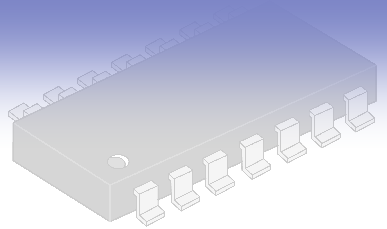
第 2 章

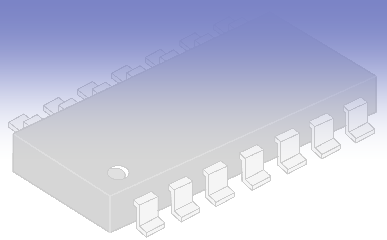
Verilog 的基本概念

内容



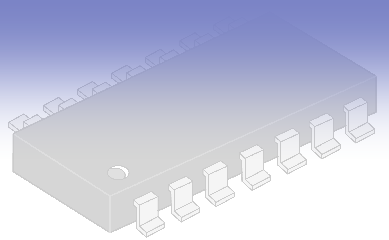
- 模块（module）
- 端口（port）
- 层次命名（Hierarchical Names）

Verilog 模块的基本概念



- Verilog 模块（module）
  - 基本描述单位
  - 代表一个基本的功能模块
  - 描述某个数字电路的功能、结构
  - 与其它模块通信的外部接口
- 模块的构成
  - 最简单的情况——一个基本器件
  - 低层次模块（子模块）的组合
- 高层的模块通过输入/输出接口调用低层的模块

Verilog 模块的结构



module 模块名(标识符) (端口列表/端口声明列表);

端口声明;
参数声明;

wire, reg 及其它类型变量声明;

assign 数据流语句;

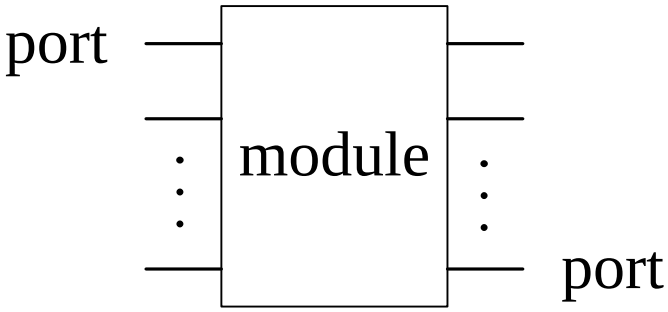
initial 块和 always块

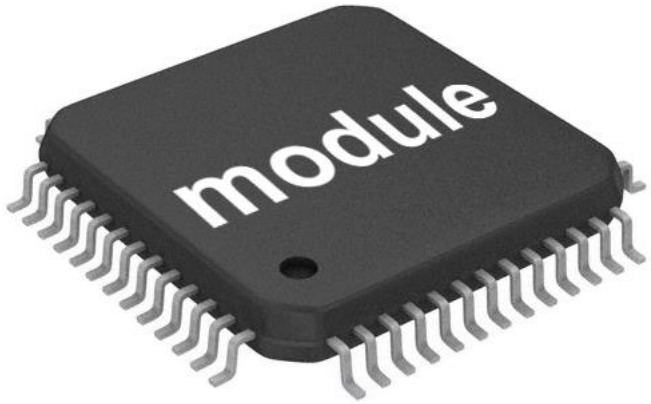
所有行为语句均在这些块中;

低层模块实例语句;

任务和函数;

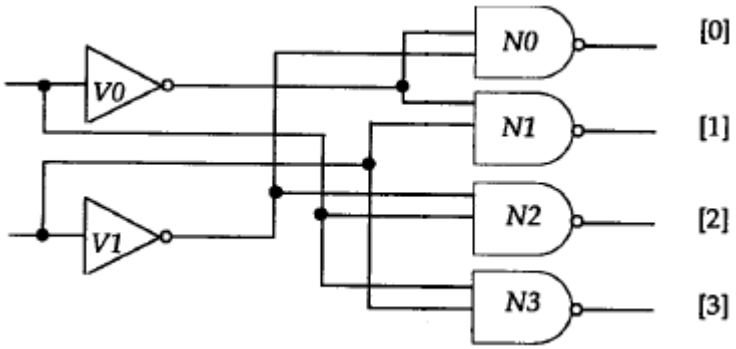
endmodule // 模块结束语句





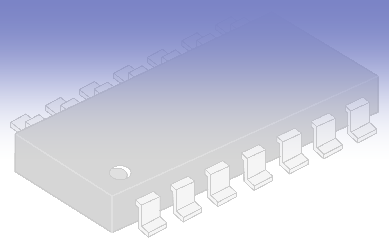
Verilog 模块的基本构成

- I/O 端口说明（declaration）
- 内部信号说明（declaration）
- 逻辑功能描述（modeling ）




```
 2    module decoder2x4( output reg [3:0] y, input [1:0] sel );
 3
 4    always @(*)
 5    begin
 6        case ( sel )
 7            0:     y = 1;
 8            1:     y = 2;
 9            2:     y = 4;
10            3:     y = 8;
11            default:  y = 4'bx;
12        endcase
13    end
14    endmodule
```

Verilog 模块的构成（1）



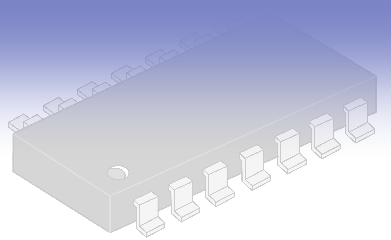
- 必须包含
  - module 开始
  - endmodule 结束

```
module tb_mux2x1();
    reg [2:0] p_data;
    wire p_dout;
    integer i;

    mux2x1 m0( .dout( p_dout ), .sel(p_data[2]), .din(p_data[1:0]) );

    initial begin
        p_data = 3'bx;
        #5 p_data = 3'b0;
        for ( i = 1; i < 8; i = i + 1 )
            #5 p_data = p_data + 3'b1;
    end

    initial
        $monitor( "At time %t, dout=%1b, sel=%1b, din=%2b",
                  $time, p_dout, p_data[2], p_data[1:0]);
endmodule
```

- 可选的…
  - 端口列表及声明
  - 模块内部5个组成部分
    - 变量声明
    - 数据流语句
    - 低层模块实例
      - 引用低层模块
    - 行为模块
    - 任务和函数

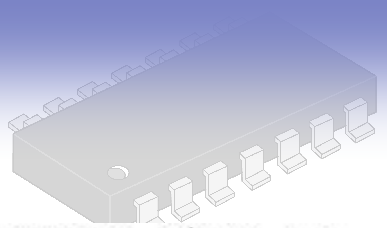
- 在一个Verilog源文件中，可以定义多个模块
  - 有些综合工具要求，一个Verilog文件只包含和定义一个模块

测试台模块



```
module tb_mux2x1();
    reg [2:0] p_data;
    wire p_dout;
    integer i;

    mux2x1 m0( .dout( p_dout ), .sel(p_data[2]), .din(p_data[1:0]) );

    initial begin
        p_data = 3'bx;
        #5 p_data = 3'b0;
        for ( i = 1; i < 8; i = i + 1 )
            #5 p_data = p_data + 3'b1;
    end

    initial
        $monitor( "At time %t, dout=%1b, sel=%1b, din=%2b",
                  $time, p_dout, p_data[2], p_data[1:0]);
endmodule
```

Verilog 模块的构成（2）



- 每一个模块包含在关键字对之间

  module

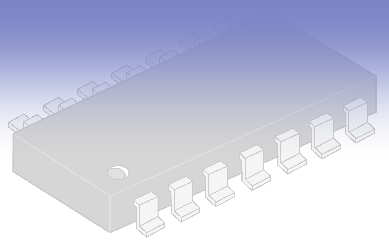
  … … … …

  endmodule

- 说明部分（Declaration）
  - 定义
    - 模块的 I/O 端口
    - 寄存器类型变量、线网变量、参数
    - 调用的函数、任务
- 语句部分
  - 定义一个设计的功能和结构
- 说明、语句可散布在模块中的任何地方
  - 说明必须在使用前出现
- 建议
  - 为了描述清晰，最好将所有说明放在语句之前，使模块具有良好的可读性

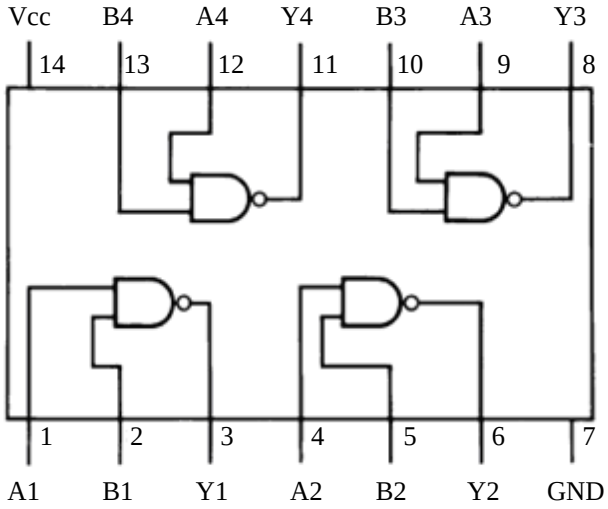
```
module module_name( port_list );
    Declaration:
        input,output, inout,
        wire, reg, parameter,
        task, function,...
    Statement:
        Initial statement;
        Continuous assignmet;
        Always statement;
        Generate statement;
        Module instantiation;
        Gate instantiation;
        UDP instantiation;
endmodule
```

例、简单模块



- 例1、 DM7400
  - 74系列的4 个 2 输入与非门
- 电路模块有3 组输入/输出端口
  - 2 组 4 位输入：a 、b
  - 1 组 4 位输出：y
- 电路模块实现的逻辑功能
  - 2 输入与非门
- Verilog 模块 dm7400
  - 与电路模块一一对应
  - 在端口列表中说明/定义所有 I/O 端口
  - 使用连续赋值语句描述逻辑功能
    - 连续赋值语句将值赋给线网（net）
    - 线网 —— 对应电路中的**连线**

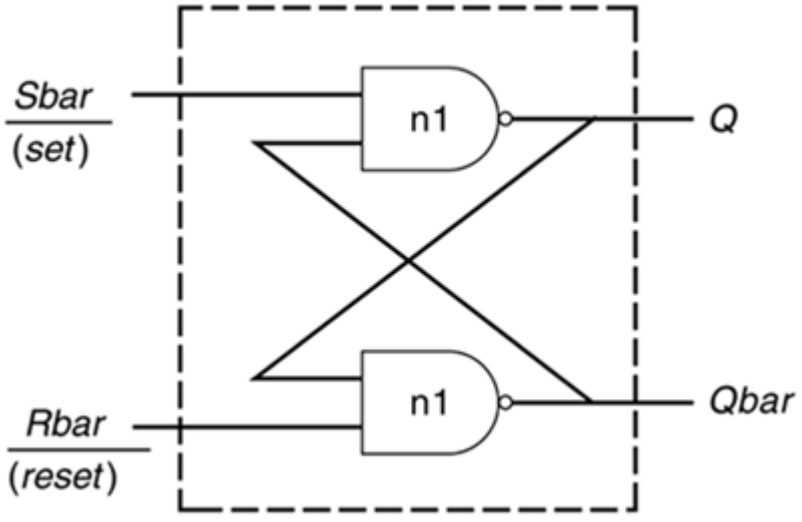
```
module dm7400( output [3:0] y,
               input [3:0] a,
               input [3:0] b );
    assign y = ~( a & b );
endmodule
```



DM7400：4 个 2 输入与非门

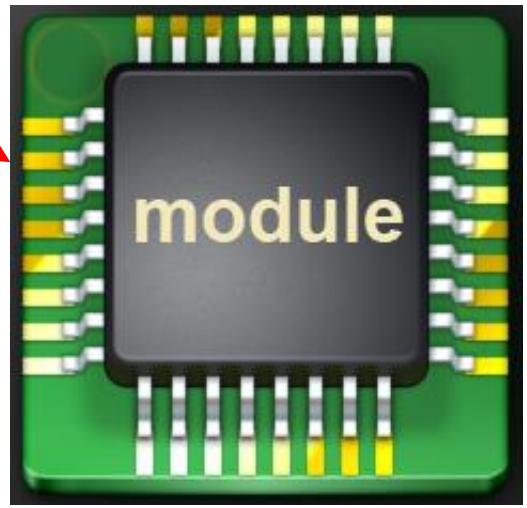
例、SR 锁存器

```
// SR_latch module
module SR_latch(Q, Qbar, Sbar, Rbar);

//Port declarations
output Q, Qbar;
input Sbar, Rbar;

// 底层模块实例，引用 Verilog 原语 (primitive) 与非门
nand n1(Q, Sbar, Qbar);
nand n2(Qbar, Rbar, Q);

endmodule // 模块 SR_latch 结束


// 激励信号模块 (Stimulus module)
module Top;

// wire, reg 变量声明
wire q, qbar;
reg set, reset;

// 底层模块实例，引用 SR_latch 模块
SR_latch m1(q, qbar, ~set, ~reset);

initial // 模块 SR_latch 结束
begin
  $monitor($time, " set = %b, reset= %b, q= %b\n",set,reset,q);
  set = 0; reset = 0;
  #5 reset = 1;
  #5 reset = 0;
  #5 set = 1;
end

endmodule // 模块 Top 结束
```

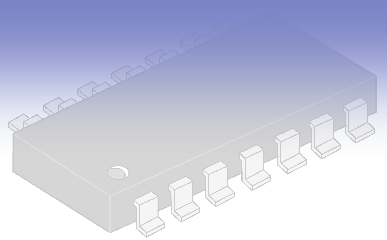


端口（ports）

- 模块的输入/输出端口是与外部环境交互的信号通道接口
  - 相当于电路中芯片的管脚（pin）
  - 通过端口对模块进行调用
    - 实例引用
- 电路中的其它模块
  - 通过输入/输出端口与模块连接
  - 信号经由端口流入/流出
- 端口列表
  - 模块的所有输入/输出信号通道的列表
    - 输入/输出端口标识符列表
  - 模块可以没有端口列表
    - 如，模块没有任何和外部环境的交换的信号
    - 最顶层模块 —— 测试台（Testbench）模块



模块的端口列表



- 声明一个模块的输入/输出端口，格式：

  `module` module_name （list_of_ports）;

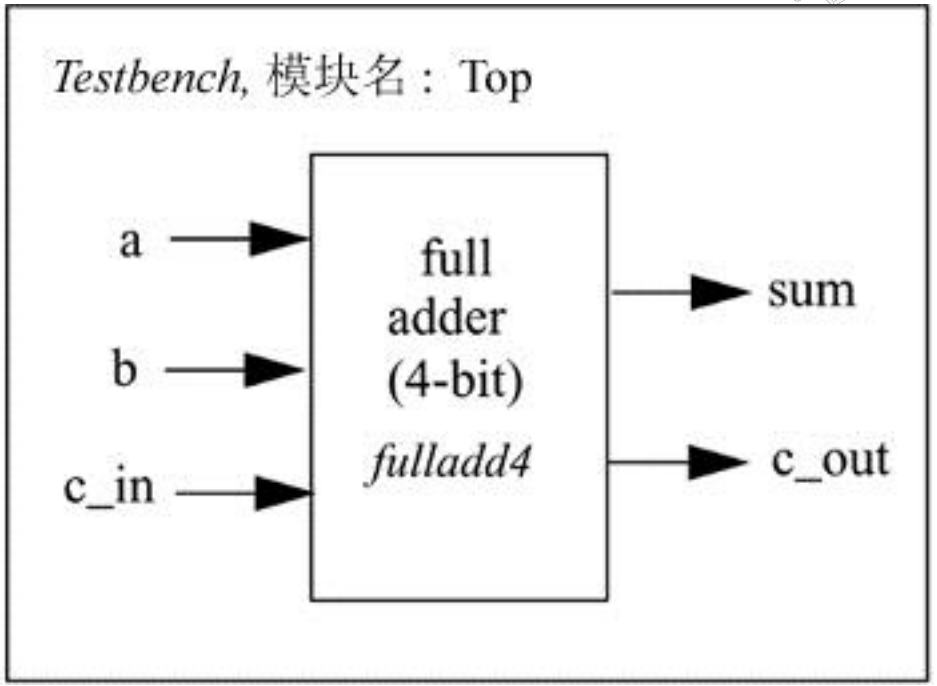
  注意：结尾分号（ ; ），

  除 endmodule 语句之外，每条语句后均有分号

- 模块名
  - 标识模块名称的标识符
- 端口列表：
  - 由 I/O 端口名（标识符）组成
  - 格式：
    - port1, por2, port3, …
- 输入/输出端口的出现顺序
  - 可任意顺序
  - 推荐：
    - 在一个设计中选定一种顺序，如：先输出端口，再输入端口
    - 例：(cout, sum, a, b)

例、端口列表

- 全加器和测试台模块的 I/O

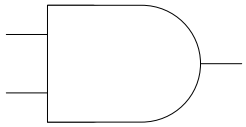


- module fulladd4(sum, c_out, a, b, c_in); // Module with a list of ports
- module Top; // No list of ports, top-level module in simulation

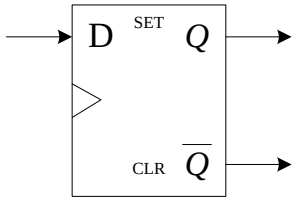
端口声明（1）

- 必须对端口列表中的所有端口的类型进行声明
- Verilog中的端口有 3 种类型（ I/O 类型）—— 表示信号流动的方向

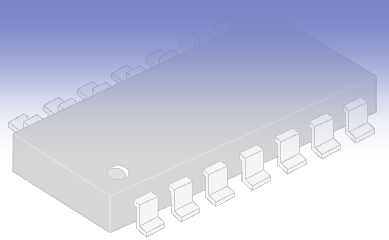
Verilog Keyword	Type of Port
input	输入端口
output	输出端口
inout	输入/输出双向端口(Bidirectional port)



- 端口变量的数据类型
  - **wire**
    - 所有端口的数据类型默认为线网数据类型
  - **reg**
    - 如果输出端口保持数值，必须显式地声明为寄存器数据类型
  - 不能将 **input** 和 **inout** 类型的端口声明为 **reg** 类型
    - 输入端口只反映与其连接的外部信号的变化，不能保存这些信号的值

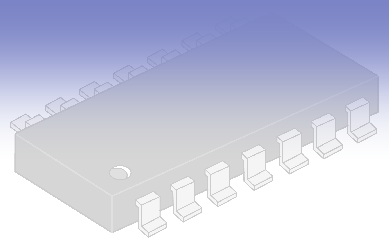


端口声明（2）



- 明确定义 I/O 端口
- 输入端口的说明（定义）
  - 例
    - input [信号位宽 - 1 : 0] 端口名1;
    - input [信号位宽 - 1 : 0] 端口名2;
    - input [信号位宽 - 1 : 0] 端口名3;
    - input [信号位宽 - 1 : 0] 端口名4;
  - 或
    - input [信号位宽 - 1 : 0] 端口名1，端口名2，端口名3，端口名4，… ;
- 输出端口的说明（定义）
  - 例
    - output [信号位宽 - 1 : 0] 端口名1;
    - output [信号位宽 - 1 : 0] 端口名2;
  - 或
    - output [信号位宽 - 1 : 0] 端口名1，端口名2，端口名3，端口名4，… ;
- 双向端口的说明（定义）
  - 例
    - inout [信号位宽 - 1 : 0] 端口名1，端口名2，端口名3，端口名4，… ;

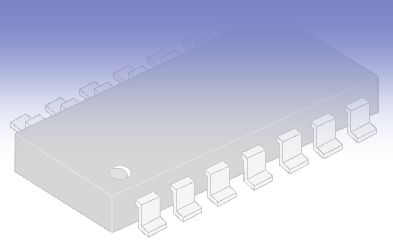
例、端口声明（1）



- 四位全加器模块的端口声明
  - 可以在端口列表中声明端口的类型，否则，端口列表中的所有端口必须在模块中声明

```
module fulladd4(sum, c_out, a, b, c_in);

//Begin port declarations section
output[3:0] sum;
output c_cout;

input [3:0] a, b;
input c_in;
//End port declarations section
...
<module internals>
...
endmodule
```

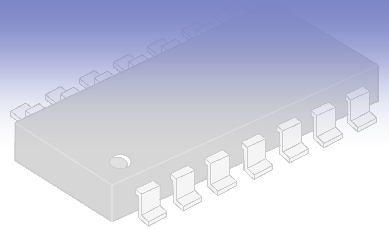
例、端口声明（2）



- DFF模块的端口声明

```
module DFF(q, d, clk, reset);
output q;
reg    q; // 输出端口 q 保持值, 因此, 它被声明为寄存器类型( reg) 的变量.
input  d, clk, reset;
...
...
endmodule
```

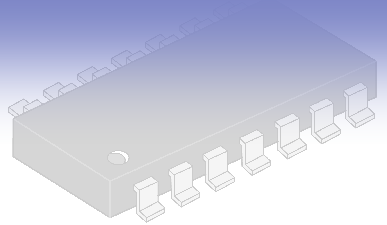
ANSI C 风格的端口声明



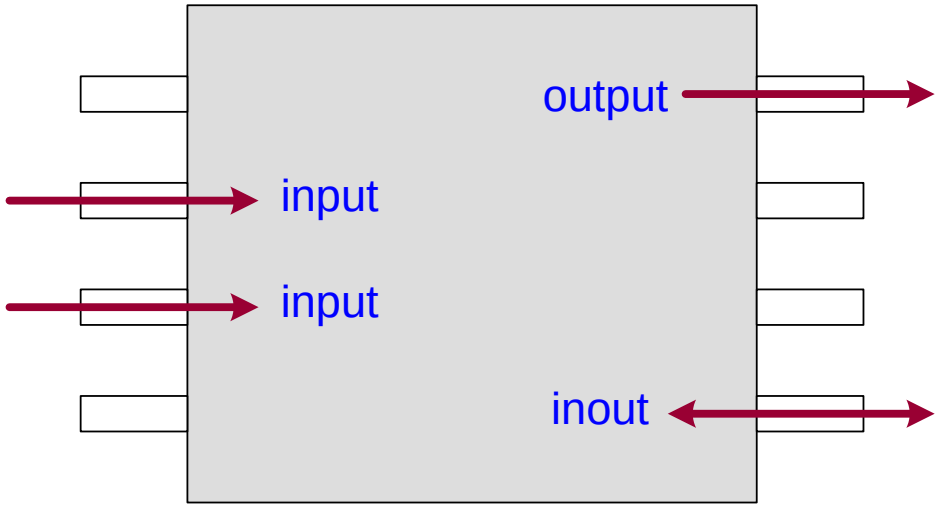
- 在端口列表中说明端口的 I/O 类型
- 例、
- module m_name (output y1, output y2, inout z1, inout z2, input a1, input a2, …);

```
module fulladd4( output reg [3:0] sum,
                 output reg c_out,
                 input [3:0] a, b, //wire by default
                 input c_in);      //wire by default
...
<module internals>
...
endmodule
```

端口的连接规则（1）

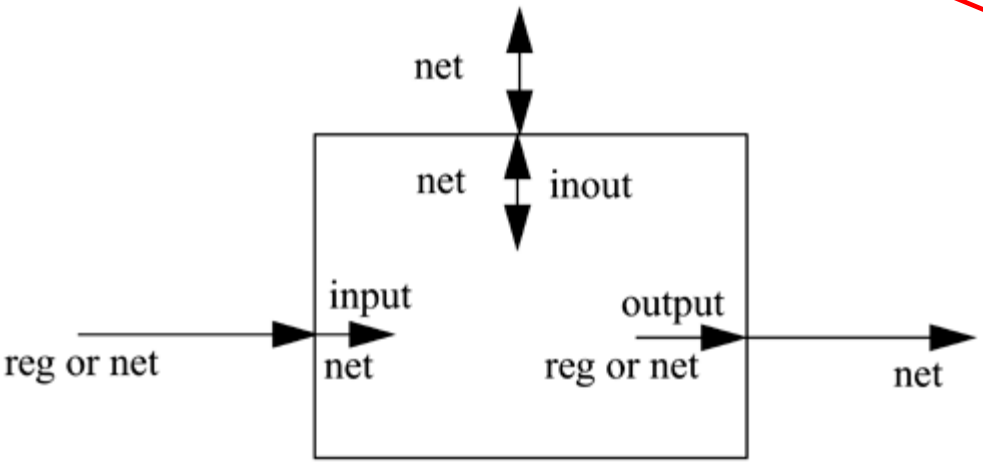


- 端口—— 由相互连接的两个部分组成
  - 对应芯片的管脚引线
- 管脚引线包括两个部分
  - 一部分在芯片内部
    - 连接内部逻辑电路
  - 一部分在芯片外部
    - 连接电路板上的其它芯片或器件的端口

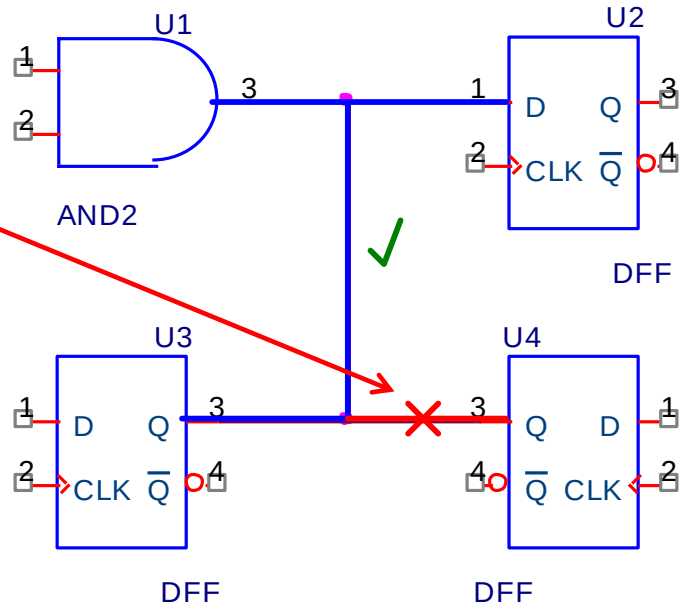


端口的连接规则（2）

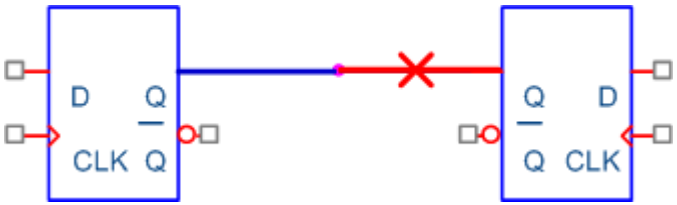
- 输入端口（input）
  - 在模块内是线网数据类型
  - 外部可以连接到
    - 线网（net）数据类型：wire
    - reg 数据类型：reg
- 输出端口（output）
  - 在模块内，可以有两种
    - 线网数据类型
    - reg 数据类型
  - 外部必须连接到线网数据类型



- 输入/输出端口（inout）
  - 在模块内是线网数据类型
  - 外部必须连接线网数据类型



例、非法端口连接

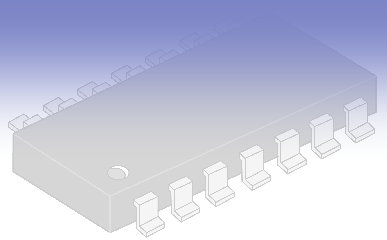



```
module Top;

      // 声明连接变量
      reg [3:0]A,B;
      reg C_IN;
      reg [3:0] SUM;
      wire C_OUT;

      // fulladd4 实例引用，实例名 fa0
      fulladd4 fa0(SUM, C_OUT, A, B, C_IN);
      // 非法连接，fulladd4 的输出端口 sum 连接到模块 Top 中的 reg 型变量 SUM 上
      …
      <stimulus>
      …
endmodule
```

```
module fulladd4(sum, c_out, a, b, c_in);

//Begin port declarations section
output[3:0] sum;
output c_cout;

input [3:0] a, b;
input c_in;
//End port declarations section
...
<module internals>
...
endmodule
```

外部必须连接到线网数据类型

端口与外部信号的连接

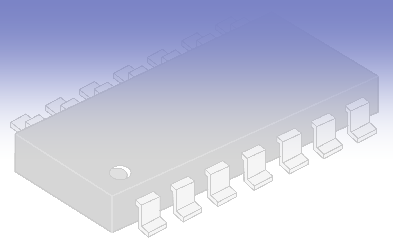


- 模块的引用
  - 被其它模块引用时，连接方法有两种
- 方法1 ——顺序端口连接
  - 模块标识符

    模块实例名（**连接端口 1 的信号名，连接端口 2的 信号名，…**）
  - 必须严格按照端口声明的顺序连接
- 方法2 ——命名端口连接
  - 模块标识符

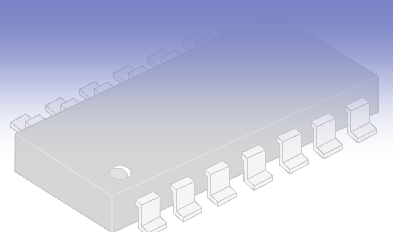
    模块实例名（.端口 1(连接信号名)， .端口 2(连接信号名) ，…）
  - 不必按照端口声明的顺序连接

例、顺序端口连接



```
module Top;

      // 声明连接变量
      reg  [3:0]A,B;
      reg  C_IN;
      wire [3:0] SUM;
      wire C_OUT;

      // 模块 fulladd4 的实例引用, 实例命名: fa_ordered.
      // 信号按端口列表次序连接
      fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
      ...
      <stimulus>
      ...
endmodule


module fulladd4(sum, c_out, a, b, c_in);
      output[3:0] sum;
      output c_cout;
      input [3:0] a, b;
      input c_in;
      ...
      <module internals>
      ...
endmodule
```

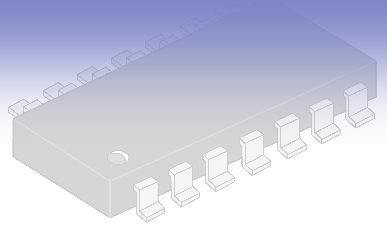
例、命名端口连接



```
1 module fa_1b ( cout, sum, a, b, cin );
2
3         input  a, b, cin;
4         output cout, sum;
5
6         assign {cout, sum } = a + b + cin;
7
8 endmodule
```

```
1 `timescale 1 ns / 100 ps
2
3 `include "fa_1b.v"
4
5 module testbench;
6
7         reg pa, pb, pcin;
8         wire pcout, psum;
9
10        reg [2 : 0] pval;
11
12        fa_1b m_fa( .cout(pcout), .sum(psum), .a(pa), .b(pb), .cin(pcin) );
13
14        initial
15        begin
16                for ( pval = 0; pval < 8; pval = pval + 1 )
17                begin
18                        { pcin, pb, pa } = pval;
19
20                        #5 $display ( "pa, pb, pcin = %b%b%b", pa, pb, pcin,
21                                      " --- pcout, psum = %b%b", pcout, psum );
22                end
23        end
24 endmodule
```

未连接的端口



- 允许模块引用（调用）实例的端口保持未连接状态

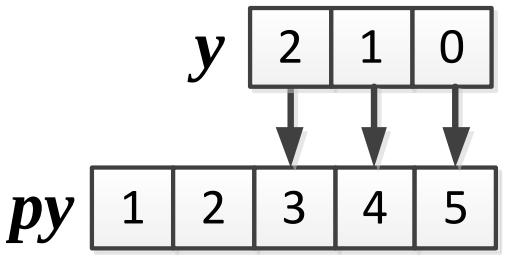
```
module dff(output q, qbar, input d, clear, clk );
    ......
endmodule
```

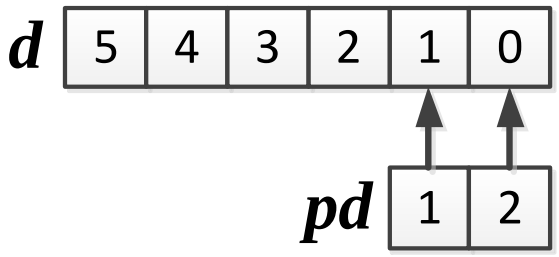
```
module top()
    wire pq;
    reg  pd, pclr, pclk;

    // 命名端口连接
    dff u0(.q(pq), .qbar(), .d(pd), .clear(pclr), .clk(pclk) )

    // 顺序端口连接
    dff u1( pq, , pd, pclr, pclk);
    ........
endmodule
```

不同的端口位宽

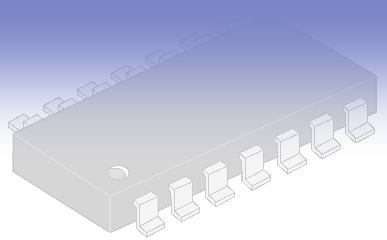
- 对模块实例引用（实例调用）时，当连接端口的变量与被调用端口位宽不同时，进行端口位宽匹配

```
module child( output [2:0] y, input [5:0] d);
    ......
endmodule

module top;
    wire [1:5] py;
    wire [1:2] pd;

    child u0(.y(py), .d(pd) )
endmodule
```





层次命名（Hierarchical Names ）



- 使用Verilog进行层次化设计
  - 每一个模块实例、信号和变量都使用一个标识符定义
  - 每个标识符都有一个唯一的层次命名
- 层次命名 —— 层次路径名
  - 从**顶层模块**开始，由句点（.）格开的名字（序列）组成
  - **顶层模块** —— 不被其它任何模块实例化的模块
- 每个标识符表示一个层次
  - 层次包括：
    - 任务定义
    - 函数定义
    - 命名程序块
    - 模块实例化

例、所有对象的层次命名

- 原始 Verilog 代码：

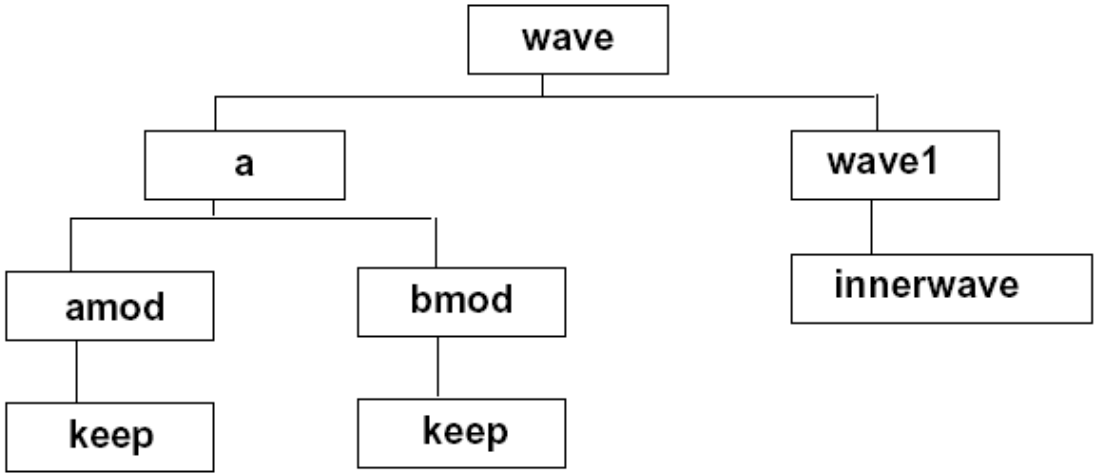
```
wave
wave.stim1
wave.stim2
wave.a
wave.a.stim1
wave.a.stim2
wave.a.amod
wave.a.amod.in
wave.a.amod.keep
wave.a.amod.keep.hold
wave.a.bmod
wave.a.bmod.in
wave.a.bmod.keep
wave.a.bmod.keep.hold
wave.wave1
wave.wave1.innerwave
wave.wave1.innerwave.hold
```

```
module mod (in);
input in;

always @(posedge in) begin : keep
reg hold;
    hold = in;
end
endmodule
```

```
module cct (stim1, stim2);
input stim1, stim2;

// instantiate mod
mod amod(stim1), bmod(stim2);
endmodule
```

```
module wave;
reg stim1, stim2;

cct a(stim1, stim2); // instantiate cct

initial begin :wave1
    #100   fork :innerwave
               reg hold;
           join
    #150   begin
               stim1 = 0;
           end
end
endmodule
```

Verilog 代码中蕴涵的层次



利用层次命名引用对象

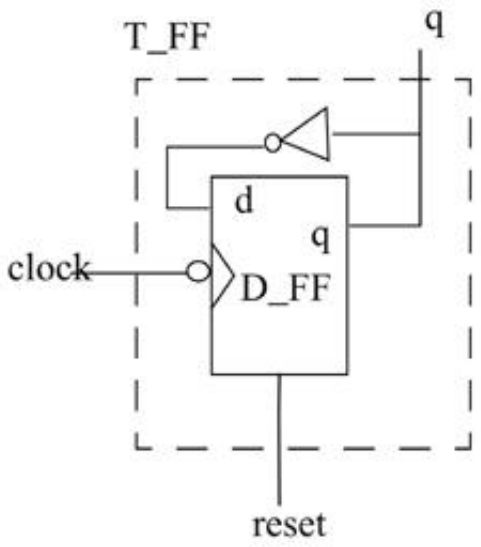
- 通过使用层次路径名
  - 可以唯一指明在一个设计（项目）中的每个标识符

```
VSIM 33> run 2us
# At 0,  m_art.y_out = x
# At 50,  m_art.y_out = 0
# At 100,  m_art.y_out = 6
# At 150,  m_art.y_out = 4
# At 200,  m_art.y_out = 13
# At 250,  m_art.y_out = 1
# At 300,  m_art.y_out = 11
# At 350,  m_art.y_out = 9
# At 400,  m_art.y_out = 13
# At 450,  m_art.y_out = 3
# At 500,  m_art.y_out = 6
# At 550,  m_art.y_out = 14
# At 600,  m_art.y_out = 2
# At 650,  m_art.y_out = 6
# At 700,  m_art.y_out = 8
# At 750,  m_art.y_out = 5
# At 900,  m_art.y_out = 1
# At 950,  m_art.y_out = 4
# At 1000,  m_art.y_out = 0
```

```
 1 `timescale 10ns / 1ns
 2
 3 module art;
 4
 5     reg [3: 0] y_out;
 6     integer seed = 10;
 7
 8     initial repeat (20) y_out = #5 { $random(seed) } % 16;
 9
10 endmodule
11
12 module top;
13
14     art m_art();
15
16     initial $monitor("At %0t,    m_art.y_out = %d", $time, m_art.y_out);
17
18 endmodule
```

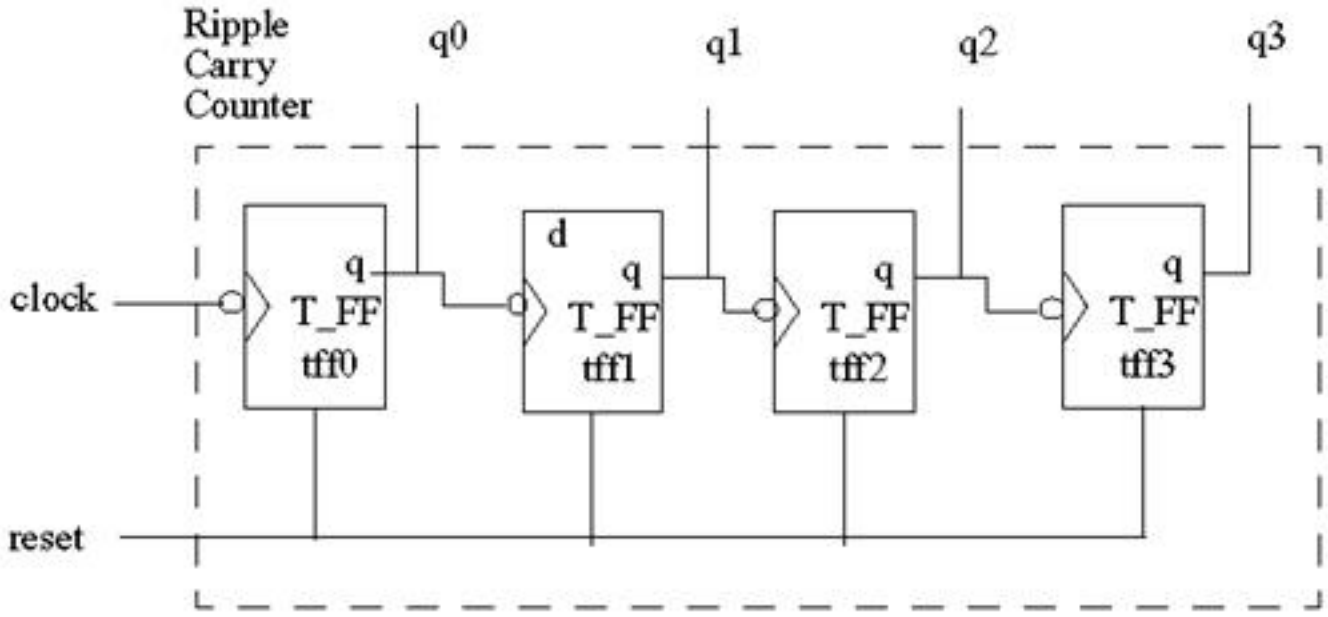
层次化设计举例

- 4 位行波进位计数器
  - 由下降沿触发的T触发器构成
    - T触发器
      - 由D触发器和反相器构成

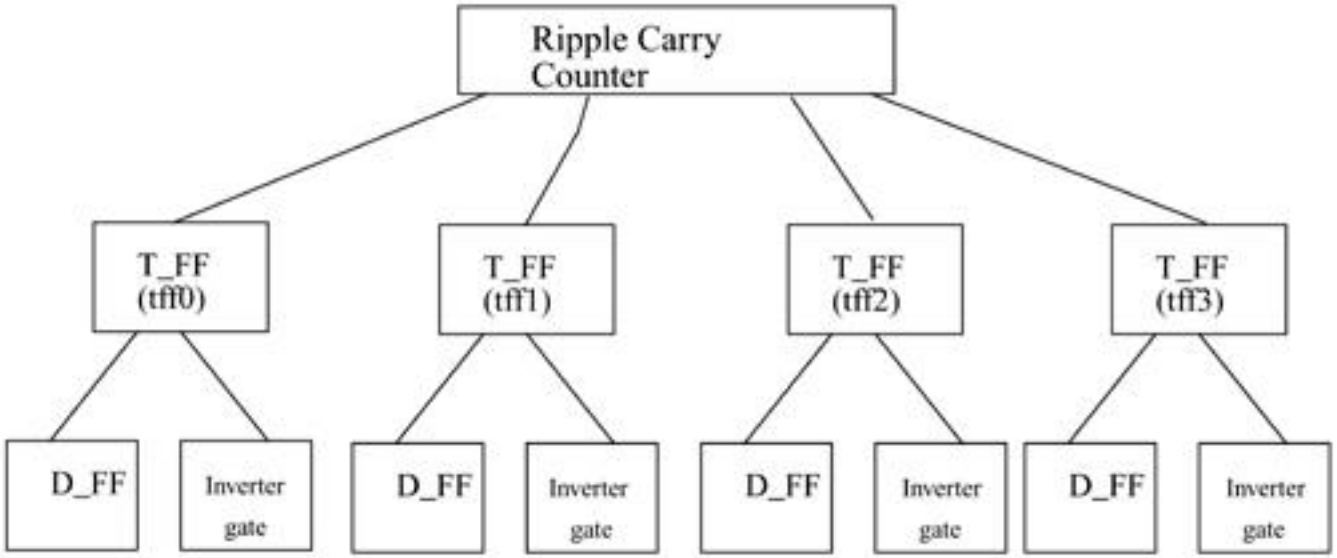


reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0

T触发器的真正表

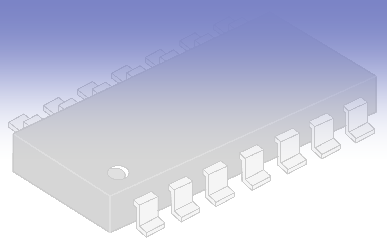


4 位行波进位计数器的设计层次



- 自顶向下
  - 行波进位计数器（功能）→T触发器→D触发器和反相器（非门）
- 自底向上
  - D触发器和反相器（非门）→T触发器→行波进位计数器（功能）

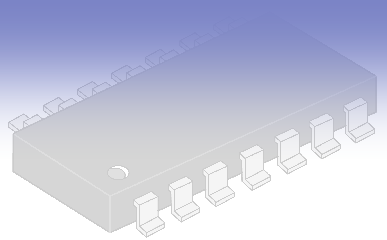
使用Verilog HDL 分层建模（设计）（1）



- 自顶向下设计
- 4 位行波进位计数器顶层模块

```
`include "T_FF.v"

// 定义行波进位计数器模块 ( the top-level module )
// 模块名: ripple_carry_counter
module ripple_carry_counter(q, clk, reset);

output [3:0] q;      // 输入/输出端口的信号或向量声明
input clk, reset;    // 输入/输出端口的信号或向量声明

// 创建 4 个T触发器实例，每个T触发器实例有其唯一的实例名
T_FF tff0(q[0],clk,  reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule
```

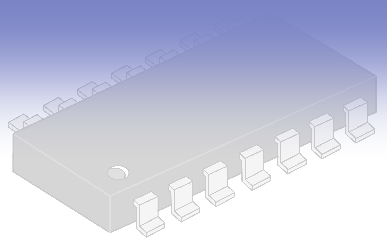
使用Verilog HDL 分层建模（设计）（2）



- T触发器

```
`include "D_FF.v"

// 定义T触发器模块
// 模块名: T_FF
module T_FF(q, clk, reset);

// 输入/输出端口的信号或向量声明
output q;
input clk, reset;

// 声明线网变量
wire d;

// 创建D触发器(D_FF)模块实例
D_FF dff0(q, d, clk, reset);

// 非门(not), 使用Verilog内置原语(primitive)
not n1(d, q);

endmodule
```

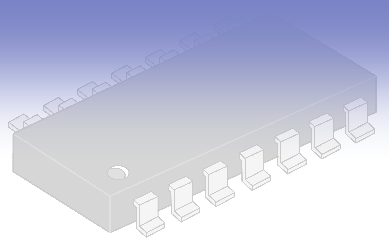
使用Verilog HDL 分层建模（设计）（3）



- D 触发器

```
// 定义具有同步复位的D触发器(D Flipflop)模块
// 模块名: D_FF
// module D_FF with synchronous reset
module D_FF(q, d, clk, reset);

output q;
input d, clk, reset;
reg q;

// Synchronous reset
always @(posedge reset or negedge clk)
if (reset)
    q <= 1'b0;
else
    q <= d;

endmodule
```

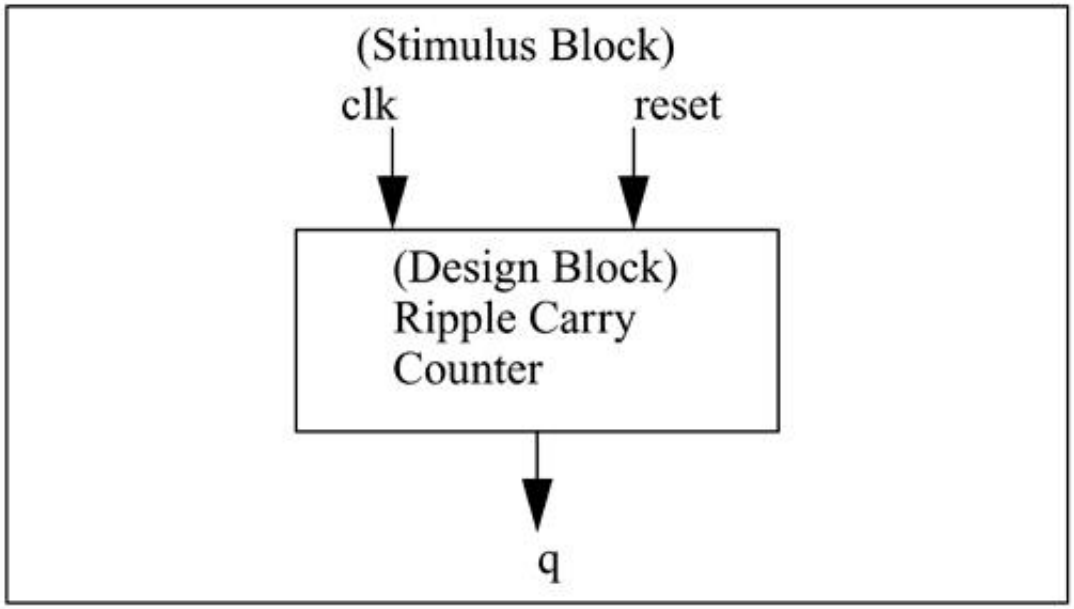
对设计结果进行仿真

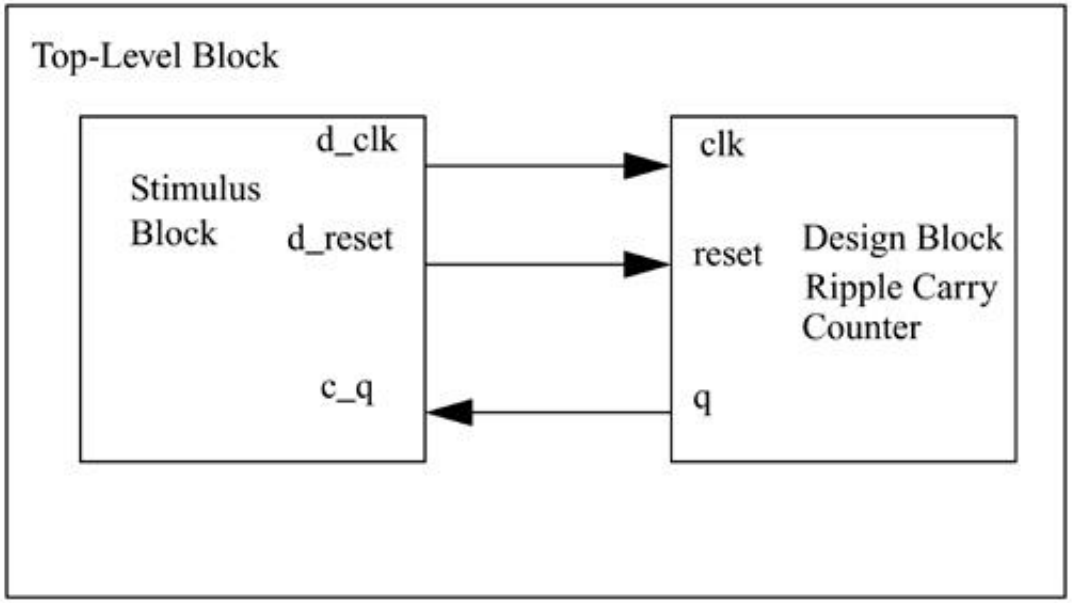


- 仿真测试
  - 在设计过程中，每一层设计完成后，都进行仿真
  - 以便在系统设计的早期发现问题
- 测试台（Testbench）
  - 对设计进行仿真测试的（Verilog）HDL模块
  - 测试模块和功能设计模块应当分开描述（设计）
  - 对一个功能模块可以设计多个测试台模块进行仿真

仿真测试模块的设计

- Testbench 的设计模式
  - 在产生激励信号的模块中调用（引用实例）
  - 设计一个顶层模块（top_level），在其中调用设计模块和激励信号模块





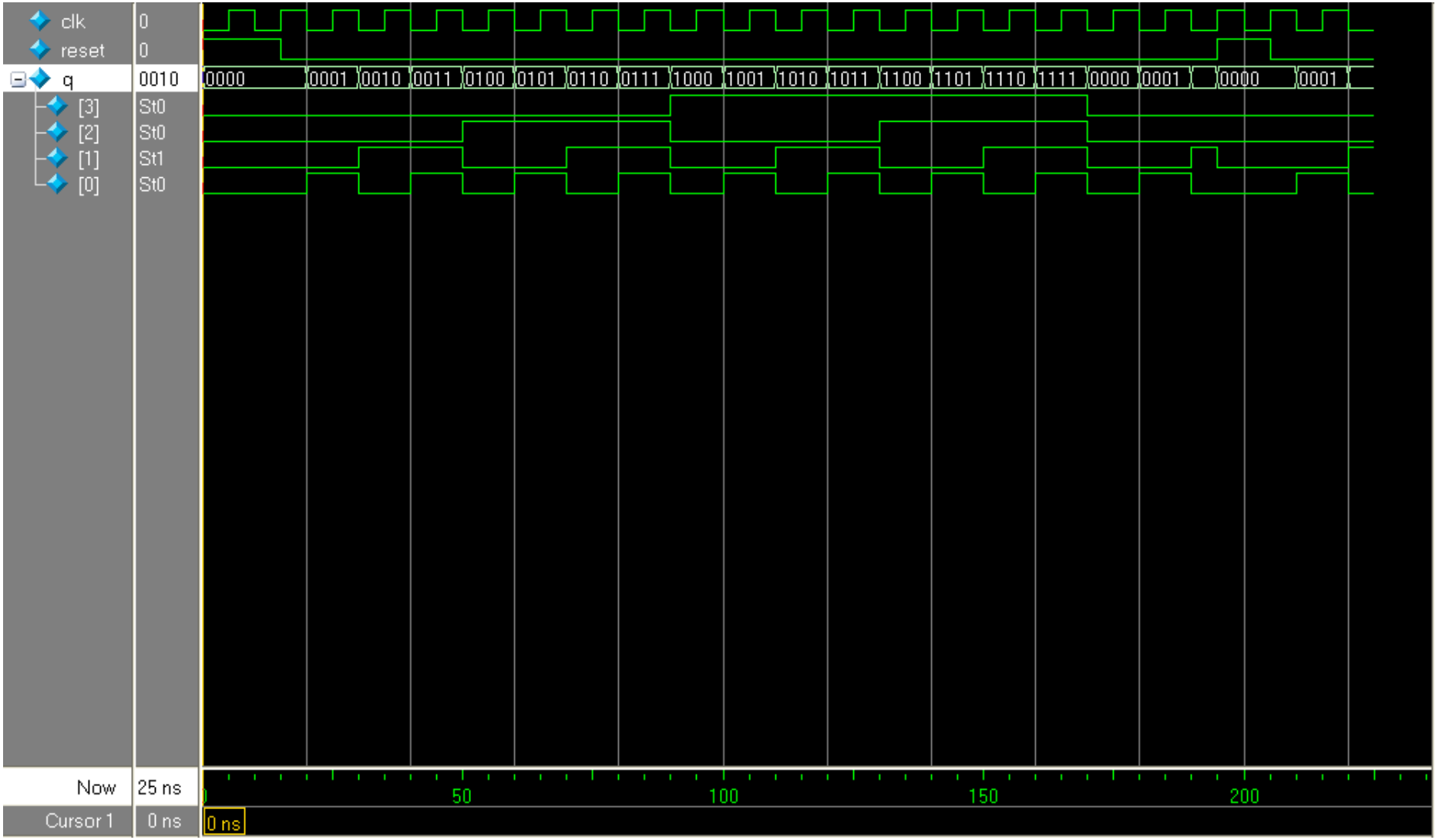
测试台设计

- 例
  - 用于仿真测试4 位行波进位计数器
  - 在产生激励信号的模块中调用（引用实例

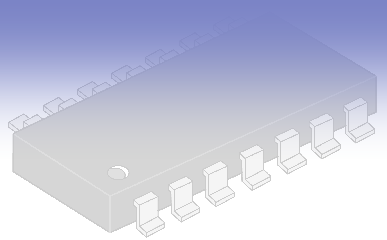
```
module testbench;

reg clk;
reg reset;
wire[3:0] q;

// 创建一个行波进位计数器模块实例
ripple_carry_counter r1(q, clk, reset);

// 创建一个周期为 10 个时间单位的时钟信号
initial
   clk = 1'b0;       //set clk to 0
always
   #5 clk = ~clk;    //toggle clk every 5 time units

// Control the reset signal that drives the design block
// reset is asserted from 0 to 20 and from 200 to 220.
initial
begin
         reset = 1'b1;
   #15   reset = 1'b0;
   #180  reset = 1'b1;
   #10   reset = 1'b0;
   #20   $finish;          //terminate the simulation
end

// Monitor the outputs
initial
   $monitor($time, " Output q = %d",   q);

endmodule
```

仿真验证设计（1）

- 仿真波形



仿真验证设计（2）



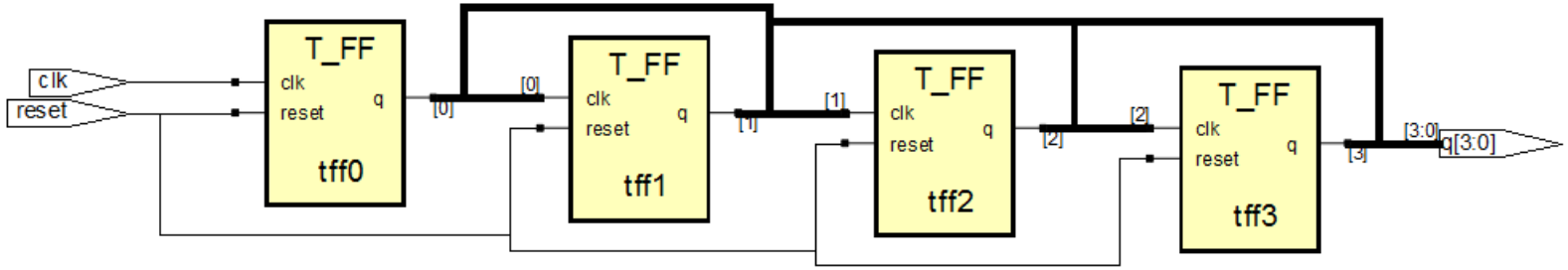
- 测试输出结果

```
run 300ns
                0 Output q =  0
               20 Output q =  1
               30 Output q =  2
               40 Output q =  3
               50 Output q =  4
               60 Output q =  5
               70 Output q =  6
               80 Output q =  7
               90 Output q =  8
              100 Output q =  9
              110 Output q = 10
              120 Output q = 11
              130 Output q = 12
              140 Output q = 13
              150 Output q = 14
              160 Output q = 15
              170 Output q =  0
              180 Output q =  1
              190 Output q =  2
              195 Output q =  0
              210 Output q =  1
              220 Output q =  2
```

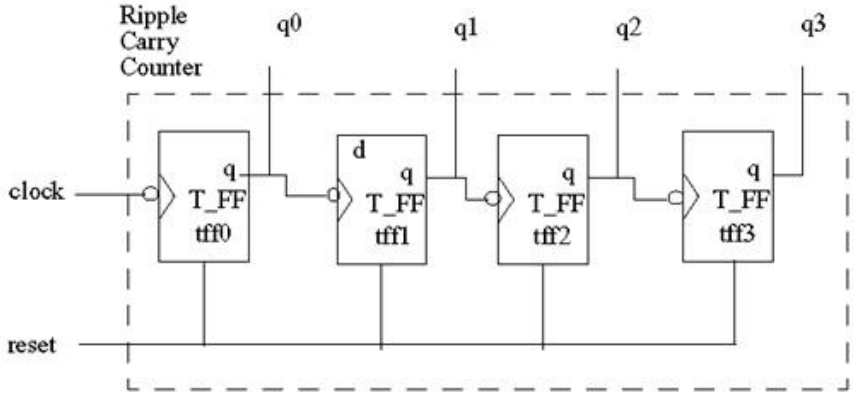
设计综合（1）

综合工具

- Synplify Pro E-2011.03-SP2， 下图：RTL级电路图

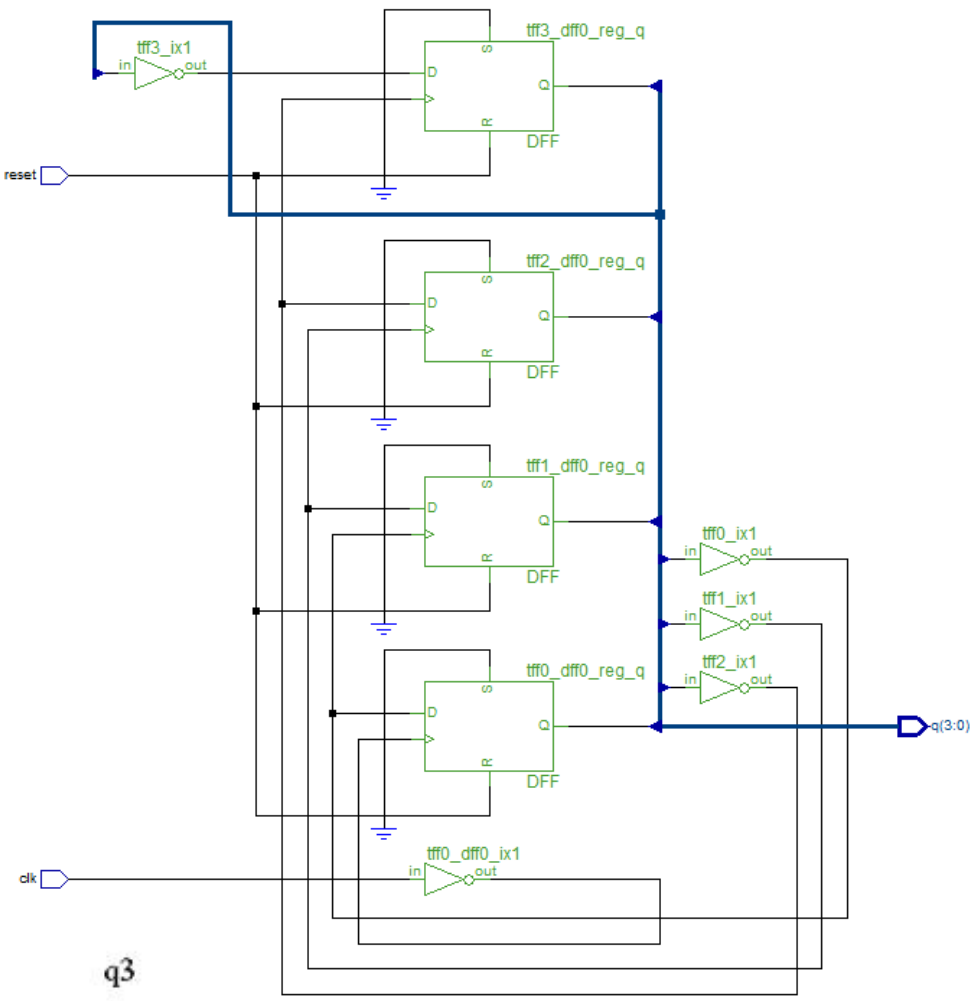


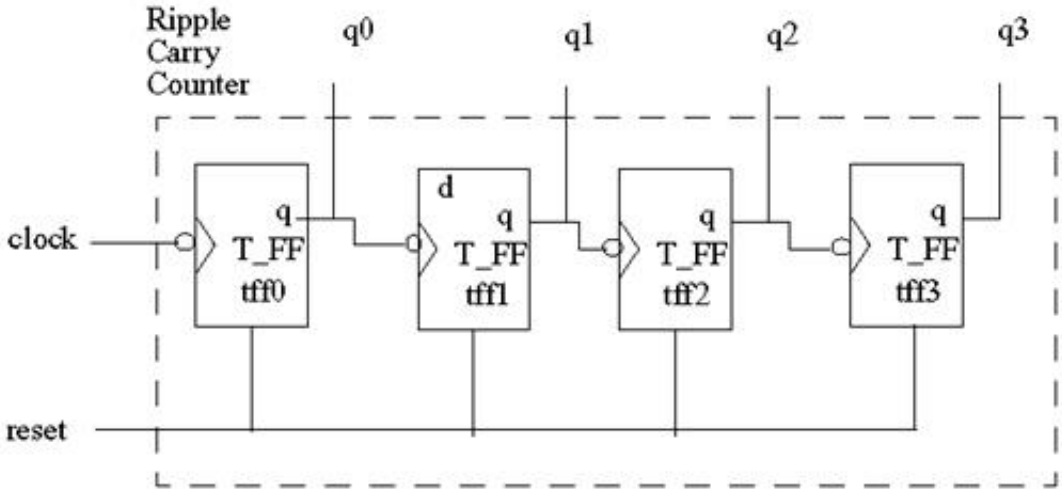

```
`include "T_FF.v"

// 定义行波进位计数器模块 ( the top-level module )
// 模块名: ripple_carry_counter
module ripple_carry_counter(q, clk, reset);

output [3:0] q;      // 输入/输出端口的信号或向量声明
input clk, reset;    // 输入/输出端口的信号或向量声明

// 创建 4 个T触发器实例, 每个T触发器实例有其唯一的实例名
T_FF tff0(q[0],clk,  reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule
```



设计综合（2）

- 综合工具
  - Precision_Synthesis RTL Plus 2012a.10
  - 右图：RTL级电路图





HDL设计方法

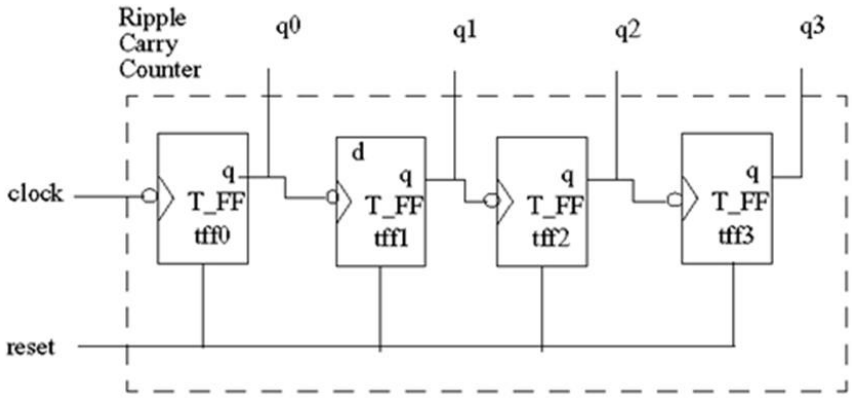
数字系统行为描述

- 在HDL语言中的行为描述通常分为两类——算法和数据流
- 算法 —— 用过程或程序描述数字系统（电路）的行为
  - 不包含任何特定的物理实现，定义 I/O 响应及所需的处理的过程
  - 可用于验证所设计的数字系统是否执行正确的功能
    - 不考虑具体实现
- 数据流描述 —— 说明数据如何在寄存器间移动

数字系统结构描述

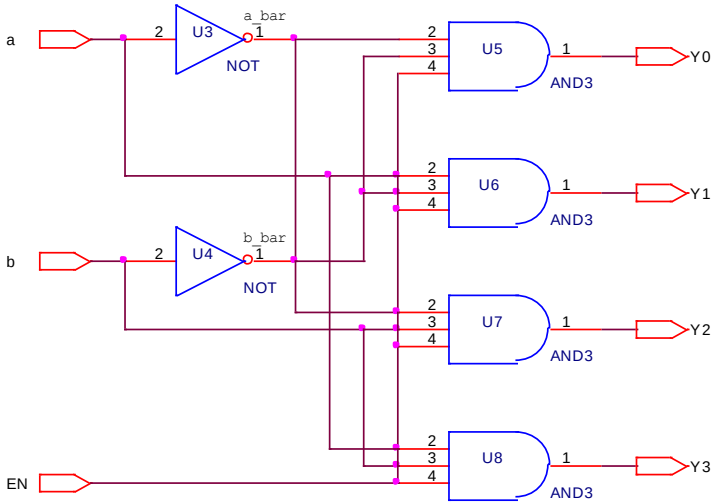
- 用基本单元（primitive）或低层模块（component，元件）的连接来描述系统

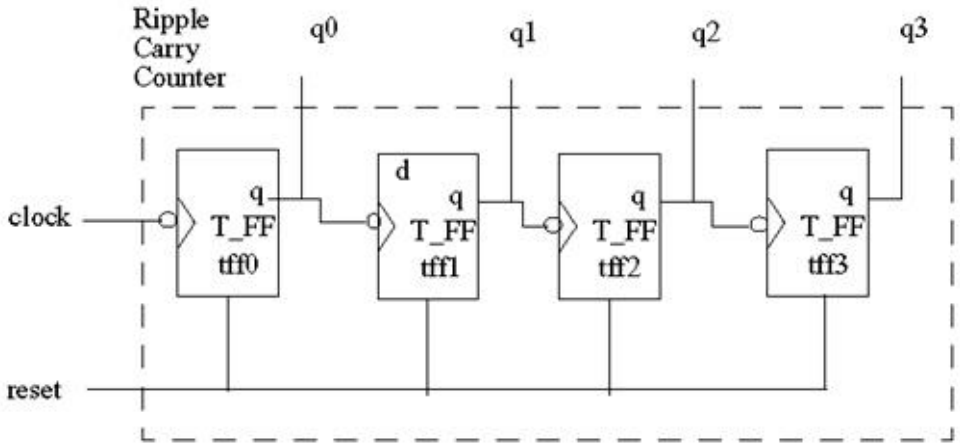
```
`include "T_FF.v"

// 定义行波进位计数器模块 ( the top-level module )
// 模块名: ripple_carry_counter
module ripple_carry_counter(q, clk, reset);

output [3:0] q;      // 输入/输出端口的信号或向量声明
input clk, reset;    // 输入/输出端口的信号或向量声明

// 创建 4 个T触发器实例, 每个T触发器实例有其唯一的实例名
T_FF tff0(q[0],clk,  reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule
```

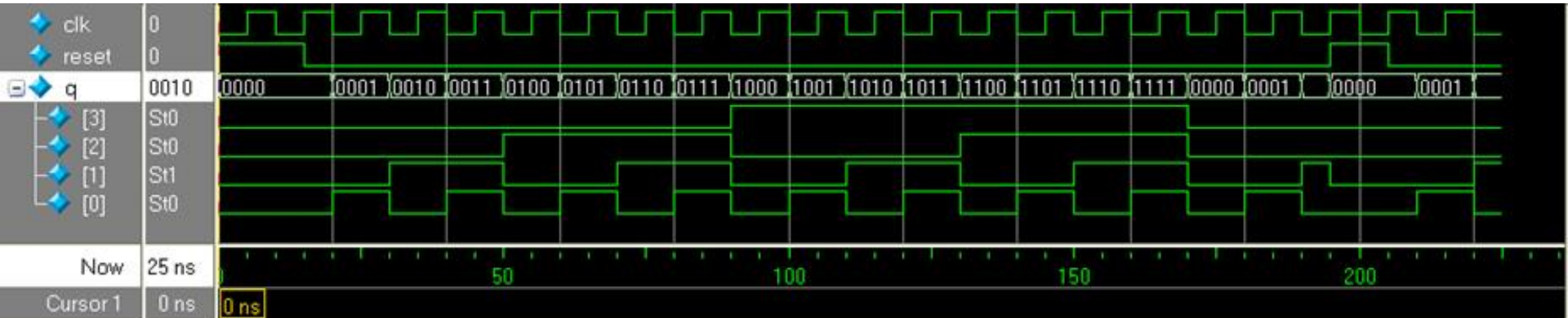


并发执行/顺序执行（1）

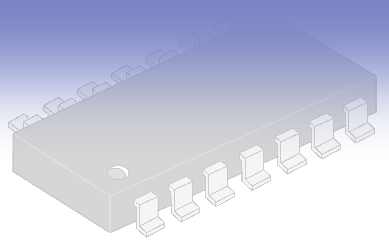
- 数字系统包括
  - 同步过程
    - 上电后，各个子电路模块从 0 时刻起，同时开始进入工作状态
  - 顺序过程 —— 时序电路
    - 对于记忆单元，满足某种条件，状态才能变化







并发执行/顺序执行（2）



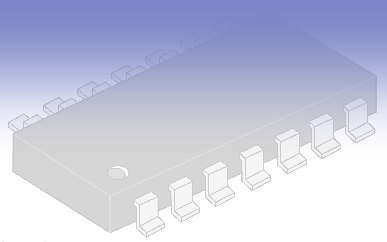
- 并发执行的语句描述电路的同步
- 在Verilog 模块中的并发执行执行语句
  - 连续赋值语句：assign
  - 过程语句：initial、always
  - 门原语、实例引用语句
- 并发执行的语句的执行顺序与其在模块中的位置无关

- 例、逻辑功能定义

```
 1 // wire py;
 2 assign py = a or b;
 3
 4 nand #2 u1( y, a, b);
 5
 6 // reg q;
 7 always @( posedge clk or posedge clr )
 8 begin
 9         if ( !clr )
10                 q <= 0;
11         else if ( en )
12                 q <= d;
13 end
```

- 模块中使用 3 个并发语句
  - —— 混合设计描述方式
  - assign
    - 数据流描述方式
    - 通常描述组合逻辑电路
  - 门实例
    - 实例名称为：u1，输出延时为 2 个时间单位
  - always
    - 行为描述方式
    - 描述时序逻辑电路/组合逻辑电路
    - 这里：
      - 描述了一个带有异步清除端 clr 的 D 触发器

并发执行/顺序执行（3）

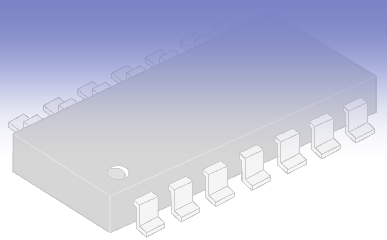


- 过程语句定义一个过程块，描述数字电路的逻辑功能
  - 过程块中的语句是顺序执行语句
  - 描述数字系统中状态变化（或事件传递）顺序

```
`timescale 1ns/100ps

module latch (input d, c, output reg q, q_b );
    always @( c or d )
        if ( c ) begin
            q <= #4 d;
            q_b <= #3 ~d;
        end
endmodule
```

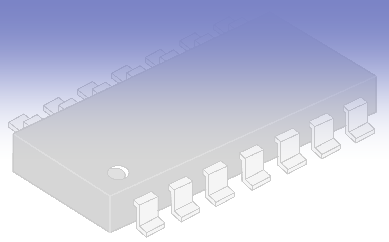
```
module dff( output reg q,
                input clk, reset, d );
    always @( posedge clk ) begin
        if ( reset ) q <= 1'b0;
        else q <= d;
    end
endmodule
```

使用Verilog设计数字电路



- 4 种描述方式
  - 数据流方式（Dataflow ）
    - 使用连续赋值语句：assign
  - 行为和算法（Behvioral or algorithmic）方式
    - 使用过程语句（procedure statement）
      - initial —— 只能用于建模与仿真
      - always —— 用于行为方式描述设计、建模和仿真
  - 结构（Structural）方式
  - 混合设计描述方式
    - —— 上述描述方式的组合
- 描述电路的并发执行/顺序执行
  - 描述数字电路信号传递和处理的物理过程

数据流描述方式



- 基本方法是使用连续幅值语句
- 语法
  - assign [delay] *LHS_net = RHS_expression*
  - 右边表达式的值被赋给一个线网（wire）变量
- 语义
  - 右边表达式中操作数无论何时变化，表达式都重新计算，并在指定延时后，将计算出的变化值赋给左边的线网变量

- 例、2 位比较器

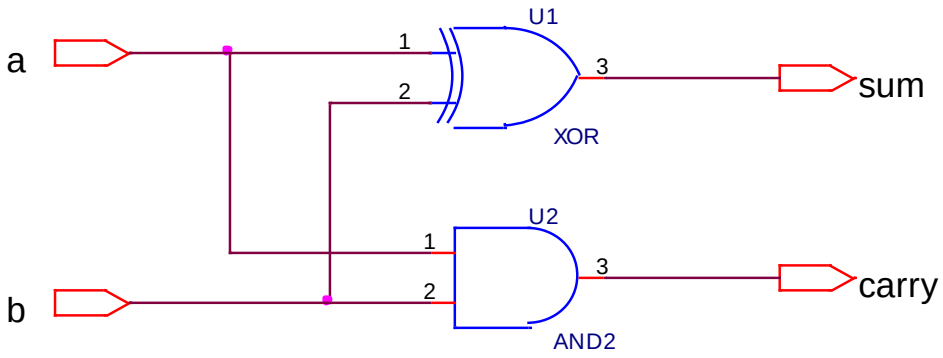
```
1 module comparator_2b (equal, a, b);
2
3         input [1:0] a, b;
4         output equal;
5
6         assign equal = (a == b) ? 1 : 0;
7
8 endmodule
```

数据流描述——简单例子

- 例1、半加器（half adder）电路模块
- 逻辑表达式

$$sum = x \oplus y$$

$$carry = x \bullet y$$




```
1 module half_adder( sum, carry, a, b);
2
3         input a, b;
4         output sum, carry;
5
6         assign #2  sum = a ^ b;
7         assign #5  carry = a & b;
8
9 endmodule
```

1、模块有4个端口 （port），两个输入端口： a 、b；两个输出端口： sum、carry

2、没有定义端口的位数，则，隐含所有端口都是 1 位（bit）

3、没有说明各个端口的数据类型，则，隐含 4 个端口都是线网（wire）数据类型

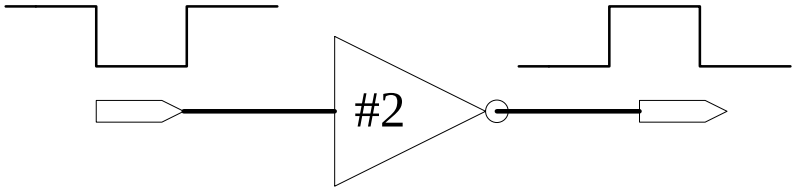
4、使用两条连续赋值语句（assign）语句，描述半加器的数据流（data flow）行为

5、连续赋值语句是并发（concurrently）执行的，它们在模块中出现的顺序无关紧要

6、每条连续赋值语句的执行顺序依赖于发生在变量 a 和 b 上的事件

延时

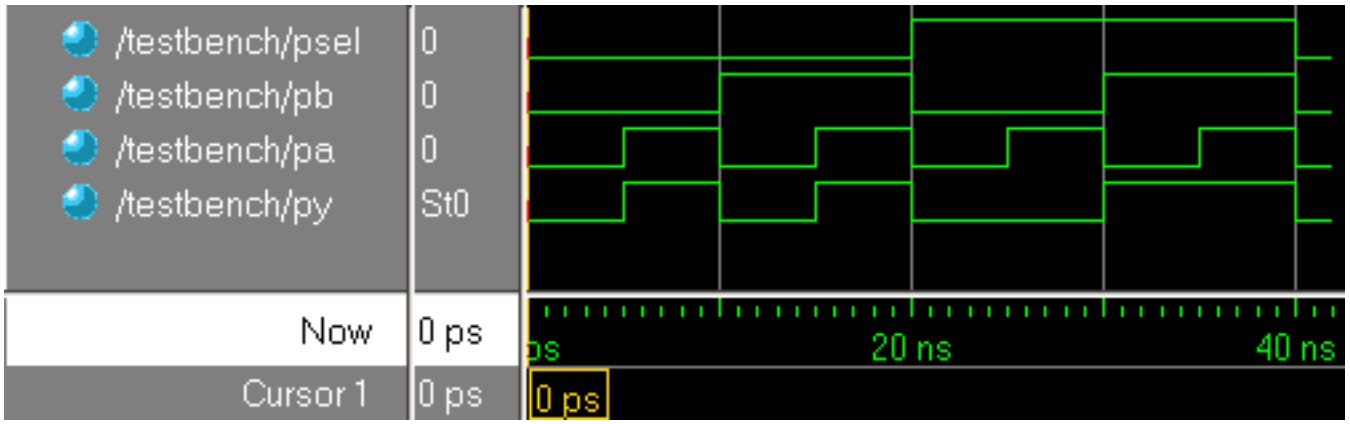
- Verilog HDL 模型中的所有延时都根据时间单位定义
  - 延时定义了右边操作数变化与赋值给左边表达式之间的持续时间
    - 由于需要经过一定的延时，门电路的值（逻辑状态）才能发生变化
  - 如没有定义持续时间，缺省值为 0
- 例、
  - assign #2 sum = a ^ b;
  - 这里，＃2 是指 2 个时间单位



- 利用编译指令定义时间单位的对应的物理时间
- 例、
  - `timescale 1ns/100 ps
  - 定义延时单位为 1 ns(10-9s)，精度为 100(10-12s) ps
  - ＃2：表示 2ns
  - ＃ 2.24：表示 2.2 ns，延时的精度为 100 ps
- 缺省时间单位
  - 没有使用编译指令定义时间单位时，仿真器会指定一个缺省时间单位
- IEEE Verilog 标准中没有指定缺省时间单位

时间单位和精度

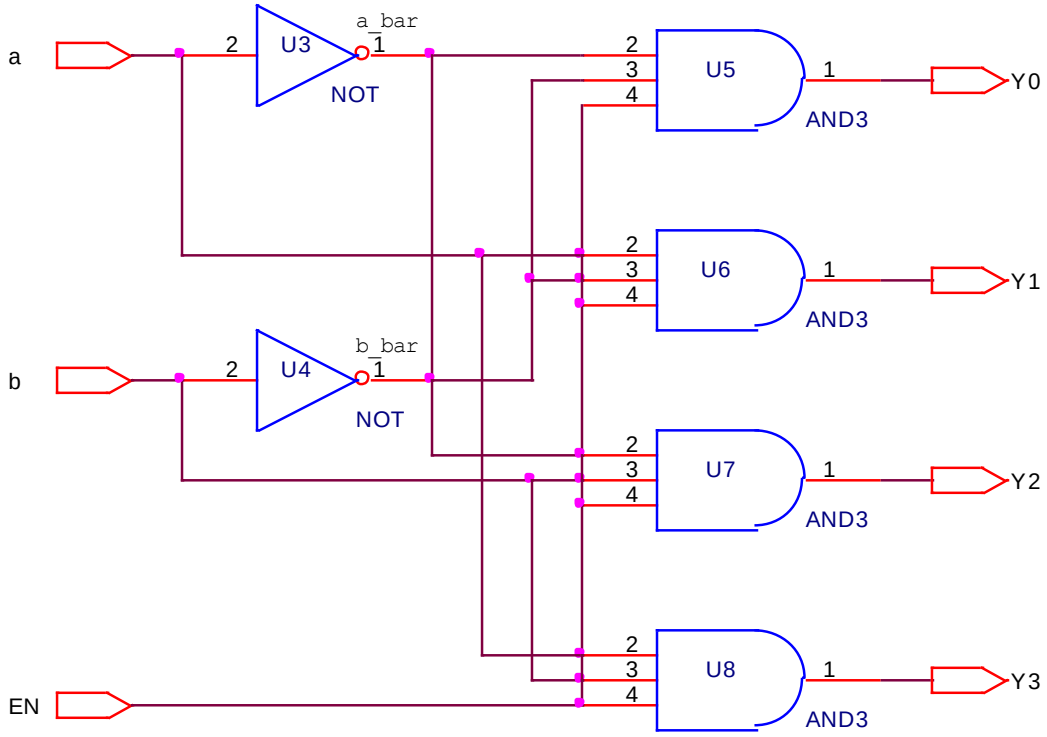
- 例、编译指令定义延时单位和延时精度
  - `timescale 1ns/100 ps
  - 此时：
    - 延时值 —— # 5.22 ➔ 5.2 ns
    - 延时值 —— # 6.17 ➔ 6.2 ns
  - `timescale 1ns/1ns
  - 此时：
    - # 36.2 ➔ 36ns
    - # 51.6 ➔ 52ns
- `timescale 指令影响后面所有模块中的延时值
  - 直至遇到另一个 `timescale 指令或 `resetall指令
- 当设计中多个模块带有自身的`timescale 时
  - 仿真器选择最小延时精度，并且所有延时精度转换为最小延时精度

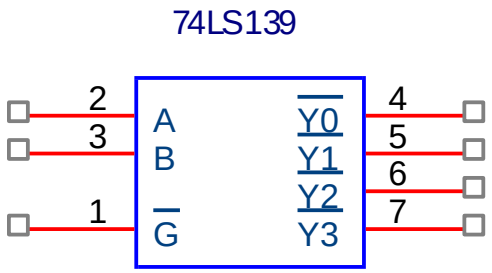


例2、2-4 译码器电路模块

- 真值表和电路原理图

输入			输出			
EN	b	a	Y3	Y2	Y1	Y0
0	x	x	0	0	0	0
1	0	0	0	0	0	**1**
1	0	1	0	0	**1**	0
1	1	0	0	**1**	0	0
1	1	1	**1**	0	0	0

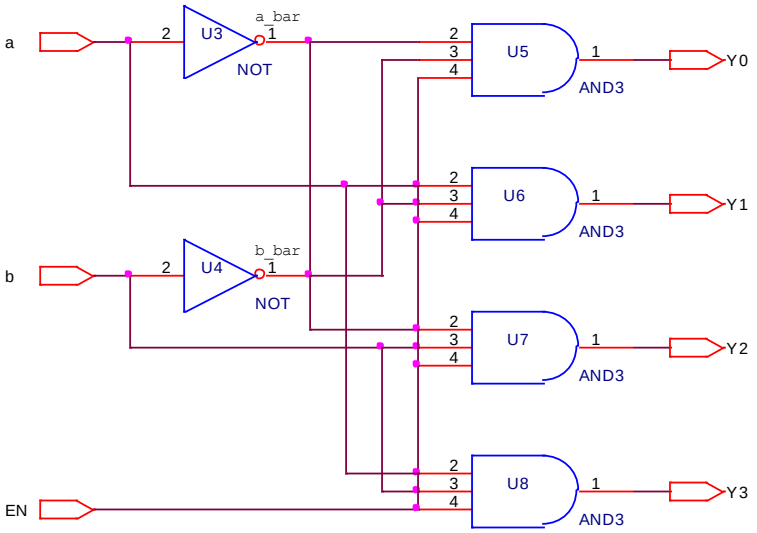




例2、2-4 译码器 Verilog 模块

```
`timescale 1ns /100ps
module decoder_2x4( output [3:0] y,
                    input a, b, en );
    wire a_bar, b_bar;

    assign #1 a_bar = ~a,
              #1 b_bar = ~b,
              #2 y[0]  = a_bar & b_bar & en,
              #2 y[1]  = a & b_bar & en,
              #2 y[2]  = a_bar & b & en,
              #2 y[3]  = a & b & en;
endmodule
```

输入			输出			
EN	b	a	Y3	Y2	Y1	Y0
0	x	x	0	0	0	0
1	0	0	0	0	0	**1**
1	0	1	0	0	**1**	0
1	1	0	0	**1**	0	0
1	1	1	**1**	0	0	0

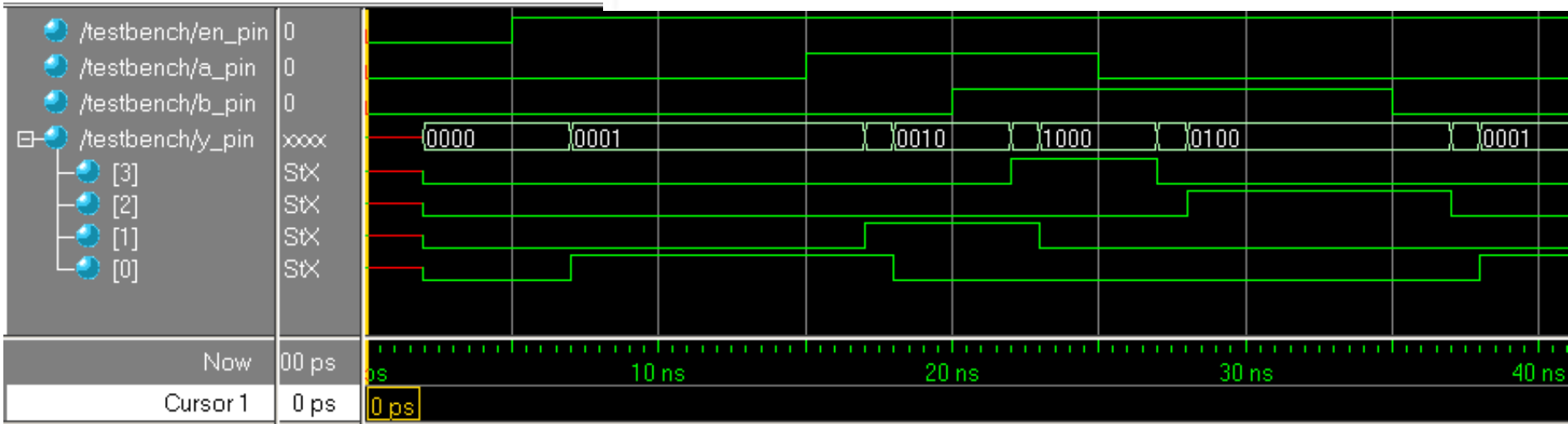


- 编译指令
  - 将模块中的时间单位定义为1ns，精度 100ps
- 端口
  - 模块有 3 个输入，1个 4 位宽度输出
- 定义了 2 个线网变量
- 6 个连续赋值语句

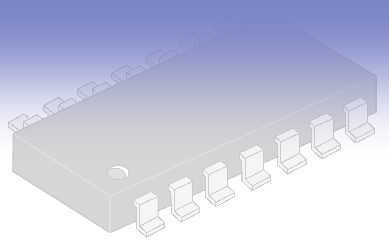
2-4 译码器电路模块中连续幅值语句的执行

- 连续赋值语句
  - 并发执行，执行顺序与它们在模块中出现的顺序无关
  - 执行顺序依赖于语句中的变量上发生的事件

```
`timescale 1ns /100ps
module decoder_2x4( output [3:0] y,
                    input a, b, en );
    wire a_bar, b_bar;

    assign #1 a_bar = ~a,
           #1 b_bar = ~b,
           #2 y[0]  = a_bar & b_bar & en,
           #2 y[1]  = a & b_bar & en,
           #2 y[2]  = a_bar & b & en,
           #2 y[3]  = a & b & en;
endmodule
```



行为描述方式



- 使用过程语句描述

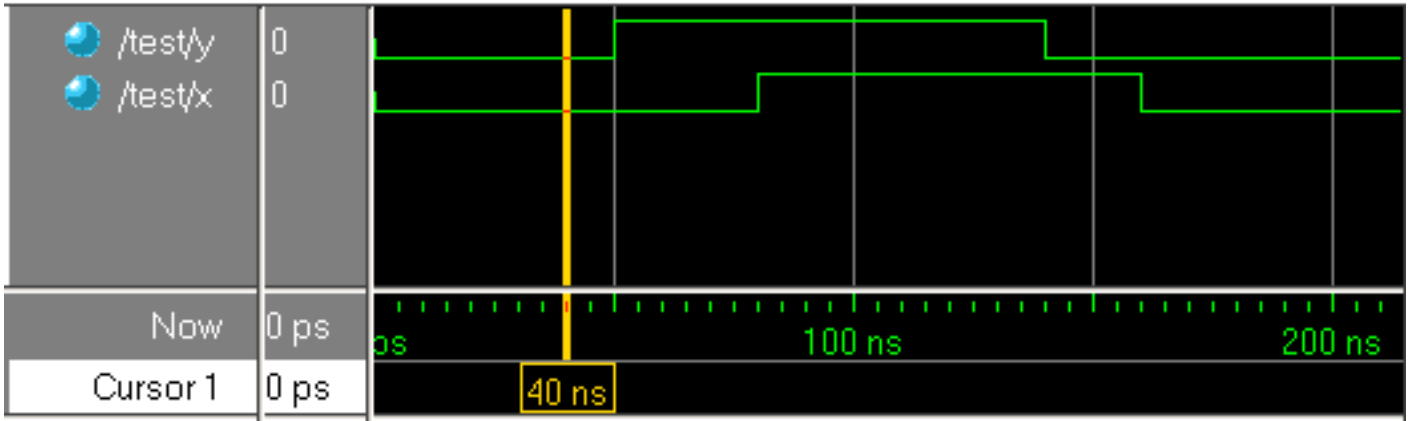
1. initial 语句
   - 此语句只执行一次
2. always 语句
   - 此语句总是循环执行

- 要点
  - 只有寄存器类型变量能够在这两种语句中被幅值
  - 寄存器类型变量在被赋新值之前保持原有值不变
  - 所有 initial 和 always 语句在 0 时刻并发执行

initial 语句

- 例4、
  - initial 语句包含一个顺序过程
    begin
    …
    end
  - 顺序过程从 0 ns 开始执行
    1. 50ns 后，y = 1
    2. 再过 30ns 后，x = 1
    3. 再过 60ns 后，y = 0
    4. 再过 20ns 后，x = 0
  - 顺序过程中所有语句执行完后，initial 语句永远挂起
  - 此模块产生如下波形

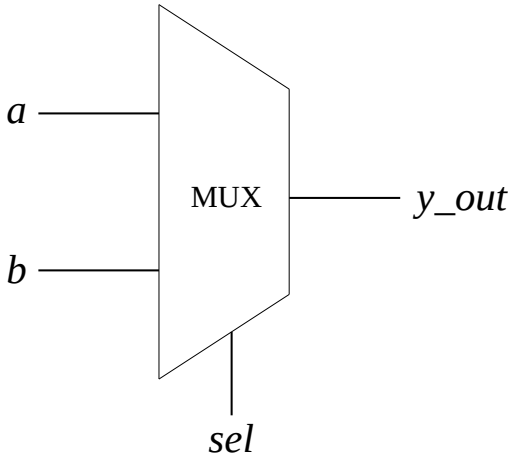
```
1 `timescale 10ns / 100ps
2
3 module test ( y, x );
4
5         output x, y;
6         reg    x, y;
7
8         initial
9                 begin
10                        y = 0;
11                        x = 0;
12                        y = #5 1;
13                        x = #3 1;
14                        y = #6 0;
15                        x = #2 0;
16                end
17 endmodule
```



例、行为描述

二选一多路器(two-channel multiplexer)

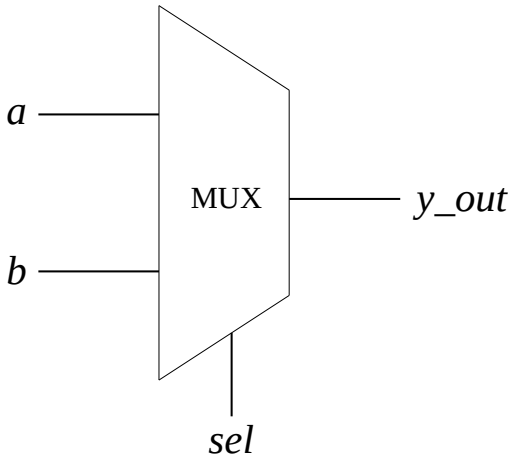
- 只要信号 a 或 b 或 sel 发生变化，输出就发生变化
- 如果 sel 为 0 则选择a 输出，否则选择 b 输出
- 特点
  - 在行为级模型中，逻辑功能描述采用高级语言结构
  - 如 while、if 、case 等




```
module mux_2x1 ( output reg y_out,
                 input sel, a, b );
    always @ (*) begin
        if ( !sel ) y_out = a;
        else y_out = b;
    end
endmodule
```

例、二选一多路器

- 行为描述 —— 算法描述



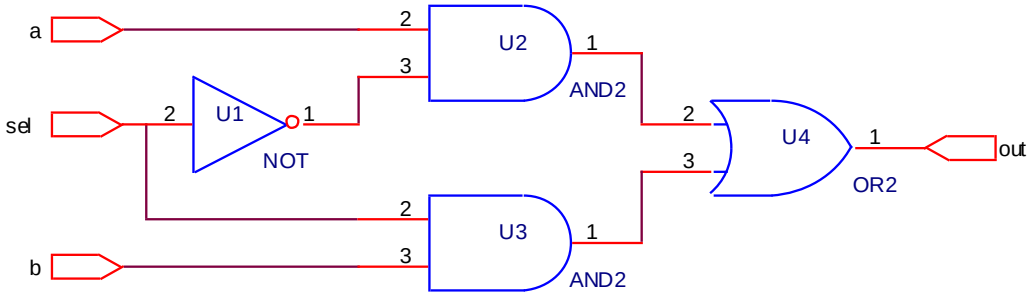

```
module mux_2x1 ( output reg y_out,
                 input sel, a, b );
    always @ (*) begin
        if ( !sel ) y_out = a;
        else y_out = b;
    end
endmodule
```

sel	y_out
0	a
1	b

- 模块 mux_2x1 有 3 个输入和 1 个输出
- 由于 y_out 要在 always 中被幅值，所以说明成 reg 类型
  - reg：是寄存器类型的一种
- 在 always 语句中，紧跟 @ 后的表达式是事件控制
- 事件控制意味着：
  只要 a、b、 sel 中任一个的值发生变化， always 中的顺序过程语句就执行

Verilog 的结构化描述形式

- 结构描述（structural description）使用
  - ① 内置门原语（门级）
  - ② 开关级原语（晶体管级）
  - ③ 用户定义原语UDP（门级）
  - ④ 模块实例（创建层次结构）
- 使用线网互相连接
- 例、二选一多路器的结构描述
  - 使用内置原语



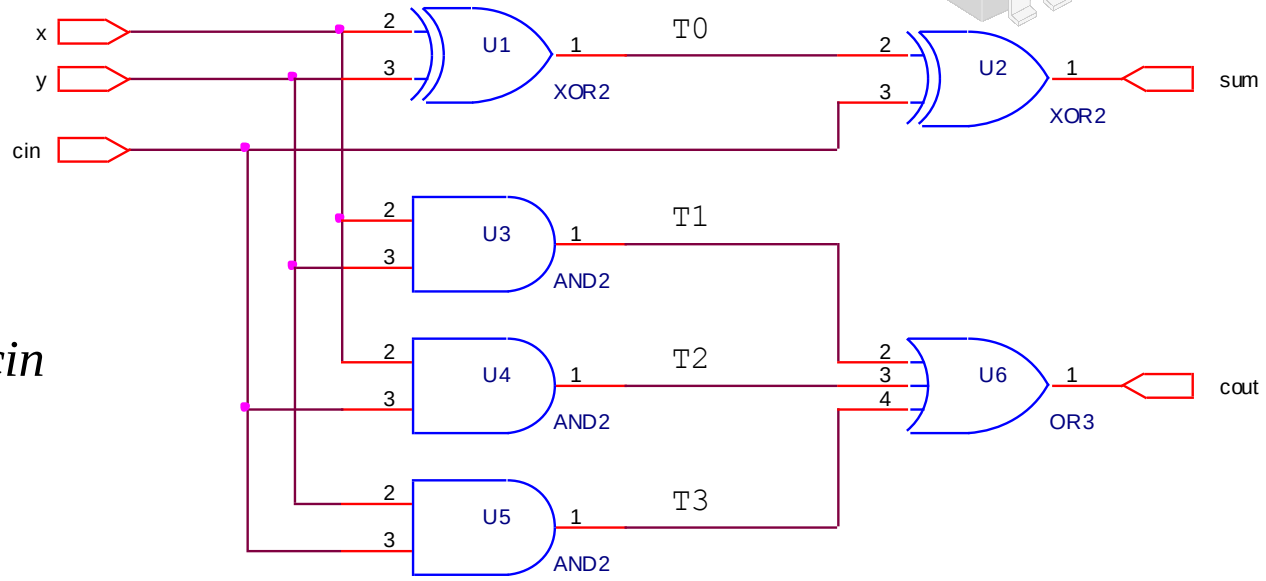

```
1 module mux_2x1 (y_out, a, b, sel);
2          input  a, b, sel;
3          output y_sel;
4
5          not       u1(n_sel, sel);              // not 内置门级原语
6          and  #1   u2(a_sel, a);                // and 内置门级原语
7          and  #1   u3(b_sel, b);
8          or   #2   u4(y_out, a_sel, b_sel);     // or  内置门级原语
9
10 endmodule
```

例6、全加器（full adder） 的结构描述

- 1位全加器逻辑表达式和电路原理图

$$sum = x \oplus y \oplus cin$$

$$cout = x \cdot y + x \cdot cin + y \cdot cin$$

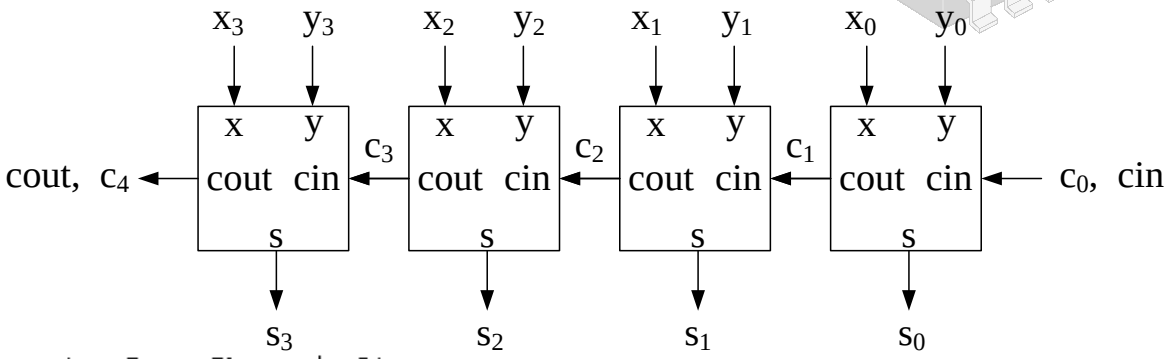



```
1 module full_adder ( cout, sum, x, y, cin );
2         input  x, y, cin;
3         output cout, sum;
4         wire t0, t1, t2, t3;
5
6         xor      u1( t0, x, y),
7                  u2( sum, t0, cin);
8         and      u3(t1, x, y),
9                  u4(t2, x, cin),
10                 u5(t3, y, cin);
11        or       u6(cout, t1, t2, t3);
12
13 endmodule
```

- 模块包含内置门：xor, and, or 的实例语句
- 门实例通过线网变量 t0, t1, t2, t3 相连接
- 门实例可以任意顺序出现
- 在门实例中：第 1 个是输出，其余是输入

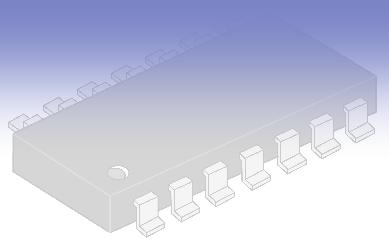
例7、4位全加器的结构描述

- 可以用 4 个 1 位全加器模块描述
- 4 位全加器结构框图




```
 1
 2 module fulladder4b (fsum, cout4, fa, fb, cin0);
 3
 4         parameter SIZE = 4;
 5
 6         input [SIZE - 1: 0] fa, fb;
 7
 8         output [SIZE - 1: 0] fsum;
 9
10         input cin0;
11
12         output cout4;
13
14         wire [SIZE - 1: 1] c;
15
16         full_adder
17
18                 fa0( .cout(c[1]),   .sum(fsum[0]), .x(fa[0]), .y(fb[0]), .cin(cin0) ),
19                 fa1( .cout(c[2]),   .sum(fsum[1]), .x(fa[1]), .y(fb[1]), .cin(c[1]) ),
20                 fa2( .cout(c[3]),   .sum(fsum[2]), .x(fa[2]), .y(fb[2]), .cin(c[2]) ),
21                 fa3( .cout(cout4), .sum(fsum[3]), .x(fa[3]), .y(fb[3]), .cin(c[3]) ),
22
23 endmodule
24
```

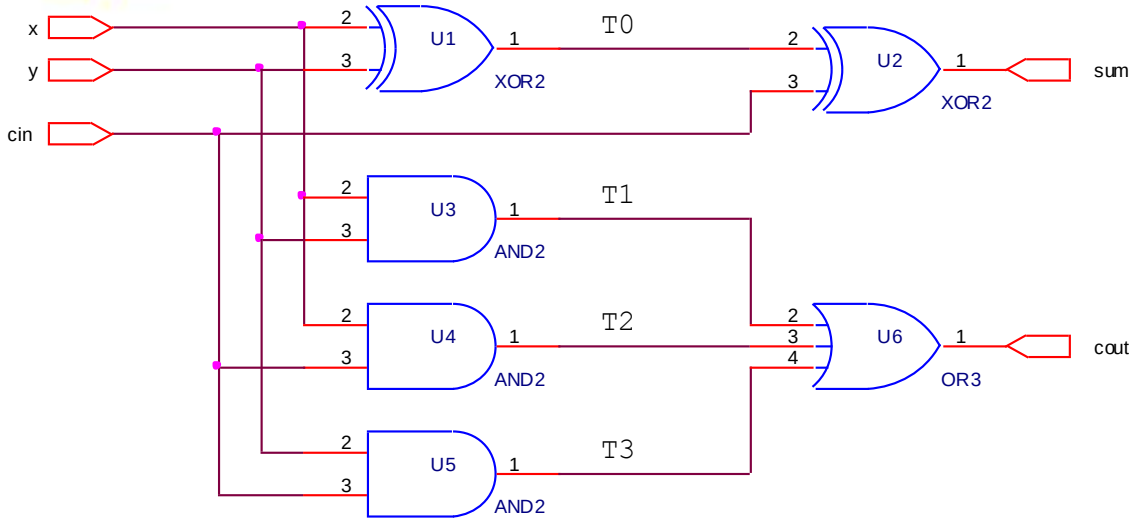
混合设计描述方式



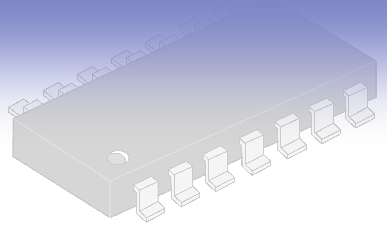
- 在 Verilog 模块设计中，行为描述和结构描述可以自由混合
- 一个模块描述可以包含
  - initial 语句
  - always 语句
  - 连续幅值语句
  - 实例化的门
  - 实例化的（子）模块语句
- 来自 initial 和 always 的值能够驱动门
  - 只有寄存器类型的数据可以在这两种语句中被幅值
- 来自门或连续幅值语句的值能够用于触发 initial 和 always 语句
  - 连续幅值语句只能驱动线网

例8、1 位全加器的混合描述

```
module fadder_1b ( output reg cout,
                   output sum,
                   input a, b, cin );
    reg t1, t2, t3;
    wire t0;

    xor x1( t0, a, b); // 门原语

    // 过程语句块
    always @( a or b or cin ) begin
        t1 = a & b;
        t2 = a & cin;
        t3 = b & cin;
        cout = t1 | t2 | t3;
    end
    // 连续赋值语句
    assign sum = t0 ^  cin;
endmodule
```



Verilog 模块的仿真



- 可以使用 Verilog 描述用于模块测试的信号
- 对激励和控制进行验证
  - 使用初始化语句产生
- 建立测试平台（ testbench ）
  - 设计一个描述测试信号的变化和测试过程的模块
- 利用测试平台对被测电路模块进行动态的、全面的测试

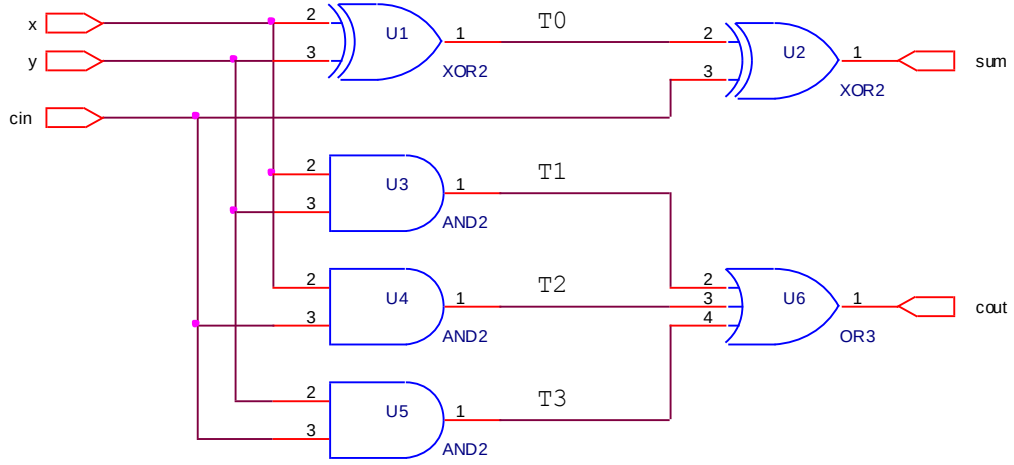
例、建立1 位全加器的测试平台（ testbench ）

- 测试模块的设计

```
1 `timescale 1 ns / 100 ps
2
3 `include "fa_1b.v"
4
5 module testbench;
6
7         reg pa, pb, pcin;
8         wire pcout, psum;
9
10        reg [2 : 0] pval;
11
12        fa_1b m_fa( .cout(pcout), .sum(psum), .x(pa), .y(pb), .cin(pcin) );
13
14        initial
15        begin
16                for ( pval = 0; pval < 8; pval = pval + 1 )
17                begin
18                        { pcin, pb, pa } = pval;
19
20                        #5 $display ( "pa, pb, pcin = %b%b%b", pa, pb, pcin,
21                                      " --- pcout, psum = %b%b", pcout, psum );
22                end
23        end
24 endmodule
```

```
1 module fa_1b ( cout, sum, x, y, cin );
2
3         input  x, y, cin;
4         output cout, sum;
5
6         assign {cout, sum } = x + y + cin;
7
8 endmodule
```

1 位全加器的测试结果



pa, pb, pcin = 000 --- pcout, psum = 00

pa, pb, pcin = 100 --- pcout, psum = 01

pa, pb, pcin = 010 --- pcout, psum = 01

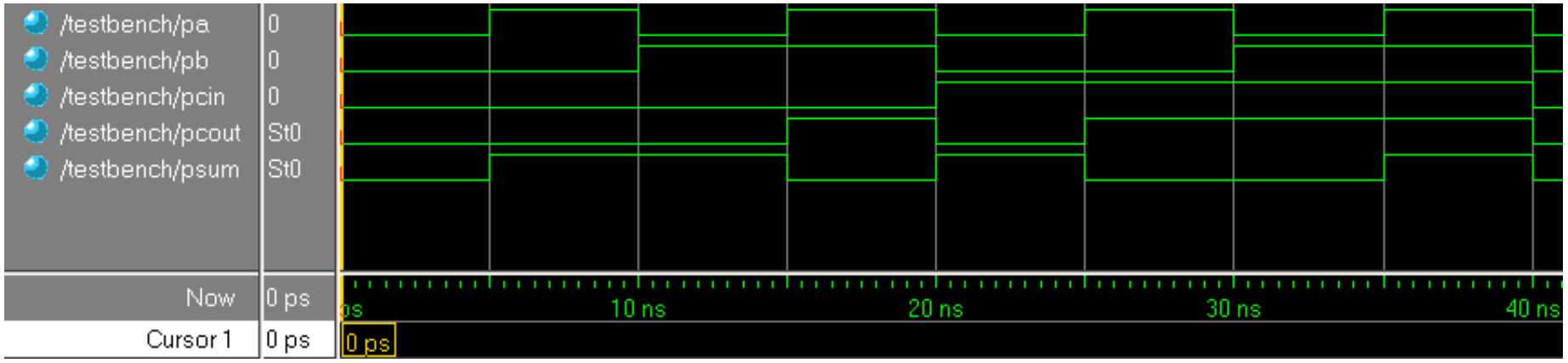
pa, pb, pcin = 110 --- pcout, psum = 10

pa, pb, pcin = 001 --- pcout, psum = 01

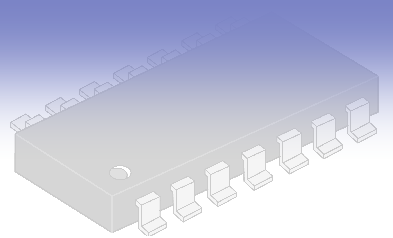
pa, pb, pcin = 101 --- pcout, psum = 10

pa, pb, pcin = 011 --- pcout, psum = 10

pa, pb, pcin = 111 --- pcout, psum = 11



例、二选一多路器的测试

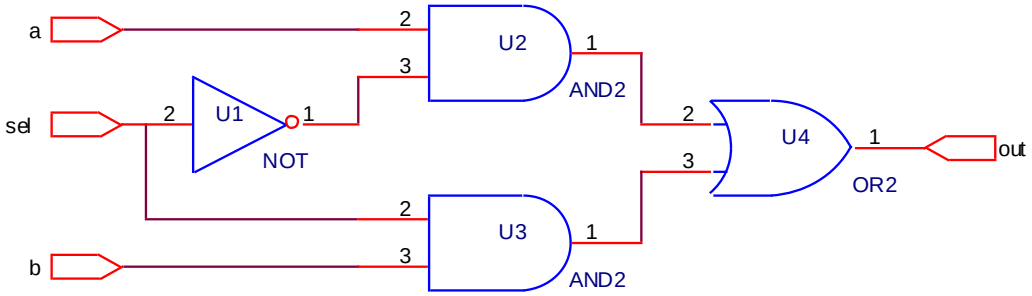


- 建立二选一多路器的测试平台

```
1 `timescale 1 ns/100 ps
2
3 `include "mux_2m1.v"
4
5 module testbench;
6         reg pa, pb, psel;
7         wire py;
8
9         reg [2 : 0] pval;
10
11        initial
12        begin
13                for ( pval = 0; pval < 8; pval = pval + 1 )
14                begin
15                        { psel, pb, pa } = pval;
16
17                        #5 $display ( "psel, pb, pa = %b%b%b", psel, pb, pa,
18                                      " --- py = %b", py );
19                end
20        end
21
22        mux_2m1  m_mux( .y_out(py), .a(pa), .b(pb), .sel( psel));
23 endmodule
```

```
1 module mux_2m1 (y_out, a, b, sel);
2         input  a, b, sel;
3         output y_out;
4
5         reg  y_out;
6
7         always @ ( sel or a or b)
8                 if ( !sel )
9                         y_out = a;
10                else
11                        y_out = b;
12
13 endmodule
```

二选一多路器的测试结果



psel, pb, pa = 000 --- py = 0

psel, pb, pa = 001 --- py = 1

psel, pb, pa = 010 --- py = 0

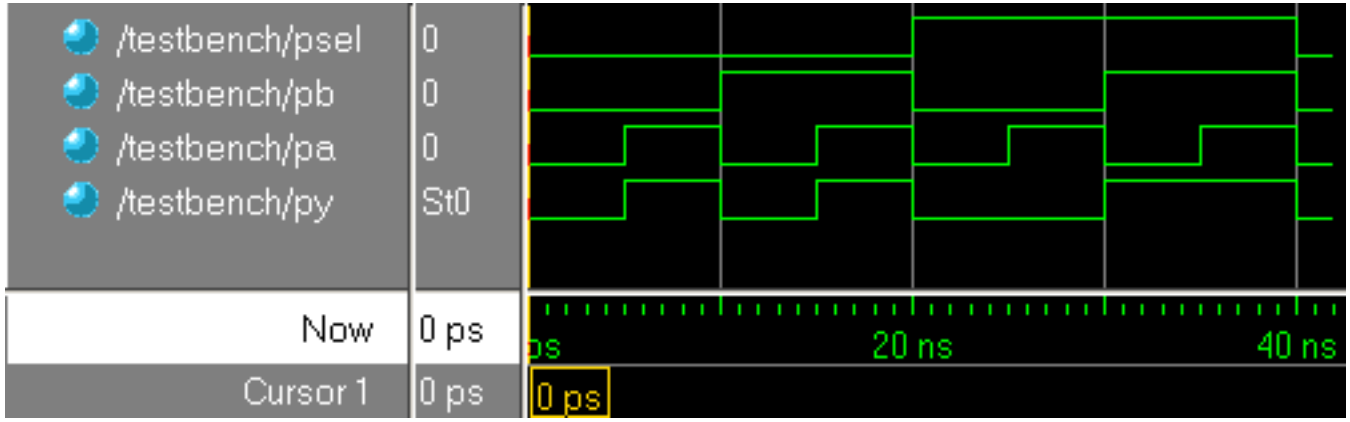
psel, pb, pa = 011 --- py = 1

psel, pb, pa = 100 --- py = 0

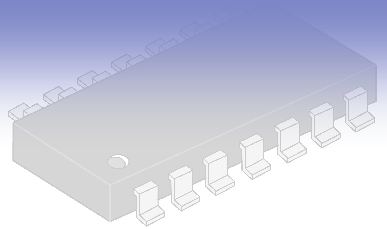
psel, pb, pa = 101 --- py = 0

psel, pb, pa = 110 --- py = 1

psel, pb, pa = 111 --- py = 1



小结



- 模块定义
- 端口
- 模块引用
- 层次命名
- 使用Verilog进行数字电路设计