算成二进制是 1010_1011；

(4) `8'b1010_1011` 表示 8bit 的二进制数，二进制就是 1010_1011。

虽然上面的表示方式不同，但都表示的是相同的值，数值经过运算后的结果也都相同。

4. 基数表示法的基本格式

(1) [换算为二进制后位宽的总长度]['][数值进制符号][与数值进制符号对应的数值]
其中[位宽的总长度]可有可无，[数值进制符号]中如果是[h]则表示十六进制，如果是[o]则表示八进制，如果是[b]则表示二进制，如果[d]则表示十进制。当[换算为二进制后位宽的总长度]比[与数值进制符号对应的数值]的实际位数多，则自动在[与数值进制符号对应的数值]的左边补足 0，如果位数少，则自动截断[与数值进制符号对应的数值]左边超出的位数。

(2) 如果将数字写成“'haa”，那么这个十六进制数的[换算为二进制后位宽的总长度]就取决于[与数值进制符号对应的数值]的长度。

(3) 在基数表示法中如果遇到 x，则在十六进制数中表示 4 个 x，在八进制中表示 3 个 x。

(4) 另外，数字中的下划线没有任何意义，但是可以很好的增强可读性，推荐每 4 个 bit 后加一个下划线，例如：4'b11011011 和 4'b1101_1011 表示的是一样的值，但是后面的看上去更容易识别。

5. **Verilog 语言中的实数型变量可以采用十进制，也可以采用科学计数法，例如：**

13_2.18e2 表示 13218

6. **字符串是指双引号中的字符序列，是 8 位 ASCII 码值的序列，例如：**

“Hello World”，该字符串包含 11 个 ASCII 符号（两个单词共 10 个符号，单词之间的空格位一个符号，共 11 个 ASCII 符号），一个 ASCII 符号需要 1 个 byte 存储，所以共需要 11 个 byte 存储。

8.3.4 变量

Verilog 语言中主要的两种变量类型

1. 线网型：表示电路间的物理连接；
2. 寄存器型：Verilog 中一个抽象的数据存储单元。

线网型和寄存器类型具体又包含很多种变量，线网型变量最常用的变量就是 wire，而寄存器型最常用的变量是 reg。wire 可以看成直接的连接，在可综合的逻辑中会被映射成一根真实的物理连线；而 reg 具有对某一个时间点状态进行保持的功能，如果在可综合的时序逻辑中表达，会被映射成一个真实的的物理寄存器，而在 Verilog 仿真器中，寄存器类型的变量通常要占据一个仿真内存空间。

因此在设计逻辑的时候要明确定义每个信号是 wire 还是 reg 属性。凡是在 always 或 initial 语句中被赋值的变量（赋值号左边的变量），不论表达的是组合逻辑还是时序逻辑，都一定是 reg 型变量；凡是在 assign 语句中被赋值的变量，一定是 wire 型变量。

8.3.5 参数

参数是一种常量，通常出现在 module 内部，常被用于定义状态机的状态、数据位宽和计数器计数个数大小等，例如：

```
parameter IDLE = 3'b001;
parameter CNT_1S_WIDTH = 4'd15
Parameter CNT_MAX = 25'd24_999_999
```

可以在编译时修改参数的值，因此它又被常用于一些参数可调的模块中，使用户在实例化模块时，可以根据需要配置参数，例如：

```
counter
#(
    .CNT_MAX    (25'd24     )    //实例化时参数可修改
)
```

```
counter_inst
(
      .sys_clk      (sys_clk   ),     //input     sys_clk
      .sys_rst_n    (sys_rst_n ),     //input     sys_rst_n

      .led_out      (led_out   )      //output    led_out
);
```

parameter 是出现在模块内部的局部定义，只作用于声明的那个文件，可以被灵活改变，这是 parameter 的一个重要特征。

8.3.6 赋值语句

赋值语句的赋值方式有两种，分别为“<=”（非阻塞赋值）和“=”（阻塞赋值）。

1. 以赋值操作符“<=”来标识的赋值操作称为“非阻塞型过程赋值（Nonblocking Assignment）”。非阻塞型过程赋值语句的特点如下：

(1) 在 begin-end 串行语句块中，一条非阻塞过程语句的执行不会阻塞下一语句的执行，也就是说在本条非阻塞型过程赋值语句对应的赋值操作执行完之前，下一条语句也可以开始执行；

(2) 仿真过程在遇到非阻塞型过程赋值语句后首先计算其右端赋值表达式的值，然后等到仿真时间结束时再将该计算结果赋值变量。也就是说，这种情况下的赋值操作是在同一仿真时刻上的其他普通操作结束后才得以执行。

2. 以赋值操作符“=”来标识的赋值操作称为“阻塞型过程赋值（Blocking Assignment）”。阻塞型过程赋值语句的特点如下：

(1) 在 begin-end 串行语句块中的各条阻塞型过程赋值语句将以它们在顺序块后排列次序依次得到执行；

(2) 阻塞型过程赋值语句的执行过程是：首先计算右端赋值表达式的值，然后立即将计算结果赋值给“=”左端的被赋值变量。

阻塞型过程赋值语句的这两个特点表明：仿真进程在遇到阻塞型过程赋值语句时将计算表达式的值并立即将其结果赋给等式左边的被赋值变量；在串行语句块中，下一条语句的执行会被本条阻塞型过程赋值语句所阻塞，只有在当前这条阻塞型过程赋值语句所对应的赋值操作执行完后下一条语句才能开始执行。

后面我们会有专门的章节通过编写代码和仿真详细的介绍其中的不同。

8.3.7 注释

Verilog 中双反斜线“//”可以实现对一行的注释，除此之外“/*.....*/”也是一种注释，进行注释时“/*.....*/”之间的语句都将被注释掉，所以“/*.....*/”不仅仅可以实现一行的注释，还可以实现对多行的注释，注释对整个代码的功能没有任何影响，只是设计者为了增强代码的可读性而增加的内容。

8.3.8 关系运算符

关系运算符种类：

(1) a < b，a 小于 b

(2) a > b，a 大于 b

(3) a <= b，a 小于或者等于 b

(4) a >= b，a 大于或者等于 b

在进行关系运算时，如果声明的关系是假的（false），则返回值是 0；如果声明的关系是真的（true），则返回值是 1；如果某个操作数的值不定，则关系是模糊的，返回值是 x。所有的关系运算符都有着相同的优先级别，但关系运算符的优先级要比算数运算符的低。例如：

```
//表达意义相同
a < size - 1
a < (size - 1)
//表达意义不同
size - (1 < a)
size - 1 < a
```

当表达式 size-(1<a)进行运算时，关系表达式先被运算，然后返回值 0 或 1 被 size 减去；而表达式 size-1<a 进行运算时，size 先被减去 1，然后再同 a 相比。

8.3.9 归约运算符、按位运算符和逻辑运算符

(1) 归约运算符和按位运算符

“&”操作符有两种用途，既可以作为一元运算符（仅有一个参与运算的量），也可以作为二元运算符（有两个参与运算的量）。

当“&”作为一元运算符时表示归约与。&m 是将 m 中所有比特相与，最后的结果为 1bit。例如：

```
&4'b1111 = 1&1&1&1 = 1'b1
&4'b1101 = 1&1&0&1 = 1'b0
```

当“&”作为二元运算符时表示按位与。m&n 是将 m 的每个比特与 n 的相应比特相与，在运算的时候要保证 m 和 n 的比特数相等，最后的结果和 m（n）的比特数相同。例如：

```
4'b1010&4'b0101 = 4'b0000
4'b1101&4'b1111 = 4'b1101
```

“~&”、“^”、“~^”、“|”、“~|”同理。

(2) 逻辑运算符

我们在写 Verilog 代码时常常当 if 的条件有多个同时满足时就执行使用“&&”逻辑与操作符。m&&n 是判断 m 和 n 是否都为真，最后的结果只有 1bit，如果都为真则输出 1’b1，如果不都为真则输出 1’b0。要注意和“&”的功能区分。

“||”、“= =（逻辑相等）”、“!=（逻辑不等）”同理。

8.3.10 移位运算符

移位运算符是二元运算符，左移符号为“<<”，右移符号为“>>”，将运算符左边的操作数左移或右移指定的位数，用 0 来补充空闲位。如果右边操作数的值为 x 或 z，则移位结果为未知数 x。在应用以为运算符的时候一定要注意它的这个特性，那就是空闲位用 0 来填充，也就是说，一个二进制数不管原数值是多少，只要一直移位，最终全部会变为 0。例如：4'b1000 >> 3 后的结果为 4'b0001，4'b1000 >> 4 的结果为 4'b0000。

移位运算符在使用时，左移一位可以看成是乘以 2，右移一位可以看成是除以 2。所以移位运算符用在计算中，代替乘法和除法。尤其是除法，使用移位的方式，可以节省资源。但使用的前提是数据位宽要进行拓展，否则就会出现移位后全为 0 的情况。

8.3.11 条件运算符

如果在条件语句中，只执行单个的赋值语句时，用条件表达式会更方便。条件运算符为“？：”，它是一个三元运算符，即有三个参与运算的量。

由条件运算符组成的条件表达式的一般形式为：表达式 1 ？表达式 2 ：表达式 3

执行过程是：当表达式 1 为真，则表达式 2 作为条件表达式的值，否则以表达式 3 作为条件表达式的值。例如：当 a = 6， b = 7，条件表达式(a > b) ? a : b 的结果为 7。

注意：

(1) 使用条件表达式时“?”和“:”是一对，不可以只是用一个；

(2) 条件运算符从右向左结合，例如：

```
a > b ? a : c > d ? c : d
//等价于
a > b ? a : (c > d ? c : d)
```

虽然后面要讲到的 if-else 也可以实现这种功能，但是 if-else 只能在 always 块中使用，不能在 assign 中使用，如果我们想在 assign 中使用就需要用到条件运算符。

8.3.12 优先级

总的优先级关系为：归约运算符 > 算数运算符 > 移位运算符 > 关系运算符 > “= =”和“!=”> 按位运算符 > “&&”和“||”> 条件运算符，总的来说是一元运算符 > 二元运算符 > 三元运算符。

如果在编写代码的时候对这些关系容易混淆，最好的方式就是使用“（ ）”增加优先级。

8.3.13 位拼接运算符

位拼接运算符由一对花括号加逗号组成“{ , }”，拼接的不同数据之间用“,”隔开。位拼接运算符的作用主要有两种，一种是将位宽较短的数据拼接成一个位宽长的数据；另一种是可以通过位拼接实现移位的效果。

1、 实现增长位宽的作用

如果需要将 8bit 的 a、3bit 的 b、5bit 的按顺序拼接成一个 16 位的 d，表示方法为：

```
wire [15:0] d;
d = {a, b, c};
```

2、 实现移位的作用

din 是 1bit 的串行数据，假如刚开始传来的数据是 1，后面的数据都是 0，则第一个时钟时 4bit dout 的值为 4'b1000，第二个时钟时 dout 的高三位放到最后，新来的 0 放到 dout 的最高位，变为 4'b0100，从而实现了数据的右移功能。

```
always@(posedge sys_clk or negedge sys_rst_n)
      if(sys_rst_n == 1'b0)
            dout <= 4'b0;
      else
            dout <= {din, dout[3:1]};      //右移
```

左移同理，din 是 1bit 的串行数据，假如刚开始传来的数据是 1，后面的数据都是 0，则第一个时钟时 4bit dout 的值为 4'b0001，第二个时钟时 dout 的低三位放到最前面，新来的 0 放到 dout 的最低位，变为 4'b0010，从而实现了数据的左移功能。

```
always@(posedge sys_clk or negedge sys_rst_n)
      if(sys_rst_n == 1'b0)
            dout <= 4'b0;
      else
            dout <= {dout[2:0], din};      //左移
```

8.3.14 if-else 与 case

Verilog HDL 语言中存在两种分支语言：

1、 if-else 条件分支语句

2、 case 分支控制语句

很多初学者会问编写代码的时候，到底是用 if 语句好还是用 case 语句好。同样的逻辑，可能我们用 if-else 语句可以实现，用 case 语句也可以实现。但是在很多的场合，我们又会发现 case 语句和 if-else 语句又总是同时出现，互相嵌套，密切配合。

if-else 条件分支语句：

if-else 条件分支语句的作用是根据指定的判断条件是否满足来确定下一步要执行的操作。它在使用时可以采用如下三种形式：

(1)

```
if(<条件表达式>)
    语句或语句块;
```

在 if-else 条件语句的这种使用形式中没有出现 else 项，这种情况下条件分支语句的执行过程是：如果指定的<条件表达式>成立（也就是这个条件表达式的逻辑值为“ 1”）,则执行条件分支语句内给出的“语句或语句块”，然后退出条件分支语句的执行； 如果<条件表达式>不成立（也就是条件的表达式的逻辑值为“0”、“x”、“z”），则不执行条件分支语句内给出的“语句或语句块”，而是直接退出条件语句的执行。这种写法如果在 always 块中表达组合逻辑时会产生 latch，所以不推荐这种写法。

(2)

```
if(<条件表达式 1>)
    语句或语句块 1;
else  if(<条件表达式 2>)
    语句或语句块 2;
    ……
else
```

在执行这种形式的 if-else 条件分支语句时，将按照各分支项的排列顺序对各个条件表达式是否成立做出判断，当遇到某一项的条件表达式成立时，就执行这一项所指定的语句或语句块；如果所有的条件表达式都不成立，则执行最后的 else 项。这种形式的 if-else 条件分支语句实现了一种多路分支选择控制。这种写法是我们在使用根据波形写代码的方法中最常用的一种写法。

(3) Verilog HDL 允许 if-else 条件分支语句的嵌套使用，但是不要嵌套太多层，也不推荐这种嵌套的写法，因为嵌套会有优先级的问题，最后导致逻辑混乱，if 和 else 的结合混乱，代码也不清晰，如果写代码时遇到这种情况往往是可以将其合并的，最终写成（2）的形式。

```
if(<条件表达式 1>)            //外层 if 语句
    if(<条件表达式 2>)        //内层 if 语句 1
        语句或语句块 1;
    else                      //内层 else 语句 2
        语句或语句块 2;
else                          //外层 else 语句 1
    语句或语句块 3;
```

case 分支控制语句

case 分支语句是另一种用来实现多路分支控制的分支语句。与使用 if-else 条件分支语句相比，采用 case 分支语句来实现多路控制将显得更为方便与直观。case 分支语句通常用于对微处理器指令译码功能的描述以及对有限状态机的描述。case 分支语句有“case”、“casez”、“casex”三种形式。

```
case(<控制表达式>)
    <分支语句 1> : 语句块 1;
    <分支语句 2> : 语句块 2;
    <分支语句 3> : 语句块 3;
    ……
    <分支语句 n> : 语句块 n;
    default      : 语句块 n+1;
```

```
endcase
```

<控制表达式>代表着对程序流向进行控制的控制信号：各个<分支表达式>则是控制表达式的某些具体状态取值，在实际使用中这些分支项表达式通常是一些常量表达式：各个“语句”则指定了在各个分支下所要执行的操作，它们也可以是由单条语句构成，处于最后的、以关键词 default 开头的那个分支项称为“default”分支项，它是可以缺省的。

case 语句的执行过程：

(1) 当“控制表达式”的取值等于“分支项表达式 1”时，执行第一个分支项所包含的语句块 1；

(2) 当“控制表达式”的取值等于“分支项表达式 2”时，执行第二个分支项所包含的语句块 2. ……；

(3) 当“控制表达式”的取值等于“分支项表达式 n”时，执行第 N 个分支项所包含的语句块 n；

(4) 在执行了某一分支项内的语句后，跳出 case 语句结构，终止 case 语句的执行。case 语句中各个“分支项表达式”的取值必须是互不相同的，否则就会出现矛盾现象。

8.3.15 inout 双向端口

在定义端口列表的时候我们知道输入用 input，输出用 output，其实还有一种双向端口，我们定义时使用 inout，在后面的实例中会用到，例如 IIC 和 SDRAM 的数据线都是双向端口。定义为 inout 的端口表示该端口是双向口，既可以作为数据的输入端口也可以作为数据的输出端口，在 Verilog 中的使用方式如下：

```
1  module  test
2  (
3
4      input   wire    sel       , /*输入输出控制信号，sel 为 1 时双向数据总线
5              向外输出数据,sel 为 0 时双向数据总线为高阻态可以向内输入数据*/
6
7      input   wire    data_out  , /*由内部模块传来要发送给双向数据总线
8                                    向外输出的数据*/
9      inout   wire    data_bus  , //双向数据总线
10     output  wire    data_in     /*接收双向数据总线从外部输入的数据后
11                                  输出到其他内部模块*/
12
13 );
14
15 //data_in:接收双向数据总线从外部输入的数据
16 assign  data_in =   data_bus;
17
18 /*data_bus:sel 为 1 时双向数据总线向外输出数据
19 sel 为 0 时双向数据总线为高阻态可以向内输入数据*/
20 assign  data_bus    =   (sel == 1'b1) ? data_out : 1'bz;
21
22 endmodule
```

8.3.16 Verilog 语言中的系统任务和系统函数

Verilog 语言中预先定义了一些任务和函数，用于完成一些特殊的功能，它们被称为系统任务和系统函数，这些函数大多数都是只能在 Testbench 仿真中使用的，使我们更方便的进行验证。

```
`timescale 1ns/1ns  //时间尺度预编译指令 时间单位/时间精度
```

时间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。

时间单位：定义仿真过程所有与时间相关量的单位。

仿真中使用“#数字”表示延时相应时间单位的时间，例#10 表示延时 10 个单位的时间，即 10ns。

时间精度：决定时间相关量的精度及仿真显示的最小刻度。

`timescale 1ns/10ps 精度 0.01，#10.11 表示延时 10110ps。

下面这种写法就是错误的，因为时间单位不能比时间精度小。

```
`timescale 100ps/1ns
```

主要的函数有如下这些，在支持 Verilog 语法的编辑器中都会显示为高亮关键字

```
$display    //打印信息，自动换行
$write      //打印信息
$strobe     //打印信息，自动换行，最后执行
$monitor    //监测变量
$stop       //暂停仿真
$finish     //结束仿真
$time       //时间函数
$random     //随机函数
$readmemb   //读文件函数
```

下面我们单独介绍它们的功能，并在 ModelSim 的 Transcript 界面中打印这些信息。

1. $display 用于输出、打印信息

使用格式为：

```
$display("%b+%b=%d",a, b, c);//格式“%b+%b=%d” 格式控制，未指定时默认十进制
%h 或%H //以十六进制的形式输出
%d 或%D //以十进制的形式输出
%o 或%O //以八进制的形式输出
%b 或%B //以二进制的形式输出
```

```
1  //a,b,c 输出列表，需要输出信息的变量
2  //每次打印信息后自动换行
3  `timescale 1ns/1ns
4
5  module tb_test();
6
7  reg [3:0] a;
8  reg [3:0] b;
9  reg [3:0] c;
10
11 initial begin
12    $display("Hello");
13    $display("EmbedFire");
```

```
14     a = 4'd5;
15     b = 4'd6;
16     c = a + b;
17     #100;
18     $display("%b+%b=%d", a, b, c);
19 end
20
21 endmodule
```

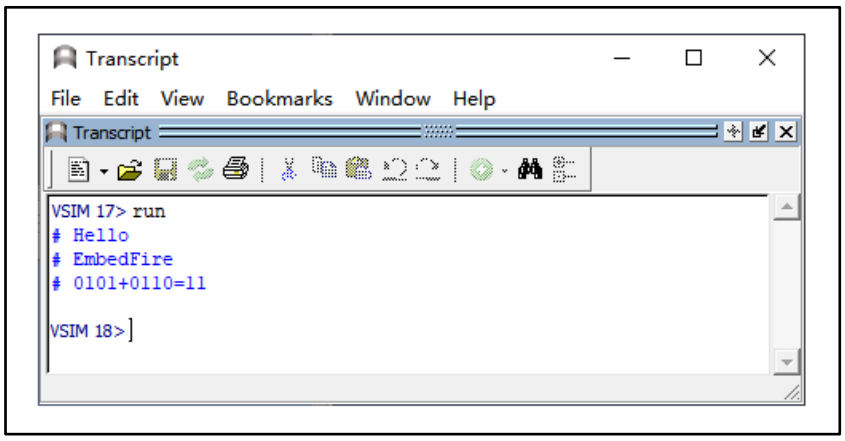

图 8-1 打印信息图

2. $write 用于输出、打印信息

使用格式为：

```
$write("%b+%b=%d\n",a, b, c); //"%b+%b=%d\n" 格式控制，未指定时默认十进制
%h 或%H //以十六进制的形式输出
%d 或%D //以十进制的形式输出
%o 或%O //以八进制的形式输出
%b 或%B //以二进制的形式输出
\n      //换行
```

```
1  //a,b,c 为输出列表，需要输出信息的变量
2  `timescale 1ns/1ns
3  module tb_test();
4
5  reg [3:0]    a;
6  reg [3:0]    b;
7  reg [3:0]    c;
8
9  initial begin
10     $write("hello ");
11      $write("EmbedFire\n");
12      a = 4'd5;
13      b = 4'd6;
14      c = a + b;
15     #100;
16     $write("%b+%b=%d\n",a, b, c);
17 end
18
19 endmodule
```

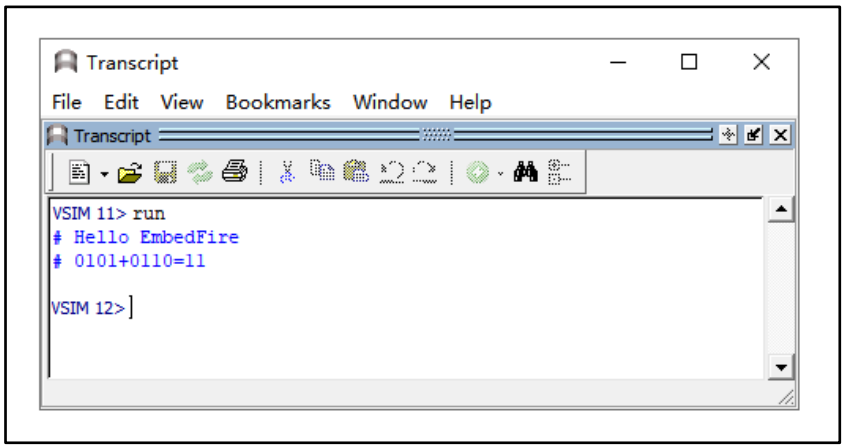

图 8-2 打印信息图

3. $strobe 用于输出、打印信息

使用格式为：

```
$strobe("%b+%b=%d",a,b,c); //"%b+%b=%d" 格式控制，未指定时默认十进制
%h 或%H //以十六进制的形式输出
%d 或%D //以十进制的形式输出
%o 或%O //以八进制的形式输出
%b 或%B //以二进制的形式输出
```

```
1  //a,b,c 输出列表，需要输出信息的变量
2  //打印信息后自动换行，触发操作完成后执行
3  `timescale 1ns/1ns
4  module tb_test ();
5
6  reg  [3:0] a;
7  reg  [3:0] b;
8  reg  [3:0] c;
9
10 initial begin
11   $strobe("strobe:%b+%b=%d", a, b, c);
12   a = 4'd5;
13   $display("display:%b+%b=%d", a, b, c);
14   b = 4'd6;
15   c = a + b;
16 end
17
18 endmodule
```

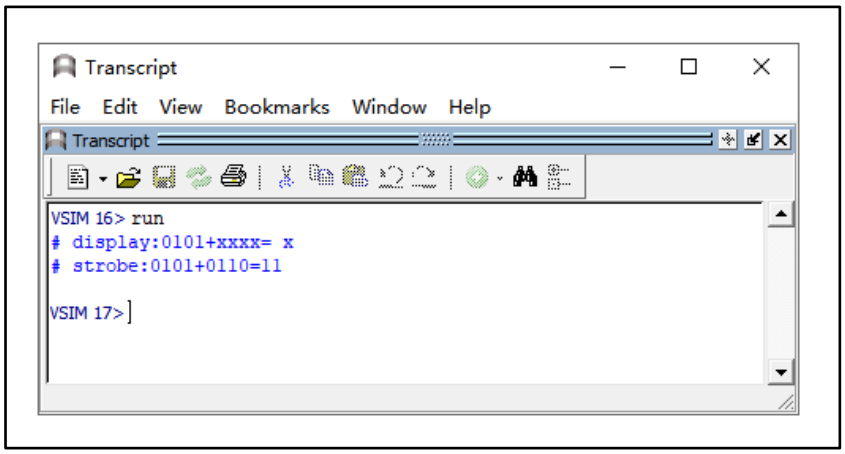

图 8-3 打印信息图

4. $monitor 用于持续监测变量

使用格式为：

```
$monitor("%b+%b=%d",a,b,c); //"%b+%b=%d" 格式控制，未指定时默认十进制
%h 或%H //以十六进制的形式输出
%d 或%D //以十进制的形式输出
%o 或%O //以八进制的形式输出
%b 或%B //以二进制的形式输出
```

```
1  //a,b,c 输出列表，需要输出信息的变量
2  //被测变量变化触发打印操作，自动换行
3  `timescale 1ns/1ns
4  module tb_test ();
5
6  reg  [3:0] a;
7  reg  [3:0] b;
8  reg  [3:0] c;
9
10 initial begin
11    a = 4'd5;
12    #100;
13    b = 4'd6;
14    #100;
15    c = a + b;
16 end
17
18 initial $monitor("%b+%b=%d", a, b, c);
19
20 endmodule
```

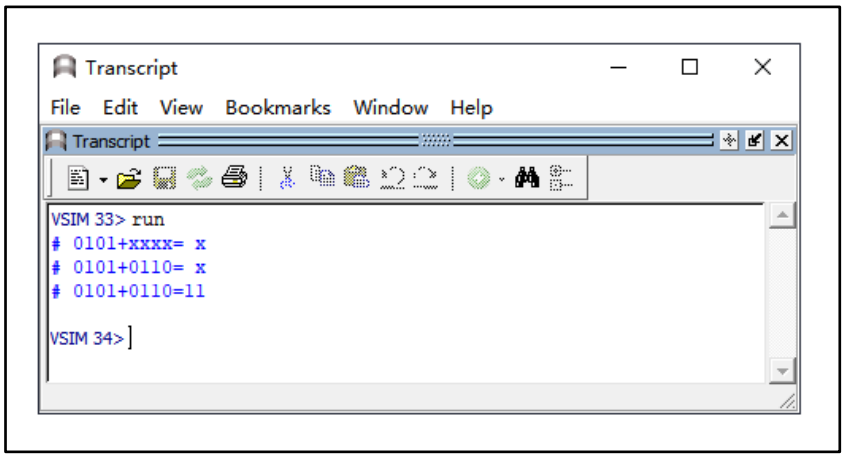

图 8-4 打印信息图

5. $stop 用于暂停仿真，$finish 用于结束仿真

```
1  `timescale 1ns/1ns
2
3  module tb_test();
4
5  initial begin
6     $display("Hello");
7     $display("EmbedFire");
8     #100;
9     $display("Stop Simulation");
10    $stop; //暂停仿真
11    $display("Continue Simulation");
12    #100;
13    $display("Finish Simulation");
14    $finish; //结束仿真
15 end
16
17 endmodule
```

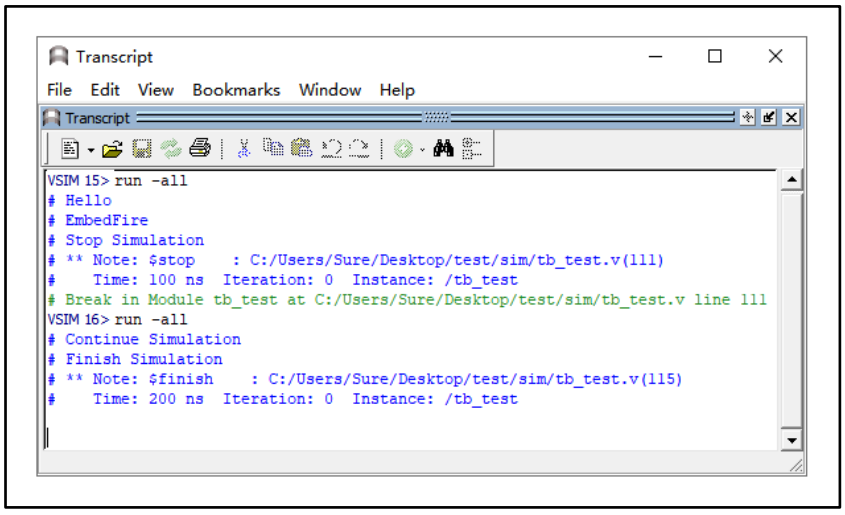

图 8-5 打印信息图

6. $time 为时间函数，返回 64 位当前仿真时间；$random 用于产生随机函数，返回随机数

```
1  `timescale 1ns/1ns
2  module tb_test ();
```

```
 3
 4 reg [3:0]   a;
 5
 6 always #10 a = $random;
 7
 8 initial $monitor("a=%d @time %d",a,$time);
 9
10 endmodule
```

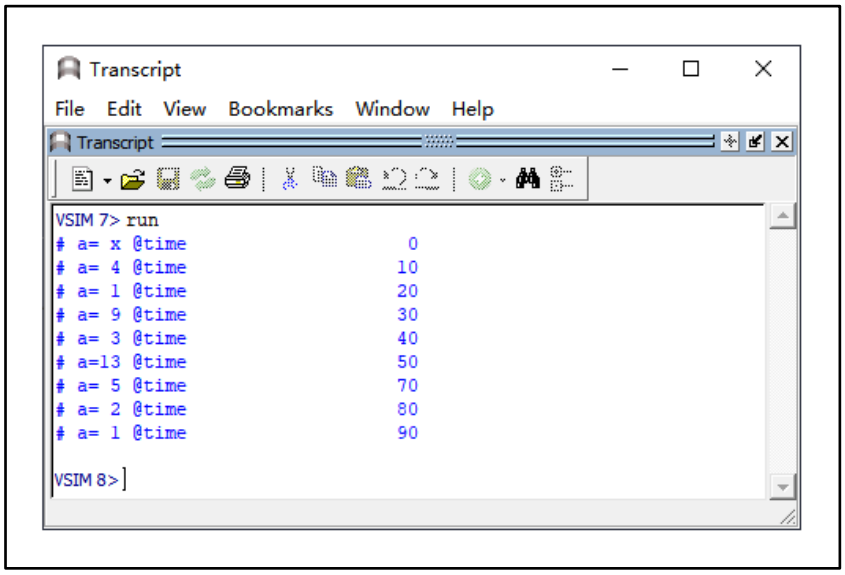

图 8-6 打印信息图

7. $readmemb 用于读二进制文件函数 ， $readmemh 用于读十六进制文件函数

使用格式为：

```
$readmemb("<数据文件名>",<存贮器名>);
$readmemh("<数据文件名>",<存贮器名>);
```

```
 1 `timescale 1ns/1ns
 2 module tb_test ();
 3
 4 integer i;
 5
 6 reg [7:0] a [20:0];
 7
 8 initial begin
 9     $readmemb("EmbedFire.txt", a);
10     for(i=0; i<=20; i=i+1) begin
11         #10;
12         $write("%s", a[i]);
13     end
14 end
15
16 endmodule
```

读取的 txt 文件为

```
01010111 // W
01100101 // e
01101100 // l
01100011 // c
01101111 // o
01101101 // m
```

```
01100101 // e
00100000 //空格
01110100 // t
01101111 // o
00100000 //空格
01000101 // E
01101101 // m
01100010 // b
01100101 // e
01100100 // d
01000110 // F
01101001 // i
01110010 // r
01100101 // e
00100001 // !
```

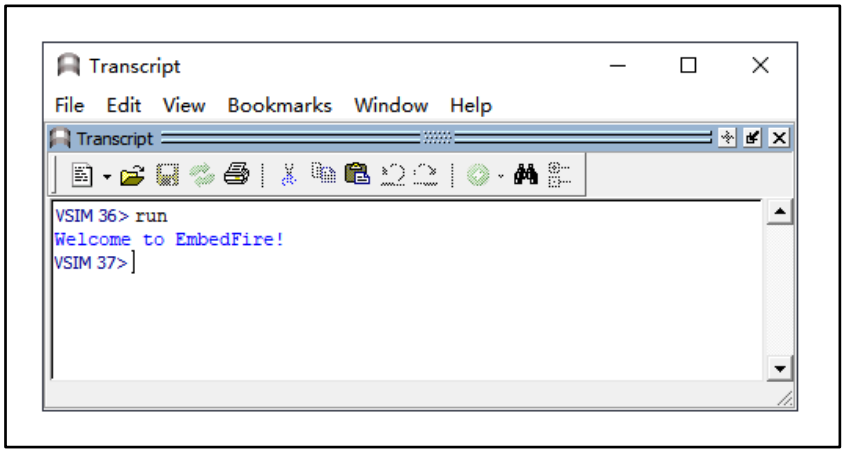

图 8-7 打印信息图

8.4 章末总结

相信大家看完了本章的语法介绍后肯定会感觉知识点太多了，也不能够一下子全部记住，这都没有关系，只要能有个大概印象就足够了，后面每一章中我们都会对遇到的新语法做详细的介绍，让大家在工程实例中学习语法，加深印象。