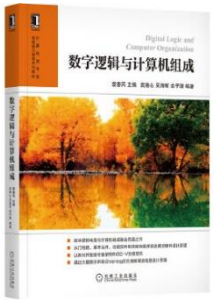
第4讲 Verilog HDL



Verilog 2005/IEEE.1364-2005

南京大学计算机学院

主要内容

- HDL概述
- Verilog语言简介
- Verilog建模方式
- Verilog代码实例
- Verilog测试文件

Verilog HDL简介

- **硬件描述语言**(Hardware Description Language, HDL)是一种用形式化方法来描述和设计数字电路的语言，支持从顶层的抽象层到底层的实现层逐步描述所设计的模块。
- Verilog HDL是由Gateway 公司于1984年作为一个逻辑模拟的编程语言开发，1990年成为公开的标准，1995年正式成为IEEE标准IEEE1364-1995 ，IEEE1364-2001，IEEE1364-2005 。
- 主要特点
  - 能形式化地抽象表示电路的结构和行为；
  - 支持逻辑设计中层次与领域的描述；
  - 可借用高级语言的精巧结构来简化电路行为的描述；
  - 具有电路仿真与验证机制以保证设计的正确性；
  - 支持电路描述由顶层到底层的综合转换；
  - 硬件描述与实现工艺无关；
  - 便于文档管理；
  - 易于理解和设计重用。

Verilog HDL简介

- Verilog HDL是应用最广泛的硬件描述语言之一，可以用于硬件建模、综合、仿真等。
- 既是一种结构描述的语言也是一种行为描述的语言。
- 支持在多个不同抽象设计层次的数字系统建模：
  - 系统级：实现设计系统整体性能的模型
  - 算法级：实现算法运行的模型
  - RTL级：描述数据在寄存器之间的流动和如何处理 、控制这些数据流动的模型
  - 门级：描述逻辑门以及逻辑门之间连接的模型
  - 开关级：描述器件中晶体管以及它们之间连接的模型
- 从语法结构上看，Verilog HDL语言与C 语言有许多相似之处，并继承和借鉴了C语言的多种操作符和语法结构。
- C语言与Verilog硬件描述语言可以配合使用，辅助设计硬件。

Verilog HDL简介

Verilog HDL与C语言的关键字与控制结构对应表

	Verilog	C
模块定义	module、function、task	sub-function
条件语句	if-then-else	if-then-else
分支语句	case	case
程序块	begin…end	{,}
For语句	for	for
While语句	while	while
中止关键字	disable	break
定义关键字	define	define
类型定义	int	int
调试仿真	monitor、display、strobe	printf

Verilog HDL简介

Verilog与C语言运算符对应表

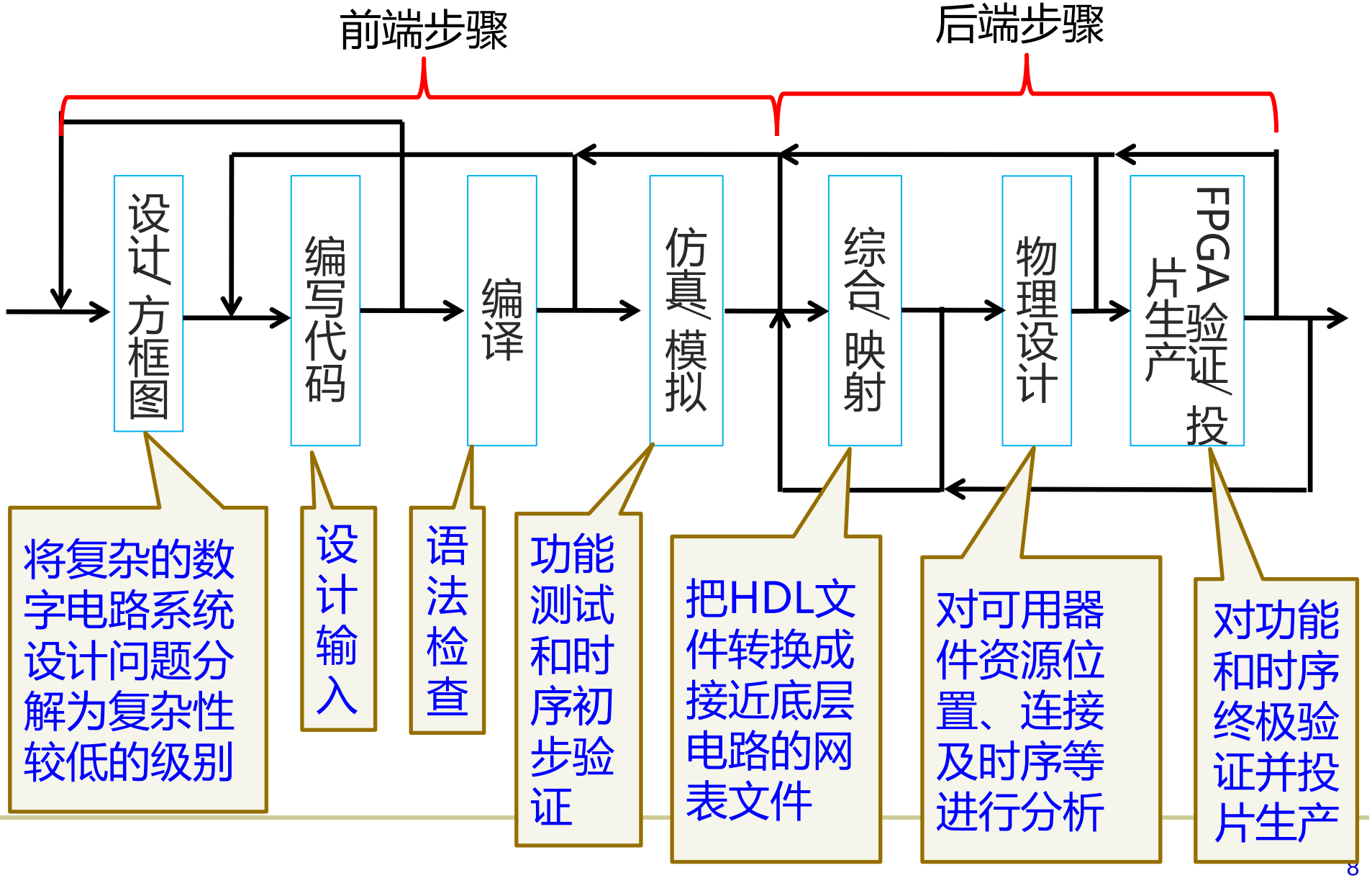
Verilog	C	功 能	Verilog	C	功 能
*	*	乘	>=	>=	大于等于
/	/	除	<=	<=	小于等于
+	+	加	==	==	等于
-	-	减	!=	!=	不等于
%	%	取模	~	~	位反相
!	!	逻辑反	&	&	按位逻辑与
&&	&&	逻辑与	|	|	按位逻辑或
||	||	逻辑或	^	^	按位逻辑异或
>	>	大于	~^	~^	按位逻辑同或
<	<	小于	>>	>>	右移
? :	? :	条件运算符	<<	<<	左移

Verilog HDL简介

Verilog HDL与高级语言比较

项目	Verilog HDL	高级语言
执行顺序	可以在同一时间并行运行。	按行依次执行的。
数据类型含义	表示数字电路中数据存储和传送的方式，如reg类型可以存储值，wire类型可以传送信号	抽象说明数据是带符号整数类型还是浮点数类型等。不涉及实现。
编程语言描述	HDL描述电路的物理结构。 对于**可综合电路**来说，HDL中描述的任何含义和行为，都由相应的电路结构来实现。	描述程序的执行流程
if-else语句	if-else对应的是一个多路选择器，注重完备性。	让程序选择其中的一个分支来执行。
for循环语句	从空间上重复描述相似的电路。	在C语言中的语义是从时间上重复执行相似的代码

1.2 基于HDL的数字电路设计流程



1.2 基于HDL的数字电路设计流程

- 设计并进行HDL编码
  - 根据目标电路的功能需求编写Verilog程序
  - 基本单元是**模块module**，包含**声明**和**语句**

例如：**与门**电路的Verilog程序代码

```
1 module top (
2   input wire a,
3   input wire b,
4   output wire c
5 );
6
7   assign c = a & b;
8
9 endmodule
```

关键字： module,assign,endmodule,input, output,wire

模块名称：top，后接I/O端口列表

两个1位输入端口a、b。

一个1位输出端口c。

assign ：后续为一个连续赋值语句，将赋值号右边的电路输出接入到左边的信号。

endmodule：模块结束

1.2 基于HDL的

- 仿真：通过软件来模拟数字电路行为，观察电路中信号的变化过程，并且检查这些变化是否符合预期。
- 测试激励：对待测试模块提供模拟输入的模块，驱动待测试模块进行工作。

右边是前述与门模块top对应的测试激励模块

3-4行：模块内部信号定义

6-10行：与门模块实例化

12-21行：仿真启动代码块

23-25行：always过程语句

begin和end之间为代码块

```
1 module sim_top ();
2
3    reg a_in, b_in;
4    wire c_out;
5
6    top dut(
7        .a(a_in),
8        .b(b_in),
9        .c(c_out)
10   );
11
12   initial begin
13       a_in = 1'b0;
14       b_in = 1'b0;
15       # 2
16       b_in = 1'b1;
17       # 2
18       a_in = 1'b1;
19       # 2
20       $finish;
21   end
22
23   always @(*) begin
24       $display("a=%d,b=%d,c=%d",a_in,b_in,c_out);
25   end
26
27 endmodule
```

initial是关键字，表示相应代码块会在仿真启动时执行

$finish是系统任务，表示结束仿真

$display是系统任务，用于控制台输出

always @(*)： always是关键字， @(*)表示a_in、b_in和c_out中任一变量的值发生变化就执行后面的代码块

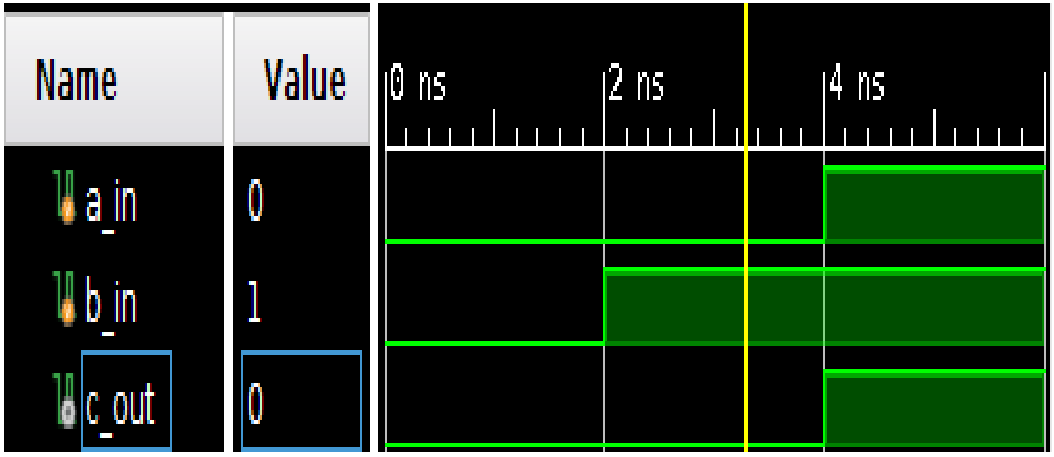
1.2 基于HDL的数字电路设计流程

- 在仿真软件（如Vivado仿真器或modelsim）中运行测试激励模块
- 在控制台中输出：

  a = 0, b = 0, c = 0

  a = 0, b = 1, c = 0

  a = 1, b = 1, c = 1
- 在波形图中显示：



该例验证对象仅是一个模块，为模块级验证

验证多模块连接的数字系统，为系统级验证

```
1 module sim_top ();
2
3    reg a_in, b_in;
4    wire c_out;
5
6    top dut(
7        .a(a_in),
8        .b(b_in),
9        .c(c_out)
10   );
11
12   initial begin
13       a_in = 1'b0;
14       b_in = 1'b0;
15       # 2
16       b_in = 1'b1;
17       # 2
18       a_in = 1'b1;
19       # 2
20       $finish;
21   end
22
23   always @(*) begin
24   $display("a=%d,b=%d,c=%d",a_in,b_in,c_out);
25   end
26
27 endmodule
```

1.2 基于HDL的数字电路设计流程

- 综合：利用EDA综合工具将HDL代码的描述转换成一种接近电路的底层描述-网表文件（netlist）
- 步骤如下：
  - 代码解析（parsing）：根据语言规范进行解析得到层次结构信息（语法树）
  - 多级综合（multi-level synthesis）：将解析到的层次结构信息转换成电路描述，对电路进行一定的优化
    - 如与门模块中的assign语句被转换为真正的与门描述
    - 若与门的一个输入端恒为0，则优化为输出恒为0
    - 并不是所有HDL代码都可综合，如$display，仅用于仿真时的控制台输出，是不可综合代码
  - 工艺映射（technology mapping）：将电路描述进一步转换成特定工艺的标准单元，输出**网表文件**。

1.2 基于HDL的数字电路设计流程

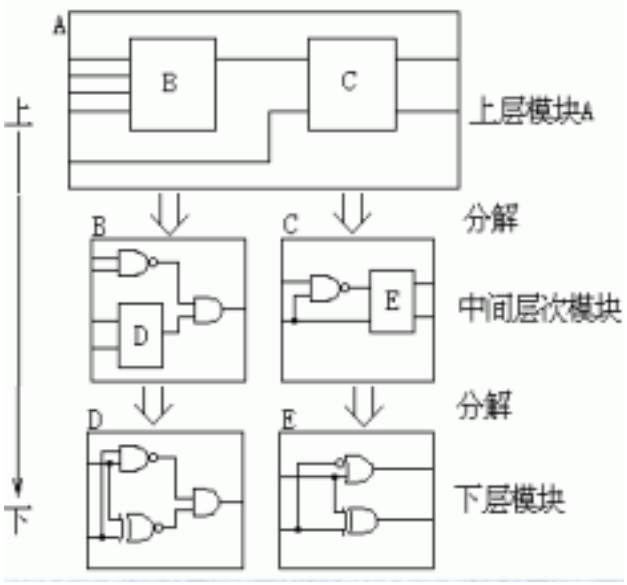
- 物理设计：确定网表中标准单元如何连接以及连线的具体位置
- 过程：
  - 布局（placement）：确定标准单元在三维空间中的位置。
  - 布线（routing）：将标准单元通过物理走线连接起来。
  - 静态时序分析（static timing analysis）：根据标准单元和物理走线的结果，分析每条数据路径的延时情况，报告电路能运行的最大频率。
    - 延时最长的路径称为关键路径，它是阻碍电路频率进一步提升的瓶颈，设计者可以根据关键路径的信息对电路设计进行迭代优化
  - 电路和规则检查：版图和原理图的一致性检查、设计规则检查
  - 生成物理设计结果：对于ASIC流程，将会生成版图文件；对于FPGA流程，将会生成相应的比特流文件。

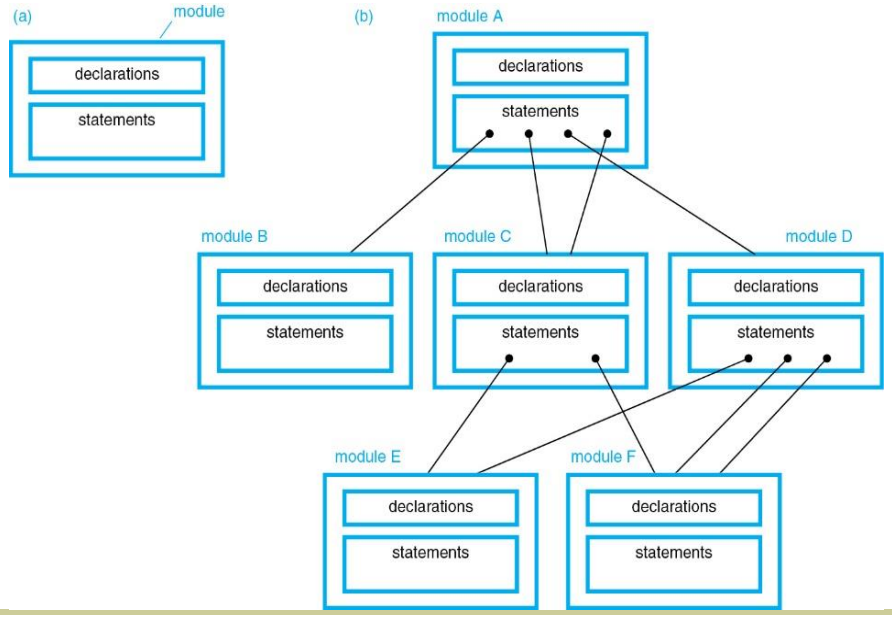
2 Verilog语言简介

- 模块、端口和实例化
- 标识符、常量和注释
- 数据类型
- 运算符及其优先级

2 Verilog语言简介

- **数字电路：器件+连线**
- **数字设计：模块+连线（结点）**
- **器件抽象成模块**
  - **模块可逐级分解**
- 物理电路中的**连线（结点）**
  - Verilog称为**网格**net，默认的网格类型是**wire**。





Verilog模块

- 模块module：Verilog进行设计和编程的基本单元。
- 模块的定义格式如下：

module 名称（端口定义列表）；

内部信号定义

语句功能描述

endmodule

```
1 module top (
2   input wire a,
3   input wire b,
4   output wire c
5 );
6
7   assign c = a & b;
8
9 endmodule
```

- 模块能够表示：
  - 整个数字系统
  - 特定功能逻辑部件，如一个CPU设计的ALU部分
  - 存在边界的物理单元，如IC或ASIC单元
- 一个Verilog源文件中可以包含多个模块。
- 只有一个顶层模块，其它为子模块。
- 模块可以分层嵌套。

端口

- 端口(Terminal)：模块通过端口与外部通信。
  - 端口是模块与外界环境交互的接口。
  - 对于外部环境来讲，模块内部是不可见的，对模块的调用(实例引用)只能通过其端口进行。
- 定义信号与端口间传输方向的关键字
  - input：输入到模块的信号。
  - output：模块输出的信号。
  - inout：该信号既可作为输入又可作为输出。
- 为设计者提供了很大的灵活性：只要端口保持不变，模块内部的修改并不会影响到外部环境。
- 模块内部语句除了endmodule语句、begin end语句和fork join等语句外，每个语句和数据定义的最后必须用分号表示结束。

模块实例化

- 模块通过实例化来调用，一个模块可以被多次实例化。
- 通过实例化语句来描述一个模块的实例。
- 实例化语句的语法如下：

  模块名称 实例标识符 (端口关联列表);
- 在一个模块内，实例标识符必须唯一。
- **端口关联列表**可以有两种形式给出：

①按顺序关联。关联列表中的端口排列顺序与模块端口定义的次序相同。

②按名字关联。关联列表中显式给出每个端口所关联的模块端口名。

  - 一方面可以提高可读性；
  - 另一方面，在模块端口定义的数量增加或者顺序调整时，也不会影响实例中其他端口的关联。

  - 例如： top dut(.a(a_in),.b(b_in),.c(c_out));

2.2 标识符、常量和注释

- **标识符：以字母或下划线开头，可包含字母、数字、下划线和美元$符号，用来描述“对象”的名称。如模块名、端口名、变量名、常量名、实例名等。**
  - 标识符不能与关键字同名。
  - 应采用有意义的名字。
  - 变量名区分大小写。
  - 合法的名字：A_99_Z，Reset，_54MHz_Clock$，Module
  - 非法的名字：123a，$data，module，7seg.v

2.2 标识符、常量和注释

- **整数常量：<size>'<base><value>**
  - size为二进制位数，用十进制数表示。
  - base为常数的基数，可为二(b)、八(o)、十(d)、十六(h)进制，缺省为十进制。
  - value为指定进位制中的任意有效数字，可通过下划线“_”来分隔数字，用于提升可读性。

    8‘b1010_1011：8位二进制常量1010 1011，等价于8’d171

    64‘hff01， 9'O17。
  - value中除0和1外，还可以包含x（非法值或不定态）和z（浮空值或高阻态）。
    - 只有0或1在不同综合工具中都可综合。
    - x和z不同的综合工具会有不同的处理方式。
- **注释**
  - 单行注释符：//
  - 多行注释符：/* … */

2.3 数据类型

- **数据类型用来表示数字电路中数据存储和传送方式，主要分为两类：**
  - 物理数据类型：与实际硬件电路的硬件关系比较明显，抽象程度比较低，主要包括网格类型（net）和寄存器类型(reg)；
  - 抽象数据类型：进行辅助设计和验证的数据类型，主要包括：整型integer、时间型time、实型real和参数型parameter；
- **数据取值集合：**0, 1，x，z。x表示不定态；z表示高阻态。

- **常用的数据类型有：**
  - 网格类型（net）：表示元件或模块之间的物理连接，最常用的是wire类型，传送信号。
  - 寄存器类型（register）：表示抽象的存储元件，最常用的是reg类型，存储数值
  - 参数类型（parameter）：用于给常量赋予有意义的标识符，提高可读性。如，parameter data_width = 5'd32 ;
  - 整数型（integer）：不对应真实的物理电路，主要搭配for循环等结构，使用integer进行变量声明，位宽一般为32bit，有符号数。

2.3 数据类型

1. wire类型

用于表示元件或模块之间的物理连接，对应物理电路中的连线。

- wire类型变量需要被元件或模块的输出端口持续驱动。
- wire类型变量主要用途：
  （1）用于指定模块的输入输出端口；
  （2）声明将要在模块内的结构描述中建立连通性信号。
- wire类型变量缺省为1位的wire类型

2. 寄存器类型reg

用于表示硬件电路元件的变量值，能够存储数据，具有状态保持作用，如用于描述触发器和锁存器的结果

- 常用于行为级描述，由过程赋值语句对其进行赋值。

reg和wire的区别：

- reg保持最后一次的赋值，wire需要持续的驱动
- reg默认初始值为不定值x，wire默认初始值为不定值z
- reg型变量一般是无符号的，若将负数赋值给reg型变量，会自动转成其补码形式

2.3 数据类型

3. 向量和数组

- **向量**：将多个1位信号组成一个整体进行操作，向量的长度称为位宽，通过有序界[MSB:LSB]来定义。

```
wire[7:0] a;    // 一根位宽为8的连线a
reg[31:0] rdata, wdata; // 位宽为32的寄存器rdata和wdata
```

对于可综合电路来说，子界范围通过常量给出。

- **数组**：具有相同数据类型的有序集合，通过索引访问各元素

定义格式：数据类型 数组名[first_addr : last_addr]

wire b[2:0]：数组b包含3根位宽为1的连线

reg r[9:0]：数组r包含10个位宽为1的寄存器

可用reg[MSB : LSB] m[first_addr : last_addr]定义存储器m

reg[15:0] mem[1023:0]：1K×16 b的存储器mem

数组变量不能整体引用，也不能子界范围引用，只能引用单个元素

如b[0], r[8], mem[10]（表示mem中地址为10的一个16位元素）

2.3 数据类型

4. parameter类型

用于声明一个参数，表示有名字的常量，提升代码的可读性。

- 参数的定义是局部的，作用域是当前模块。

```
parameter WORD_WIDTH = 16, ADDR_WIDTH = 10;
```

5. 有符号数&&无符号数

- 无符号数unsigned：表示数值大小的2进制数据格式
- 有号数signed：带有符号位的2进制数据补码格式
- 除integer类型之外的所有数据类型默认都是无符号类型；
- signed可以和reg和wire联合使用，用于定义有符号数。
- signed的自动扩符号位。
- 可以通过系统命令函数$signed对一个unsigned变量在运算过程中作为signed变量处理。
- 只有表达式右侧变量全部是signed变量，整个操作才按signed处理。

2.3 数据类型

5. 模块端口类型

- 模块实例化建立了父模块信号和子模块端口之间的关联，可看成物理连接。

  （1）子模块输入端口只能定义为wire类型，而与之关联的父模块中实例化输入信号作为驱动源，可以是wire类型或reg类型

  （2）子模块输出端口可以是wire或reg类型，而与之关联的父模块中实例化输出信号作为被驱动源，只能定义为wire类型

- 模块端口可以定义成向量，但不能定义成数组。
- 允许关联信号的位宽与端口位宽不同，但会报警。
- 允许端口保持未关联状态，其缺省值为高阻态，但被综合成0，可能会导致非预期结果，故应避免未关联。

2.3 数据类型

- **子模块端口类型定义**

```
module Right (
  input in1,                  // 正确，未声明数据类型时缺省为wire类型
  input wire in2,             // 正确，输入端口只能为wire类型
  input [3:0] in3,            // 正确，端口可定义成向量
  output out1,                // 正确，未声明数据类型时缺省为wire类型
  output [3:0] out2,          // 正确，端口可定义成向量
  output reg [1:0] out3       // 正确，输出端口可为reg类型
);
  ......
endmodule

module Wrong (
  input reg in1,              // 错误，输入端口只能为wire类型
  output out1 [2:0]           // 错误，端口不能为数组类型
);
  ......
endmodule
```

2.3 数据类型

- **父模块端口类型定义**

```
module Top (.....);
  wire w1, w2;
  wire [3:0] wv1;
  wire [1:0] wv2;
  reg r1, r2;
  reg [3:0] rv1;
  wire [5:0] too_long;

  Right warning_or_wrong (     // 对Right模块进行实例化，按端口名字关联
    .in1(1'b0),          // 正确，输入端口可与常量关联
    .in2(),              // 正确但有警告，输入端口可处于未关联状态
    .in3(),              // 输入端口in3未给出，正确但有警告，可处于未关联状态
    .out1(r2),           // 错误，输出端口不能与reg类型信号关联
    .out2(),             // 正确，输出端口可处于未关联状态
    .out3(too_long)            // 正确但有警告，向量位宽与端口定义不一致，
                               // too_long[5:2]处于未关联状态
  );
endmodule
```

```
module Right (
  input in1,
  input wire in2,
  input [3:0] in3,
  output out1,
  output [3:0] out2,
  output reg [1:0] out3
);
```

2.4 运算符及其优先级

- 数字电路设计中的数据本质上是**位串**，数据类型含义表示的是数字电路中**数据存储**和**传送的方式**。
- HDL描述的是电路的**物理结构**，高级编程语言描述的是**程序的执行流程**。
- Verilog中每一种运算符都有其对应的**电路结构**

1. 算术运算符

- 包括“+”、“-”、“*”、“/”和“%”，表示加、减、乘、整除和取模运算。
- “+”和“-”运算结果的位宽为两个操作数中位宽较长者再加**1**，多出来一位用于表示进位或借位；将综合出补码加减运算电路。
- “*”运算结果的位宽为两个操作数的**位宽之和**；将综合出阵列乘法器电路。
- “/”和“%”运算结果的位宽与第一个操作数的位宽相同。有的综合器会综合出阵列除法器电路。
- 阵列乘法器、阵列除法器比较复杂、延迟长。因此在高性能处理器设计中，通常不直接使用“*”、“/”、“%”运算符，而是会编写高性能乘法器和除法器的DHL代码。

2.4 运算符及其优先级

2. 位运算符

- 包括“~”、“&”、“|”、“^”和“^~”（或“~^”），分别表示按位取反、按位与、按位或、按位异或和按位同或运算。
- 位运算结果的位宽与操作数的位宽相同，如两个操作数的位宽不同，则短操作数先进行零扩展（即高位补“0”），再进行运算。
- 综合出与位宽数量相同的一个或多个门电路

3. 归约运算符

- 包括“&”、“~&”、“|”、“~|”、“^”和“^~”（或“~^”），分别表示与归约、与非归约、或归约、或非归约、异或归约和同或归约运算。
- 无论操作数的位宽是多少，归约运算结果的位宽均为1。
- 将操作数最低位依次与前面各位进行与、或、异或等运算。

  若信号a的位宽为4，则&a = ((a[0] & a[1]) & a[2]) & a[3]
- 综合出一个输入端口数量与操作数位宽相同的门电路。

2.4 运算符及其优先级

4. 逻辑运算符

- 包括“&&”、“||”和“!”，分别表示逻辑与、逻辑或和逻辑非运算。
- 将操作数当做布尔值，非零为真，结果位宽为1。
- 其行为可通过归约运算符和位运算符表达，从而得到综合出的电路。

如：a && b = (|a) & (|b)，a || b = (|a) | (|b)，!a = ~(|a)

5. 等式运算符

- 主要包括“==”和“!=”，分别表示等于判断和不等于判断。
- 等式运算将两个操作数的每一位分别进行比较，若均相同，结果为1。
- 等式运算符的行为可以通过位运算符和归约运算符表达，从而得到等式运算符综合出的电路。

如：(a == b) = ~(|(a ^ b))，(a != b) = |(a ^ b)

2.4 运算符及其优先级

6. 关系运算符

- 包括“<”、“<=”、“>”和“>=”，分别表示小于、小于等于、大于和大于等于判断。
- 关系运算结果的位宽均为1。
- 综合出带标志位的补码加减运算电路，通过输出标志位来判断关系。
- 通过数据类型来区分带符号数和无符号数的关系运算。

7. 位拼接运算符

- “{ }”用于将两个或多个信号按顺序拼接起来。
- “{n{m}}”表示将n个m拼接。
- 是唯一一个操作数数量可变的运算符，操作数之间用“,”分开。
- 位拼接运算结果的位宽为所有操作数的位宽之和。

  如：若a = 2'b11，b = 4'b1101，则{a, b[1:0], 3'b0} = 7'b1101000；{a, {2{c, b}}} = {a, c, b, c, b}。
- 位拼接运算符的行为相当于信号集线器，无需综合出逻辑门器件。

2.4 运算符及其优先级

8. 移位运算符

- “>>”和“<<”，分别进行逻辑右移和逻辑左移运算。
- “>>>”和“<<<”，分别进行算术右移和算术左移运算。
- 移位位数由右边的操作数给出，并用相应数量的0填补移出的空位。
- 对于右移，结果位宽和被移位操作数的位宽相同；
- 对于左移，结果位宽为被移位操作数的位宽加上左移的位数。

  如：4'b1001 >> 3 = 4'b0001；4'b1001 << 2 = 6'b100100；1 << 4 = 36'b10000。
- 若移位位数为常量，则移位运算符的行为可通过位拼接运算符和下标选择来表达。

  如：假设被移位操作数a的位宽为wa，移位位数为b，wa和b皆为常量，则 (a >> b) = {{b{1'b0}}, a[wa-1:b]}

  (a << b) = {a, {b{1'b0}}}
- 若移位位数为变量，则移位运算符将综合出移位器电路（组合逻辑电路，如桶型移位器）。

2.4 运算符及其优先级

7. 条件运算符

- “条件？表达式1：表达式2”：若条件为真，则将表达式1的作为条件运算的结果；否则将表达式2的作为条件运算的结果。
- 条件运算结果的位宽与表达式的位宽相同。
- 若两个表达式的位宽不同，则位宽较短的表达式将进行零扩展。
- 条件运算符综合出二路选择器，条件对应的电路输出作为其控制信号

2.4 运算符及其优先级

- 运算符的优先级

类 别	运 算 符	优先级
逻辑、位运算符	! ~	高
算术运算符	* / %	
	+ -	
移位运算符	<< >> >>>	
关系运算符	< <= > >=	
等式运算符	= = != === !==	
归约、位运算符	& ~&	
	^ ^~或~^	
	| ~|	
逻辑运算符	&&	
	||	
条件运算符	? :	低

为提高程序的可读性，建议使用括号来控制运算的优先级！

1 + a << 2

3 verilog的建模方式

- 三种建模方式
  - 结构化建模
  - 数据流建模
  - 行为建模
- 行为建模中的过程语句

3.1 三种建模方式

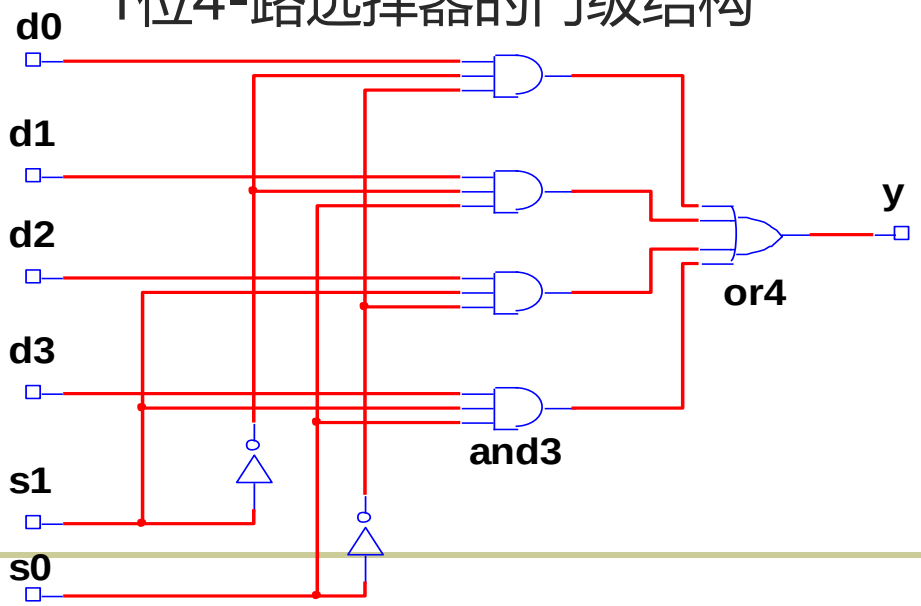
- **结构化建模**
  - 将电路描述成一个分级的子模块系统，并通过逐层调用子模块来构成功能复杂的数字系统。
  - 根据子模块不同的抽象级别，结构化建模方式分成以下三类：

模块级：调用用户设计的子模块(如前例sim_top模块调用top子模块)

门级：调用Verilog提供的基本门级元件 (以下4-路选择器举例)

开关级：调用Verilog内建的基本开关元件 (无须了解)

1位4-路选择器的门级结构




```
……
// 调用非门，生成s1_n 和 s0_n
not (s1_n, s1);
not (s0_n, s0);
// 调用三输入与门
and (y0, d0, s1_n, s0_n);
and (y1, d1, s1_n, s0);
and (y2, d2, s1, s0_n);
and (y3, d3, s1, s0);
// 调用四输入或门
or (y, y0, y1, y2, y3);
……
```

3.1 三种建模方式

- 数据流建模
  - 从数据的视角出发，通过描述数据在电路中的流动方向来给出电路的功能。
  - 适合组合逻辑电路的描述，不适合时序逻辑电路的描述。
  - 主要通过连续赋值语句进行建模，语法为：

    assign 网线 = 表达式;

    用赋值号右边的表达式对应电路的输出来驱动赋值号左边的网线
  - 注意点
    - 赋值号右边数据类型可以是wire和reg，左边则不能是reg类型
    - 多个连续赋值语句之间是并行关系，没有前后之分
    - 对一个未定义的变量赋值，相当于驱动一个位宽为1的wire型变量
    - 相同的网线不能进行重复驱动。
    - 描述的电路不能出现组合回路。

3.1 三种建模方式

例：使用数据流模式实现1位四路选择器

使用布尔表达式来代替基本门级元件的实例化

```
module mux4_to_1 (
    output y,
    input d0, d1, d2, d3,
    input s0, s1
);
    assign y=(~s1 & ~s0 & d0) | (~s1 & s0 & d1) | (s1 & ~s0 & d2) | (s1 & s0 & d3);
endmodule
```

采用条件运算符来描述

```
module mux4_to_1 (
      output y,
      input d0, d1, d2, d3,
      input s0, s1
);
      assign y = (s1) ? (s0 ? d3 : d2) : (s0 ? d1 : d0);
endmodule
```

数据流建模方式比门级结构化建模方式的代码更简洁。

在现代数字系统的开发过程中，大部分组合逻辑电路都采用数据流建模方式来描述。

3.1 三种建模方式

- **行为建模方式**
  - 通过高级编程语句和结构编写的过程块来描述数字系统的功能。
  - 从系统功能出发进行描述，不关注电路的具体结构，由综合器根据过程块所描述的行为**综合出**电路结构。
  - 关键要素是always语句，基本语法：

    always @(事件信号列表) 过程语句
    - 含义：当事件信号列表中的任意一个信号发生变化时，过程语句中的信号将按照所描述的行为进行更新。
    - 过程语句中被赋值的左边变量只能是reg类型变量。
    - 事件信号列表中有多个信号时，需用关键字or或者逗号来连接。
    - （*）隐式事件信号列表，*表示过程语句赋值号右侧所有信号集合，含义是赋值号右侧任一信号发生变化都会导致左侧信号更新
    - 可通过隐式事件信号列表的always语句对组合逻辑电路进行建模

3.1 三种建模方式

例：使用行为建模方式实现1位四路选择器

```
module mux4_to_1 (
      output reg y,          // 注意此处为reg类型
      input d0, d1, d2, d3,
      input s0, s1
);
      always @(*)            // 相当于 @(s1, s0, d0, d1, d2, d3)
            case ({s1, s0})
                  2'b00: y = d0;
                  2'b01: y = d1;
                  2'b10: y = d2;
                  2'b11: y = d3;
             endcase
endmodule
```

3.2 行为建模中的过程语句

1. **begin-end复合语句**
   - 在语法上将多条语句看成一条语句。类似{ }。
2. **两种过程赋值语句**
   - 阻塞赋值语句，其语法为：寄存器 = 表达式;
     - 立即更新，后续语句中按照新值计算
     - 按在代码中出现的顺序执行
     - 通常用来描述组合逻辑电路
   - 非阻塞赋值语句，其语法为：寄存器 <= 表达式;
     - 滞后更新，新值在其它所有操作都结束后才会进行更新
     - 一组非阻塞赋值语句是并行执行的
     - 利用时钟变量作为事件信号时，赋值符合数据存储的特点
     - 通常用于描述时序逻辑电路

3.2 行为建模中的过程语句

2. 两种过程赋值语句（续）

- 例1：用两种过程赋值语句对全加器建模有何不同？

```
always @(*)
    begin
        p = a ^ b;
        g = a & b;
        s = p ^ cin;
        cout = g | (p & cin);
    end
```

```
always @(*)
    begin
        p <= a ^ b;
        g <= a & b;
        s <= p ^ cin;
        cout <= g | (p & cin);
    end
```

若开始a、b、cin都是0，则输出p、g、s、cout都是0。随后a从0改为1，则（阻塞方式下按顺序执行赋值语句，非阻塞方式下并行执行赋值语句）

阻塞赋值方式下，p=1⊕0=1，g=1•0=0，s=1⊕0=1，cout=0+(1•0)=0

非阻塞赋值方式，p=1⊕0=1，g=1•0=0，s=0⊕0=0，cout=0+(0•0)=0。

由此可见，采用非阻塞方式实现的加法器有问题！

3.2 行为建模中的过程语句

2. 两种过程赋值语句（续）

- 例2：用非阻塞方式对计数器建模。

```
reg [31:0] cnt;
    always @ (posedge clk)
        cnt <= cnt + 1'd1;
```

- 上述代码可以综合出一个“每来一个时钟自增1”的计数器
- 赋值号右边的cnt是时钟到来前的值，左边cnt是增量后的
- posedge表示在时钟clk的上升沿触发
- negedge表示在时钟clk的下降沿触发

3.2 行为建模中的过程语句

3. if-else语句

- 根据判断条件的真假更新相应分支中的寄存器变量。
- 格式：
  - if (表达式) 语句1；
  - if (表达式) 语句1；else 语句2；
  - if (表达式1) 语句1；
    else if （表达式2）语句2；
    else if （表达式n） 语句n；
    - 这里“表达式”为逻辑表达式或关系表达式，值为0或z，判定结果为“假”；值为1，判断结果为“真”。
    - 多语句时使用“begin-end”，形成复合块语句。
- 可转化为条件运算符。
- 可综合成选择电路，即多路选择器。

3.2 行为建模中的过程语句

4. case语句

- 根据表达式的值从多个分支中选择其中一个并按照所选择分支的描述来更新寄存器变量。
- case语句的语法为：

```
case（选择表达式）
    值1：语句1；
    值2：语句2；
      …
    值n：语句n；
    [default: 语句n+1；]
endcase
```

- 每个“值”可以是表达式
- 可转换为嵌套的if-else
- 用于译码器、多路选择器等
- 在组合逻辑电路设计中，case要覆盖所有分支情况，否则会综合成锁存器保持原值，与预期电路不符。

3.2 行为建模中的过程语句

5. 循环语句

- 提供了对于相似结构电路的一种简洁描述方式。
- 在高级语言中，循环语句表示时间上重复执行的相似代码；而在HDL中，循环语句表示在空间上重复描述的电路。
- 使用较多的是for循环结构：

  for （表达式1；表达式2；表达式3）语句
- 并不存在“执行”for循环的电路，只是空间上有多个相似电路
- 若循环结束条件表达式中除了循环变量外还有其他变量的话，电路会变得很复杂

例如，for (i = 0; i < threshold; i = i + 1) 将综合出比较器

for (i = 0; i < 8; i = i + 1) 仅综合出8个相似的电路即可

4 verilog代码实例

- 组合逻辑代码实例
- 时序逻辑代码实例

4.1 组合逻辑电路代码设计

- **设计3-8译码器**

```
module decode3to8 (
    output reg [7:0] out,
    input [2:0] in
);
    always @(*)
      case (in)
        0: out = 8'b00000001;
        1: out = 8'b00000010;
        2: out = 8'b00000100;
        3: out = 8'b00001000;
        4: out = 8'b00010000;
        5: out = 8'b00100000;
        6: out = 8'b01000000;
        7: out = 8'b10000000;
      endcase
endmodule
```

行为建模方式

```
module decode3to8 (
      output reg [7:0] out,
      input [2:0] in
);
      integer i;
      always @(*)
          for (i = 0; i < 8; i = i + 1)
                out[i] = (in == i);
   endmodule
```

行为建模方式

```
module decode3to8 (
      output [7:0] out,
      input [2:0] in
);
      assign out = 1 << in;
endmodule
```

数据流建模方式

4.1 组合逻辑电路代码设计

- **七段数码管译码电路**

仅包含0~9这10种情况，因而在case语句最后需加default分支，使得当I为0~9以外的数时，能够将O设置为0。否则，综合器将会综合出锁存器，使得在输入0~9以外的数时保留上次的输出，这与预期的功能不符。

```
module decode7seg (
    output reg [0:6] O,
    input [0:3] I
);
    always @(*)
        case (I)
            0: O = 7'b1111110;
            1: O = 7'b0110000;
            2: O = 7'b1101101;
            3: O = 7'b1111001;
            4: O = 7'b0110011;
            5: O = 7'b1011011;
            6: O = 7'b1011111;
            7: O = 7'b1110000;
            8: O = 7'b1111111;
            9: O = 7'b1110111;
            default: O = 7'b0;
        endcase
endmodule
```

4.2 时序逻辑电路设计

- 设计带复位端rst和使能端en的D触发器

```
module Dff (
    input clk, rst, en, d,
    output reg q
);
    always @(posedge clk)
        begin
            if (rst) q <= 1'b0;
            else if (en) q <= d;
        end
endmodule
```

同步复位方式

只有当时钟上升沿到达后，才能进行复位

```
module AsyncDff (
    input clk, rst, en, d,
    output reg q
);
    always @(posedge clk or posedge rst)
        begin
            if (rst) q <= 1'b0;
            else if (en) q <= d;
        end
endmodule
```

异步复位方式

只要复位信号rst从0变为1就可以复位，无需等待时钟上升沿到来

4.2 时序逻辑电路设计

- **自动售货机出售1.5元的零食，只接收五角和一元两种输入**

这里用case语句实现了一个有限状态机

```
module VendingMachine (
  input clk, arst,  //clk-时钟信号，arst-异步复位信号
  input in5, in10, // 是否投了五角或一元硬币
  output snack,   // 是否出货
  output out5    // 是否找五角零钱
);
  reg [2:0] state;  //5个状态，需要3位编码
  parameter s_idle=0,s_05=1,s_10=2,s_15=3,s_20=4;  //状态编码赋值给状态名称
  always @(posedge clk or posedge arst) begin
    if (arst) state <= s_idle;
    else begin
     case (state)
        s_idle: begin
            if (in5) state <= s_05;
            if (in10) state <= s_10;
         end
       s_05: begin
            if (in5) state <= s_10;
            if (in10) state <= s_15;
       end
            s_10: begin
                if (in5) state <= s_15;
                if (in10) state <= s_20;
            end
            s_15: state <= s_idle;
            s_20: state <= s_idle;
        endcase
    end
  end
  assign snack=(state==s_15)||(state==s_20);
  assign out5=(state == s_20);
endmodule
```

5 仿真测试模块

5 仿真测试模块

- 仿真测试模块用于对Verilog设计的电路进行逻辑仿真测试，设置一个输入序列，加载到一个HDL设计中，验证电路功能，测试电路性能。
- initial语句：在零时刻执行一次。用于初始化或生成激励信号
  - 在仿真的初始状态对各变量进行初始化；
  - 在测试文件中生成激励波形作为电路的仿真信号。
- 测试模块的结构:
  - 1、产生时钟信号：缺省使用系统时钟来产生时钟。
  - 2、准备激励信号：在测试设计中使用的并行激励块提供必要的激励。通常仿真变量被描述为相对于仿真时间零点。通过设置延迟时间来观察输出。
  - 3、显示结果：通过观察输入输出波形图来验证；也可以调用$display和 $write 等任务去打印出每一执行后的结果。

5 仿真测试模块

- 激励输入序列
  - 交互式/典型输入值/随机值$random()
  - 自检测试平台：将测试平台的输出与所期望的输出一一比较，如果有错误的话，跟踪数量，并显示差异。
- 测试模块结构：

module 测试模块名； //没有输入和输出端口

数据类型声明

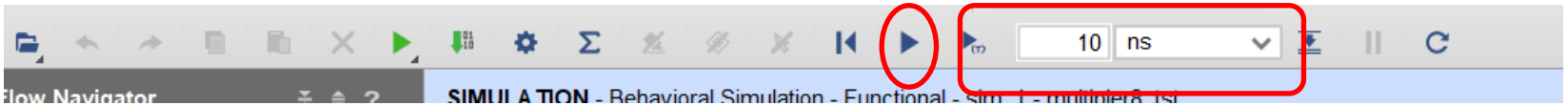
实例化被测试模块

产生测试激励信号

对输出响应进行收集

endmodule

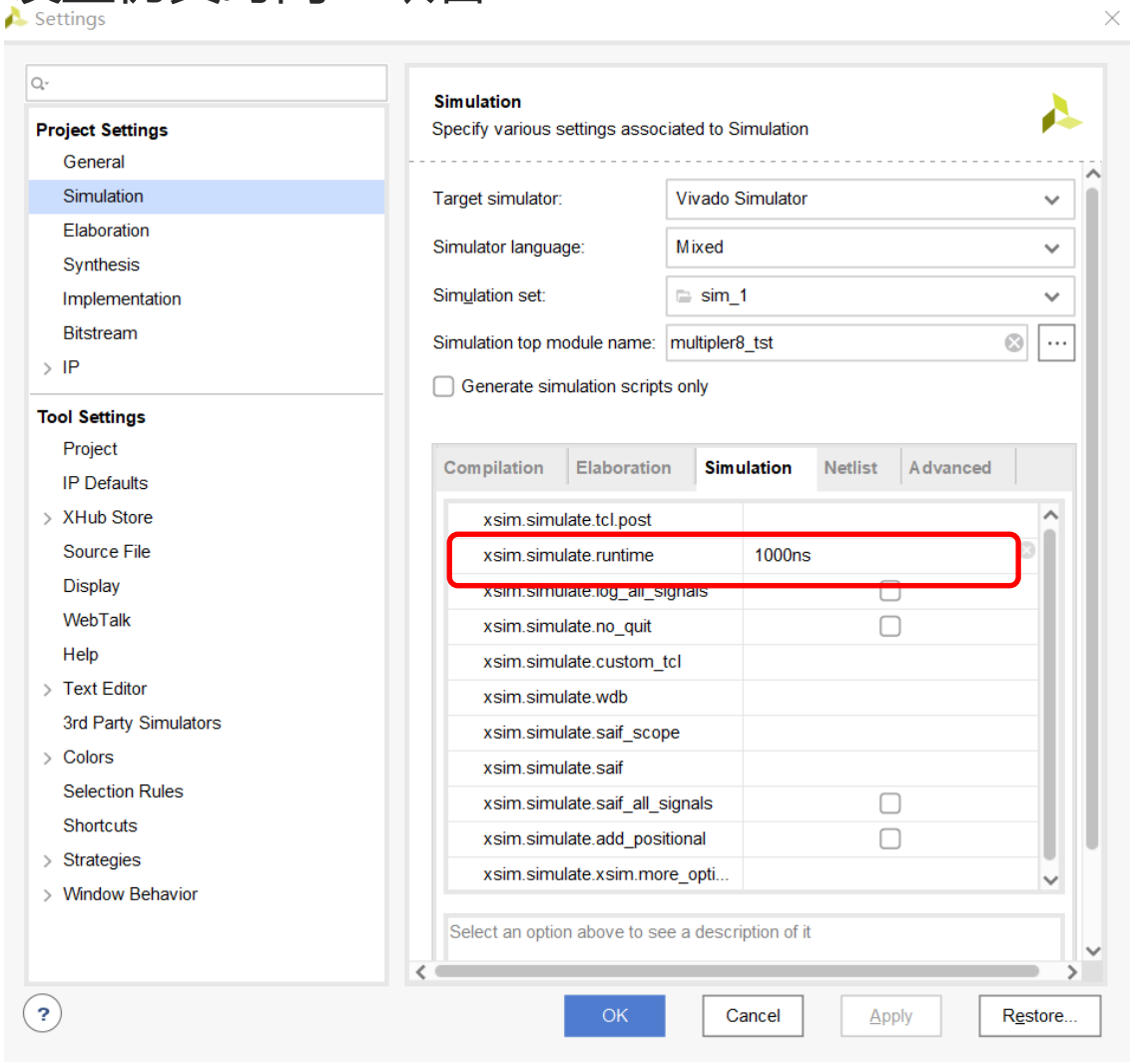
- #n:延迟n个时间单位
- $stop任务：暂停当前仿真。$finish：仿真停止。



持续执行

5 仿真测试模块

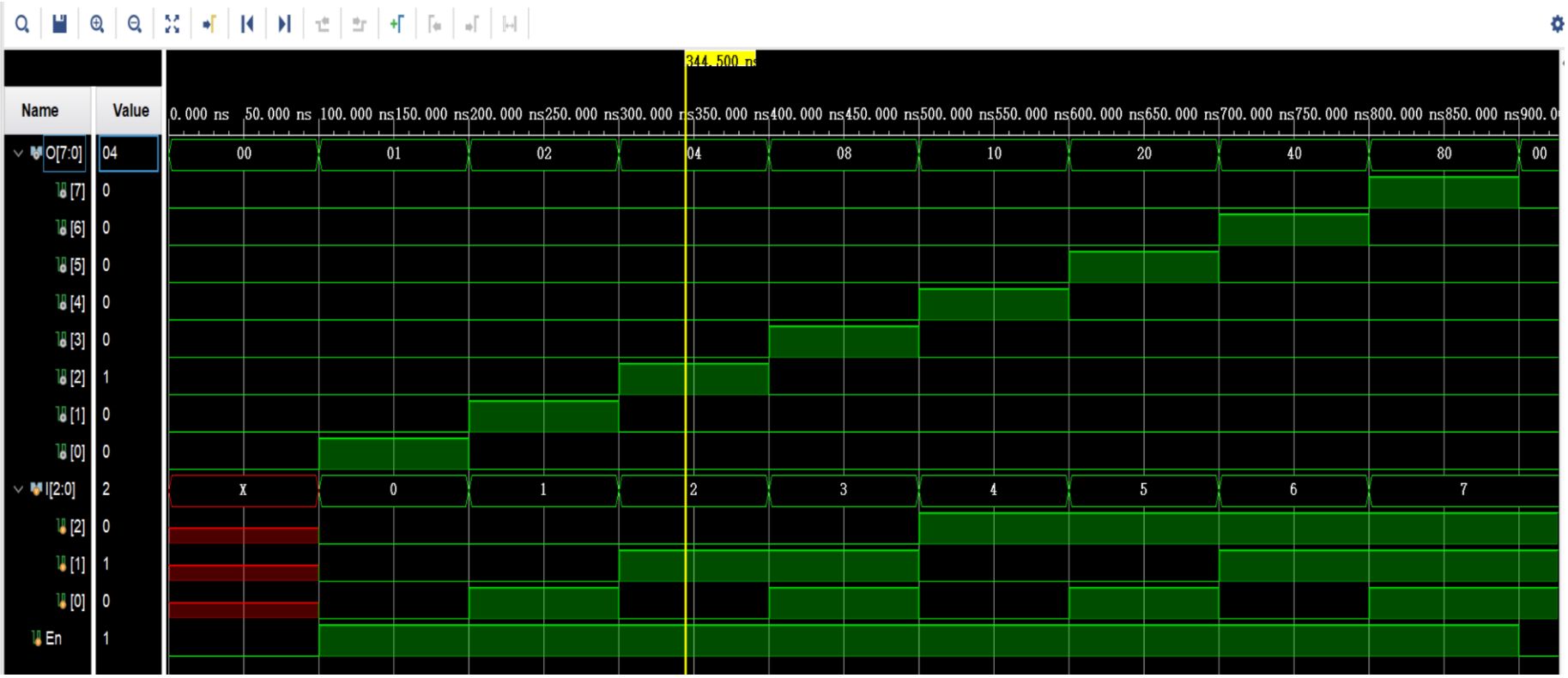
- 在Vivado设置仿真时间：缺省1000ns



3-8译码器测试模块

```
`timescale 1ns / 1ps
module decode3to8_tst(  );
    wire [7:0] O;
    reg [2:0] I;
    reg En;
 decode3to8 decode3to8_impl(.O(O),.I(I),.En(En));
    initial begin
    begin En = 1'b0; end
    #100 begin En = 1'b1; I = 3'b000; end
    #100 begin En = 1'b1; I = 3'b001; end
    #100 begin En = 1'b1; I = 3'b010; end
    #100 begin En = 1'b1; I = 3'b011; end
    #100 begin En = 1'b1; I = 3'b100; end
    #100 begin En = 1'b1; I = 3'b101; end
    #100 begin En = 1'b1; I = 3'b110; end
    #100 begin En = 1'b1; I = 3'b111; end
    #100 begin En = 1'b0; end
    end
endmodule
```

测试波形

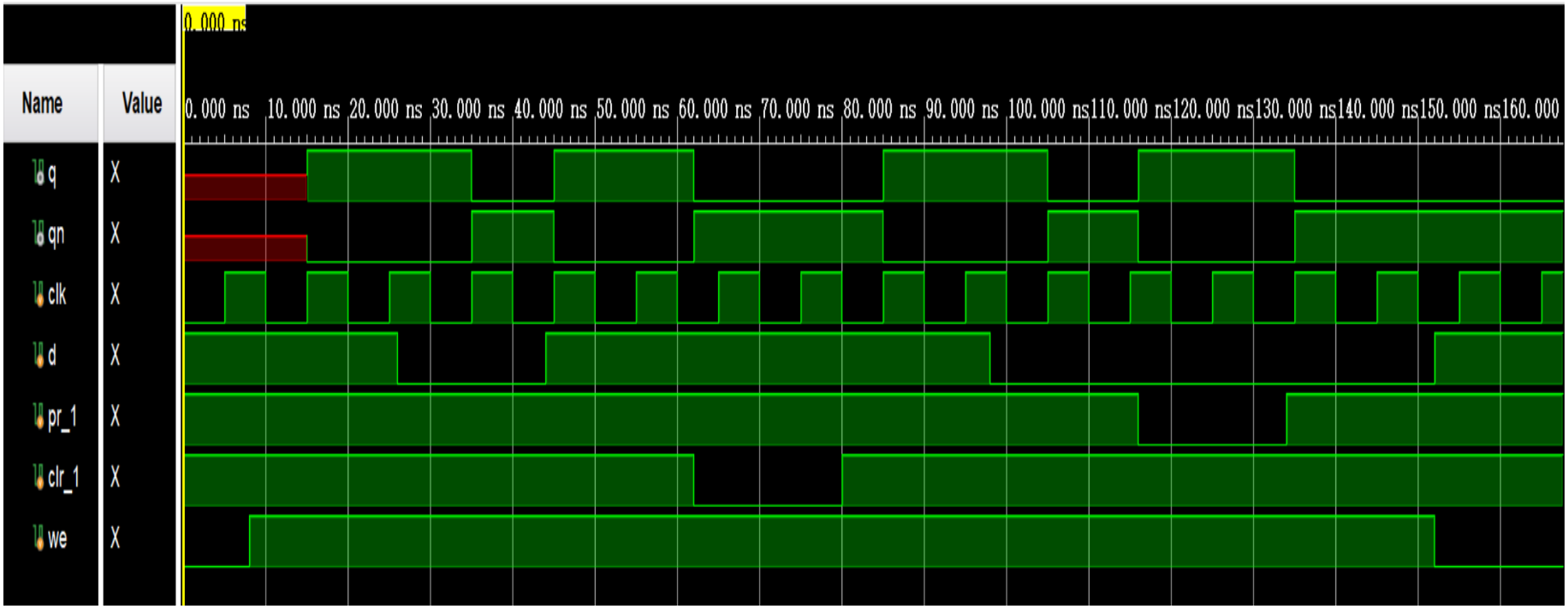

344.500 ns
Name
Value
0.000 ns
50.000 ns
100.000 ns
150.000 ns
200.000 ns
250.000 ns
300.000 ns
350.000 ns
400.000 ns
450.000 ns
500.000 ns
550.000 ns
600.000 ns
650.000 ns
700.000 ns
750.000 ns
800.000 ns
850.000 ns
900.000 ns
O[7:0]
04
00
01
02
04
08
10
20
40
80
00
O[7]
O[6]
O[5]
O[4]
O[3]
O[2]
O[1]
O[0]
I[2:0]
2
X
0
1
2
3
4
5
6
7
I[2]
I[1]
I[0]
En
1

1位异步清零置位寄存器

```
module d_register(
    output q,qn,
    input clk,d,pr_1,clr_1,we
);
    reg q_r;
    wire set;
    assign set = pr_1 & clr_1;
    always@(posedge clk or negedge set)
begin
        if(!set)
            q_r <= clr_1;
        else if(we == 1)
            q_r <= d;
    end
    assign q = q_r;
    assign qn = ~q_r;
endmodule
```

```
`timescale 1ns / 1ps
module d_register_tst();
wire q,qn;
reg clk,d,pr_1,clr_1,we;
d_register d_register_inst(q,qn,clk,d,pr_1,clr_1,we);
always
# 5 clk=~clk;
initial begin
clk = 0;d = 1;clr_1 = 1;pr_1 = 1;we = 0;
#8 we = 1;
#18 d = 0;
#18 d = 1;
#18 clr_1 = 0;
#18 clr_1 = 1;
#18  d = 0;
#18  pr_1 = 0;
#18  pr_1 = 1;
#18  we = 0; d = 1;
#18  pr_1 = 0;
#18  $stop;
end
endmodule
```

测试波形


Name
Value
q
qn
clk
d
pr_1
clr_1
we
X
0.000 ns
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
70.000 ns
80.000 ns
90.000 ns
100.000 ns
110.000 ns
120.000 ns
130.000 ns
140.000 ns
150.000 ns
160.000

分析波形图

4位移位寄存器测试模块

```
`timescale 1ns / 1ps
module shrg4u_tst(    );
reg Tclk, CLR, S0, S1, RIN, LIN;
reg [3:0] I;                        // A-D = I[3:0]
wire [3:0] Q;                       // QA-QD = Q[3:0]

shrg4u UUT ( .CLK(Tclk), .CLR(CLR), .RIN(RIN), .LIN(LIN), .S0(S0), .S1(S1),
             .A(I[3]), .B(I[2]), .C(I[1]), .D(I[0]),
             .QA(Q[3]), .QB(Q[2]), .QC(Q[1]), .QD(Q[0]) );

always begin
  #0.5 ; Tclk = 1'b1; #5 ;    // 定义时钟周期10ns，上升沿发生在 at 10.5ns, 20.5ns,....
  Tclk = 1'b0; #4.5 ;
end
```

4位移位寄存器测试模块

```
initial begin : TB
integer ii, j;
  #10 ;                                          // 等待实验板系统复位时间
  RIN = 1'b0; LIN = 1'b0;                        // 设置左、右移入位为0
  for (ii=0; ii<=15; ii=ii+1) begin              // 验证载入、保持和清零功能
    CLR = 1'b0; {S1,S0} = 2'b11; I[3:0] = ii; #10 ;     // 载入数据，并等待有效后验证
    if (Q != I[3:0]) $display("S1S0 = 11, ABCD = %4b, QA-QD = %4b, load failed", I, Q);
    {S1,S0} = 2'b00; #10 ;                       // 保持数据,并等待时钟信号有效后验证
    if (Q != I[3:0]) $display("S1S0 = 00, ABCD = %4b, now QA-QD = %4b, hold failed",
I, Q);
    CLR = 1'b1; #10 ;                            // 清零信号有效，并等待时钟信号有效后验证
    if (Q != 4'b0) $display("CLR = 1, QA-QD = %4b, clear failed", Q);
  end
  $display("Clear, load, and hold test completed");
  CLR = 1'b0; LIN=1'b1;                          //清零信号无效，进入移位状态
```

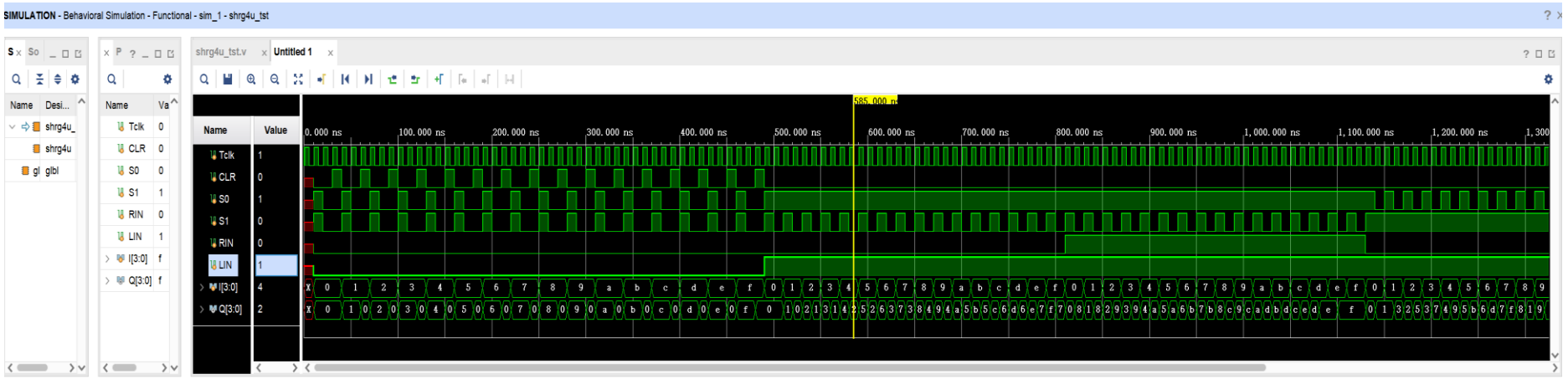
4位移位寄存器测试模块

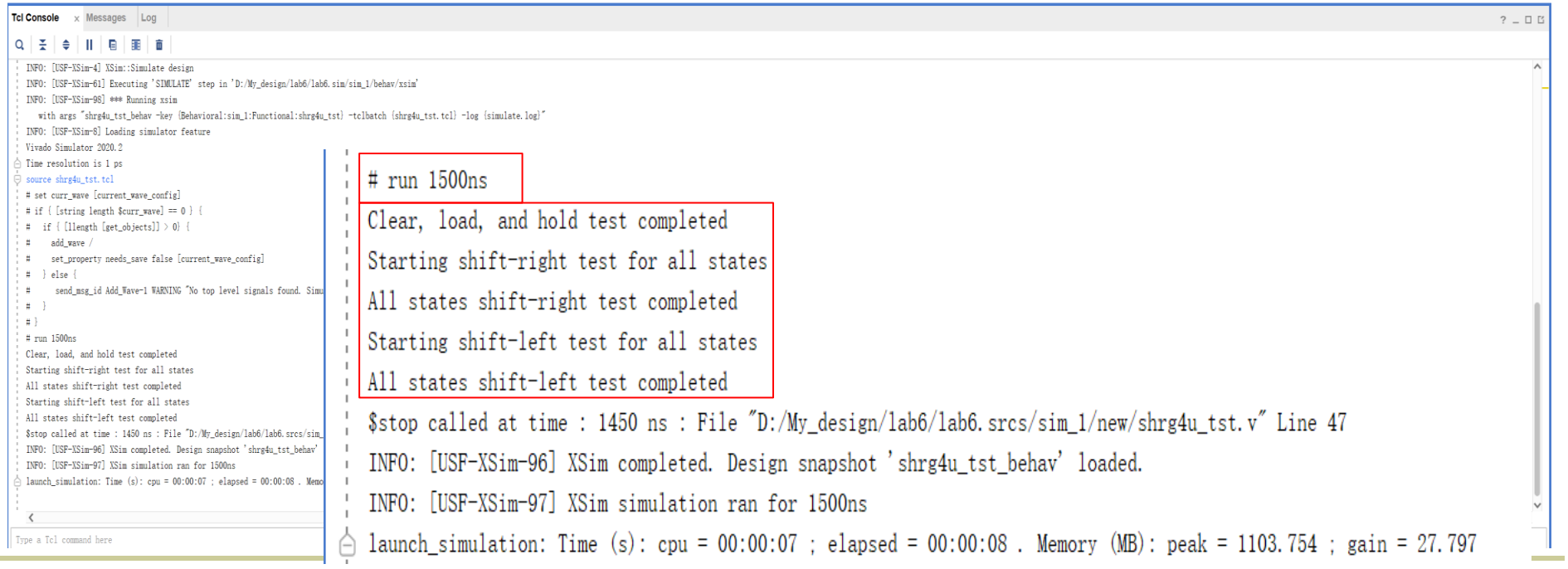
```
$display("Starting shift-right test for all states");
  for (ii=0; ii<=31; ii=ii+1) begin          // 右移测试
    {S1,S0} = 2'b11; { RIN, I[3:0]} = ii[4:0]; #10 ; // 载入初始数据，等待时钟信号
    {S1,S0} = 2'b01;  #10 ;                  // 右移，等待时钟信号有效后验证
    if (Q != {RIN,ii[3:1]})
        $display("S1S0 = 01, old QA-QD = %4b, RIN,LIN = %2b, QA-QD = %4b,
shift-right failed", ii[3:0], {RIN,LIN}, Q);
  end
  $display("All states shift-right test completed");
```

4位移位寄存器测试模块

```
 $display("Starting shift-left test for all states");
  RIN = 1'b0; LIN = 1'b1;                    // 左移移入位置 1
  for (ii=0; ii<=15; ii=ii+1) begin          // 左移测试
    {S1,S0} = 2'b11; I[3:0] = ii; #10 ;      //载入初始数据，等待时钟信号
    {S1,S0} = 2'b10;  #10 ;                  // 左移，等待时钟信号有效后验证
    if (Q != {ii[2:0],LIN})
        $display("S1S0 = 10, old QA-QD = %4b, RIN,LIN = %2b, QA-QD = %4b,
shift-left failed", ii[3:0], {RIN,LIN}, Q);
  end
  $display("All states shift-left test completed");
  $stop;
end
endmodule
```

4位移位寄存器测试模块





更多内容

- Verilog 2005/IEEE.1364-2005

咋写Verilog

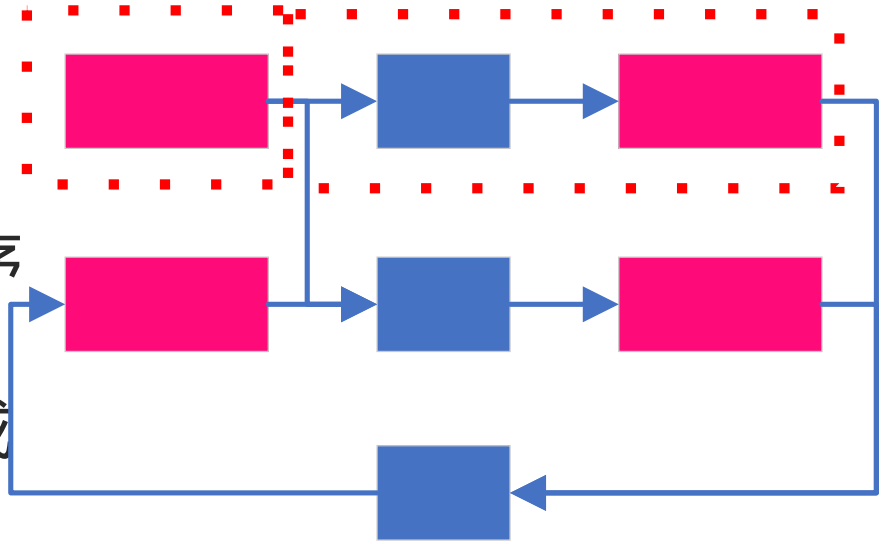
许嘉帆

软件&硬件

- 软件
  - 写文章
  - 执行顺序有先后
- 硬件（时序设计）
  - 画图
  - 执行顺序无先后
  - 代码块的顺序不重要，重要的是每块代码要写清楚

逻辑电路和时序电路

- 逻辑电路
  - 类似软件的逻辑
  - 赋值有先后顺序
- 测试电路
  - 类似软件逻辑，有先后顺序
- 时序电路
  - 主要由Reg和逻辑电路组成
  - 编写以always块为单位
  - 每个块编写一个(或多个)reg的前驱电路



reg变量和wire变量的区别

- reg能够在always语句块中进行赋值，wire不能
- assign和wire只是对变量的重命名
- reg变量不都会变成register
  - 通常在设计声明时就已经确认这个变量是啥类型，我一般在真reg的变量后添加_r来做一个标记，比如:state_r
  - 真reg变量的赋值全部都使用阻塞赋值，假reg变量的赋值全部都使用非阻塞赋值
- 事实上，大家可以把所有实体对应变量分为register和wire，其中register的声明一定是reg，而wire的命名可能是wire，也可能是reg。

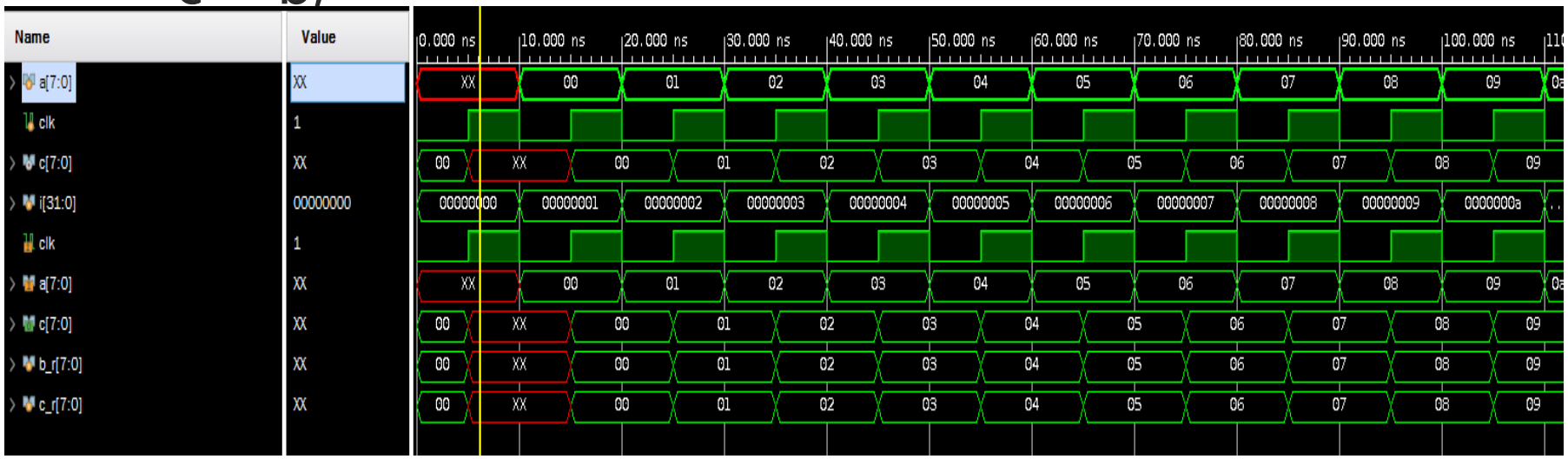
重点

- 一个Reg变量的赋值不能出现在2个always块中
- 一个always语句块中不要出现2种赋值
- 一个Reg变量不要出现2种赋值
- 对于reg类型最后变成wire的变量，需要对出现的所有情况考虑，再赋值，不允许出现某个if下面没有对其赋值。
- always @(posedge clk) 块下不要出现 =
- 一个always要么用于描述时序电路，要么用于描述逻辑电路
  - always @(posedge clk) begin … end
  - always @(*) begin … end

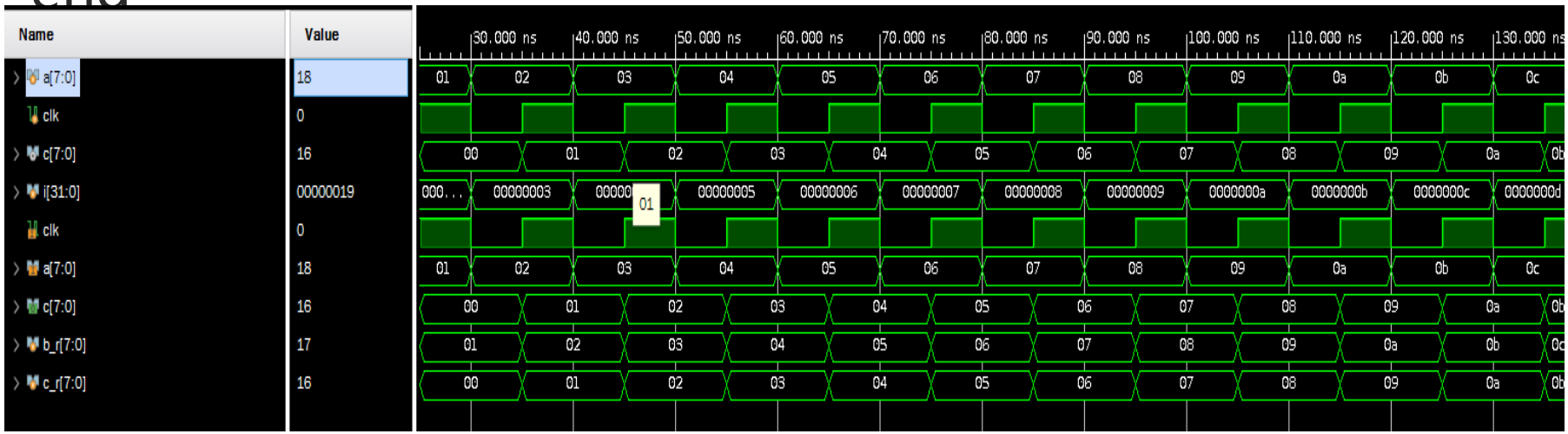
阻塞赋值 = 和非阻塞赋值 <=

- 阻塞赋值有先后
- 非阻塞赋值在时钟上升沿事件结束后统一赋值

```
input [7:0]a;
reg [7:0]b; reg [7:0]c;
always @(posedge clk)
begin
        b = a;
        c = b;
```



```
input [7:0]a;
reg [7:0]b; reg [7:0]c;
always @(posedge clk) begin
        b <= a;
        c <= b;
end
```



always @(posedge clk) 不要出现 =

- https://blog.csdn.net/weixin_38679924/article/details/102524574

Verilog中的for循环

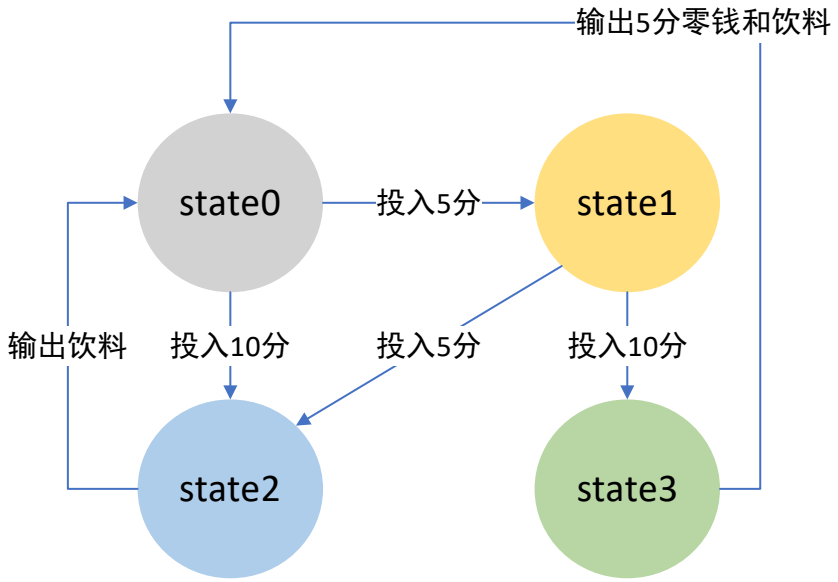
- 设计阶段的for循环只是为了简化很多条类似的语句，不要将其和软件代码中的for循环搞混
  - 前导0

- 更多的for循环代码是用于编写测试代码时（类似软件）

代码鉴赏环节

单状态机设计

- https://www.educoder.net/tasks/lmy39o4sfeu8
- 设计一个自动饮料售卖机，饮料10分钱，硬币有5分和10分两种，并考虑找零。
  - a.画出fsm（有限状态机）

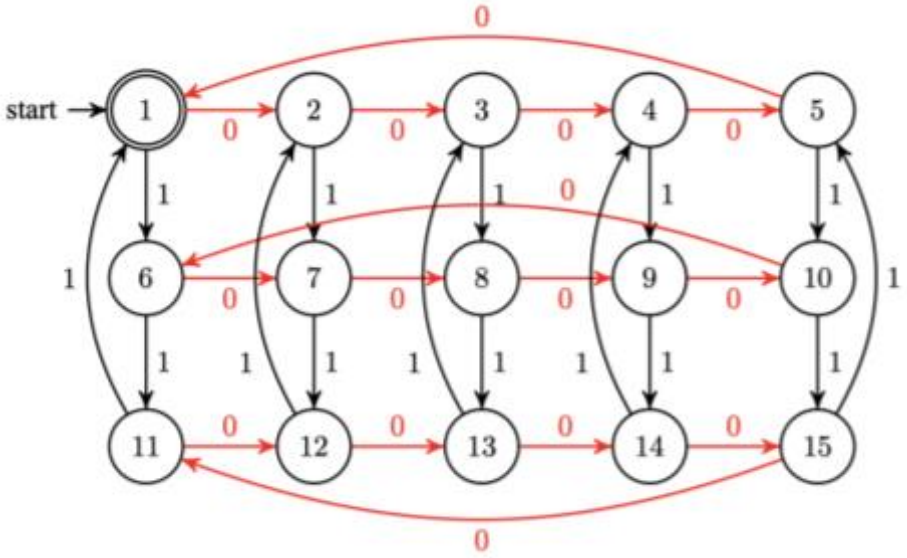


状态机代码模板

```
module xx(xxx);
    reg [2:0]state_r;
    always @(posedge clk) begin
        //这里用Case也可以。
        if (state_r == 0) begin
            if(xxx) begin
                  state_r <= ??
            end
        end
        else if(state_r == 1) begin
            if(xxx) begin
              state_r <= ??
            end
        end
        ....
        else begin
            ...
        end
    end
endmodule
```

多标识的状态机

- 模块接受一个01串，判断当前已经输入的01串中，是否0的个数是5的倍数且1的个数是3的倍数。



我们使用俩个标识来标记这个状态机。