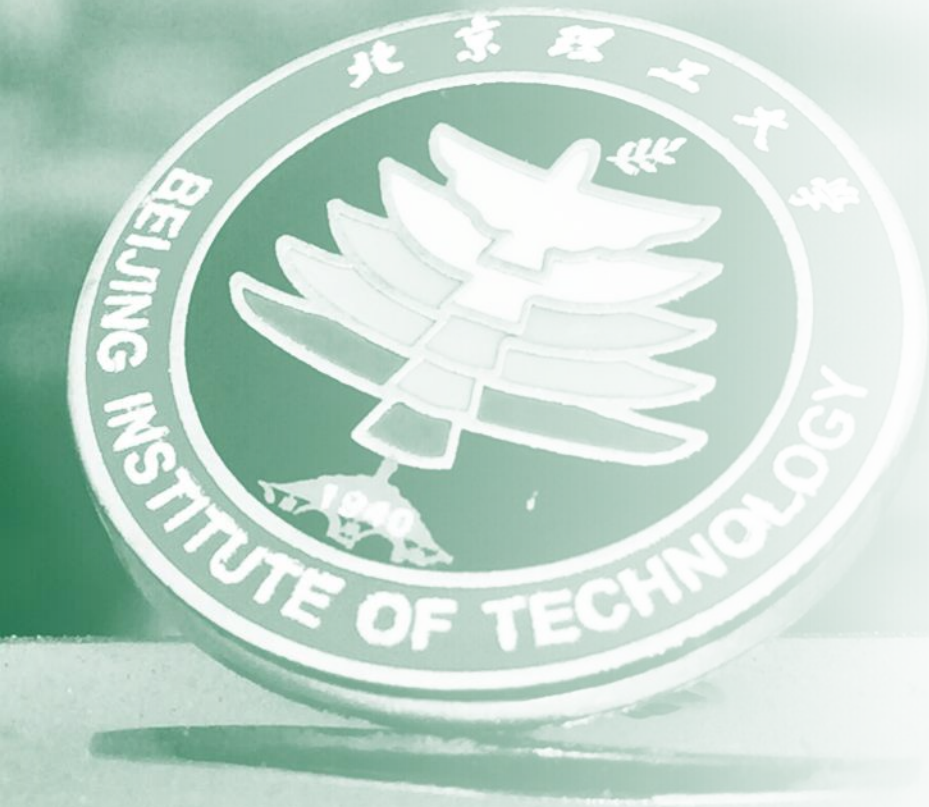
VerilogHDL 语言简介

主讲人：王赞



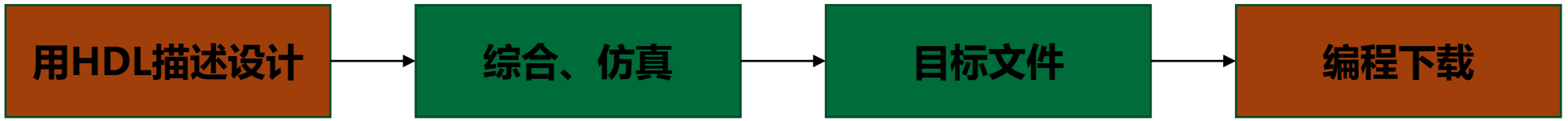
目录 | CONTENTS

1 概述
2 数据类型
3 模块构建
4 功能描述
5 系统函数
6 代码示例

1 概述

硬件描述语言简介

- 硬件描述语言（Hardware Description Language）是一种用形式化方法（即文本形式）来描述和设计数字电路和数字系统的高级模块化语言。它是设计人员和EDA工具之间的一个桥梁，主要用于编写设计文件，在EDA工具中建立电路模型；也用来编写测试文件进行仿真
- HDL发展至今已有近三十年的历史，到20世纪80年代，已出现了数十种硬件描述语言。80年代后期，HDL向着标准化、集成化的方向发展，VHDL、Verilog HDL先后成为IEEE标准



Verilog HDL 特点

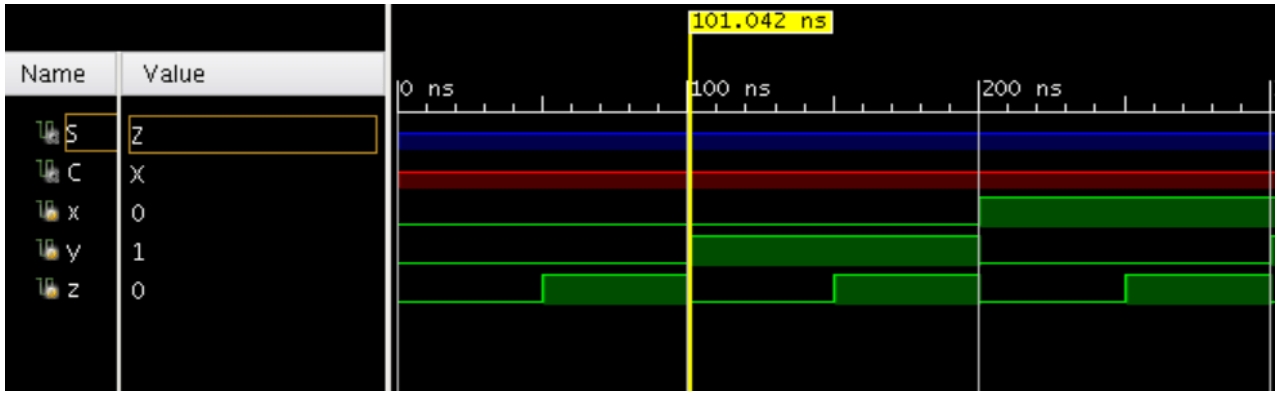
- 可以形式化地表示电路的行为和结构
- 可以在多个层次上对所设计的系统加以描述
- 具有混合建模的能力，一个模块可以使用不同级别抽象模型来描述
- 基本逻辑门、开关级结构模型可直接调用
- 易创建用户定义原语
- 仿真工具比较好用
- 较多第三方工具支持
- 易学易用，功能强

2 数据类型

数据类型——变量值

值	含义
0	代表逻辑0或否条件
1	代表逻辑1或真条件
X	代表未知的逻辑值（可能是0也可能是1）
Z	代表一个高阻态

- 0在电路中一般是低电平
- 1在电路中是高电平
- X一般是寄存器类型（reg）未初始化
- Z是线型（wire）变量未接驱动



数据类型——wire & reg

- Verilog 中最常用的两种数据类型就是线网（wire）和寄存器（reg）
- wire 类型用来表示硬件单元之间的物理连线

```
2  wire    a;
3  wire    b;
4  wire    c = 1'b0;
```

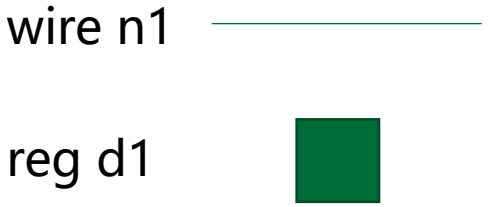
- reg 类型用来表示存储单元

```
2  reg       rstn;
3  initial begin
4      rstn = 1'b0;
5      #100;
6      rstn = 1'b1;
7  end
```

数据类型——向量 Vector

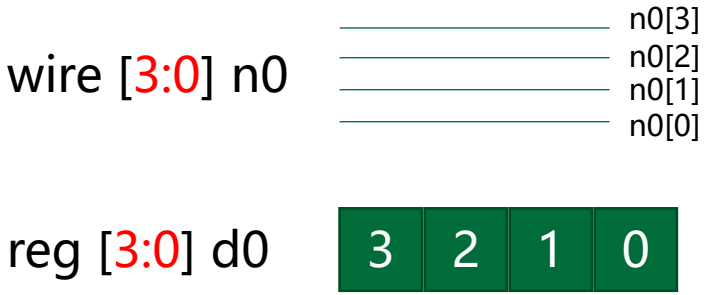
标量Scalar

1-bit位宽的线网或寄存器类型



向量Vector

n-bit位宽的线网或寄存器类型



数据类型——数组 Array

- 在 Verilog 中允许声明 reg, wire, integer, time, real 及其向量类型的数组

```
reg        y1 [11:0];        // y is an scalar reg array of depth=12, each 1-bit wide
wire [0:7] y2 [3:0]          // y is an 8-bit vector net with a depth of 4
reg  [7:0] y3 [0:1][0:3];    // y is a 2D array rows=2,cols=4 each 8-bit wide
```

- 存储器：寄存器数组模拟存储器，可用来描述 RAM 或 ROM 的行为

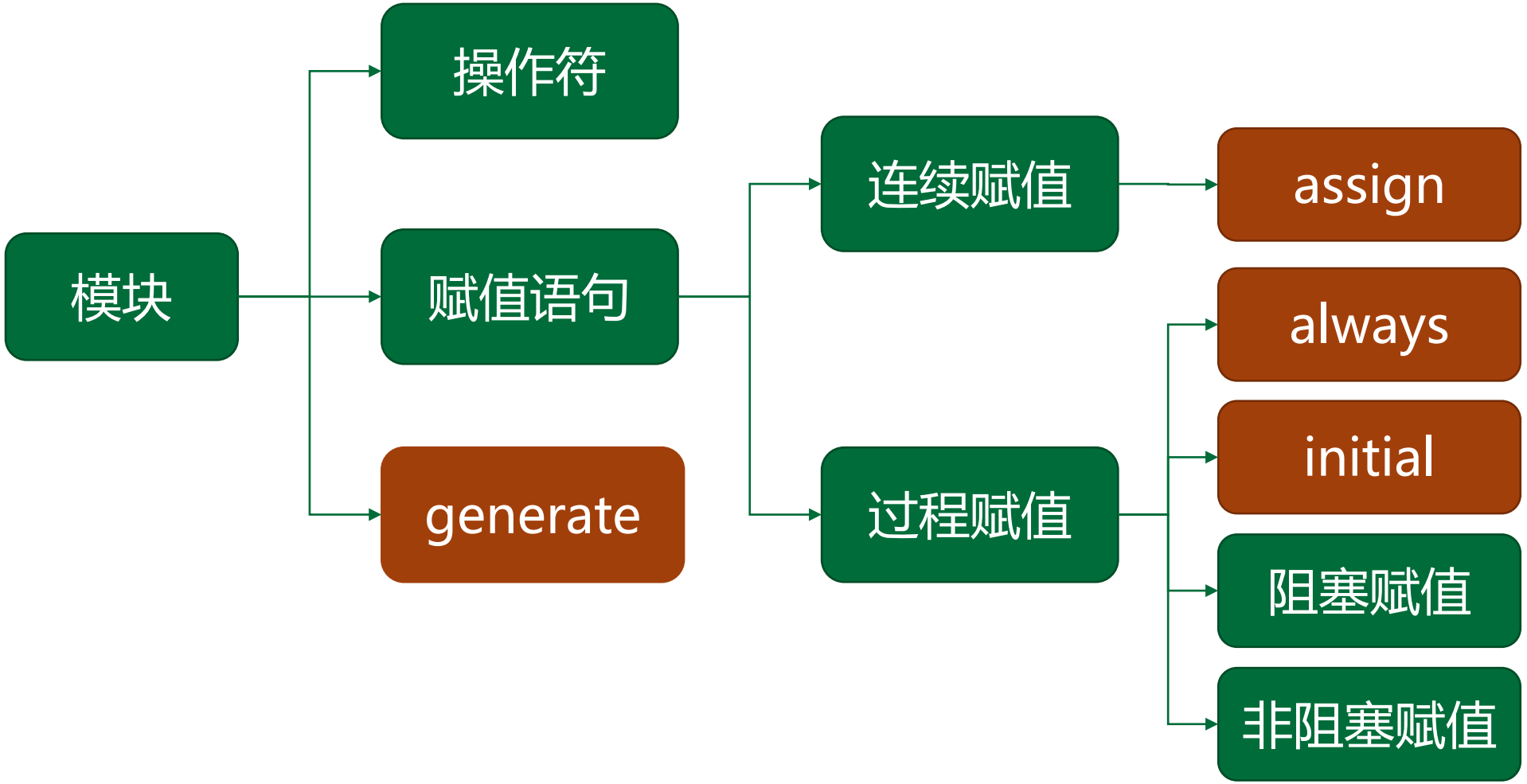
```
module des ();
  reg [7:0]  mem1;                          // reg vector 8-bit wide
  reg [7:0]  mem2 [0:3];                 // 8-bit wide vector array with depth=4
  reg [15:0] mem3 [0:3][0:1];   // 16-bit wide vector 2D array with rows=4,cols=2

  initial begin
    mem1 = 8'ha9;
    $display ("mem1 = 0x%0h", mem1);
    ...
```

数据类型——其他类型

- integer 是32位宽的通用整型变量，可在对硬件建模时用于其他目的（可综合）
- time 变量是无符号的，64位宽，可用于存储仿真时间量以进行仿真调试，realtime 变量是将时间存储为浮点数（不可综合）
- real 实数变量可以存储浮点值，可以与 integer 和 reg 相同的方式进行赋值（不可综合）
- string 字符串存储在 reg 中，reg 变量的宽度必须足够大以容纳字符串（可综合）

3 模块构建

模块构建——内容概要


模块
操作符
赋值语句
generate
连续赋值
过程赋值
assign
always
initial
阻塞赋值
非阻塞赋值

模块构建——模块 Module

Verilog 的基本设计单元是“模块”。

Verilog 模块的结构由在module和 endmodule 关键词之间的4个主要部分组成：

```
module block1(a,b,c,d );
    input a,b,c;
    output d;
    wire x;
    assign d = a | x;
    assign x = ( b & ~c );
endmodule
```

- **端口定义** module 模块名(端口1,端口2,...)
- **I/O 说明** 包括输入(input)、输出(output)和双向(inout)
- **信号类型声明** 声明信号的数据类型和函数声明wire, reg, integer, real, time
- **功能描述** 用来描述设计模块的内部结构和模块端口间的逻辑关系。常用 assign 语句、always 块语句等方法实现

模块构建——模块 Module

- 模块是具一个有特定功能的设计单元，在电路综合时模块会被转换为相应的数字电路
- 给定模块一组输入，模块会返回一组输出，这意味着模块可以被重复使用，由此来实现更加复杂的电路
- 按照如下形式实例化模块：

```
module mod1 (input d, ...);
    // Contents of the module
endmodule

module mod2;
    wire data;
    mod1 u0 (.d(data), ...);
    // Contents of the module
endmodule
```

模块构建——运算符 Operator

功能

- 算术运算符
- 逻辑运算符
- 关系运算符
- 等值运算符
- 缩减运算符
- 条件运算符
- 位运算符
- 移位运算符
- 位拼接运算符

运算符也称**操作符**，是Verilog HDL预定义的函数符号，这些函数对被操作的对象（即操作数）进行规定的运算，得到一个结果。

操作数个数

- 单目运算符
- 双目运算符
- 三目运算符

模块构建——运算符 Operator

算术运算符	功能
+	加
-	减
*	乘
/	除
%	求模
**	幂

逻辑运算符	功能
&&	逻辑与
||	逻辑或
!(单目)	逻辑非

位运算符	功能
~(单目)	按位取反
&	按位与
|	按位或
^	按位异或
^~,~^	按位同或

- 在逻辑运算中，如果操作数不止一位，应将操作数作为一个整体来对待
- 两个不同长度的操作数进行位运算时，将自动按右端对齐，位数少的操作数会在高位用0补齐。

模块构建——运算符 Operator

关系运算符	功能
<	小于
<=	小于或等于
>	大于
>=	大于或等于

等值运算符	功能
==	等于
!=	不等于
===	全等
!==	不全等

缩减运算符	功能
&(单目)	与
~&(单目)	与非
|(单目)	或
~|(单目)	或非
^(单目)	异或
^~,~^(单目)	同或

- 关系运算符优先级低于算术运算符，返回结果为逻辑值，0或1或x
- **等于运算符(==)和全等运算符(===)的区别**：使用等于运算符时，两个操作数必须逐位相等，结果才为1，若某些位为x或z，则结果为x；使用全等运算符时，若两个操作数的相应位形式上完全一致，则结果为1，否则为0

模块构建——运算符 Operator

移位运算符	功能
>>	右移
<<	左移

- 在移位运算中，将操作数左移或者右移后空位补0

条件运算符	功能
信号 = 条件？表达式1：表达式2;	当条件为真，信号取表达式1的值，未假，则取表达式2的值

位拼接运算符	功能
{信号1的某几位, …, 信号n的某几位}	用于将两个或多个信号的某些位拼接起来

模块构建——运算符 Operator

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

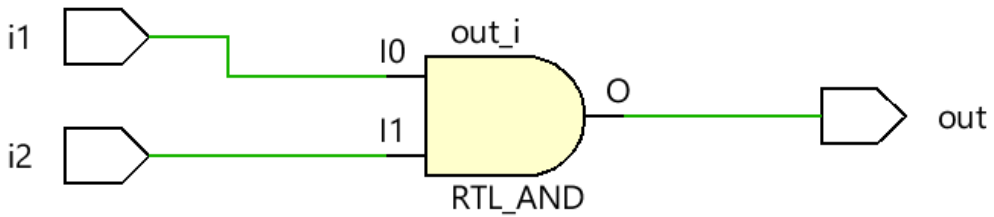
模块构建——连续赋值

- 连续赋值语句是 Verilog 数据流建模的基本语句，用于对 wire 型变量进行赋值。其格式如下：

```
1 | assign <net_expression> = <expression of different signals or constant value>
```

  - 等式左边必须是一个标量或者线性向量，而不能是寄存器类型
  - 等式右边的类型没有要求，等式右边的值一旦发生变化，就会立刻重新计算并同时赋值给左侧

```
2  module xyz ;
3      wire i1, i2;
4      wire out;
5
6      assign out = i1 & i2;
7  endmodule
```



模块构建——过程赋值

- 过程赋值是在 initial 或 always 语句块里的赋值，主要用于对寄存器类型变量进行赋值
- 寄存器变量在被赋值后，其值将保持不变，直到重新被赋予新值
- 过程赋值只有在语句执行的时候，才会起作用
- Verilog 过程赋值包括 2 种语句：阻塞赋值与非阻塞赋值

模块构建——过程赋值

- 过程赋值语句块——always 语句块
  - 通常带有触发条件
  - 语句块中的语句会重复执行
  - 一个变量不能在多个 always 块中被赋值
  - 在 always 块中被赋值的只能是 register 型变量
  - always 语句块即可以用来实现组合逻辑也可以用来实现时序逻辑

模块声明

```
always @ (event)
    [statement]

always @ (event) begin
    [multiple statements]
end
```

- 不带有敏感信号的 always 语句块会一直执行
  - 可用于仿真时钟信号生成

```
always #10 clk = ~clk;
```

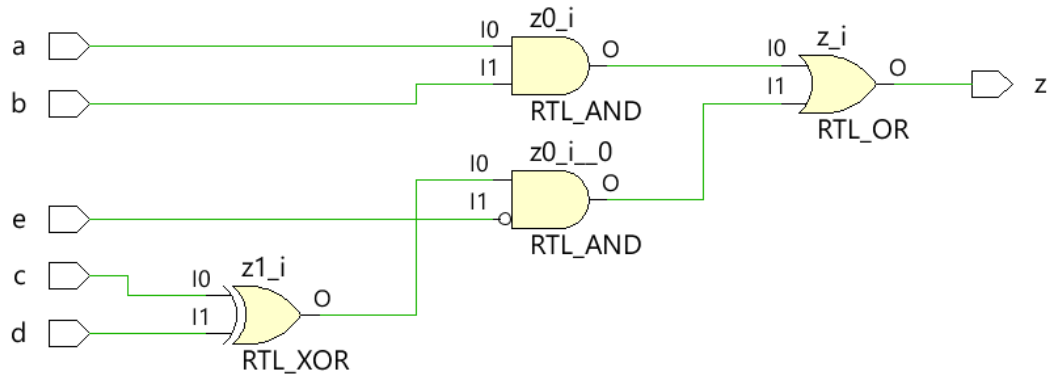
模块构建——过程赋值

- always 设计组合逻辑电路

代码示例

```
1  module combo ( input   a, b, c, d, e,
2                                output  reg z);
3
4      always @ ( a or b or c or d or e) begin
5          z = ((a & b) | (c ^ d) & ~e);
6      end
7
8  endmodule
```

综合电路



模块构建——过程赋值

- always 设计时序逻辑电路

代码示例

- 模N计算器，计数器从0开始，每个时钟周期上升沿自加1，计算器加到N-1之后重新从0开始计数。
- 模N计数器需要 $\log_2 N$ 个触发器来保存计数值

```
1  module mod10_counter ( input   clk,
2                         input   rstn,
3                         output  reg[3:0] out);
4
5      always @ (posedge clk) begin
6          if (!rstn) begin
7              out <= 0;
8          end else begin
9              if (out == 10)
10                 out <= 0;
11             else
12                 out <= out + 1;
13         end
14     end
15 endmodule
```

模块构建——过程赋值

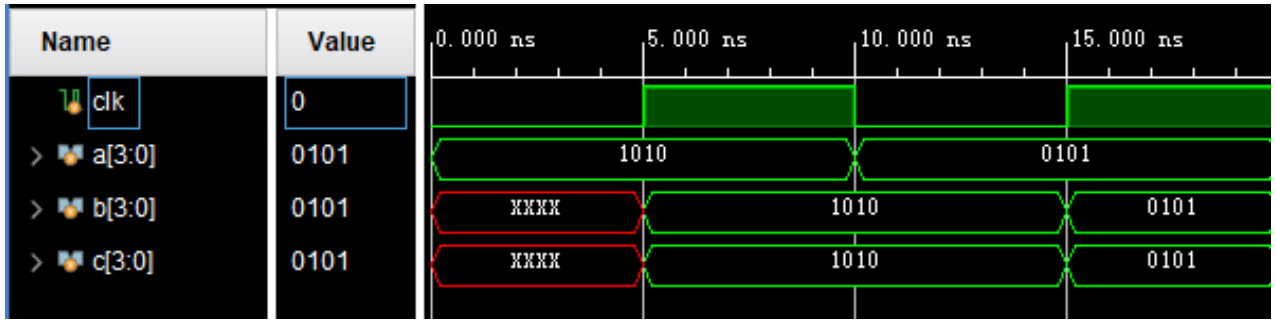
阻塞赋值

- 阻塞赋值属于顺序执行，即下一条语句执行前，当前语句一定会执行完毕
- 阻塞赋值语句使用等号 = 作为赋值符
- 仿真中，initial 里面的赋值语句都是用的阻塞赋值

```
1  reg_variable = expression;
```

代码示例

```
5  always @ (posedge clk) begin
6      b = a;
7      c = b;
8  end
```



模块构建——过程赋值

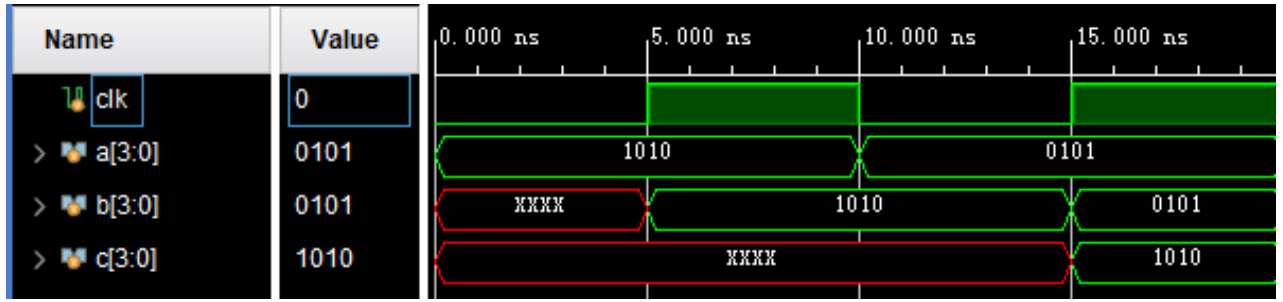
非阻塞赋值

- 非阻塞赋值属于并行执行语句，即下一条语句的执行和当前语句的执行是同时进行的，它不会阻塞位于同一个语句块中后面语句的执行
- 非阻塞赋值语句使用小于等于号 <= 作为赋值符

```
1  reg_variable <= expression;
```

代码示例

```
5  always @ (posedge clk) begin
6      b <= a;
7      c <= b;
8  end
```



模块构建——过程赋值

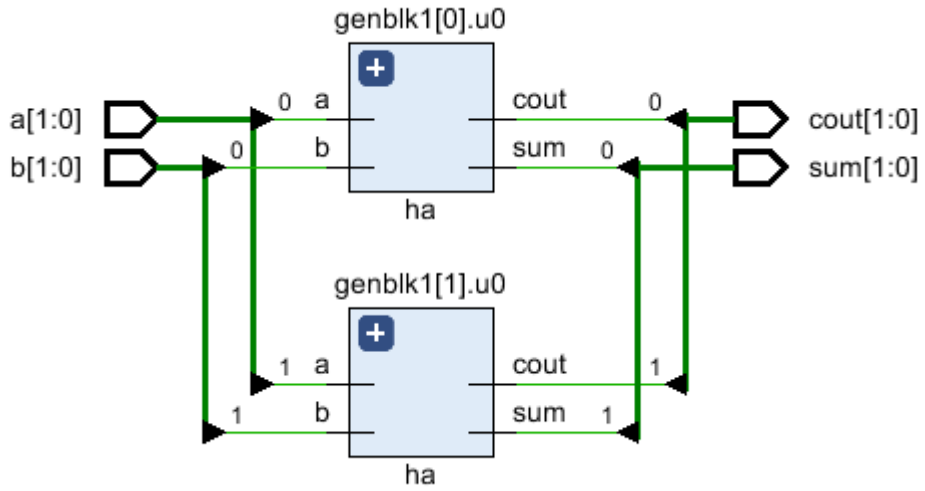
- 过程赋值语句块——initial 语句块
  - 仅用于仿真，沿时间轴只执行一次
  - 用途主要是在仿真的初始状态对各变量进行初始化
  - 不可综合，常用于测试文件中生成激励波形（如复位信号）作为电路的仿真信号

```
initial
    [single statement]

initial begin
    [multiple statements]
end
```

```
module behave;
  reg rstn;
  reg [2:0] a;

  initial begin
    rstn = 1'b0;
    #10 rstn = 1'b1 ;
        a = 3'b001 ;
    #30 $finish;
  end
endmodule
```

模块构建——generate

- generate 可以用来循环实例化模块或条件实例化模块
  - generate 与 for loop，用来构造循环结构，多次实例化某个模块
  - generate 与 if else 或 case，用来在多个块之间选择一个代码块




```
1  // Design for a half-adder
2  module ha ( input   a, b,
3              output  sum, cout);
4
5    assign sum  = a ^ b;
6    assign cout = a & b;
7  endmodule
8
9  // A top level design that contains N instances of half adder
10 module my_design
11     #(parameter N=4)
12         (   input [N-1:0] a, b,
13             output [N-1:0] sum, cout);
14
15     // Declare a temporary loop variable to be used during
16     // generation and won't be available during simulation
17     genvar i;
18
19     // Generate for loop to instantiate N times
20     generate
21         for (i = 0; i < N; i = i + 1) begin
22           ha u0 (a[i], b[i], sum[i], cout[i]);
23         end
24     endgenerate
25 endmodule
```

4 功能描述

功能描述——内容概要

结构描述

- 结构（Structural）描述是对设计电路的结构进行描述，即描述设计电路使用的元件及这些元件之间的连接关系，属于低层次的描述方法

数据流描述

- 数据（Data Flow）流描述采用持续赋值语句，抽象级别位于结构描述和行为描述之间

行为描述

- 行为（Behavioural）描述是对设计电路的逻辑功能的描述，并不用关心设计电路使用哪些元件以及这些元件之间的连接关系，属于高层次的描述方法

功能描述——结构描述

- **结构描述**是对设计电路的结构进行描述，即描述设计电路使用的元件及这些元件之间的连接关系
- 结构描述通过调用电路元件（如逻辑门，甚至晶体管）来构建电路，属于低层次的描述方法
- 一个逻辑网络由许多逻辑门组成，用逻辑门的模型来描述逻辑网络最直观
- 结构描述在一些电路设计中也有一定的实际意义（系统速度快）
- 直接调用门原语进行结构描述

功能描述——结构描述

- 基本逻辑门关键字是Verilog HDL预定义的逻辑门，包括and、or、not、xor、nand、nor等
- Verilog HDL内置了26个基本元件，其中14个门级元件，12个开关级元件
- 调用门原语句法：

```
gate_keyword < instance > ( output, input1, ..., inputn );
```

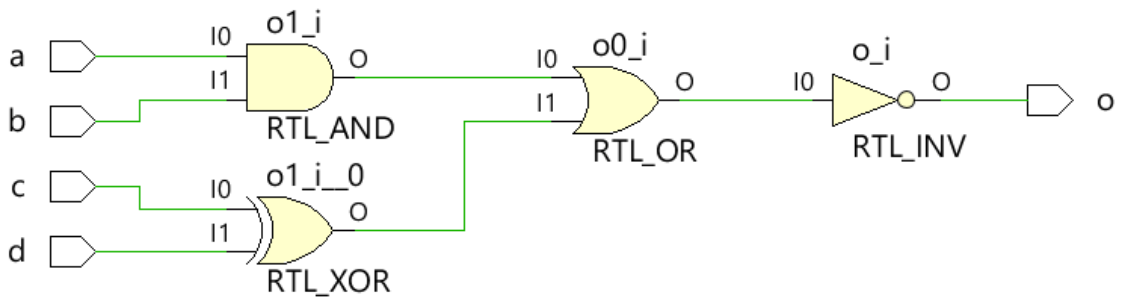
- 示例：

```
module gates (  input a, b, c, d,
                output o);
    // assign o = !(a & b & c & d)
    nand (o, a, b, c, d);   // o is the output, a, b, c and d are inputs
endmodule
```

功能描述——数据流描述

- **数据流描述**方式要比结构化描述方式抽象级别要高一些，因为它不在需要清晰地刻画出具体的数字电路架构，而是可以比较直观地表达底层的逻辑行为
- 数据流描述方式又称为 RTL 级描述方式，即寄存器传输级描述
- 从数据的变换和传送的角度来描述设计模块，常用 assign 连续赋值语句实现，因此抽象级别没有行为描述方式高，纯数据流的描述方式只适用于小规模的电路设计

```
1  module combo ( input   a, b, c, d,
2                 output  o);
3
4    assign o = ~((a & b) | c ^ d);
5
6  endmodule
```



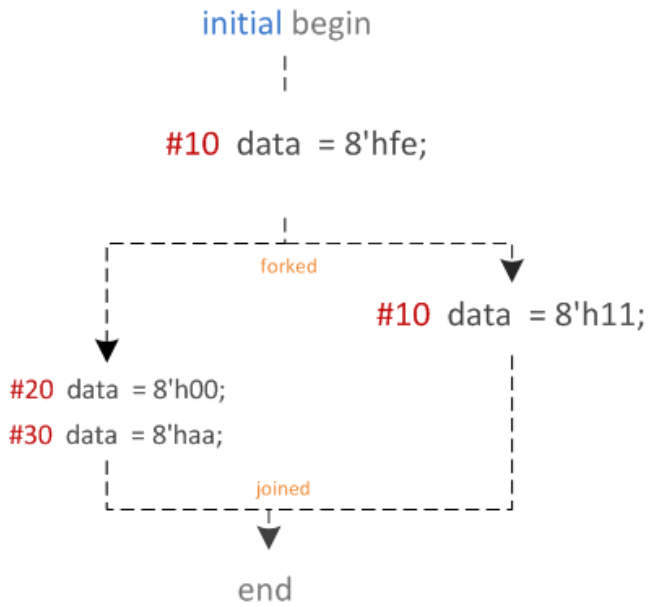
功能描述——行为描述

- **行为描述**是对设计电路的逻辑功能的描述，并不关心设计电路使用哪些元件以及这些元件之间的连接关系
- 最能体现 EDA 风格的硬件描述方式，既可以描述简单的逻辑门，也可以描述复杂的数字系统乃至微处理器；既可以描述组合逻辑电路，也可以描述时序逻辑电路
- 属于高层次的描述方法，类似高级语言可以使用控制流和循环语句等功能

功能描述——行为描述

- 语句块将一组语句组合在一起，这些语句在语法上等效于单个语句
- 顺序执行的语句常包装在 begin end 关键字中，并且将以给定的顺序依次执行。出现多组 begin end 应注意其对应匹配。
- 并行执行的语句常包装在fork join关键字内，多用于仿真

```
1  initial begin
2      #10 data = 8'hfe;
3      fork
4          #10 data = 8'h11;
5          begin
6              #20 data = 8'h00;
7              #30 data = 8'haa;
8          end
9      join
10 end
```



功能描述——行为描述

- Control Flow 语句在 Verilog 中主要包括条件语句、循环语句。
- 其中条件语句主要为 if-else 和 case 两种关键字。

if-else语句

```
// if statement without else part
if (expression)
    [statement]

// if statment with an else part
if (expression)
    [statement]
else
    [statement]

// if-else-if statement
if (expression)
    [statement]
else if (expression)
    [statement]
else
    [statement]
```

case 语句

```
// Here 'expression' should match one of the items
case (<expression>)
    case_item1 :    <single statement>
    case_item2,
    case_item3 :    <single statement>
    case_item4 :    begin
                        <multiple statements>
                        end
    default      : <statement>
endcase
```

循环语句

- for语句——有条件的循环语句。通过3个步骤来决定语句的循环执行：

1. 给控制循环次数的变量赋初值
2. 判定循环执行条件，若为假则跳出循环；若为真，执行指定的语句后，转到第3步
3. 修改循环变量的值，返回第2步

```
module my_design;
    integer i;

    initial begin
        // Note that ++ operator does not exist in Verilog !
        for (i = 0; i < 10; i = i + 1) begin
            $display ("Current loop#%0d ", i);
        end
    end
endmodule
```

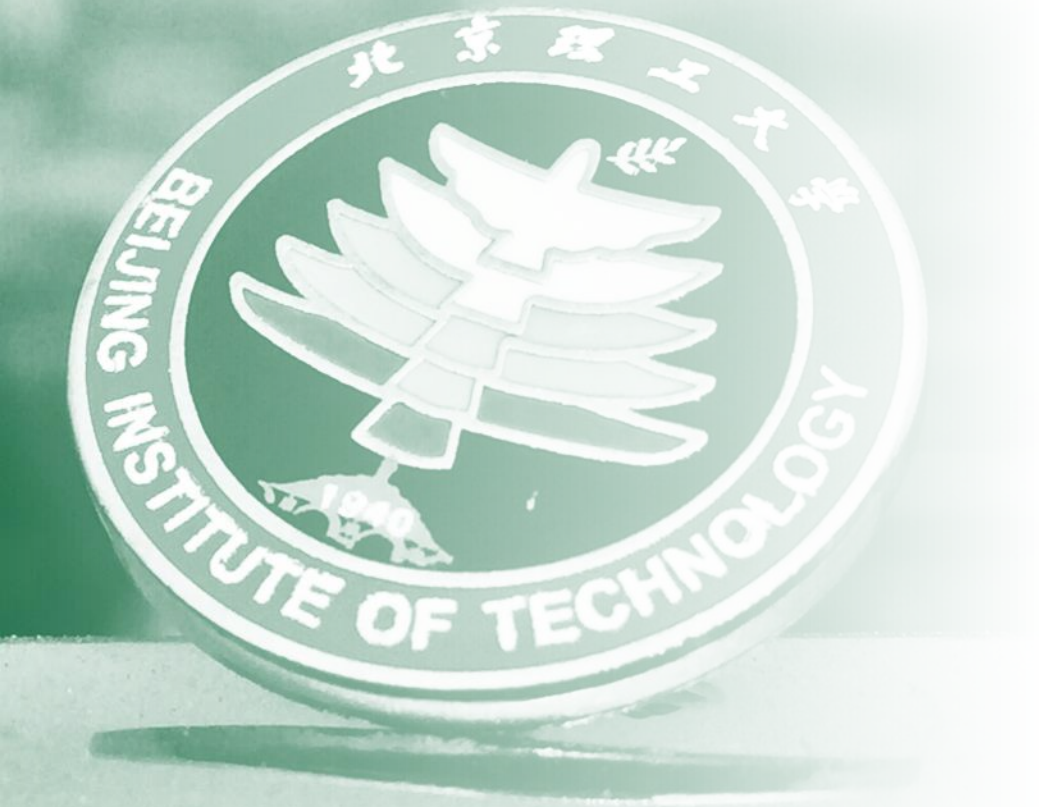
功能描述——行为描述

循环语句

- repeat 语句——连续执行一条语句 n 次
- while 语句——执行一条语句直到某个条件不满足。首先判断循环执行条件表达式是否为真，若为真，则执行后面的语句或语句块，直到条件表达式不为真；若不为真，则其后的语句一次也不被执行
- forever 语句——无限连续地执行语句，可用 disable 语句中断！多用在 initial 块中，以生成时钟等周期性波形

功能描述——行为描述

- task 语句
  - 用来由用户定义任务，任务类似高级语言中的子程序，用来单独完成某项具体任务，并可以被模块或其他任务调用
  - 当希望能够对多个信号进行一些运算并输出多个结果（即有多个输出变量）时，宜采用任务结构
- function 语句
  - 用来定义函数，函数的目的是通过返回一个用于某表达式的值，来响应输入信号，适于对不同变量采取同一运算的操作
  - 函数在模块内部定义，通常在本模块中调用，也能根据按模块层次分级命名的函数名从其他模块调用，而 task 只能在同一模块内定义与调用



5 系统函数

系统函数与代码示例——系统函数

- 在仿真时将一定内容显示出来是非常常用的，$display 和 $write 函数就主要用于显示信息和调试信息，其区别是 $display会在字符串末尾追加一个换行符，$write则不会

```
1  module tb;
2    initial begin
3      $display ("This ends with a new line ");
4      $write ("This does not,");
5      $write ("like this. To start new line, use newline char");
6      $display ("This always start on a new line !");
7    end
8  endmodule
```

- Verilog 还提供一个连续监视器 $monitor 函数，每当其参数列表中的变量或表达式发生更改时，会自动打印出变量或表达式的值。

系统函数与代码示例——系统函数

- Verilog 也有系统函数来处理文件相关操作，包括打开文件，将值输出到文件，从文件中读取值和关闭文件等。
  - 打开和关闭文件
  - 向文件中输入信息
  - 读取文件

  - 读取文件数据到存储器：

$readmemb（“<数据文件名>”,<存储器名>);
$readmemh（“<数据文件名>”,<存储器名>);

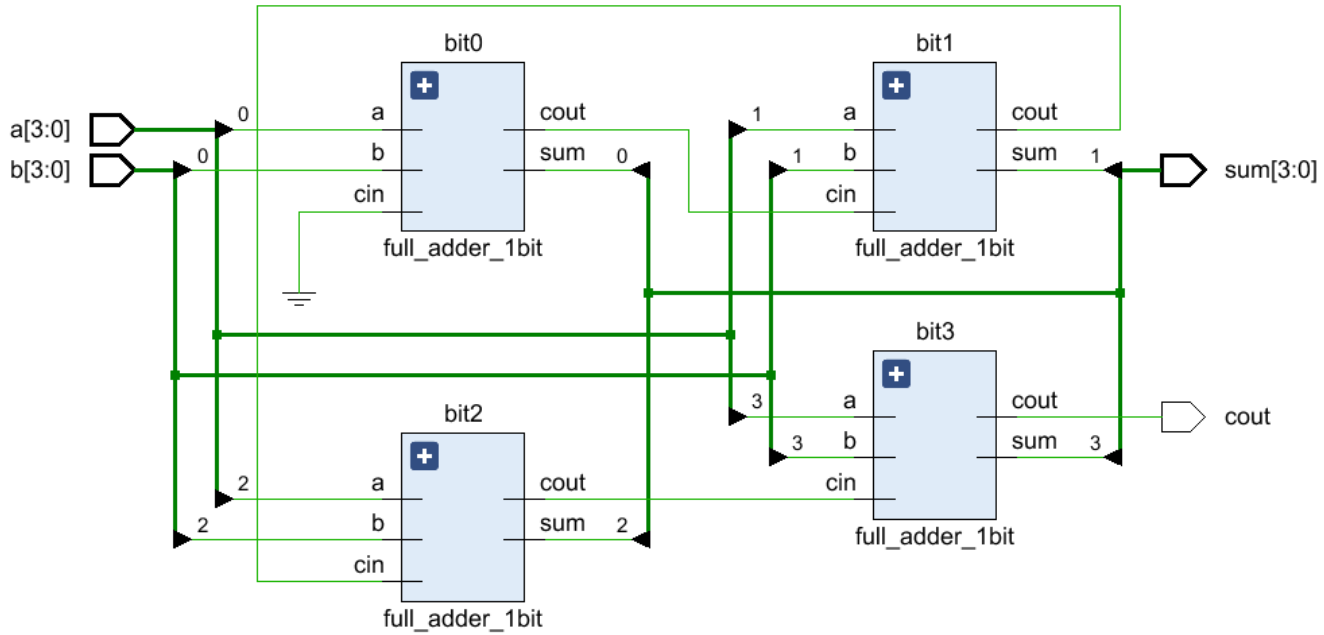
```
module tb;
    reg[8*45:1] str;
    integer     fd;

    initial begin
      fd = $fopen("my_file.txt", "r");

      // Keep reading lines until EOF is found
      while (! $feof(fd)) begin

        // Get current line into the variable 'str'
        $fgets(str, fd);

        // Display contents of the variable
        $display("%0s", str);
      end
      $fclose(fd);
    end
endmodule
```

6 代码示例

系统函数与代码示例——代码示例

4-bit 全加器——设计源码

```
1   `timescale 1ns / 1ps
2
3   module full_adder_1bit(
4       input wire a, b, cin,
5       output wire sum, cout
6       );
7
8       assign sum = (a ^ b) ^ cin;
9       assign cout = (a & b) | ((a ^ b) & cin);
10  endmodule
11
12  module full_adder_4bit(
13      input wire[3:0] a, b,
14      output wire[3:0] sum,
15      output wire cout
16      );
17
18      wire [3:0] carry;
19      full_adder_1bit bit0(a[0], b[0],     1'b0, sum[0], carry[0]);
20      full_adder_1bit bit1(a[1], b[1], carry[0], sum[1], carry[1]);
21      full_adder_1bit bit2(a[2], b[2], carry[1], sum[2], carry[2]);
22      full_adder_1bit bit3(a[3], b[3], carry[2], sum[3], carry[3]);
23
24      assign cout = carry[3];
25  endmodule
26
```

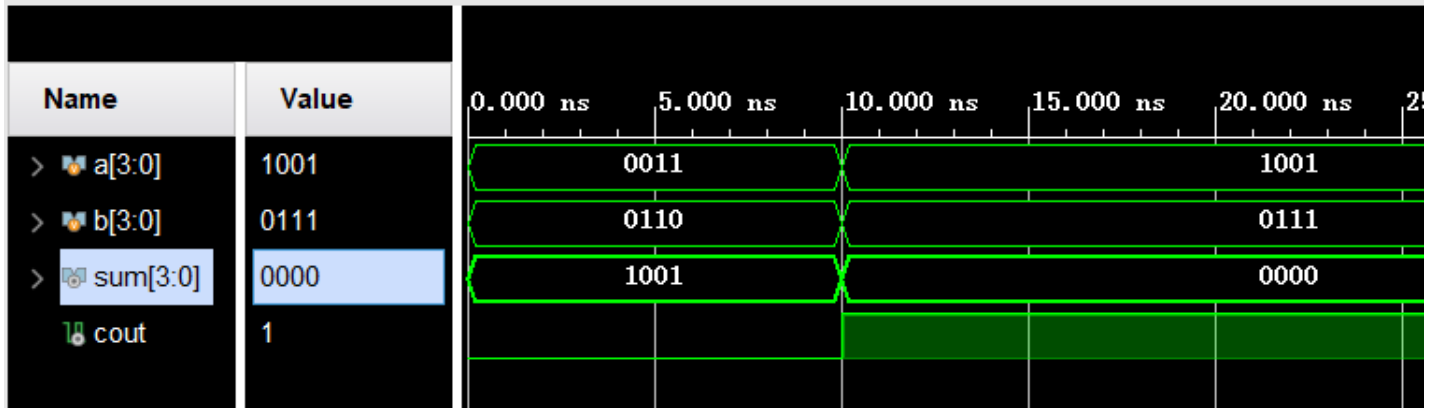


系统函数与代码示例——代码示例

4-bit 全加器——仿真源码

```
1   `timescale 1ns / 1ps
2
3   module testbench();
4   reg[3:0] a, b;
5   wire[3:0] sum;
6   wire cout;
7
8   initial begin
9       a = 4'b0011;
10      b = 4'b0110;
11      #10;
12      a = 4'b1001;
13      b = 4'b0111;
14  end
15
16  full_adder_4bit adder(a, b, sum, cout);
17
18  endmodule
19
```

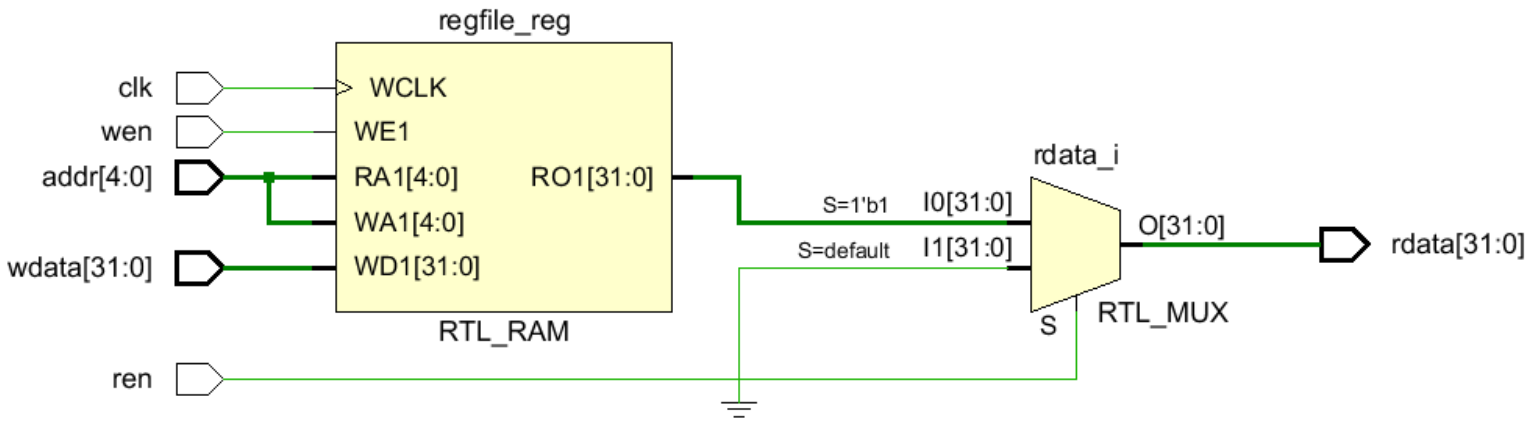
- 初始a和b分别赋值4’b0011，4’b0110，计算结果sum和cout分别是4’b1001，1’b0
- 经过10ns，a和b分别赋值4’b1001，4’b0111，计算结果sum和cout分别是4’b0000，1’b1 ，即发生溢出



系统函数与代码示例——代码示例

可读写存储器 regfile —— 设计源码

```
1   `timescale 1ns / 1ps
2
3   module regfile(
4       input wire        clk,
5       input wire        ren,
6       input wire        wen,
7       input wire[31:0]  wdata,
8       input wire[4:0]   addr,
9       output wire[31:0] rdata
10      );
11
12      reg[31:0] regfile[31:0];
13
14      initial begin
15          $readmemh("C:\\ram_data.txt", regfile);
16      end
17
18      assign rdata = (ren == 1'b1) ? regfile[addr] : 32'b0;
19
20      always @(posedge clk) begin
21          if (wen)
22              regfile[addr] = wdata;
23      end
24  endmodule
25
```



- initial 语句在仿真开始时执行，不可综合！
- 通过$readmemh 系统函数，将文本文件中定义的值加载到reg变量中，文本文件内容如下：

ram_data.txt - 记事本

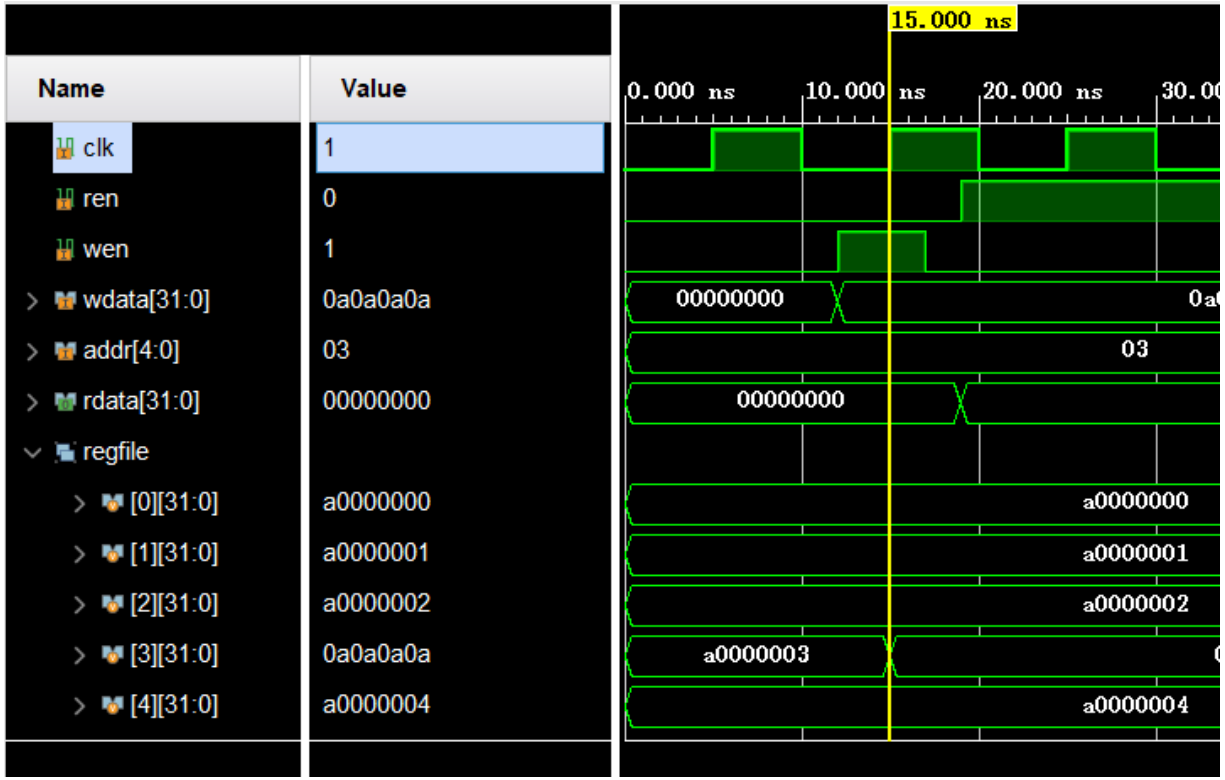
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
a0000000
a0000001
a0000002
a0000003
a0000004
```

系统函数与代码示例——代码示例

可读写存储器 regfile —— 仿真源码

```
1   `timescale 1ns / 1ps
2
3   module testbench3();
4
5       reg         clk;
6       reg         ren;
7       reg         wen;
8       reg[4:0]    addr;
9       reg[31:0]   wdata;
10      wire[31:0] rdata;
11
12      regfile regfile0(clk, ren, wen, wdata, addr, rdata);
13
14      initial begin
15          clk = 1'b0;
16          ren = 1'b0;
17          wen = 1'b0;
18          addr = 5'b00011;
19          wdata = 32'h0;
20          #12
21          wen = 1'b1;
22          wdata = 32'h0a0a0a0a;
23          #5
24          wen = 1'b0;
25          #2
26          ren = 1'b1;
27      end
28
29      always #5 clk = ~clk;
30
31  endmodule
32
```



感谢聆听