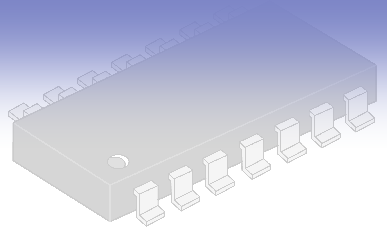
第 8 章

用户定义原语(UDP)

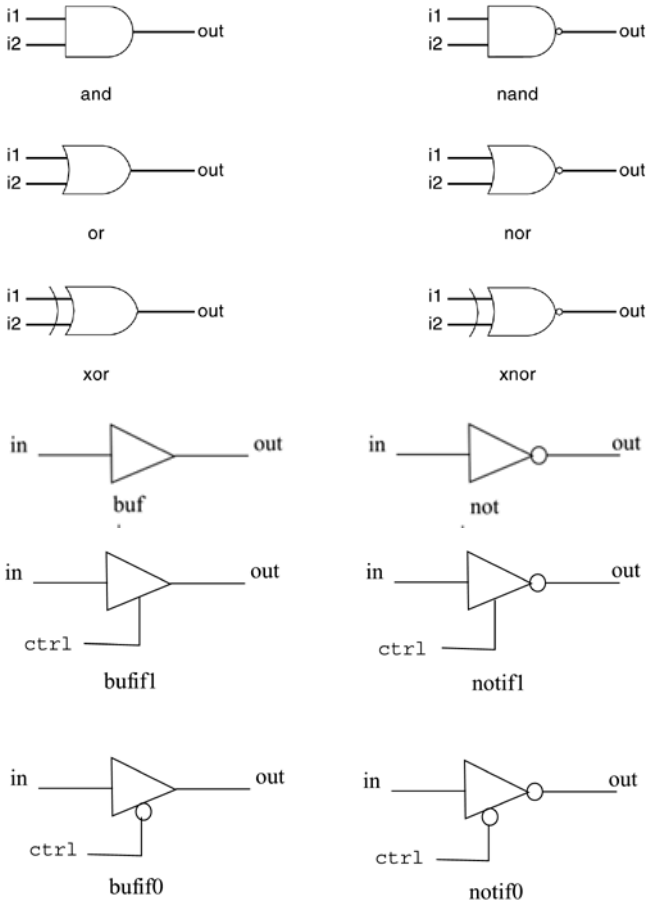
内容



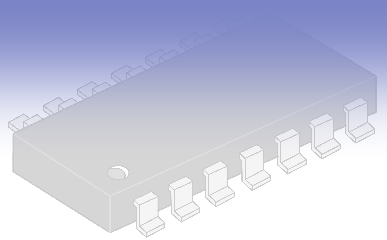
- UDP基础
- 组合逻辑的UDP
- 时序逻辑的UDP
- UDP表中的缩写符号
- UDP设计指南

Verilog 原语

- Verilog 提供的内置原语
  - 门原语
    - **and、nand、or、nor、xor、xnor**
  - 缓冲器/非门
    - **buf（缓冲器），not （非门）**
  - 三态门
    - **bufif1，bufif0，notif1， notif0**
- 内置原语可以综合

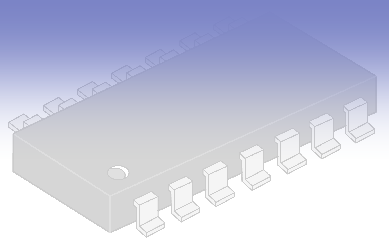


用户定义原语（UDP）基础



- Verilog 允许用户定义自己的原语
- UDP —— User-Defined Primitives
- UDP的类型有两种
  - 表示组合逻辑的UDP
    - 输出由输入信号的组合逻辑确定
  - 表示时序逻辑的UDP
    - 输出由当前输入信号和内部状态确定
  - UDP**不能综合**，只可用于仿真

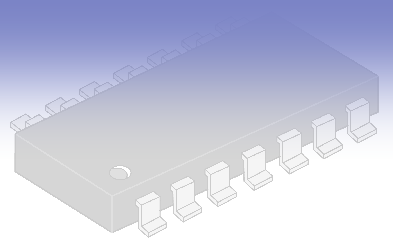
UDP定义的组成



```
// UDP 名（标识符）和端口列表
primitive <udp_name> ( <输出端口名>, // 只许一个输出端口
                       <输入端口名> );

// 端口声明
output <输出端口名>;
input <输入端口名>;
reg <输出端口明>; // 可选，只用于表示时序逻辑的 UDP

// UDP 初始化，可选，只用于表示时序逻辑的 UDP
initial <输出端口名> = <初始值>;

// UDP 状态表
table
    <表项>
endtable

// UDP 定义结束
endprimitive
```

- 主要定义规则
  - 只有一个1 bit 输出端，端口列表中的第一个
  - 如果定义的是表示时序逻辑的原语，输出端口必须声明为 reg 类型
  - 使用 input 声明输入端口，不支持 inout 端口
  - 时序逻辑 UDP 的输出可以用 initial 初始化
  - 状态表中可包含的值为 0、1和 x，不能有 z
  - UDP与模块同级

组合逻辑的UDP



- 状态表中每一行的语法

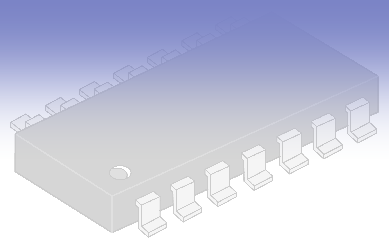
```
<input1> <input2> ..... <inputN> : <output>;
```

- 例、自定义与门 —— udp_and

```
primitive udp_and(out, a, b);

output out; // 组合逻辑的输出端不能声明成 reg 类型
input a, b; // 输入端口声明

// 定义状态表
table
    // a   b   :   out;
       0   0   :   0;
       0   1   :   0;
       1   0   :   0;
       1   1   :   1;
endtable

endprimitive
```

所有输入组合必须在状态表中列出，否则，在状态表中找不到对应输入的项，产生输出为 x

例、列出所有输入组合（1）

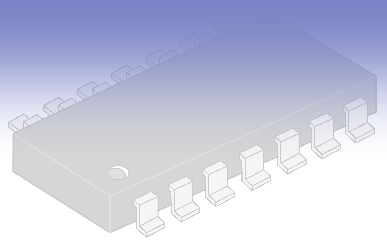


- 状态表中必须列出所有可能的输入组合
- 如果在状态表中找不到对应输入的项，产生输出为 x

```
primitive udp_and(out, a, b);

output out; // 组合逻辑的输出端不能声明成 reg 类型
input a, b; // 输入端口声明

// 定义状态表
table
    // a   b   :   out;
       0   0   :   0;
       0   1   :   0;
       1   0   :   0;
       1   1   :   1;
endtable

endprimitive
```

& 与	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

例、列出所有输入组合（2）

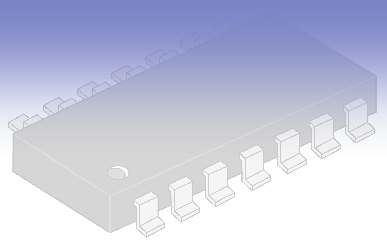


- 状态表中必须列出所有可能的输入组合
- 如果在状态表中找不到对应输入的项，产生输出为 x

```
primitive udp_or(out, a, b);

output out;
input a, b;

table
  //  a    b    :    out;
      0    0    :    0;
      0    1    :    1;
      1    0    :    1;
      1    1    :    1;
      x    1    :    1;
      1    x    :    1;
endtable

endprimitive
```

或	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

无关项的缩写表示

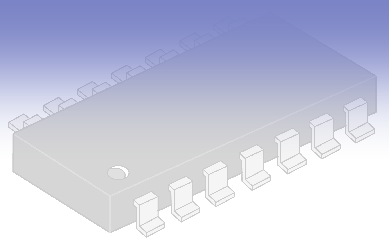


- 无关项
  - 不影响输出值的输入项
- 无关项可用符号“？”表示
  - ？——自动展开为 0、1或 x
- 例

```
primitive udp_or(out, a, b);

output out;
input a, b;

table
  //   a    b    :    out
       0    0    :    0;
       1    ?    :    1; // ? 展开为 0, 1, x
       ?    1    :    1; // ? 展开为 0, 1, x
       0    x    :    x;
       x    0    :    x;
endtable

endprimitive
```

或	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

例、所有可能的输入组合（3）



- 2选1多路选择器

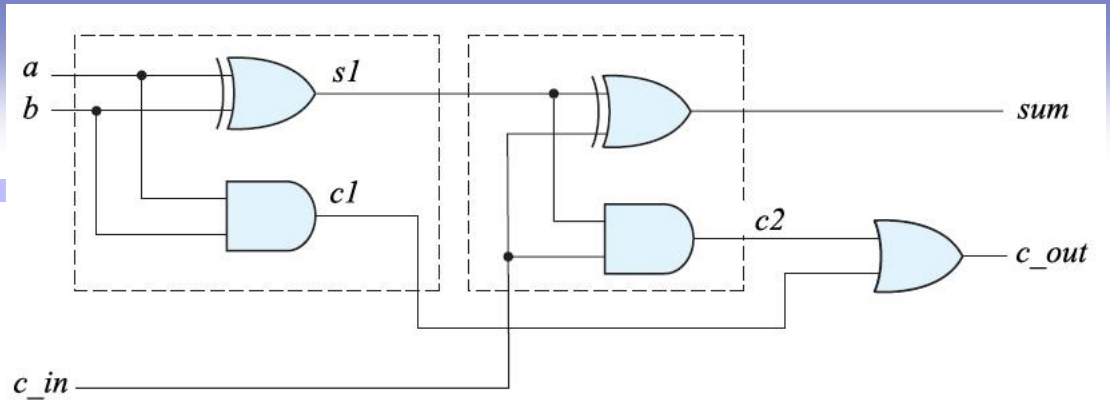
```
primitive mux2x1_udp(output f, input s, I0, I1 );

// 定义 UDP 状态表
table
// input are in the same order as the input list
//   s    I0   I1   :    f    // 标识符用于增加可读性
     0    0    ?    :    0;
     0    1    ?    :    1;
     1    ?    0    :    0;
     1    ?    1    :    1;
     ?    0    0    :    0;
     ?    1    1    :    1;
endtable
endprimitive
```

s	f
0	I0
1	I1

UDP原语的实例引用

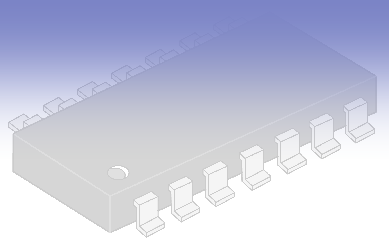
```
primitive udp_and(out, a, b);
output out;
input a, b;
table
    // a   b   :   out;
       0   0   :   0;
       0   1   :   0;
       1   0   :   0;
       1   1   :   1;
endtable
endprimitive
```

```
primitive udp_or(out, a, b);
output out;
input a, b;
table
  //   a   b   :   out
       0   0   :   0;
       1   ?   :   1;
       ?   1   :   1;
       0   x   :   x;
       x   0   :   x;
endtable
endprimitive
```




```
// 1-bit 全加器
module fulladd(sum, c_out, a, b, c_in);
output sum, c_out;
input a, b, c_in;

// 内部线网变量
wire s1, c1, c2;

xor (s1, a, b); // 使用 Verilog 内置原语
udp_and (c1, a, b); // 使用 UDP

xor (sum, s1, c_in); // 使用 Verilog 内置原语
udp_and (c2, s1, c_in); // 使用 UDP

udp_or  (c_out, c2, c1); // 使用 UDP

endmodule
```

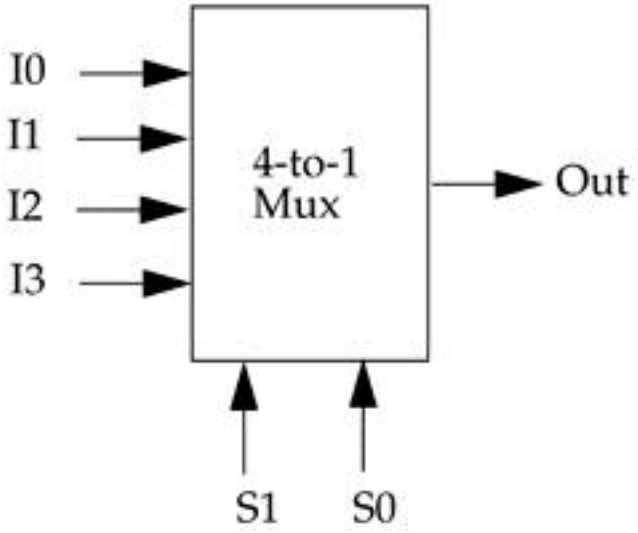
组合逻辑 UDP 设计



- 四选一多路器

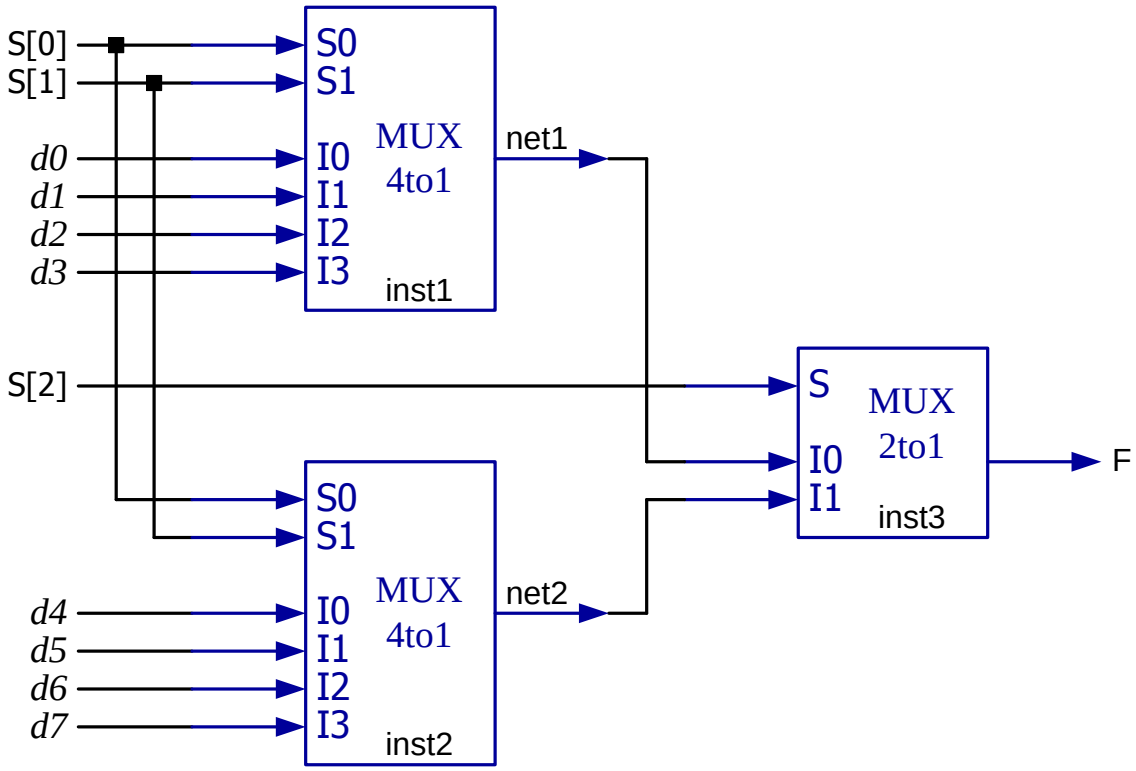
```
primitive mux4x1_udp( output out,
                      input i0, i1, i2, i3, s1, s0);
table
  //  i0   i1   i2   i3,  s1   s0   : out
      1    ?    ?    ?    0    0    : 1 ;
      0    ?    ?    ?    0    0    : 0 ;
      ?    1    ?    ?    0    1    : 1 ;
      ?    0    ?    ?    0    1    : 0 ;
      ?    ?    1    ?    1    0    : 1 ;
      ?    ?    0    ?    1    0    : 0 ;
      ?    ?    ?    1    1    1    : 1 ;
      ?    ?    ?    0    1    1    : 0 ;
      ?    ?    ?    ?    x    ?    : x ;
      ?    ?    ?    ?    ?    x    : x ;
endtable

endprimitive
```

S1	S0	Out
0	0	I0
0	1	I1
1	0	I2
1	1	I3



八选一多路器（1）

- 由 2个4x1多路器和1个2x1多路器
  - 使用用户定义原语
    - mux4x1_udp
    - Mux2x1_udp

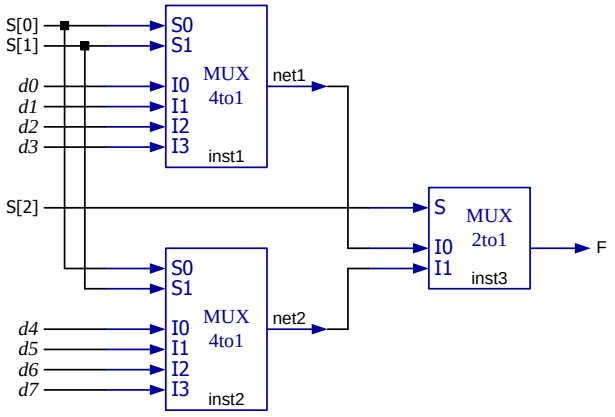


八选一多路器（2）

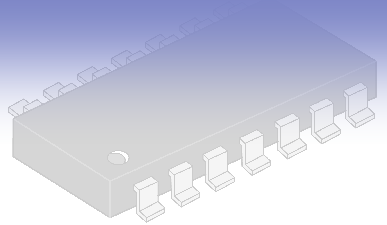
```
primitive mux4x1_udp( output out,
                      input i0, i1, i2, i3, s1, s0);
table
  //  i0   i1   i2   i3, s1   s0   : out
      1    ?    ?    ?    0    0    : 1 ;
      0    ?    ?    ?    0    0    : 0 ;
      ?    1    ?    ?    0    1    : 1 ;
      ?    0    ?    ?    0    1    : 0 ;
      ?    ?    1    ?    1    0    : 1 ;
      ?    ?    0    ?    1    0    : 0 ;
      ?    ?    ?    1    1    1    : 1 ;
      ?    ?    ?    0    1    1    : 0 ;
      ?    ?    ?    ?    x    ?    : x ;
      ?    ?    ?    ?    ?    x    : x ;
endtable

endprimitive
```

```
primitive mux2x1_udp(output f, input s, I0, I1 );

// 定义 UDP 状态表
table
// input are in the same order as the input list
//   s    I0   I1   :    f    // 标识符用于增加可读性
     0    0    ?    :    0;
     0    1    ?    :    1;
     1    ?    0    :    0;
     1    ?    1    :    1;
     ?    0    0    :    0;
     ?    1    1    :    1;
endtable
endprimitive
```

```
// Using two 4x1 multiplexer UDPs and one 2x1 multiplexer UDP
`include "mux4x1_udp.v"
`include "mux2x1_udp.v"
module mux8x1(output F, input [2:0] s, input [7:0] d );

    wire net1, net2;

    // 引用 UDP 实例
    mux4x1_udp inst1( net1, d[0], d[1], d[2], d[3], s[1], s[0] );
    mux4x1_udp inst2( net1, d[4], d[5], d[6], d[7], s[1], s[0] );
    mux2x1_udp inst3( F, s[2], net1, net2 );
endmodule
```



时序逻辑的UDP

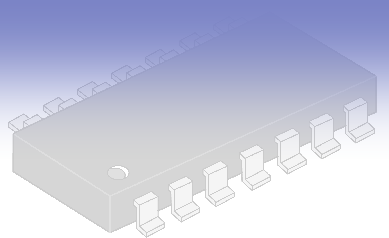


- 特点
  - 输出必须为 reg 类型
  - 输出可用 initial 初始化
  - 状态表格式

```
<input1> <input2> ..... <inputN> : <current_state> : <next_state>;
```

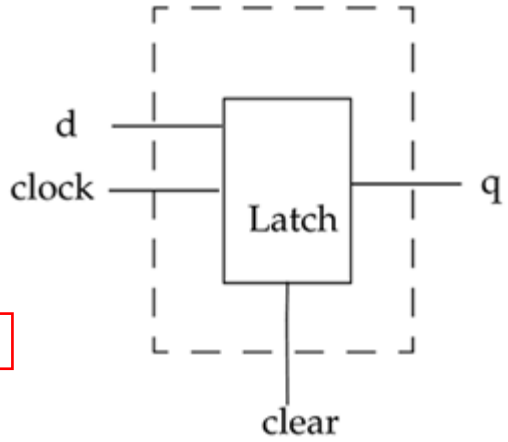
- 两种时序逻辑 UDP
  - 电平敏感
    - 对输入信号的电平敏感
  - **边沿敏感**
    - 对输入信号的边沿敏感

电平敏感的时序逻辑UDP



- 带清零端的电平敏感锁存器
  - 根据输入电平改变状态
- 功能
  - 若 clear 为 1，输出 q 恒为 0
  - 若 clear 为 0
    - 如果 clock 为 1，q = d
    - 如果 clock 为 0，保持 q

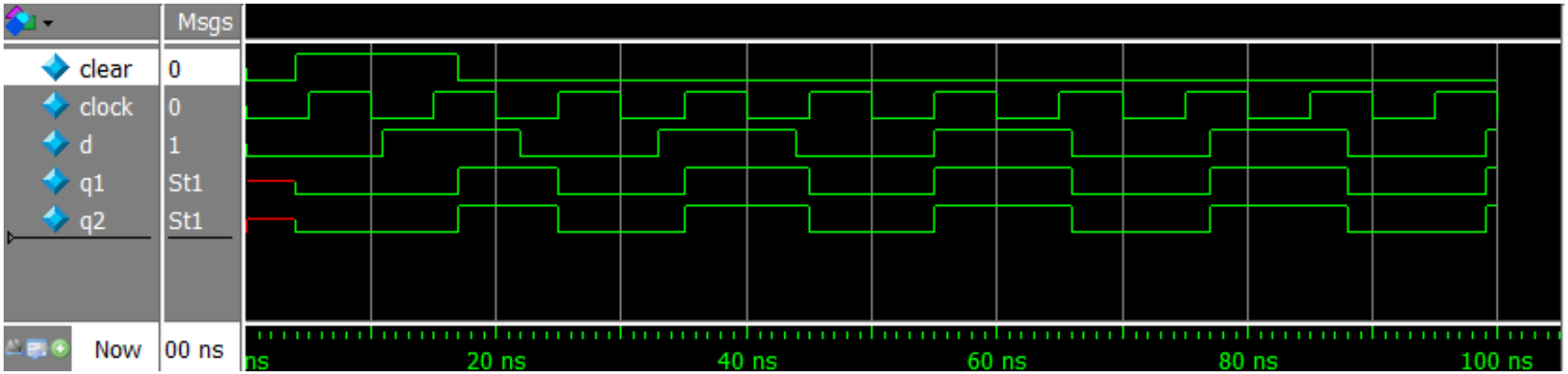
```
primitive latch(q, d, clock, clear);

output q;
reg q;      // 声明 q 为 reg 类型保存内部数据
input d, clock, clear;

// 初始化时序 UDP，只允许有一条 initial 语句
initial
    q = 0; // 初始化输出为 0

//state table
table
  //                    当前状态     下一状态
  // d clock clear :    q      :    q+
     ?    ?      1   :    ?      :    0 ; // 清零
     1    1      0   :    ?      :    1 ; // 将 d 的值锁存在 q = 1
     0    1      0   :    ?      :    0 ; // 将 d 的值锁存在 q = 0
     ?    0      0   :    ?      :    - ; // - 表示 q 保持原状态不变
endtable

endprimitive
```



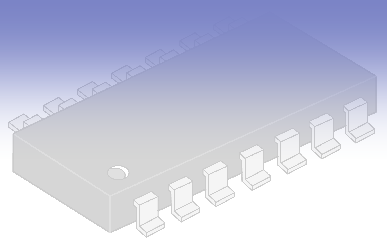
电平敏感的时序逻辑UDP仿真

```
primitive latch(q, d, clock, clear);
output q;
reg q;      // 声明 q 为 reg 类型保存内部数据
input d, clock, clear;

// 初始化时序 UDP，只允许有一条 initial 语句
initial
    q = 0; // 初始化输出为 0

//state table
table
  //                  当前状态     下一状态
  // d clock clear :    q      :    q+
     ?    ?      1   :    ?      :    0 ; // 清零
     1    1      0   :    ?      :    1 ; // 将 d 的值锁存在 q = 1
     0    1      0   :    ?      :    0 ; // 将 d 的值锁存在 q = 0
     ?    0      0   :    ?      :    - ; // - 表示 q 保持原状态不变
endtable
endprimitive
```

```
module latch(output reg q, input d, clock, clear);
      always @(*)
            q = (clear) ? 1'b0 : ((clock) ? d : q);
endmodule
```



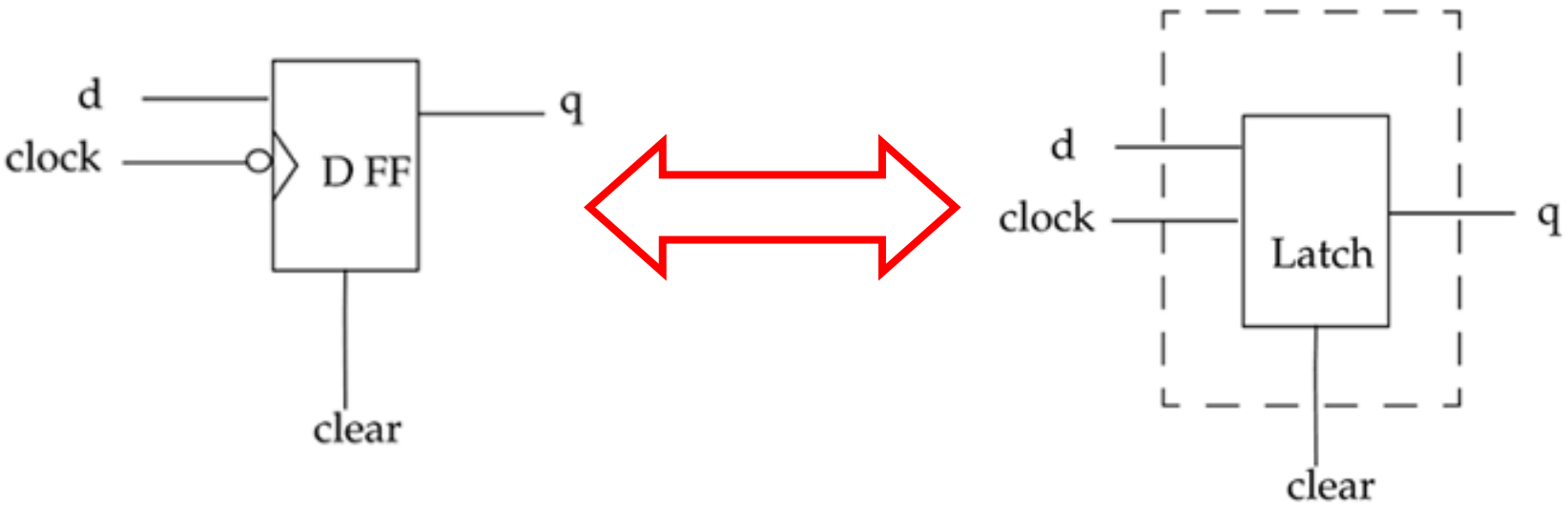
边沿敏感UDP



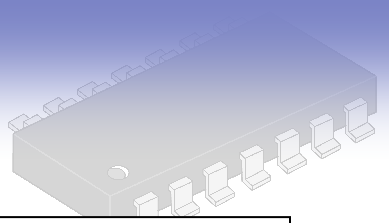
- 描述上升沿/下降沿有效的行为
- 必须考虑时钟的所有变化
  - 0→1，1→0，0→x，1→x，x→0，x→1，0→？，？→0，？→?
  - 如果出现没有明确指定的边沿，输出结果为 x
- 边沿跳变
  - (01) —— 从 0 到 1 正边沿跳变
  - (10) —— 从 1 到 0 负边沿跳变
  - (0x) —— 从 0 到 x 的跳变
  - (1x) —— 从 1 到 x 的跳变
  - (x0) —— 从 x 到 0 的跳变
  - (x1) —— 从 x 到 1 的跳变
  - (0?) —— 从 0 到 0、1或 x 的跳变
  - (?0) —— 从 0、1或 x 到 0 的跳变
  - (??) —— 从 0、1和 x 到 0 、1和 x 任意跳变

边沿敏感时序逻辑UDP

- 根据边沿跳变与/或输入电平改变其状态
- 例、带清零端的下降沿触发的D触发器
- 功能
  - 若 clear = 1，则 q 的输出恒为 0
  - 若 clear = 0
    - 当 clock 从 1 跳变到 0 时，则 q = d，否则，q 保持不变
    - 当 clock 保持稳定时，而 d 改变值，q 不变



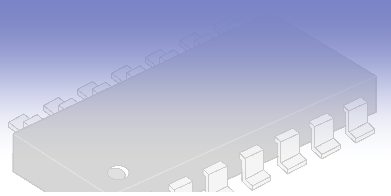
带清零端的下降沿触发的D触发器



```
primitive edge_dff( output reg q = 0,
                    input d, clock, clear );
table
    //  d clock clear  : q : q+
        ?    ?     1   : ? : 0 ; // 当 clear = 1，输出 q = 0
        ?   (??)   1   : ? : 0 ; // 当 clear = 1，输出 q = 0
        ?    ?    (10) : ? : - ; // 忽略 clear 的负跳变
      (??)   ?     0   : ? : - ; // 当 clock 没有变化时，忽略 d 的任何变化
        1   (10)   0   : ? : 1 ; // 在 clock 的负跳沿锁存数据 q = d = 1
        0   (10)   0   : ? : 0 ; // 在 clock 的负跳沿锁存数据 q = d = 0
        ?   (1x)   0   : ? : - ; // clock 变化到不定态时，q 保持不变
        ?   (0?)   0   : ? : - ; // 忽略 clock 的正跳变
        ?   (x1)   0   : ? : - ; // 忽略 clock 正跳沿
endtable
endprimitive
```

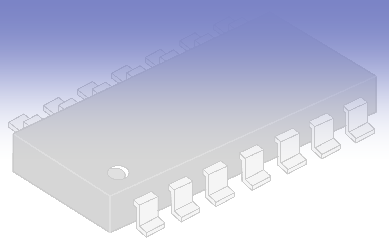
- 边沿跳变
  - (10) —— 从 1 到 0 负边沿跳变
  - (1x) —— 从 1 到 x 的跳变
  - (0?) —— 从 0 到 0、1或 x 的跳变
  - (x1) —— 从 x 到 1 的跳变
  - (??) —— 从 0、1和 x 到 0 、1和 x 任意跳变
- 状态表 —— 每行只能输入一个跳变沿

上升沿有效的D触发器原语



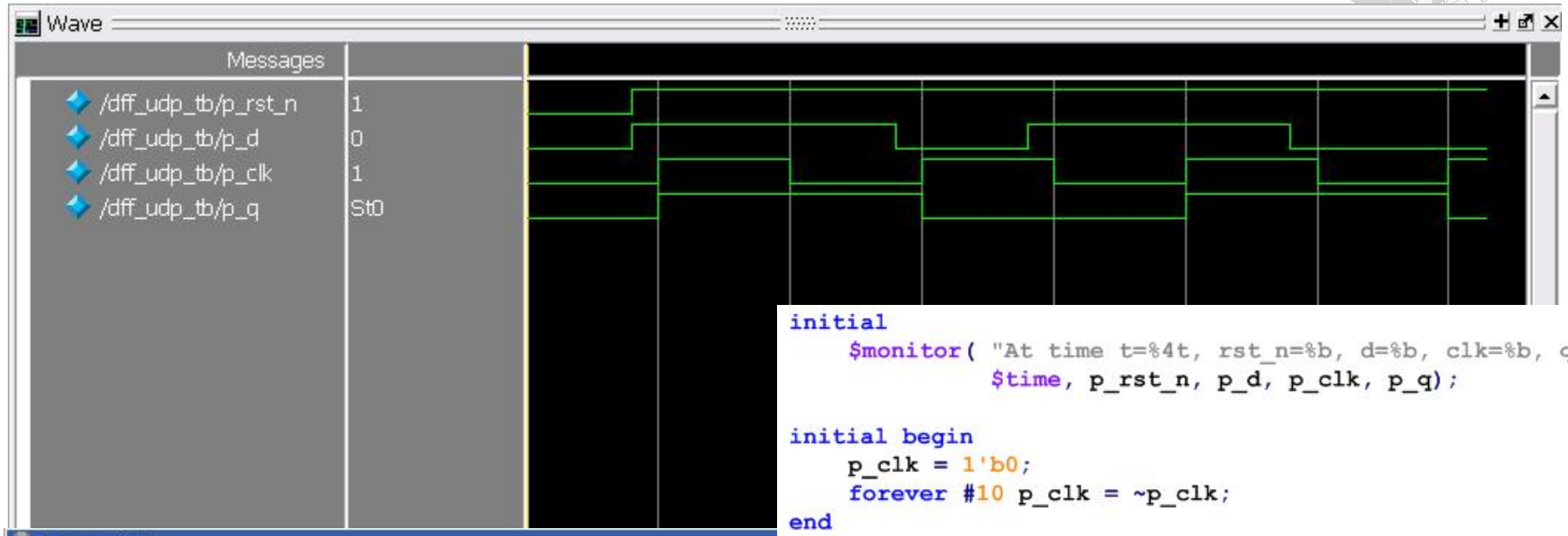
```
// A positive-edge-sensitive D flip-flop
primitive dff_udp( output reg q, input  d, clk, rst_n );
    // initialize q to 0
    initial q = 0;

    // define state table
    table
    // input are in the same order as the input list
    //  d       clk     rst_n   :   q   :   q+  // q+ is the next state
        0       (01)        1       :   ?   :   0;
        1       (01)        1       :   ?   :   1;
        ?       (0x)        1       :   ?   :   -;
        ?       (?0)        1       :   ?   :   -;
        ?       (x0)        1       :   ?   :   -;
    // reset case when rst_n is 0 and clk has any transition
        ?       (??)        0       :   ?   :   0;
        ?       ?           0       :   ?   :   0;
    // reset case when 0 --> 1 transition on rst_n, hold q+ state
        ?       ?         (01)      :   ?   :   -;
        (??)    ?           1       :   ?   :   -;
    endtable
endprimitive
```

上升沿有效的D触发器原语测试模块



```
// test bench for the positive-edge-sensitive D flip-flop
`timescale 1ns/1ns
`include "DFF_udp.v"
module dff_udp_tb;

    reg p_d, p_clk, p_rst_n;
    wire p_q;

    // UDP ports cannot be connected by name
    dff_udp inst( p_q, p_d, p_clk, p_rst_n);

    initial
        $monitor( "At time t=%4t, rst_n=%b, d=%b, clk=%b, q=%b",
                  $time, p_rst_n, p_d, p_clk, p_q);

    initial begin
        p_clk = 1'b0;
        forever #10 p_clk = ~p_clk;
    end

    initial begin
            p_rst_n=1'b0; p_d=1'b0;
        #8  p_rst_n=1'b1; p_d=1'b1;
        #20 p_d=1'b0;
        #10 p_d=1'b1;
        #20 p_d=1'b0;
        #15 $stop;
    end
endmodule
```

上升沿有效的D触发器原语仿真结果



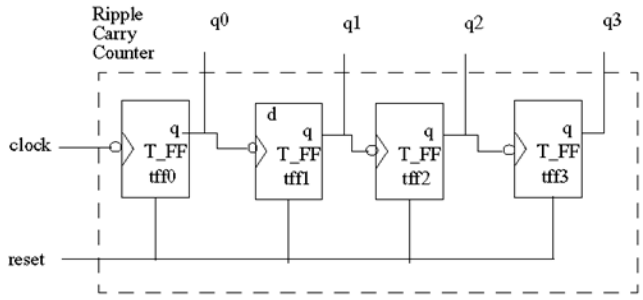

```
initial
    $monitor( "At time t=%4t, rst_n=%b, d=%b, clk=%b, q=%b",
              $time, p_rst_n, p_d, p_clk, p_q);

initial begin
    p_clk = 1'b0;
    forever #10 p_clk = ~p_clk;
end

initial begin
        p_rst_n=1'b0; p_d=1'b0;
    #8  p_rst_n=1'b1; p_d=1'b1;
    #20 p_d=1'b0;
    #10 p_d=1'b1;
    #20 p_d=1'b0;
    #15 $stop;
end
```

Transcript

```
VSIM 5> run
# At time t=    0, rst_n=0, d=0, clk=0, q=0
# At time t=    8, rst_n=1, d=1, clk=0, q=0
# At time t=   10, rst_n=1, d=1, clk=1, q=1
# At time t=   20, rst_n=1, d=1, clk=0, q=1
# At time t=   28, rst_n=1, d=0, clk=0, q=1
# At time t=   30, rst_n=1, d=0, clk=1, q=0
# At time t=   38, rst_n=1, d=1, clk=1, q=0
# At time t=   40, rst_n=1, d=1, clk=0, q=0
# At time t=   50, rst_n=1, d=1, clk=1, q=1
# At time t=   58, rst_n=1, d=0, clk=1, q=1
# At time t=   60, rst_n=1, d=0, clk=0, q=1
# At time t=   70, rst_n=1, d=0, clk=1, q=0
# Break in Module dff_udp_tb at G:/Zest_DSD/PPT_2014_ex/ch08/DFF_UDP/dff_udp_tb.v line 30
```

时序逻辑UDP举例

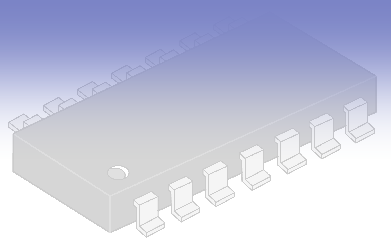
- 使用UDP设计一个4位行波计数器
  - 用UDP描述一个 T 触发器
  - 引用 T 触发器




```
// 行波计数器
module counter(Q , clock, clear);

output [3:0] Q;
input clock, clear;

T_FF tff0(Q[0], clock, clear);
T_FF tff1(Q[1], Q[0], clear);
T_FF tff2(Q[2], Q[1], clear);
T_FF tff3(Q[3], Q[2], clear);

endmodule
```

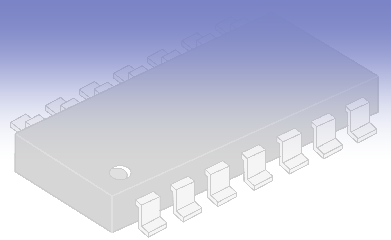
```
primitive T_FF( output reg q,
                input clk, clear );

// 没有使用初始化语句，使用 clear 对 TFF 进行初始化
table
  //  clk    clear :    q    : q+
      ?        1   :    ?    : 0 ;  // 异步清零
      ?      (10)  :    ?    : - ;  // 忽略 clear 的负跳变
      (10)     0   :    1    : 0 ;  // 当 clk 负跳变时，触发器翻转
      (10)     0   :    0    : 1 ;  // 当 clk 负跳变时，触发器翻转
      (0?)     0   :    ?    : - ;  // 忽略 clk 的正跳变
endtable

endprimitive
```

UDP表中的缩写符号



缩写符	含义	解释
?	0, 1, x	不能用于输出部分
b	0, 1	不能用于输出部分
-	保持原值不变	只能用于时序逻辑 UDP
r	(01)	信号的上升沿
f	(10)	信号的下降沿
p	(01), (0x) or (x1)	可能是上升沿
n	(10), (1x) or (x0)	可能是下降沿
*	(??)	信号值的任意变化

UDP设计指南



- 限制
  - 主要用于功能建模
  - 只能有唯一的输出端口
  - 输入端口的数目由仿真器决定
- 要点
  - 应当完整地描述UDP的状态表
  - 注意——电平敏感输入项的优先级高于边沿敏感的优先级

```
primitive T_FF( output reg q,
                input clk, clear );

// 没有使用初始化语句，使用 clear 对 TFF 进行初始化
table
  //  clk    clear :    q    : q+
      ?        1   :    ?    : 0 ;   // 异步清零
      ?      (10)  :    ?    : - ;   // 忽略 clear 的负跳变
      (10)     0   :    1    : 0 ;   // 当 clk 负跳变时，触发器翻转
      (10)     0   :    0    : 1 ;   // 当 clk 负跳变时，触发器翻转
      (0?)     0   :    ?    : - ;   // 忽略 clk 的正跳变
endtable

endprimitive
```