Verilog HDL语言

主讲：卢　萍

Email:　luping06@hust.edu.cn
QQ:　　1269175330

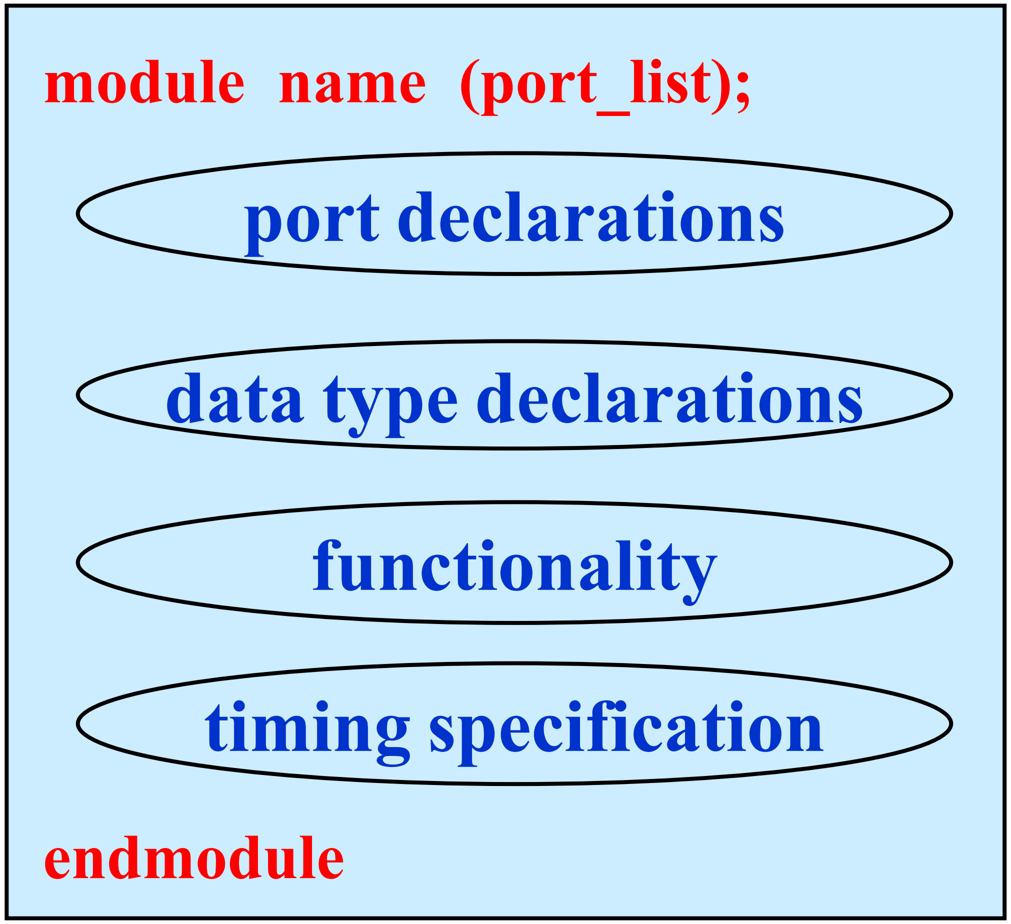
华中科技大学计算机科学与技术学院

Verilog HDL语言
设计入门

Verilog程序的结构

- 模块（module）是Verilog的基本单位，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使用。
- 模块由关键词module和endmodule进行定义

- Verilog程序由关键词module和endmodule进行定义
- Verilog HDL 大小写敏感



Verilog程序的组成部分

module Name,
port list, port declarations (if ports present)
parameters (optional) ,

Declarations of wires,
regs and other variables

Data flow statements
(assign)

Instantiation of lower
level modules

Tasks and functions

Always and initial blocks,
All behavioral statements go in these blocks.

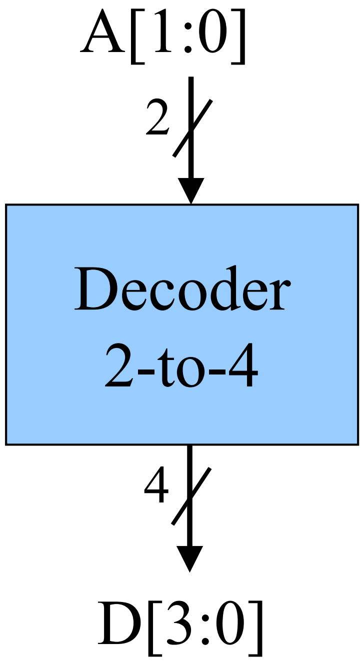
endmodule

这5个组件的排列顺序是任意的，可以选择其中的一个或几个组件构成一个Verilog程序。

Verilog 模块

- 在Verilog语言中，一个电路就是一个module。

```
module decoder_2_to_4 (A, D) ;

input [1:0] A ;
output [3:0] D ;

assign D =      (A == 2'b00) ? 4'b0001 :
                (A == 2'b01) ? 4'b0010 :
                (A == 2'b10) ? 4'b0100 :
                (A == 2'b11) ? 4'b1000 : 4'b0000 ;
endmodule
```



Verilog 模块

module name

ports names of module

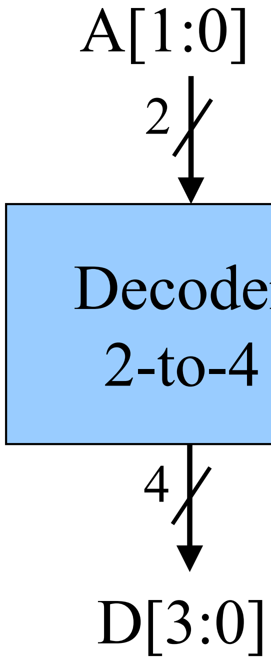
port types

port sizes

module contents

keywords underlined

```
module decoder_2_to_4 (A, D) ;

input [1:0] A ;
output [3:0] D ;

assign D =      (A == 2'b00) ? 4'b0001 :
                (A == 2'b01) ? 4'b0010 :
                (A == 2'b10) ? 4'b0100 :
                (A == 2'b11) ? 4'b1000 :4'b0000 ;
endmodule
```



如何实现一个模块

- Verilog语言的关键词不能用作模块名
  （例如：module/port/signal 等）
  - Choose a *descriptive* module name
- 定义模块的端口（connectivity）
- 定义模块内部连接到端口的信号类型
  - Choose *descriptive* signal names
- 定义内部信号
- 描述模块内部实现的功能（ functionality）

定义端口

- 每个端口都会连接一个信号（Signal）
- 申明端口的类型
  - **input**
  - **output**
  - **inout** （双向）
- Scalar (single bit) - 不需要给出信号的位数
  - **input** cin**;**
- Vector (multiple bits) - 需要定义具体的位数
  - Range is MSB to LSB (left to right)
  - Don’t have to include zero if you don’t want to… (**D[2:1]**)
  - **output** [7:0] OUT;
  - **input** [0:4] IN;

模块端口列表

- 模块端口的多种申明方法（例1）

```
module Add_half(c_out, sum, a, b);
    output sum, c_out;
    input a, b;
    ...
endmodule
```

```
module Add_half(output c_out, sum,
                input a, b);
    ...
endmodule
```

模块端口列表

- 模块端口的多种申明方法（例2）

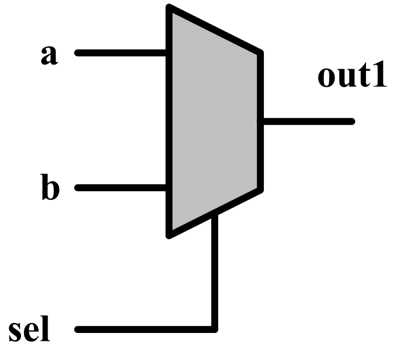
```
module xor_8bit(out, a, b);
    output [7:0] out;
    input [7:0] a, b;
    …
endmodule
```

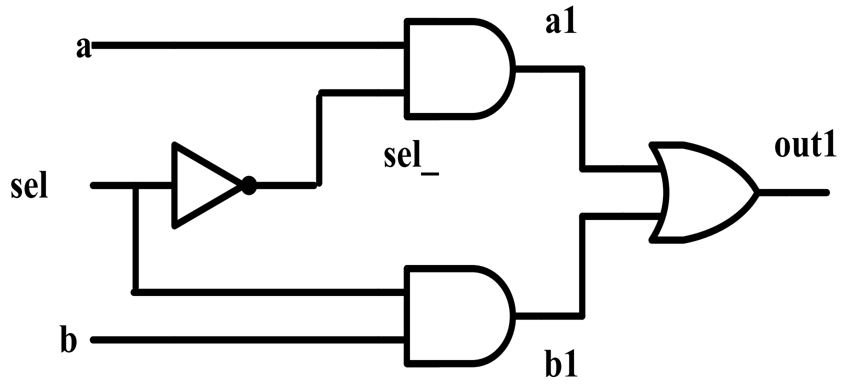
```
module xor_8bit(output [7:0] out,  input [7:0] a, b);
                          …
endmodule
```

结构描述

- 一般使用内部元件（**Primitive**）、自定义的下层模块对电路进行描述。主要用于层次化设计中。

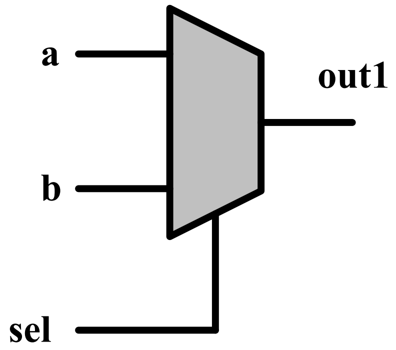
```
module mux2_1(out1,a,b,sel);
    output out1;
    input a,b,sel;

   wire a1,a2, sel_;

  not ( sel_, sel );
  and ( a1, a, sel_ );
  and ( b1, b, sel );
  or ( out1, a1, b1 );

endmodule
```





数据流描述

- 一般使用连续赋值assign语句描述，主要用于组合逻辑电路建模。

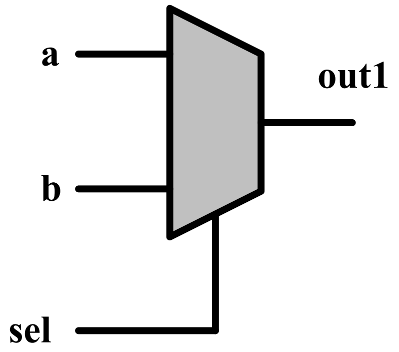


```
module mux2_1(out1, a, b, sel) ;
   output   out1;
   input  a, b;
   input sel;

assign out1= sel ? b : a;

endmodule
```

```
module mux2_1(out1, a, b, sel) ;
   output   out1;
   input  a, b;
   input sel;

assign out1=(sel & b) | (~sel & a);

endmodule
```

行为描述

- 一般使用Initial或Always语句描述，可以对组合、时序逻辑电路建模。



```
module mux2_1(out1, a, b, sel) ;
    output  reg out1;
    input  a, b;
    input sel;
always @(sel or a or b)
begin
    if (sel)
        out1 = b;
    else
        out1 = a;
end
endmodule
```

```
module mux2_1(out1, a, b, sel) ;
    output   reg out1;
    input  a, b;
    input sel;
always @(sel or a or b)
begin
    case (sel)
        1’b0 :  out1 = a;
        1’b1 :  out1 = b;
    endcase
end
endmodule
```

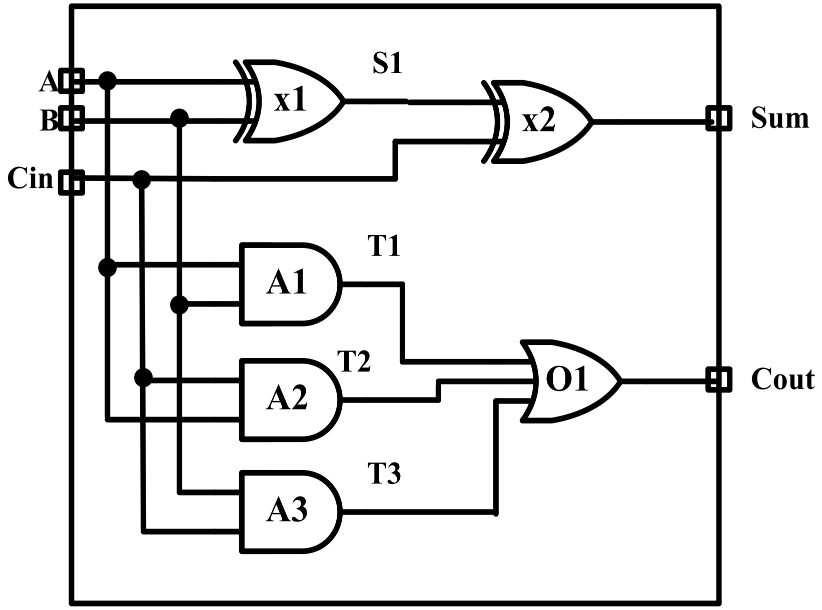
混合设计描述

- 结构、数据流和行为描述方式可以自由混合。模块描述中可以包含实例化的门、模块实例化语句、连续赋值语句以及a l w a y s语句和i n i t i a l语句的混合。它们之间可以相互包含。

- 来自a l w a y s语句和i n i t i a l语句（切记只有寄存器类型数据可以在这两种语句中赋值）的值能够驱动门或开关。

- 而来自于门或连续赋值语句（只能驱动线网）的值能够反过来用于触发a l w a y s语句和i n i t i a l语句。

❑ 实例：混合设计方式的1位全加器



```
module FA_Mix(A,B,Cin,Sum,Cout);
   input A,B,Cin;
   output Sum,Cout;
   reg Cout;
   reg T1,T2,T3;
   wire S1;
xor X1(S1,A,B);                  // 门实例语句
always @ (A or B or Cin)   // always 语句
begin
   T1 = A & B;
   T2 = A & Cin;
   T3 = B & Cin;
   Cout = (T1 | T2) | T3;
end
assign Sum = S1 ^ Cin;     // 连续赋值语句
endmodule
```

设计验证与仿真

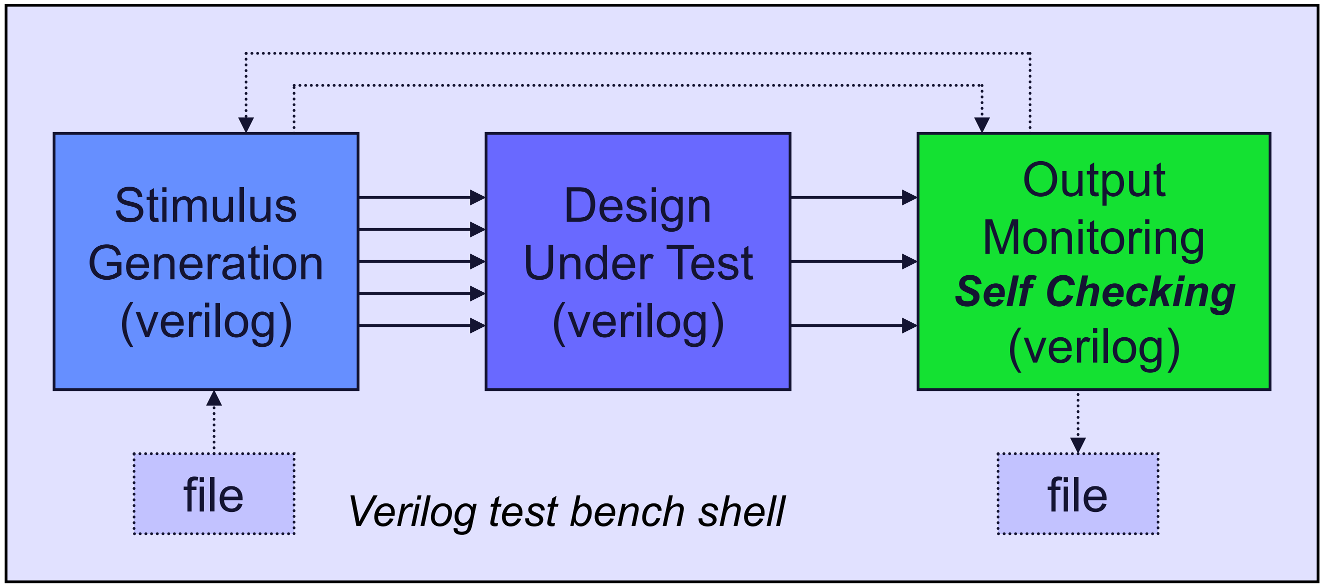
- Verilog HDL不仅提供描述设计的能力，而且提供对激励、控制、存储响应和设计验证的建模能力。
- 激励和控制可用初始化语句产生。验证运行过程中的响应可以作为“变化时保存”或作为选通的数据存储。
- 最后，设计验证可以通过在初始化语句中写入相应的语句自动与期望的响应值比较完成。
- 要测试一个设计块是否正确，就要用Verilog再写一个测试模块。这个测试模块应包括以下三个方面的内容：
  - 测试模块中要调用到设计块，只有这样才能对它进行测试；
  - 测试模块中应包含测试的激励信号源；
  - 测试模块能够实施对输出信号的检测，并报告检测结果。

HDL电路的仿真与验证

- 残酷的现实……
  - 设计用时10%，而用于设计验证的时间则占 90%
  - 如果方法不当验证用时甚至更多

Testbenchs也是用Verilog语言写的。

Testbench Verilog不是用来描述硬件的，更像是一个测试程序。


Stimulus Generation (verilog)
Design Under Test (verilog)
Output Monitoring *Self Checking* (verilog)
file
file
Verilog test bench shell

decoder_2_to_4的Testbench

```
`timescale 1ns / 1ps     // `（反引号）这个字符位于主键盘的左上角

module decoder_2_to_4_tb;     //  测试模块没有输入输出端口

    reg [1:0]   x;
    wire [3:0]  y;
    integer k;

    decoder_2_to_4  DUT (x, y);       // Circuit under test


    initial
    begin
      x = 0;
      for (k=0; k < 4; k=k+1)
            #5 x=k;                          // 在5个时间单位后k赋值给x

      #10 $stop;             // 在10个时间单位后暂停仿真
    end

endmodule
```

mux2_1的Testbench

```
module mux2_1_tb;     //  测试模块没有输入输出端口
    reg [1:0]  x, y;
    reg     s;
    wire [1:0]  m;

    mux2_1 DUT (m, x, y, s);        // Circuit under test

    initial
    begin
      x = 0; y = 0; s = 0;
      #10 x = 1;                  // 在10个时间单位后x的值为1
      #10 y = 1;
      #10 x = 3; y = 0;
      #10 x = 2; y = 3;
      #10 s = 1;
      #10 x = 1;
      #10 y = 1;
      #10 x = 3; y = 0;
      #10 x = 2; y = 3;
     #20 $stop;
    end

endmodule
```

Testbench实例（仅用于演示）

```
module test_and;
integer file, i, code;
reg a, b, expect, clock;
wire out;
parameter cycle = 20;
and #4 a0(out, a, b);                    // Circuit under test

initial begin : file_block
   clock = 0;
   file = $fopen("compare.txt", “r” );
   for (i = 0; i < 4; i=i+1) begin
     @(posedge clock)      // Read stimulus on rising clock
     code = $fscanf(file, "%b %b %b\n", a, b, expect);
     #(cycle - 1)          // Compare just before end of cycle
     if (expect !== out)
        $strobe("%d %b %b %b %b", $time, a, b, expect, out);
   end // for
   $fclose(file); $stop;
end // initial
always #(cycle /2) clock = ~clock; // Clock generator
endmodule
```

Comments（注释） in Verilog

- Commenting is important
  - In industry many other poor schmucks are going to read your code
  - Some poor schmuck (perhaps you 4 years later) are going to have to reference your code when a customer discovers a bug.
- The best comments document why you are doing what you are doing, not what you are doing.
  - Any moron who knows verilog can tell what the code is doing.
  - Comment why (motivation/thought process) you are doing that thing.

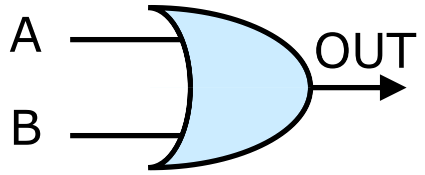
Slide taken direct from Eric Hoffman

Verilog语言基础

- 间隔符：空格、TAB键、换行符及换页符
- 注释
  - 单行注释：用//标志起头和回车符结尾
  - 多行注释：用/*标志起头和*/标志结尾
- 标识符
  - 可以是任意一组字母、数字、$符号和_(下划线)符号的组合；
  - 必须是由字母或下划线开头，长度小于1024字符；
  - 转义标识符以反斜杠“\”开头，以空白符结尾的任何字符序列；
  - 标识符区分大、小写。
- 关键词：Verilog HDL 内部已使用的词，关键词都是小写。
- 格式：区分大小写。自由格式，即结构可以跨越多行编写。

Identifiers (Signal Names)

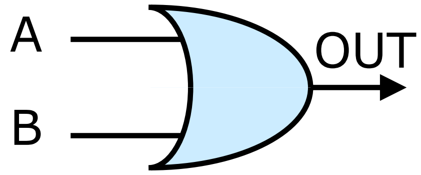
- Identifiers are the names you choose for your signals
- In a programming language you should choose descriptive variable names. In a HDL you should choose descriptive signal names.
  - Use mixed case and/or _ to delimit descriptive names.
    - ✓assign parityErr = ^serial_reg;
    - ✓nxtState = returnRegister;
  - Have a convention for signals that are active low
    - ✓Many errors occur on the interface between blocks written by 2 different people. One assumed a signal was active low, and the other assumed it was active high
    - ✓I use **_n** at the end of a signal to indicate active low
    - ✓**rst_n** = 1'b0   // assert reset



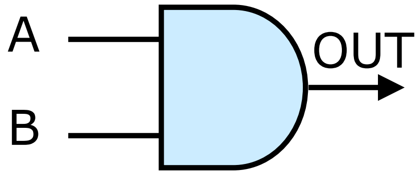
A	B	OUT
0	0	0
0	1	1
1	1	1
0	x	x
0	z	x
1	x	1
1	z	1

- 四种基本的逻辑值
  - 0：逻辑0或"假"
  - 1：逻辑1或"真"
  - x：未知
  - z：高阻
- 三类常量
  - 整型数：简单的十进制格式，**基数格式**（5'O37，4'B1x_01）
  - 实数：十进制计数法，科学计数法
  - 字符串：字符串是双引号的字符序列，字符串不能分成多行书写
- 参数
  - 参数是一个常量，经常用于定义时延和变量的宽度。

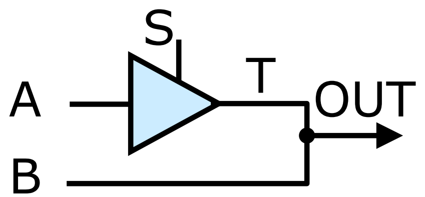
Resolving 4-Value Logic



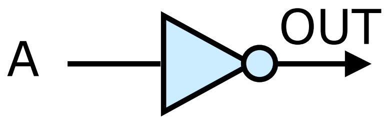
A	B	OUT
0	0	0
0	1	1
1	1	1
0	x	x
0	z	x
1	x	1
1	z	1



A	B	OUT
0	0	0
0	1	0
1	1	1
0	x	0
0	z	0
1	x	x
1	z	x



S	A	T	B	OUT
0	0	z	z	z
0	1	z	x	x
0	x	z	1	1
0	z	z	0	0
1	0	0	1	x
1	0	0	z	0
1	1	1	z	1
1	x	x	z	x
1	z	x	0	x



A	0	1	x	z
OUT	1	0	x	x

Slide taken direct from Eric Hoffman

Numbers in Verilog

- General format is: <size><'base><number>
- Examples:
  - 4'b1101 // this is a 4-bit binary number equal to 13
  - 10'h2e7 // this is a 10-bit wide number specified in hex

- Available bases:
  - d = decimal (please only use in test benches)
  - h = hex (use this frequently)
  - b = binary (use this frequently for smaller #'s)
  - o = octal (who thinks in octal?, please avoid)

Slide taken direct from Eric Hoffman

Numbers in Verilog

- Numbers can have x or z characters as values
  - x = unknown, z = High Impedance
  - 12'h13x // 12-bit number with lower 4-bits unknown

- If size is not specified then it depends on simulator/machine.
  - Always size the number for the DUT verilog
    - Why create 32-bit register if you only need 5 bits?
    - May cause compilation errors on some compilers

- Supports negative numbers as well
  - -16'h3A // this would be -3A in hex (i.e. FFC6 in 2's complement)
  - I rarely if ever use this. I prefer to work 2's complement directly

Slide taken direct from Eric Hoffman

参数parameter

有时候希望模块成为一般化的模块，即希望端口位数可选。

parameter可实现此功能，在调用模块时可改变该参数的值

```
module mux2_1(out1, a, b, sel) ;
    parameter N=2；    //本模块内不可变
    output [N-1:0]   out1;
    input [N-1:0]  a, b;
    input sel;

   assign out1= sel ? b : a;
endmodule
```

parameter：是个常量
范围 -> 本module内有效
作用 -> 模块间参数传递

在顶层模块中：

```
mux2_1 #(4)  dut0( out, x,y,s );    //实例化时参数N值为4
```

编译指令`define

可以跨模块的定义，写在模块名称上面，在整个设计工程都有效。在一个文件中通过`define指令定义的常量可以被其他文件使用。

举例：定义： `define UART_CNT 10'd1024

使用 ： `UART_CNT

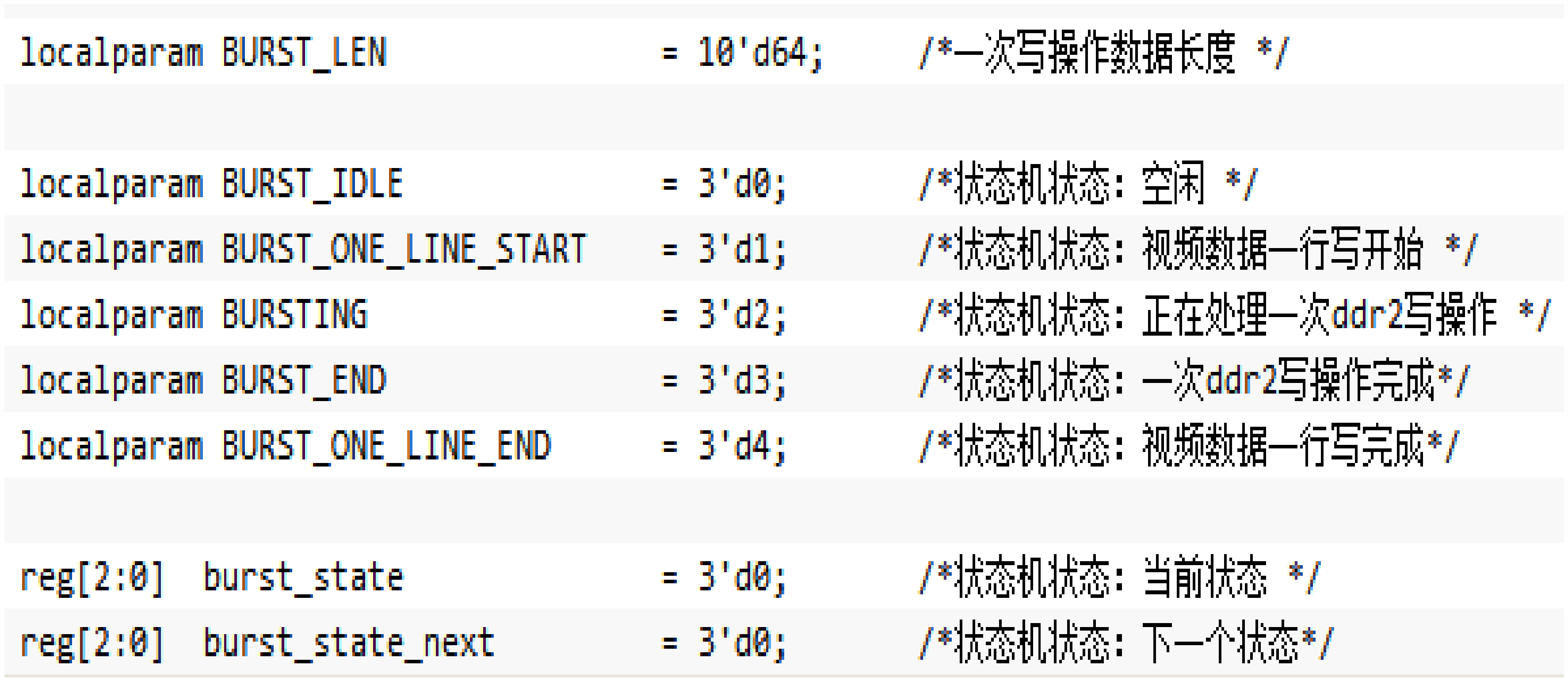
`define：定义常量
范围 -> 整个工程
作用 -> 常用于定义常量可以跨模块、跨文件

本地参数localparam

localparam：常量

范围 -> 本module内有效的定义，不可用于参数传递;

作用 -> 常用于状态机的参数定义;



```
localparam BURST_LEN                = 10'd64;     /*一次写操作数据长度 */

localparam BURST_IDLE               = 3'd0;       /*状态机状态：空闲 */
localparam BURST_ONE_LINE_START     = 3'd1;       /*状态机状态：视频数据一行写开始 */
localparam BURSTING                 = 3'd2;       /*状态机状态：正在处理一次ddr2写操作 */
localparam BURST_END                = 3'd3;       /*状态机状态：一次ddr2写操作完成*/
localparam BURST_ONE_LINE_END       = 3'd4;       /*状态机状态：视频数据一行写完成*/

reg[2:0]  burst_state               = 3'd0;       /*状态机状态：当前状态 */
reg[2:0]  burst_state_next          = 3'd0;       /*状态机状态：下一个状态*/
```

Parameters & Define

- Parameters are useful to make your code more generic/flexible. Read about it in text. More later…
- `define statement can make code more readable

```
`define idle        2'b00;    // idle state of state machin
`define conv        2'b01;    // in this state while A2           g
`define avg         2'b10;    // in this state wh               g samples
  .
  .
  .
  case (state)
    `idle : if (start_conv)                  conv;
            else                       te = `idle

    `conv : i                    ate = `avg;
                            xt_state = `conv;
```

Bad Example...Don't Use `define for state assignment
Use parameters instead. `define should be truly global thing

Slide taken direct from Eric Hoffman

Parameters & Define

```
localparam idle =           2'b00;    // idle state of state machine
localparam conv =           2'b01;    // in this state while A2D converting
localparam avg =            2'b10;    // in this state while averaging samples
  .
  .
  .
 case (state)
  idle : if (start_conv) nxt_state = conv;
             else              nxt_state = idle

  conv : if (gt) nxt_state = avg;
              else    nxt_state = conv;
  .
  .
  .
```

Slide taken direct from Eric Hoffman

❑ 数据类型

- ➢ 线网类型（wire）。net type表示Verilog结构化元件间的物理连线。它的值由驱动元件的值决定；如果没有驱动元件连接到线网，线网的缺省值为z。

- ➢ 寄存器类型（reg）。register type表示一个抽象的数据存储单元，它只能在always语句和initial语句中被赋值，并且它的值从一个赋值到另一个赋值被保存下来。寄存器类型的变量具有x的缺省值。

❑ Nets数据类型：表示元件之间的结构化连接

- wire和tri线网：是最常见的线网类型。
- wor和trior线网：如果某个驱动源为1，那么线网的值也为1。
- wand和triand线网：如果某个驱动源为0，那么线网的值为0。
- trireg线网：此线网存储数值（类似于寄存器），并且用于电容节点的建模。
- tri0和tri1线网：这类线网可用于线逻辑的建模，即线网有多于一个驱动源。
- supply0和supply1线网：supply0用于对“地”建模，即低电平0；supply1用于对电源建模，即高电平1。

- Register数据类型：在程序块中作变量用，对信号赋值需要用该数据类型，赋值时用关键字**initial**或**always**开始。
  - reg寄存器类型：是最常见的数据类型.
  - integer寄存器类型：整数寄存器包含整数值，可以作为普通寄存器使用，典型应用为高层次行为建模。
  - time类型：用于存储和处理时间。
  - real和realtime类型：实数寄存器（或实数时间寄存器）。

Registers in Verilog

- Registers are storage nodes
  - They retain their value till a new value is assigned
  - Unlike a net (wire) they do not need a driver
  - Can be changed in simulation by assigning a new value

- Registers are not necessarily FlipFlops
  - In your DUT Verilog registers are typically FlipFlops
  - Anything assigned in an *always* or *initial* block must be assigned to a register
  - You will use registers in your testbenches, but they will not be FlipFlops

Slide taken direct from Eric Hoffman

寄存器型（reg）

- reg： 最常见的数据类型

  reg cnt;   // 1位寄存器

  reg [3:0] v;   // 4位寄存器

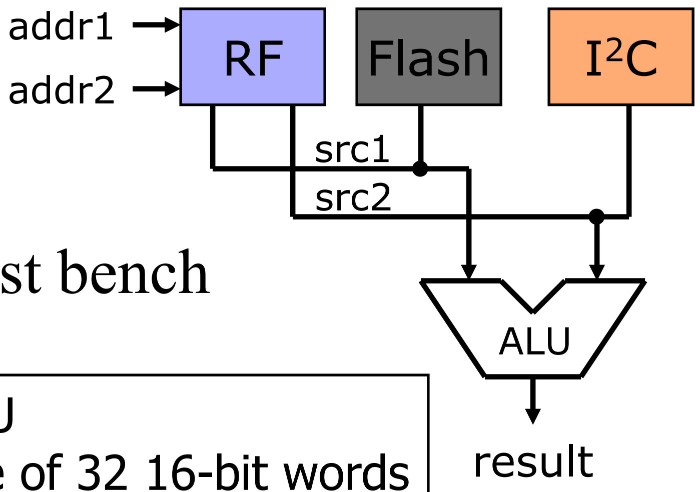
- 存储器： 寄存器数组

  reg [7:0] mem[0:63];   // 64个8位寄存器的数组

Slide taken direct from Eric Hoffman

Arrays & Memories

- Can have multi-dimentional arrays
  - reg [7:0] mem[0:99][0:3];   // what is this?
- Often have to model memories
  - RF, SRAM, ROM, Flash, Cache
  - Memories can be useful in your test bench



```
wire [15:0] src1,src2;        // source busses to ALU
reg [15:0] reg_file[0:31];        // Register file of 32 16-bit words
reg [4:0] addr1,addr2;        // src1 and src2 address
 .
 .
 .
src1 = reg_file[addr1];       // transfer addressed register file
                              // to src1 bus
```

Slide taken direct from Eric Hoffman

Vectors in Verilog

- Vectors are a collection of bits (i.e. 16-bit wide bus)

```
////////////////////////////////////////////////////////////////////////
// Define the 16-bit busses going in and out of the ALU //
////////////////////////////////////////////////////////////////////////
wire [15:0] src1_bus,src2_bus,dst_bus;
```

```
/////////////////////////////////////////////////////////////////
// State machine has 8 states, need a 3-bit encoding //
/////////////////////////////////////////////////////////////////
reg [2:0] state,nxt_state;
```

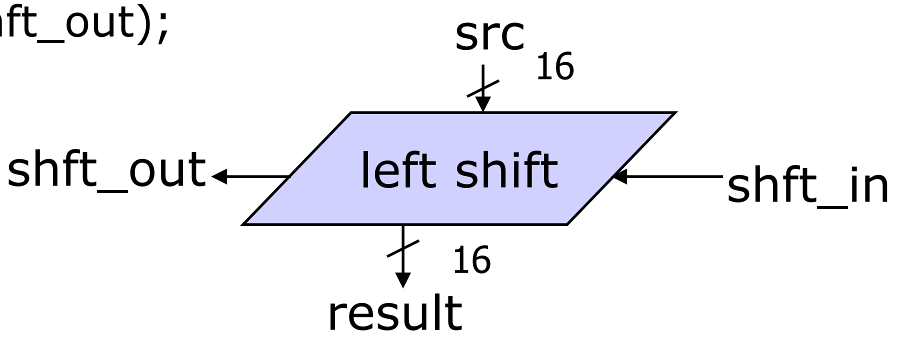
- Bus ≠ Vector (how are they different?)

Slide taken direct from Eric Hoffman

Vectors in Verilog

- Can select parts of a vector (single bit or a range)

```
module lft_shift(src,shft_in,result,shft_out);

input [15:0] src;
input shft_in;
output [15:0] result;
output shft_out;

//////////////////////////////////////////////////////////////////////////////
// Can access 15 MSB's of src with [14:0] selector                         //
// { , } is a process of concatenating two vectors to form one //
//////////////////////////////////////////////////////////////////////////////
assign result = {src[14:0],shft_in};

assign shft_out = src[15]; // can access a single bit MSB with [15]

endmodule
```



Slide taken direct from Eric Hoffman

- 系统任务和函数
  - 以$字符开始的标识符表示系统任务或系统函数；
  - 任务提供了一种封装行为的机制，任务可以返回0个或多个值；
  - 函数除只能返回一个值以外与任务相同；
  - 函数在0时刻执行，即不允许延迟，而任务可以带有延迟。

- 编译指令：以`（反引号）开始的某些标识符
  - `define和`undef，`ifdef、`else和 `endif，`default_nettype
  - `include，`resetall，`timescale
  - `unconneted_drive和`nounconnected_drive
  - `celldefine和`endcelldefine

Useful System Tasks

- $display □ Like printf in C. Useful for testbenches and debug

```
$display("At time %t count = %h",$time,cnt);
```

- $stop → Stops simulation and allows you to still probe signals and debug
- $finish → completely stops simulation, simulator relinquishes control of thread.
- Also useful is `include for including code from another file (like a header file)
- Read about these features of verilog in your text

Slide taken direct from Eric Hoffman

时延

- Verilog HDL模型中的所有时延都根据单位定义。
- 下面是带时延的连续赋值语句实例：

```
assign #2 Sum = A ^ B;    //  #2指2个时间单位。
and #1 A0(OUT, A, B);
```

- 如果没有说明时延时间单位，Verilog HDL模拟器会指定一个缺省时间单位。
- IEEE Verilog HDL标准中没有规定缺省时间单位。

Timing Controls For Simulation

- Can put "delays" in a Verilog design
  - Gates, wires, & behavioral statements
- Delays are useful for **Simulation only**!
  - Used to approximate "real" operation while simulating
  - Used to control testbench
- SYNTHESIS
  - Synthesis tool IGNORES these timing controls
    - ✓ Cannot tell a gate to wait 1.5 nanoseconds
    - ✓ Delay is a result of physical properties
  - Only timing (easily) controlled is on *clock-cycle* basis
    - ✓ Can tell synthesizer to attempt to meet cycle-time restriction

Slide taken direct from Eric Hoffman

•Zero Delay vs. Unit Delay

- When no timing controls specified: zero delay
  - Unrealistic – even electrons take time to move
  - OUT is updated same time A and/or B change:

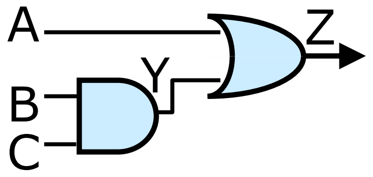
```
and A0(OUT, A, B)
```

- Unit delay often used
  - Not accurate either, but closer…
  - "Depth" of circuit does affect speed!
  - Easier to see how changes propagate through circuit
  - OUT is updated 1 "unit" after A and/or B change:

```
and #1 A0(OUT, A, B);
```

Slide taken direct from Eric Hoffman

Zero/Unit Delay Example



Zero Delay:
Y and Z change at same "time" as A, B, and C!

Unit Delay:
Y changes 1 unit after B, C

Unit Delay:
Z changes 1 unit after A, Y

Zero Delay					
T	A	B	C	Y	Z
0	0	0	0	0	0
1	0	0	1	0	0
2	0	1	0	0	0
3	0	1	1	1	1
4	1	0	0	0	1
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1
8	0	0	0	0	0
9	0	0	1	0	0
10	0	1	0	0	0
11	0	1	1	1	1
12	1	0	0	0	1
13	1	0	1	0	1
14	1	1	0	0	1
15	1	1	1	0	1

Unit Delay					
T	A	B	C	Y	Z
0	0	1	0	x	x
1	0	1	0	0	x
2	0	1	0	0	0
3	0	1	1	0	0
4	0	1	1	1	0
5	0	1	1	1	1
6	1	0	0	1	1
7	1	0	0	0	1
8	1	1	1	0	1
9	1	1	1	1	1
10	1	0	0	1	1
11	1	0	0	0	1
12	0	1	0	0	1
13	0	1	0	0	0
14	0	1	1	0	0
15	0	1	1	1	0
16	0	1	1	1	1

Slide taken direct from Prof. Schulte

Types Of Delays

- Inertial Delay (Gates)：惯性延迟
  - Suppresses pulses shorter than delay amount
  - In reality, gates need to have inputs held a certain time before output is accurate
  - This models that behavior
- Transport Delay (Nets)：传输延迟
  - "Time of flight" from source to sink
  - Short pulses transmitted
- Not critical for our project, however, in industry
  - After APR(Automatic Placement & Routing) an SDF(Standard Delay Format) is applied for accurate simulation
  - Then corner simulations are run to ensure design robust

Slide taken direct from Eric Hoffman

Delay Examples

- wire #5 net_1;   // 5 unit transport delay

- and #4 (z_out, x_in, y_in);   // 4 unit inertial delay
- assign #3 z_out = a & b;   // 3 unit inertial delay

- wire #2 z_out;   // 2 unit transport delay
- and #3 (z_out, x_in, y_in);   // 3 for gate, 2 for wire

- wire #3 c;   // 3 unit transport delay
- assign #5 c = a & b;   // 5 for assign, 3 for wire

Slide taken direct from Eric Hoffman

Verilog 运算符

- 运算符（9类）
  - 算术运算符：　　+、-、*、/、%
  - 位运算符：　　~、&、|、^、^~ 或 ~^（异或非）
  - 缩位运算符（单目）：　　&、~&(与非)、|、~|、^、^~、~^
  - 逻辑运算符：　　!、&&、||
  - 关系运算符（双目）：　　<、>、<=、>=
  - 相等与全等运算符：　　==（逻辑相等）、!=（逻辑不等）、
    ===（全等）、!==（非全等）
  - 逻辑移位运算符：　　<<、>>
  - 连接运算符：　　{ }
  - 条件运算符：　　?:

Reduction Operators
（缩位运算符）

- Reduction operators are reduce all the bits of a vector to a single bit by performing an operation across all bits.
  - Reduction AND
    - ✓ assign all_ones = &accumulator;   // are all bits set?
  - Reduction OR
    - ✓ assign not_zero = |accumulator;   // are any bits set?
  - Reduction XOR
    - ✓ assign parity = ^data_out;   // even parity bit

Vector concatenation

（向量拼接）

- Can "build" vectors using smaller vectors and/or scalar values
- Use the {} operator
- Example 1

becomes
8-bit vector:
$a_1a_0b_1b_0c_1c_0da_2$

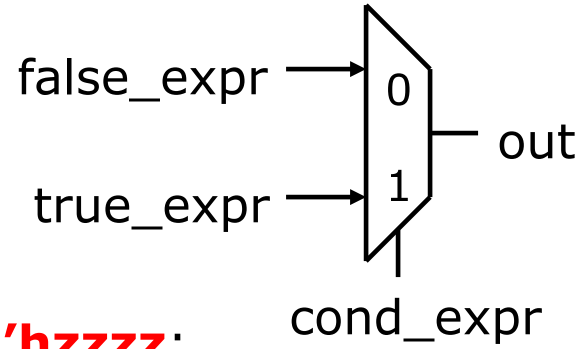
```
module concatenate(out, a, b, c, d);
    input [2:0] a;
    input [1:0] b, c;
    input d;
    output [9:0] out;

    assign out = {a[1:0],b,c,d,a[2]};

endmodule
```

Conditional Operator

- This is a favorite!
  - The functionality of a 2:1 Mux
  - assign out = conditional_expr ? true_expr : false_expr;

Examples:

```
// a 2:1 mux
assign out = select ? in0 : in1;

// tri-state bus
assign src1 = rf2src1 ? Mem[addr1] : 16'hzzzz;

// Either true_expr or false_expr can also be a conditional operator
// lets use this to build a 4:1 mux
assign out = sel[1] ? (sel[0] ? in3 : in2) : (sel[0] ? in1 : in0);
```



Slide taken direct from Eric Hoffman

Conditional assign (continued)

Examples: (nesting of conditionals)

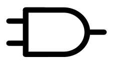
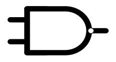
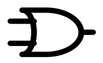
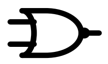
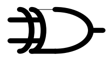
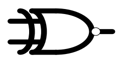
```
`define add 3'b000
`define and 3'b001
`define xor 3'b010
`define shft_l 3'b011
`define shft_r 3'b100

// an ALU capable of arithmetic,logical, shift, and zero
assign {cout,dst} = (op==`add)     ?         src1+src2+cin :
                    (op==`and)     ?         {1'b0,src1 & src2} :
                    (op==`xor)     ?         {1'b0,src1 ^ src2} :
                    (op==`shft_l)  ?         {src1,cin} :
                    (op==`shft_r)  ?         {src1[0],src1[15],src1[15:1]} :
                                             17'h00000;
```

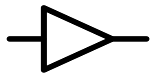
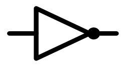
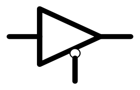
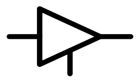
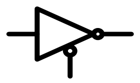
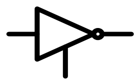
This can be very confusing to read if not coded with proper formating

Slide taken direct from Eric Hoffman

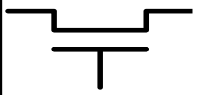
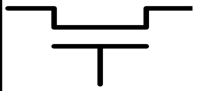
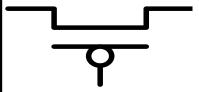
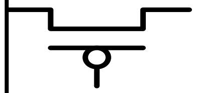
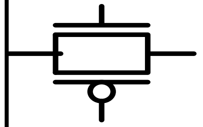
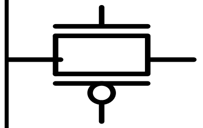
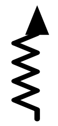
门电平模型化

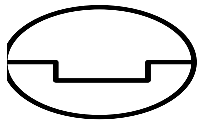
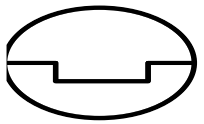
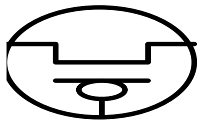
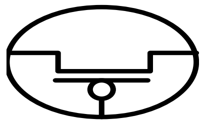
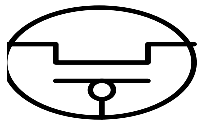
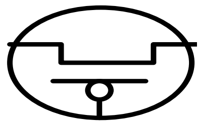
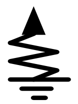
- 在Verilog HDL语言中已预定义了门级原型

	and	n-input AND gate
	nand	n-input NAND bate
	or	n-input OR gate
	nor	n-input NOR gate
	xor	n-input exclusive OR gate
	xnor	n-input exclusive NOR gate

	buf	n-output buffer
	not	n-output inverter
	bufif0	tri-state buffer; Io enable
	bufif1	tri-state buffer; hi enable
	notif0	tri-state inverter; Io enable
	notif1	tri-state inverter; hi enable

在Verilog HDL语言中已预定义了单向和双向的晶体管级原型

	Pmos	uni-directional PMOS switch
	rpmos	resistive PMOS switch
	nmos	uni-directional NMOS switch
	rnmos	resistive NMOS switch
	cmos	uni-directional CMOS switch
	rcmos	resistive CMOS switch
	pullup	pullup resistor

	tran	bi-directional pass transistor
	rtran	resistive pass transistor
	tranif0	bi-directional trnasistor;Io enable
	rtranif	resistive transitor; Io enable
	tranif1	bi-directional transistor;hi enable
	rtranif1	resistive transistor; hi enable
	pulldow	pulldown resistor

用户定义的原语

- UDP（User Defined Primitives）的定义

```
primitive  UDP 名 ( 输出端口名, 输入端口名);
    output    输出端口名;    //只允许一个输出端口
    input     输入端口名;
    reg       输出端口名;  //可选，时序逻辑的UDP才需要
    initial   输出端口名=值  //可选，时序逻辑的UDP才需要
    table
              状态表
    endtable
endprimitive
```

- 在组合电路UDP中，表规定了不同的输入组合和相对应的输出值。
- 在时序电路UDP中，使用1位寄存器描述内部状态。该寄存器的值是时序电路UDP的输出值。共有两种不同类型的时序电路UDP：
  - 模拟电平触发行为
  - 模拟边沿触发行为
- 时序电路UDP使用寄存器当前值和输入值决定寄存器的下一状态（和后继的输出）。