硬件描述语言 Verilog and SystemVerilog

2022年秋

内容

- Verilog简介
- 组合逻辑硬件描述
- 时序逻辑时序逻辑硬件描述

- 大部分内容来自于Onur Mutlu教授（ETH）的上课内容

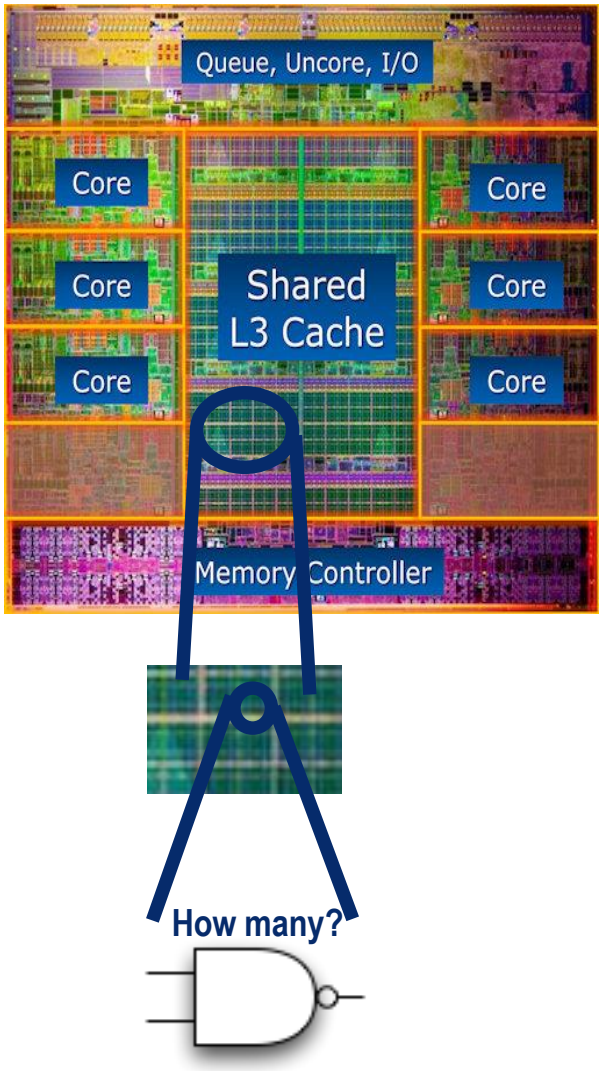
为什么需要硬件描述语言?

- 硬件极其复杂，不可能直接用门电路搭接出最终的硬件电路，通过硬件描述语言进行抽象，使得对硬件的描述成为了可能
  - 晶体管（门级），连线
- 硬件描述语言
  - 可用于描述复杂的硬件设计
  - 可用于行为仿真（包括功能和时序）
  - 可用于硬件的综合
- 硬件描述语言被设计出来完成上述的目的
  - 不同的硬件描述语言有很多的相似性（VHDL, Verilog），完成相同的目标
  - 语言功能通常可以直接映射，特别是对于常用的子集
  - Verilog（SystemVerilog）在工业界常用

硬件设计的准则——层次化设计

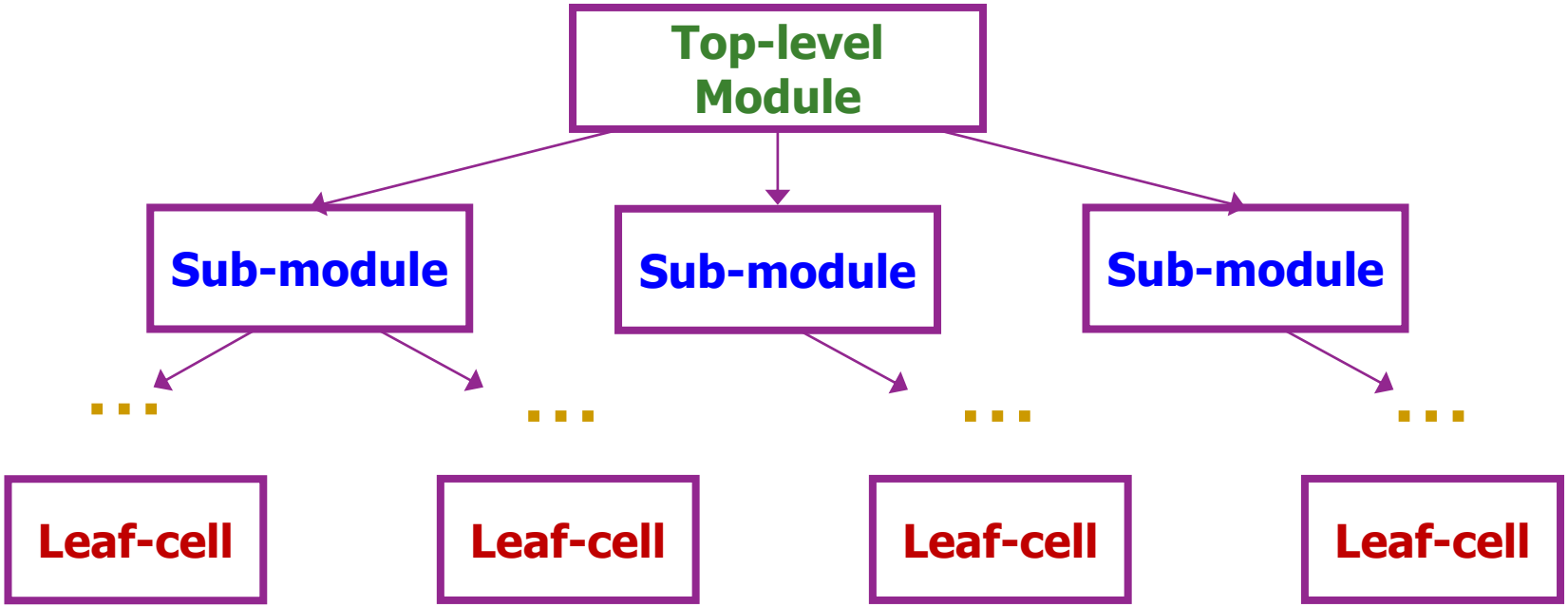
https://techreport.com/review/21987/intel-core-i7-3960x-processor

- **按照层次来组织硬件模块**
  - 底层为 “基本原语” 门 (AND, OR, …)
  - 通过基本门的原件例化来实现简单模块 (e.g., 例如多路选择器 MUXes)
  - 通过简单模块例化来实现复杂模块…
- **层次化设计控制了复杂性**
  - 类似于编程中的函数/方法的抽象
- **复杂性是一个很大的问题**
  - Sandy Bridge-E有2.27B个晶体管



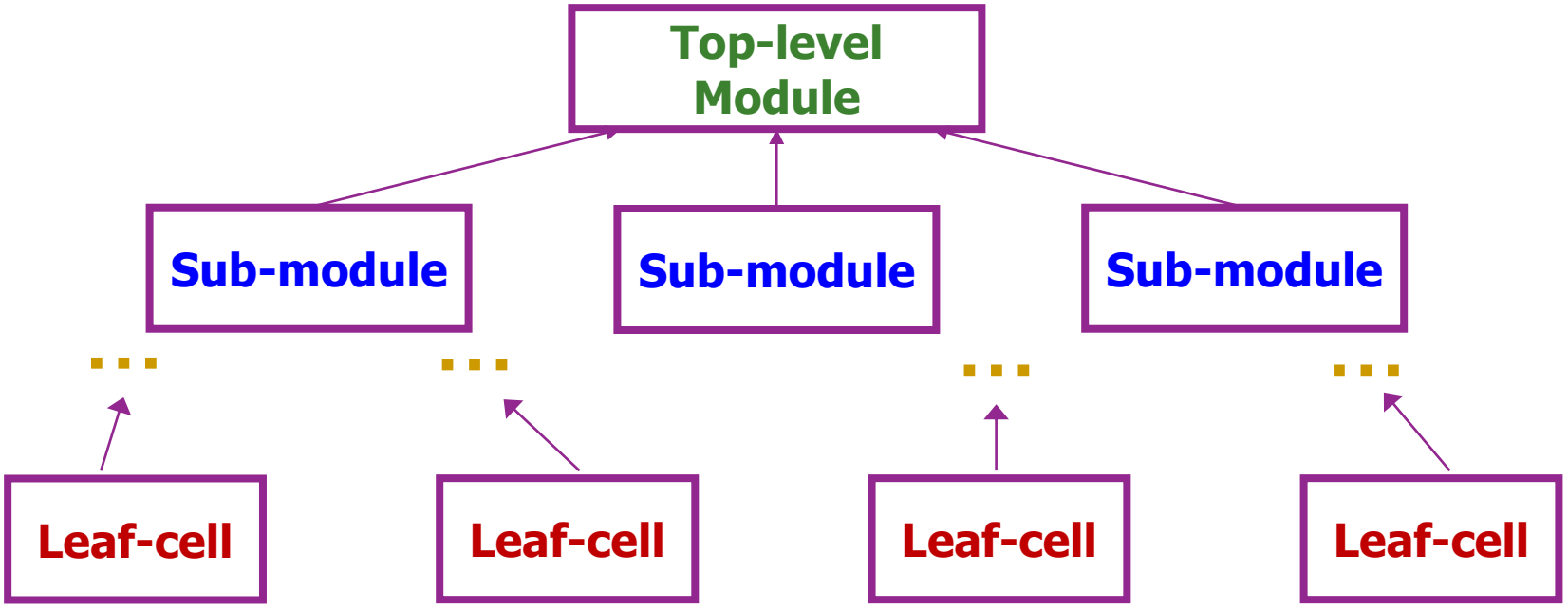
自顶向下的设计方法

- 定义顶层模块，并确定构建顶层模块所需的子模块
- 对子模块进行细分，直到最基本的门级电路（叶子单元 leaf-cell）
  - Leaf cell: 不能进一步分割的电路元件（例如，逻辑门、原始单元库元件等）。



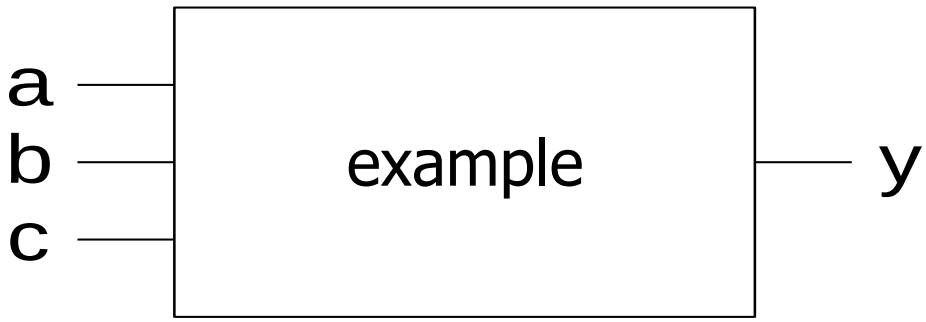
自底向上的设计方法

- 从基本模块开始构建
- 逐步从基本模块向上构造更加复杂的模块
- 直到最终设计完成最上层的顶层模块
- 两种方法不是互斥的，实际设计中需要综合使用

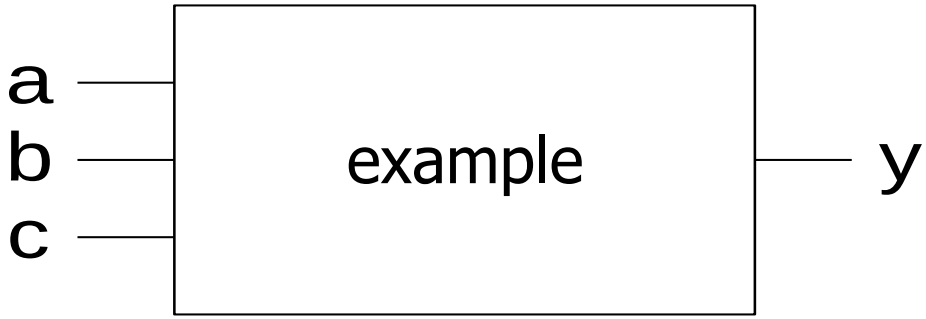


Verilog中定义一个模块

- 模块module是verilog(systemverilog)中的基本编程单元
- 首先是进行模块的定义：
  - 模块的名字
  - 模块端口的名字
  - 模块端口的方向（输入端口还是输出端口）
- 在模块定义的基础上描述模块的功能



模块实现



模块的名字

端口的列表
(输入输出端口)

```
module example (a, b, c, y);
        input a;
        input b;
        input c;
        output y;

// here comes the circuit description

endmodule
```

端口的类型

模块的定义

端口接口定义

- 下面的代码是等价的

```
module test ( a, b, y );
        input a;
        input b;
        output y;

endmodule
```

```
module test ( input a,
              input b,
              output y );

endmodule
```

端口名字和方向可以放在一起

信号数组

- **多位的输入输出Input/Output (Bus)**
  - [range_end : range_start]
  - **位数:** range_end – range_start + 1
- **举例**:

```
input  [31:0] a;   // a[31], a[30] .. a[0]
output [15:8] b1;  // b1[15], b1[14] .. b1[8]
output [7:0]  b2;  // b2[7], b2[6] .. b2[0]
input         c;   // single signal
```

- **a** 代表了一个32位的值 `[31:0] a`
- 通常使用 [31:0] 比 [0:31]普遍
- 无论如何定义，要保持在程序中是一致的

位操作

```
// You can assign partial buses
   wire [15:0] longbus;
   wire [7:0] shortbus;
   assign shortbus = longbus[12:5];

// Concatenating is by {}
   assign y = {a[2],a[1],a[0],a[0]};

// Possible to define multiple copies
   assign x = {a[0], a[0], a[0], a[0]};
   assign y = { 4{a[0]} };
```

基本的语法

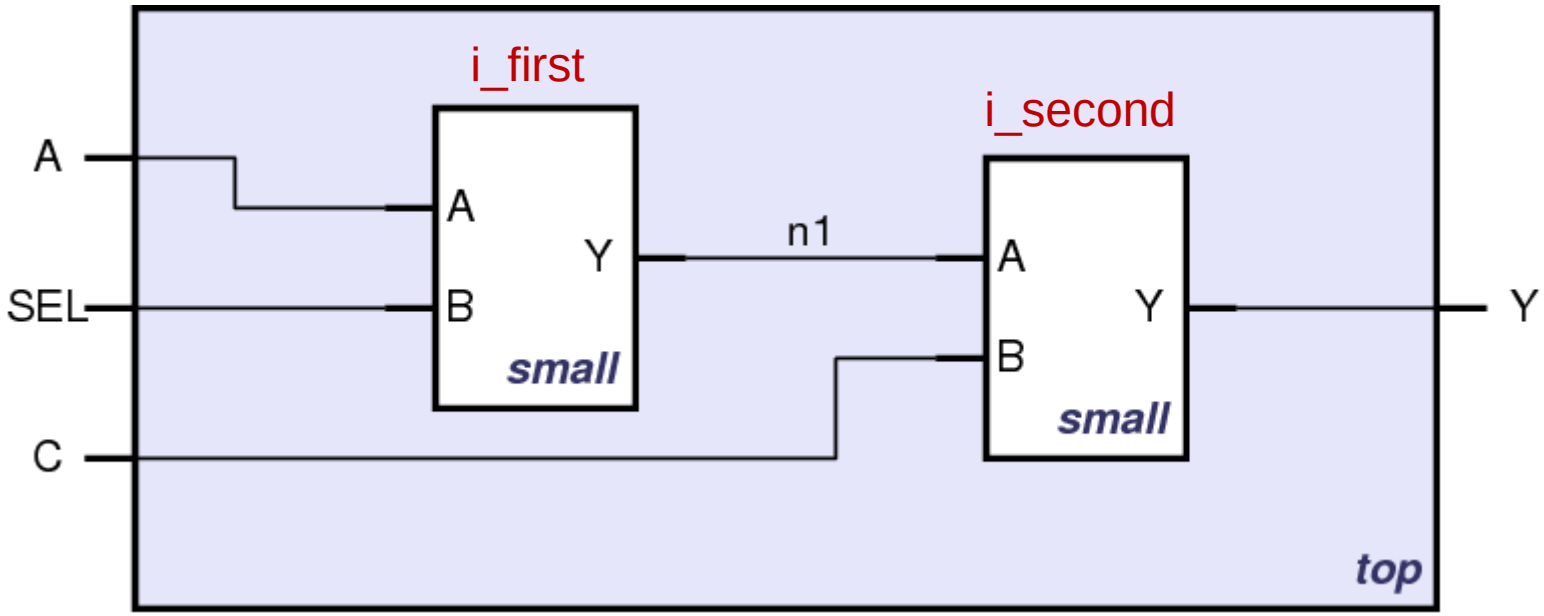
- 区分大小写
- 标识符不可以以数字开始（和c语言一致）
- 空白字符忽略
- 注释

```
// Single line comments start with a //

/* Multiline comments
   are defined like this */
```

Verilog两种基本实现风格

- 结构描述(门级描述)
  - 更像进行硬件的设计
  - 模块主体包含电路的门级描述
  - 描述模块是如何相互连接的
  - 每个模块包含其它模块（实例） 以及这些模块之间的互连关系
- 行为描述
  - 描述模块的行为，而不设计具体的模块
  - 包含逻辑和数学运算符
  - 抽象水平高于门级描述
    - 相同的行为描述会有多种可能的门级实现方式
- 实际的系统会同时使用上述的两种实现方式

结构描述：实例化一个模块

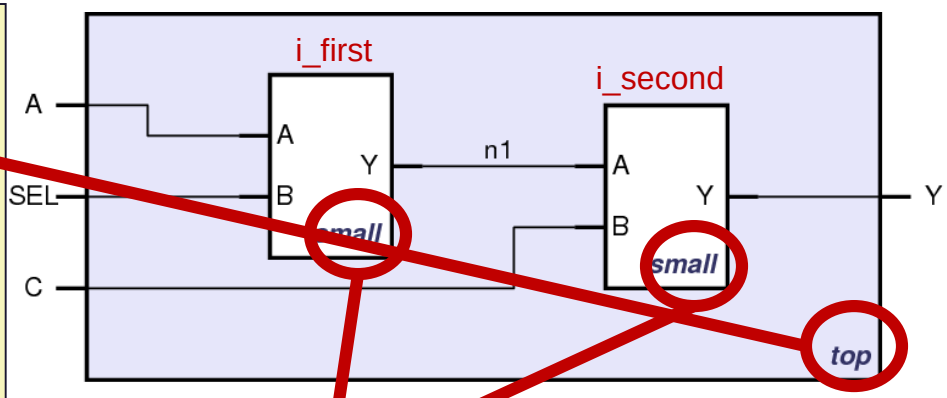


top的模块由两个small模块组成，上图是它们直接的连线关系

结构描述的举例1

模块的定义

```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;









endmodule
```

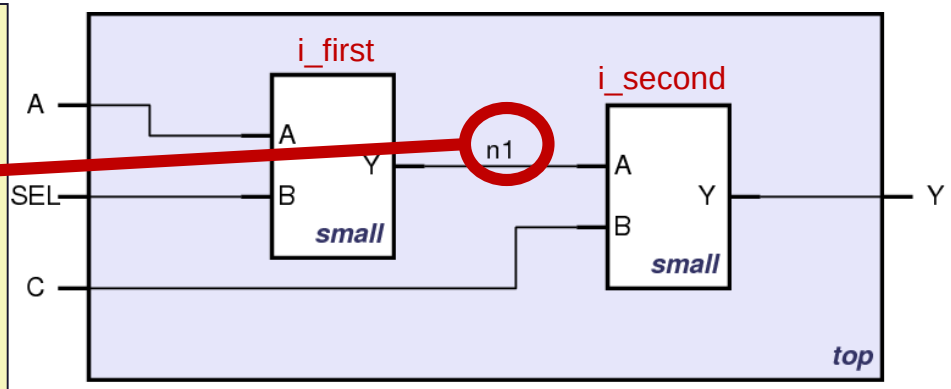



```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

结构描述的举例2

定义连线关系 (模块互联)

```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;


endmodule
```

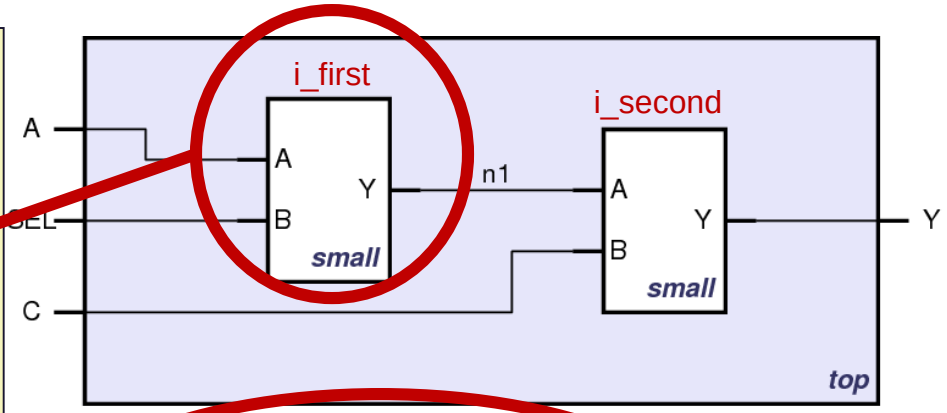



```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

结构描述的举例3

第一个例化的 “small”模块

```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;

// instantiate small once
small i_first ( .A(A),
                .B(SEL),
                .Y(n1)   );




endmodule
```

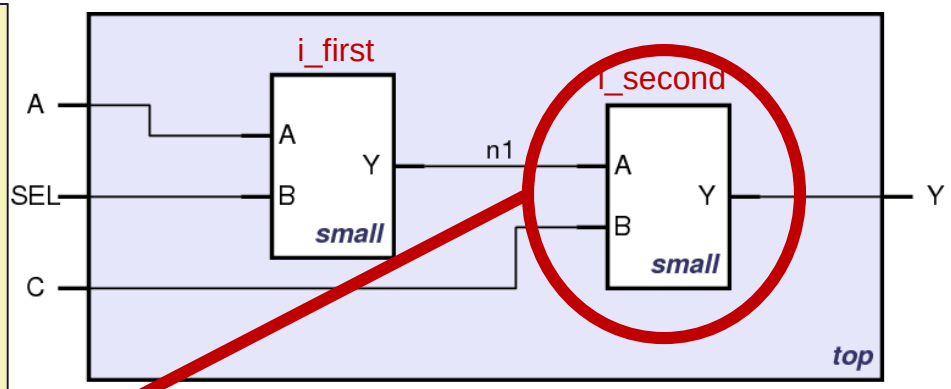



```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

结构描述的举例4

第二个例化的 “small”模块

```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;

// instantiate small once
small i_first ( .A(A),
                .B(SEL),
                .Y(n1)   );

// instantiate small second time
small i_second ( .A(n1),
           .B(C),
           .Y(Y) );

endmodule
```

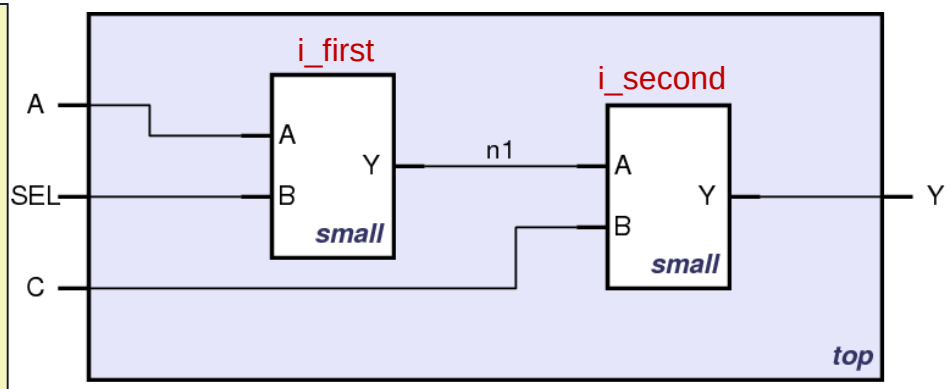



```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

结构描述的举例5

模块实例化的简短形式

```
module top (A, SEL, C, Y);
  input A, SEL, C;
  output Y;
  wire n1;

// alternative short form
small i_first ( A, SEL, n1 );

/* In short form above,
   pin order very important */

// safer choice; any pin order
small i_second ( .B(C),
                 .Y(Y),
                 .A(n1) );

endmodule
```




```
module small (A, B, Y);
  input A;
  input B;
  output Y;

// description of small

endmodule
```

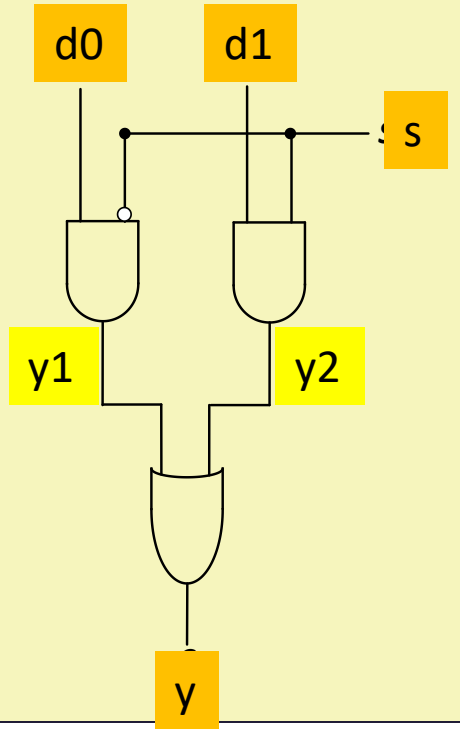
简短的形式不是好的做法

降低了代码的可维护性

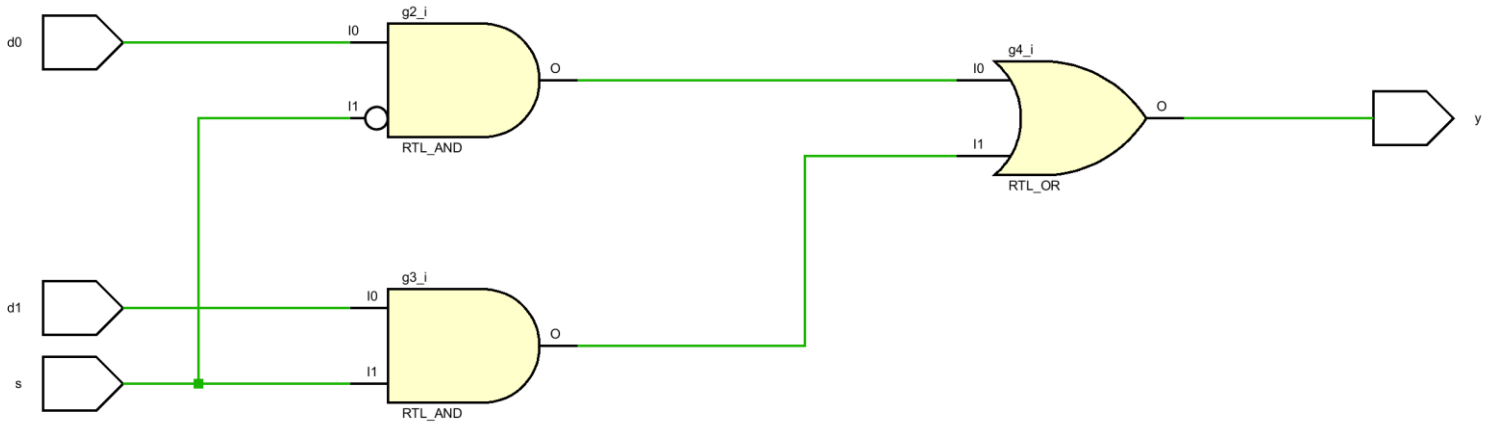
结构描述的举例6

- 支持基本的逻辑门作为预定义的基本模块
  - 这些基本模块像其它模块一样被实例化，它们在Verilog中被预定义，不需要模块定义

```
module mux2(input d0, d1,
            input s,
            output y);
   wire ns, y1, y2;

   not  g1 (ns, s);
   and  g2 (y1, d0, ns);
   and  g3 (y2, d1, s);
   or   g4 (y, y1, y2);

endmodule
```



- **RTL ANALYSIS**
  - **Open Elaborated Design**
    - Report Methodology
    - Report DRC
    - Report Noise
    - Schematic



- **SYNTHESIS**
  - Run Synthesis
  - **Open Synthesized Design**
    - Constraints Wizard
    - Edit Timing Constraints
    - Set Up Debug
    - Report Timing Summary
    - Report Clock Networks
    - Report Clock Interaction
    - Report Methodology
    - Report DRC
    - Report Noise
    - Report Utilization
    - Report Power
    - Schematic

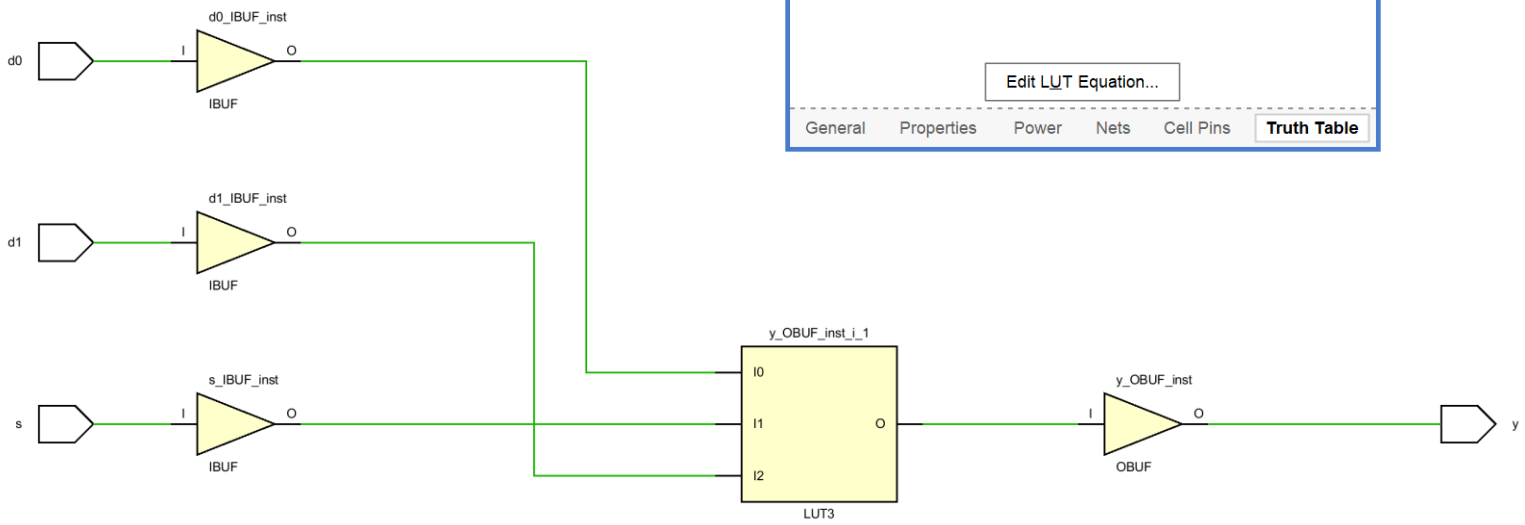
Cell Properties

y_OBUF_inst_i_1

I2	I1	I0	O=I0 & !I1 + I1 & I2
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Edit LUT Equation...

General | Properties | Power | Nets | Cell Pins | **Truth Table**

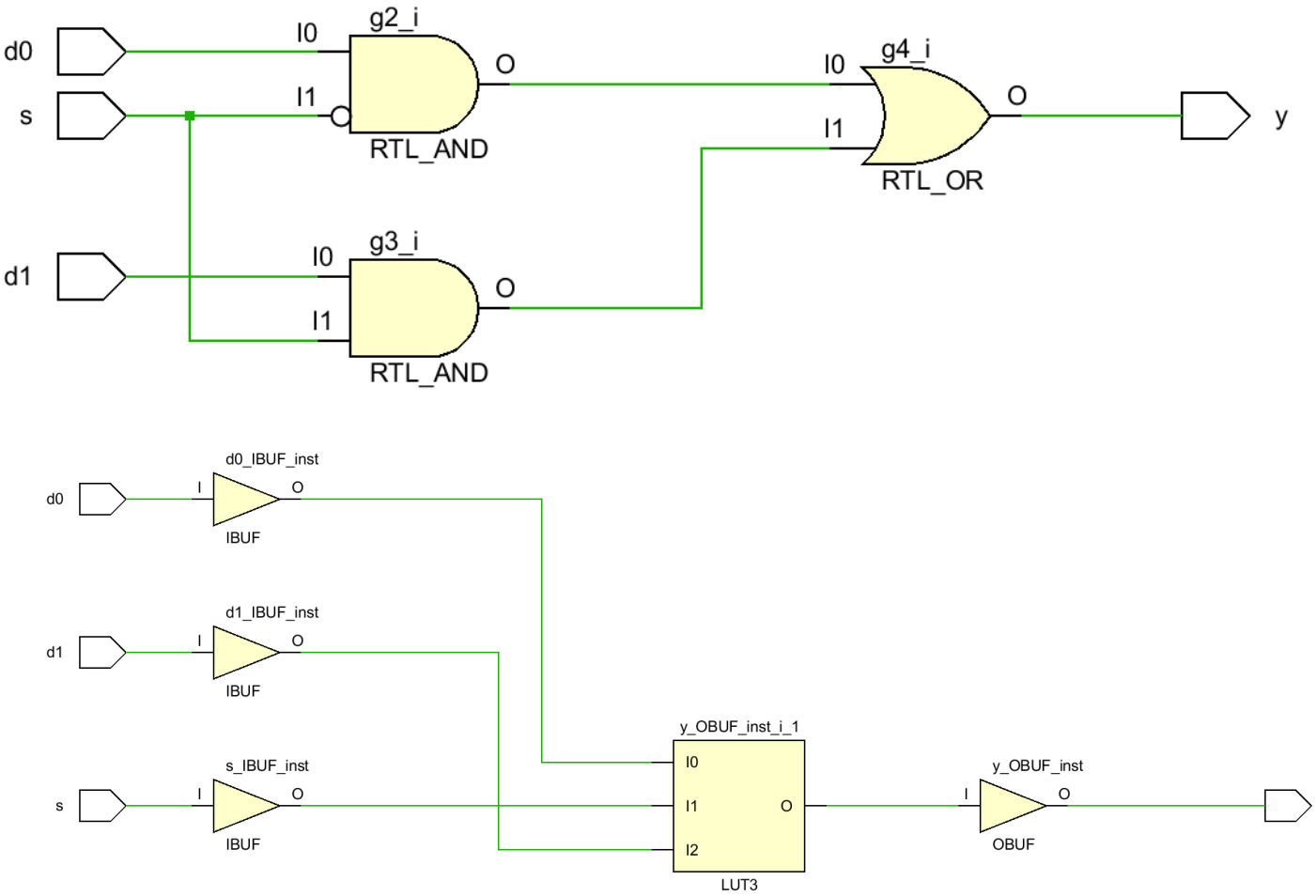


行为描述

```
module mux2(input d0, d1,
            input s,
            output y);

// here comes the circuit description
assign y =  d0 & !s |
            d1 & s;
endmodule
```

行为描述：实现结果

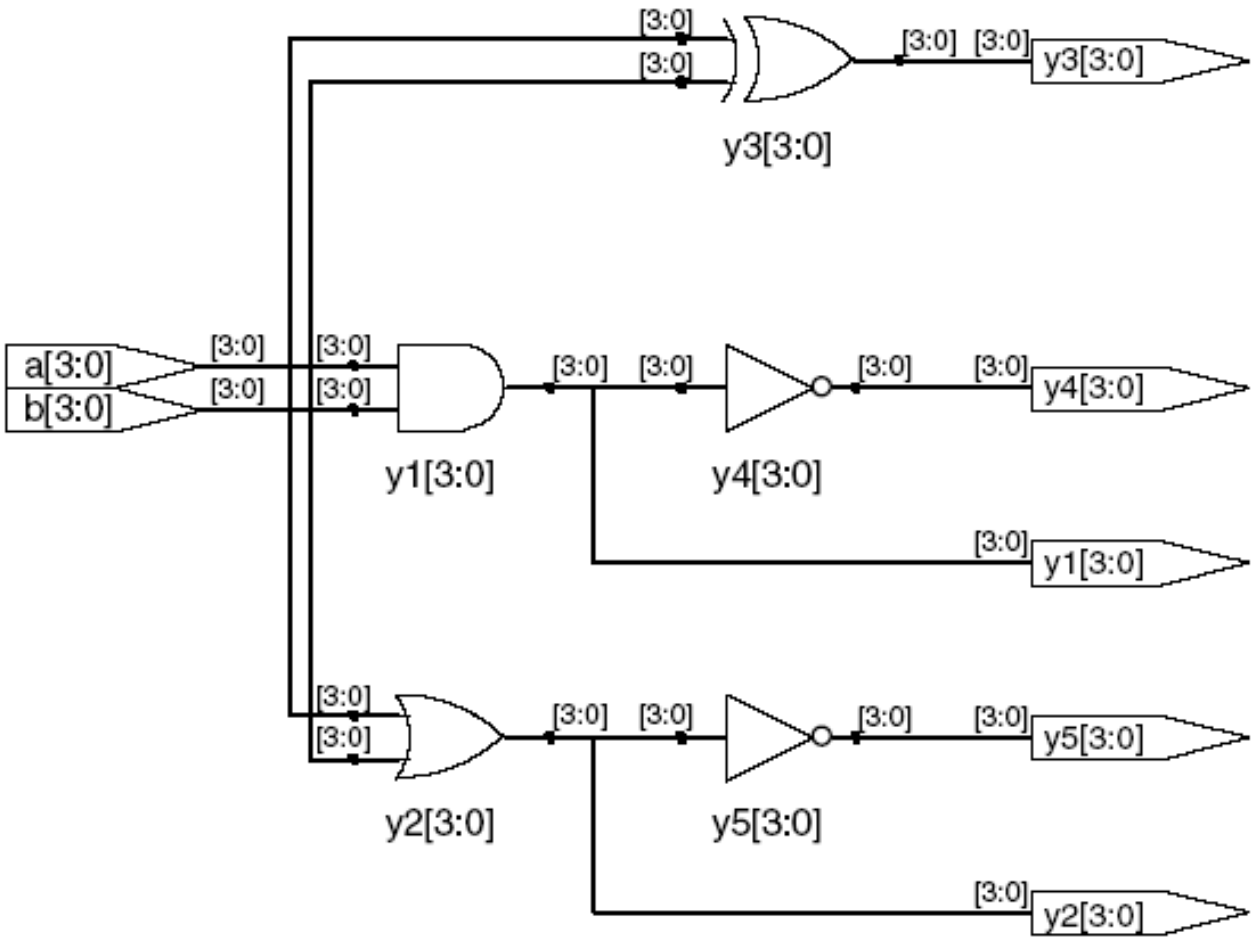
行为描述同样定义了硬件电路的模型



行为描述中的位操作

```
module gates(input  [3:0]  a, b,
             output [3:0] y1, y2, y3, y4, y5);

   /* Five different two-input logic
      gates acting on 4 bit buses */

   assign y1 = a & b;     // AND
   assign y2 = a | b;     // OR
   assign y3 = a ^ b;     // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR

endmodule
```

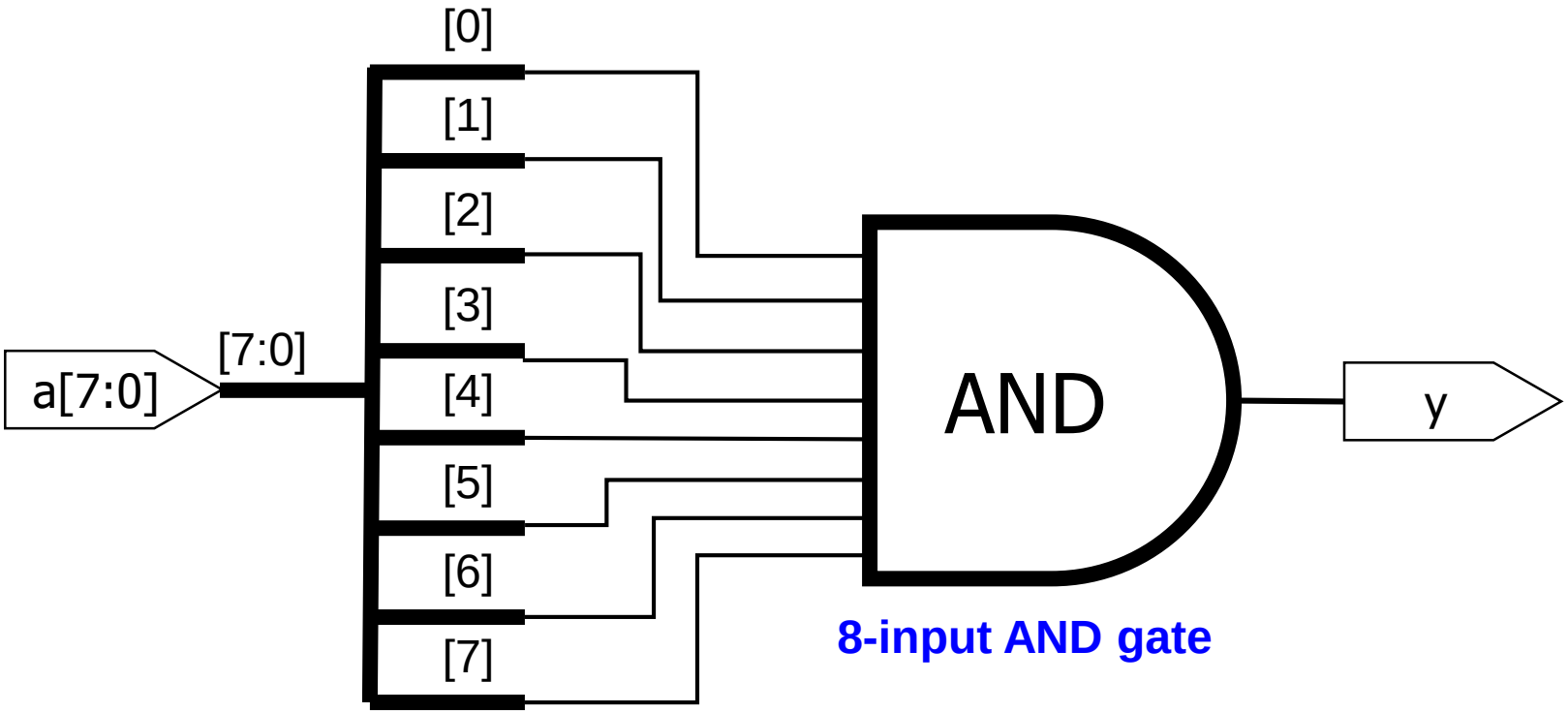
实现结果


[3:0]
[3:0]
[3:0]
[3:0]
y3[3:0]
y3[3:0]
a[3:0]
b[3:0]
[3:0]
[3:0]
[3:0]
[3:0]
y1[3:0]
[3:0]
[3:0]
[3:0]
[3:0]
y4[3:0]
y4[3:0]
[3:0]
y1[3:0]
[3:0]
[3:0]
[3:0]
[3:0]
y2[3:0]
[3:0]
[3:0]
[3:0]
[3:0]
y5[3:0]
y5[3:0]
[3:0]
y2[3:0]

行为描述中的归并操作

```
module and8(input  [7:0] a,
            output       y);

   assign y = &a;

   // &a is much easier to write than
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];

endmodule
```

实现结果

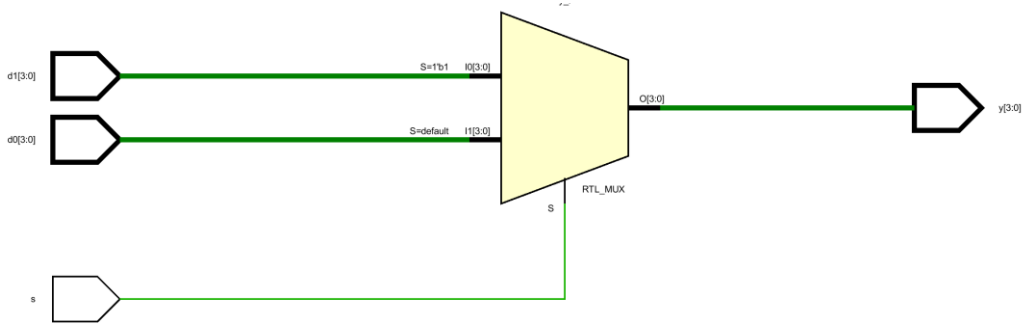


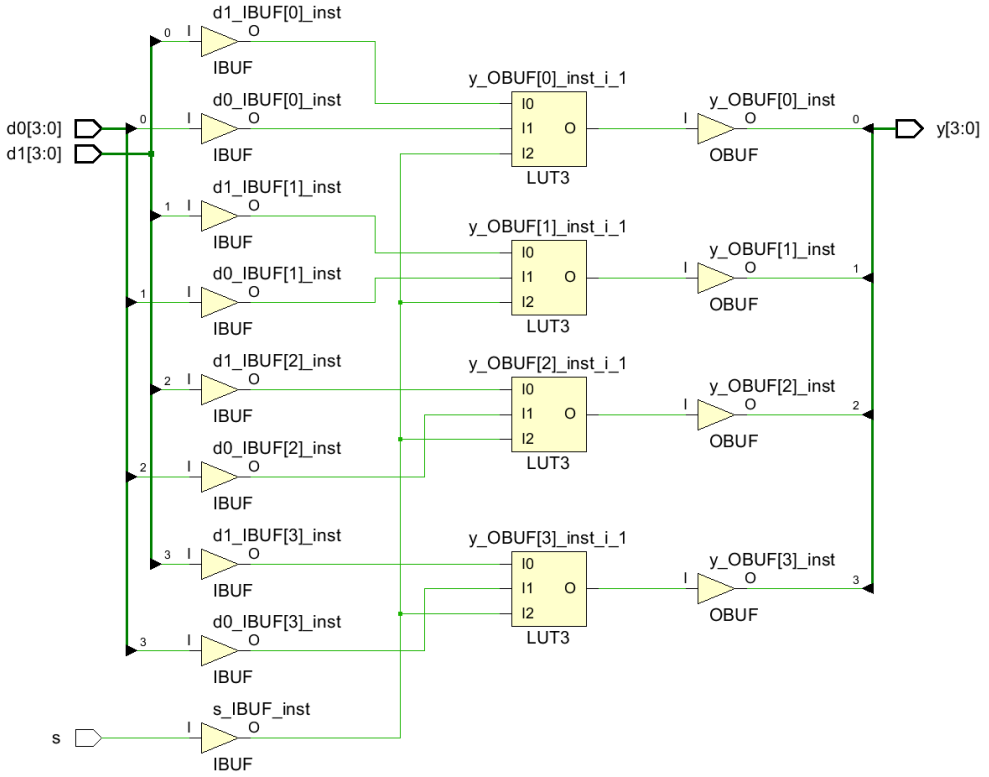
行为描述中的条件赋值

```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

   assign y = s ? d1 : d0;
   // if (s) then y=d1 else y=d0;

endmodule
```

- ?: 是三元操作符:
  - s
  - d1
  - d0

实现结果


d1[3:0]
d0[3:0]
s
S=1'b1
I0[3:0]
S=default
I1[3:0]
O[3:0]
y[3:0]
RTL_MUX
S


d0[3:0]
d1[3:0]
s
d1_IBUF[0]_inst
d0_IBUF[0]_inst
d1_IBUF[1]_inst
d0_IBUF[1]_inst
d1_IBUF[2]_inst
d0_IBUF[2]_inst
d1_IBUF[3]_inst
d0_IBUF[3]_inst
s_IBUF_inst
IBUF
y_OBUF[0]_inst_i_1
y_OBUF[1]_inst_i_1
y_OBUF[2]_inst_i_1
y_OBUF[3]_inst_i_1
I0
I1
I2
O
LUT3
y_OBUF[0]_inst
y_OBUF[1]_inst
y_OBUF[2]_inst
y_OBUF[3]_inst
OBUF
y[3:0]

条件赋值2

```
module mux4(input  [3:0] d0, d1, d2, d3
            input  [1:0] s,
            output [3:0] y);

   assign y = s[1] ? ( s[0] ? d3 : d2)
                   : ( s[0] ? d1 : d0);
  // if (s1) then
  //       if (s0) then y=d3 else y=d2
  // else
  //       if (s0) then y=d1 else y=d0

endmodule
```

条件赋值3

```
module mux4(input  [3:0] d0, d1, d2, d3
            input  [1:0] s,
            output [3:0] y);

   assign y = (s == 2'b11) ? d3 :
              (s == 2'b10) ? d2 :
              (s == 2'b01) ? d1 :
              d0;
// if      (s = “11” ) then y= d3
// else if (s = “10” ) then y= d2
// else if (s = “01” ) then y= d1
// else                     y= d0

endmodule
```

操作符优先级

Highest

~	NOT
*, /, %	mult, div, mod
+, -	add,sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	ternary operator

Lowest

Verilog中的数字表达

N’ Bxx

8’ b0000_0001

- (N) 位数
  - 表示将使用多少比特来存储数值
- (B) Base
  - 基数 b (binary), h (hexadecimal), d (decimal), o (octal)
- (xx) 数值
  - 以基数表示的值
  - 也可以有X（无效的，不关心的）和Z（高阻态），作为数值
  - 可以使用下划线 _ 来提高可读性

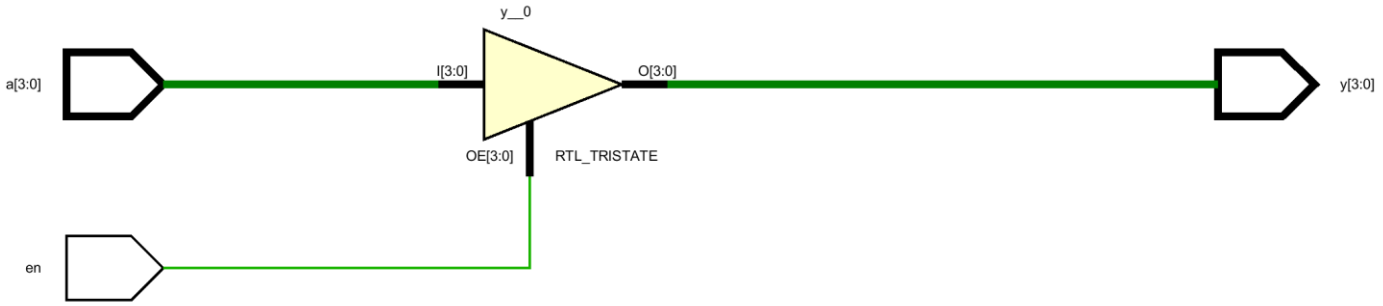
数值举例

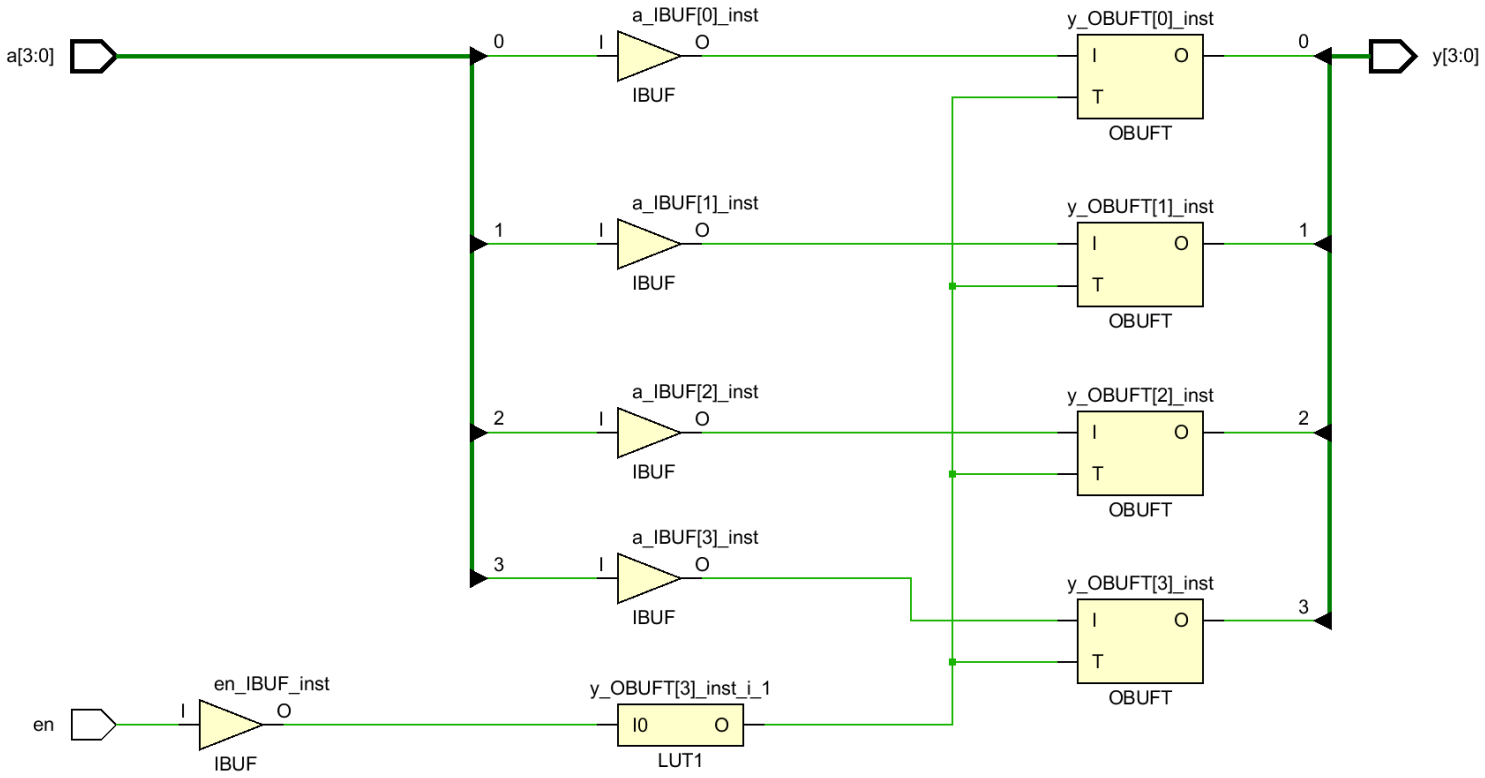
Verilog	Stored Number	Verilog	Stored Number
4'b1001	1001	4'd5	0101
8'b1001	0000 1001	12'hFA3	1111 1010 0011
8'b0000_1001	0000 1001	8'o12	00 001 010
8'bxX0X1zZ1	XX0X 1ZZ1	4'h7	0111
'b01	0000 .. 0001	12'h0	0000 0000 0000

32 bits (default)

高阻态的表达

```
module tristate_buffer(input  [3:0] a,
                       input        en,
                       output [3:0] y);

   assign y = en ? a : 4'bz;

endmodule
```




a[3:0]
0
a_IBUF[0]_inst
I
O
IBUF
y_OBUFT[0]_inst
I
O
T
OBUFT
0
y[3:0]
1
a_IBUF[1]_inst
IBUF
y_OBUFT[1]_inst
OBUFT
1
2
a_IBUF[2]_inst
IBUF
y_OBUFT[2]_inst
OBUFT
2
3
a_IBUF[3]_inst
IBUF
y_OBUFT[3]_inst
OBUFT
3
en
en_IBUF_inst
IBUF
y_OBUFT[3]_inst_i_1
I0
LUT1

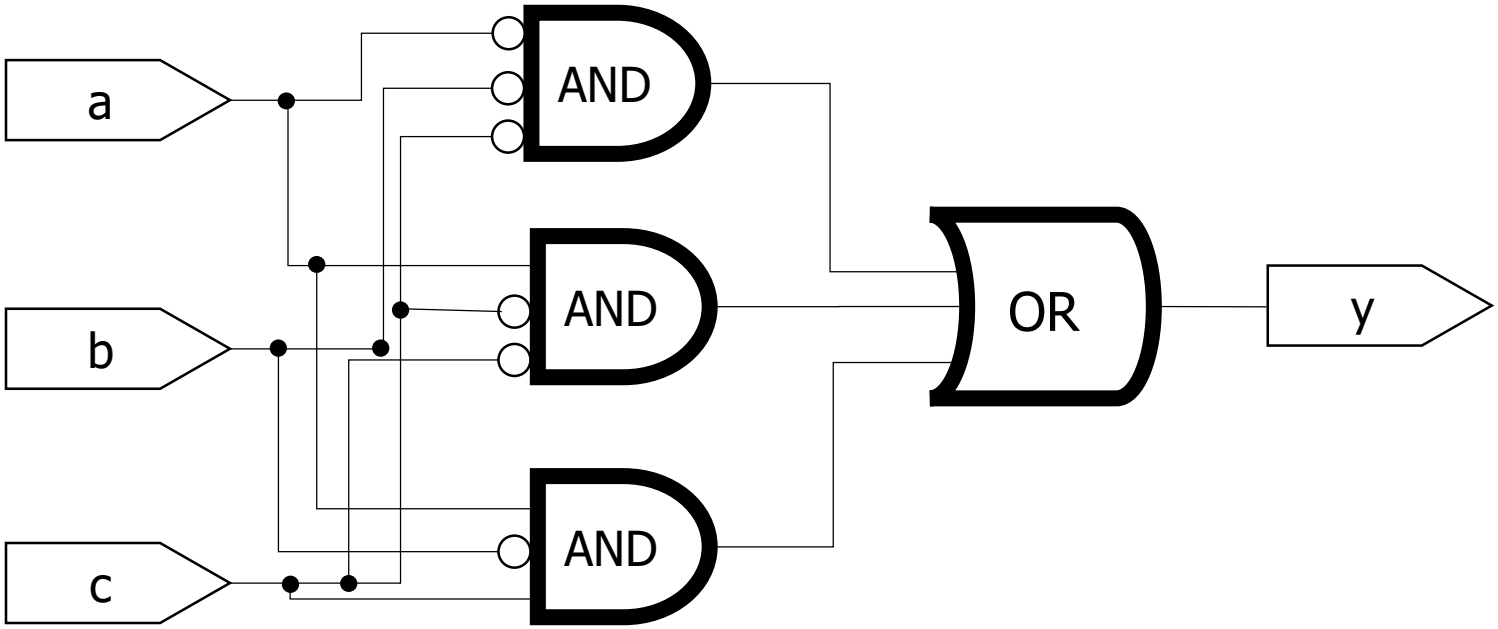
高阻态下的真值表

AND		A			
		0	**1**	**Z**	**X**
B	**0**	0	0	0	0
	1	0	1	X	X
	Z	0	X	X	X
	X	0	X	X	X

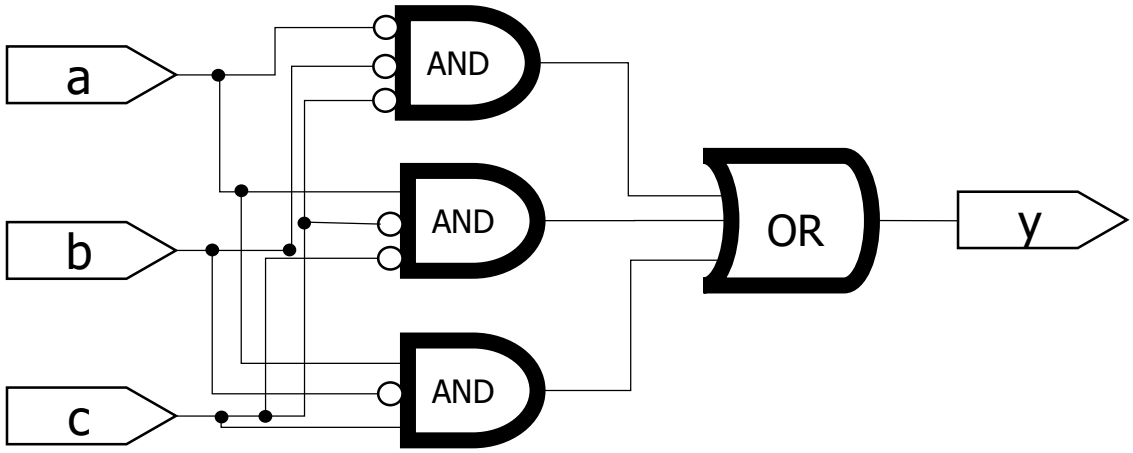
数字逻辑电路中的波形

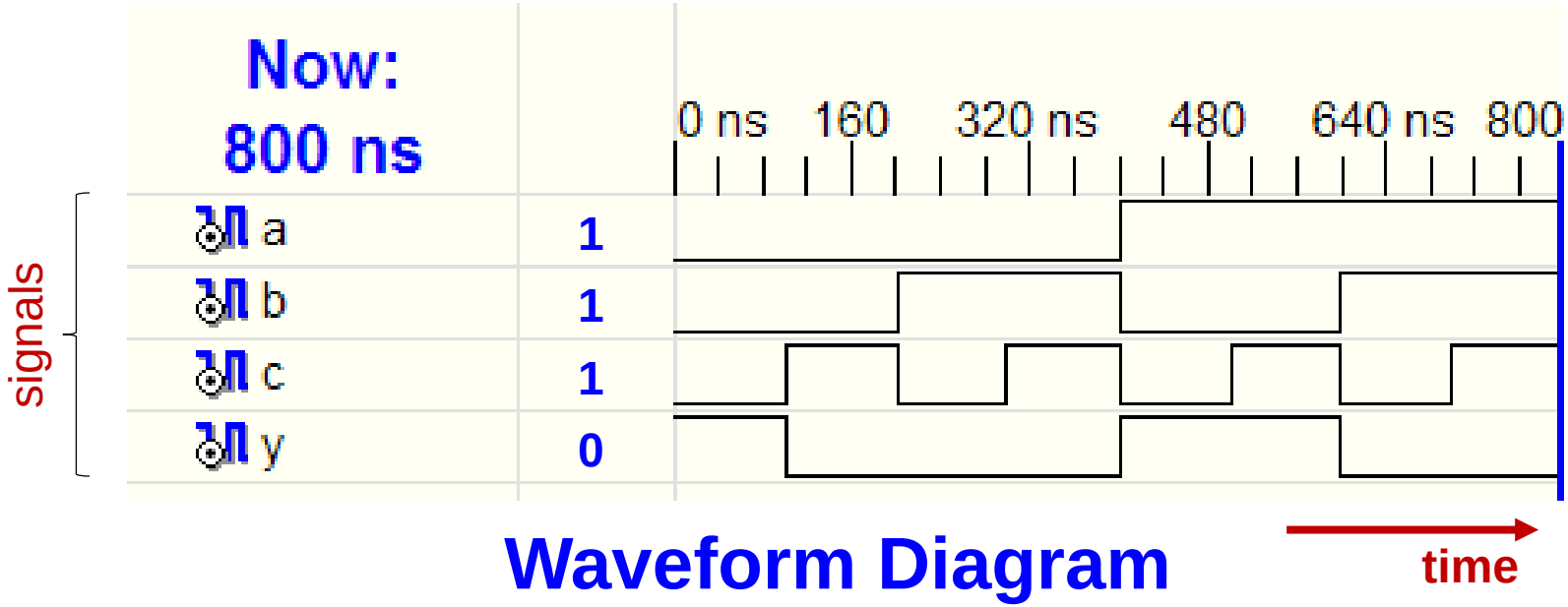
```
module example (a, b, c, y);
       input a;
       input b;
       input c;
       output y;

// here comes the circuit description
assign y = ~a & ~b & ~c |
            a & ~b & ~c |
            a & ~b &  c;

endmodule
```

综合的结果



仿真波形


a
b
c
AND
AND
AND
OR
y


Now:
800 ns
0 ns
160
320 ns
480
640 ns
800
signals
a
b
c
y
1
1
1
0
Waveform Diagram
time

硬件编程注意事项

One of the most common mistakes for beginners is to think of HDL as a computer program rather than as a shorthand for describing digital hardware. If you don't know approximately what hardware your HDL should synthesize into, you probably won't like what you get. You might create far more hardware than is necessary, or you might write code that simulates correctly but cannot be implemented in hardware. Instead, think of your system in terms of blocks of combinational logic, registers, and finite state machines. Sketch these blocks on paper and show how they are connected before you start writing code.

- 初学者通常的错误是认为HDL是计算机编程语言，而不是数字硬件的描述语言
- 如果不知道被综合成什么样的硬件，那很有可能就是不对的
- 正确的做法：从组合逻辑，寄存器，有限状态机的角度来考虑系统。可以在写代码之前在纸上绘制模块以及它们之间的连线情况

比较器的多种不同代码实现

- 可能使用到的模块定义

An XNOR gate

```
module MyXnor (input A, B,
               output Z);

   assign Z = ~(A ^ B); //not XOR

endmodule
```

An AND gate

```
module MyAnd (input A, B,
              output Z);

   assign Z = A & B;     // AND

endmodule
```

结构描述实现

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
       wire c0, c1, c2, c3, c01, c23;

MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
MyAnd haha (.A(c0), .B(c1), .Z(c01) ); // AND
MyAnd hoho (.A(c2), .B(c3), .Z(c23) ); // AND
MyAnd bubu (.A(c01), .B(c23), .Z(eq) ); // AND

endmodule
```

逻辑操作符实现

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
       wire c0, c1, c2, c3, c01, c23;

MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
assign c01 = c0 & c1;
assign c23 = c2 & c3;
assign eq  = c01 & c23;

endmodule
```

不使用中间连线信号名字

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
       wire c0, c1, c2, c3;

MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
// assign c01 = c0 & c1;
// assign c23 = c2 & c3;
// assign eq  = c01 & c23;
assign eq  = c0 & c1 & c2 & c3;


endmodule
```

使用信号数组（总线）

```
module compare (input [3:0] a, input [3:0] b,
                output eq);
       wire [3:0] c; // bus definition

MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0]) ); // XNOR
MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1]) ); // XNOR
MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2]) ); // XNOR
MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3]) ); // XNOR

assign eq  = &c; // short format


endmodule
```

按位操作运算符

```
module compare (input [3:0] a, input [3:0] b,
                output eq);
       wire [3:0] c; // bus definition

MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0]) ); // XNOR
MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1]) ); // XNOR
MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2]) ); // XNOR
MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3]) ); // XNOR

assign eq  = &c; // short format


endmodule
```

最高层的抽象

```
module compare (input [3:0] a, input [3:0] b,
                output eq);



// assign c = ~(a ^ b); // XNOR

// assign eq  = &c; // short format

assign eq = (a == b) ? 1 : 0; // really short



endmodule
```

模块参数

- 如果需要N位的比较器呢？

```
module mux2
  #(parameter width = 8)  // name and default value
   (input  [width-1:0] d0, d1,
    input              s,
    output [width-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

参数化模块的例化

```
module mux2
  #(parameter width = 8)  // name and default value
   (input  [width-1:0] d0, d1,
    input              s,
    output [width-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

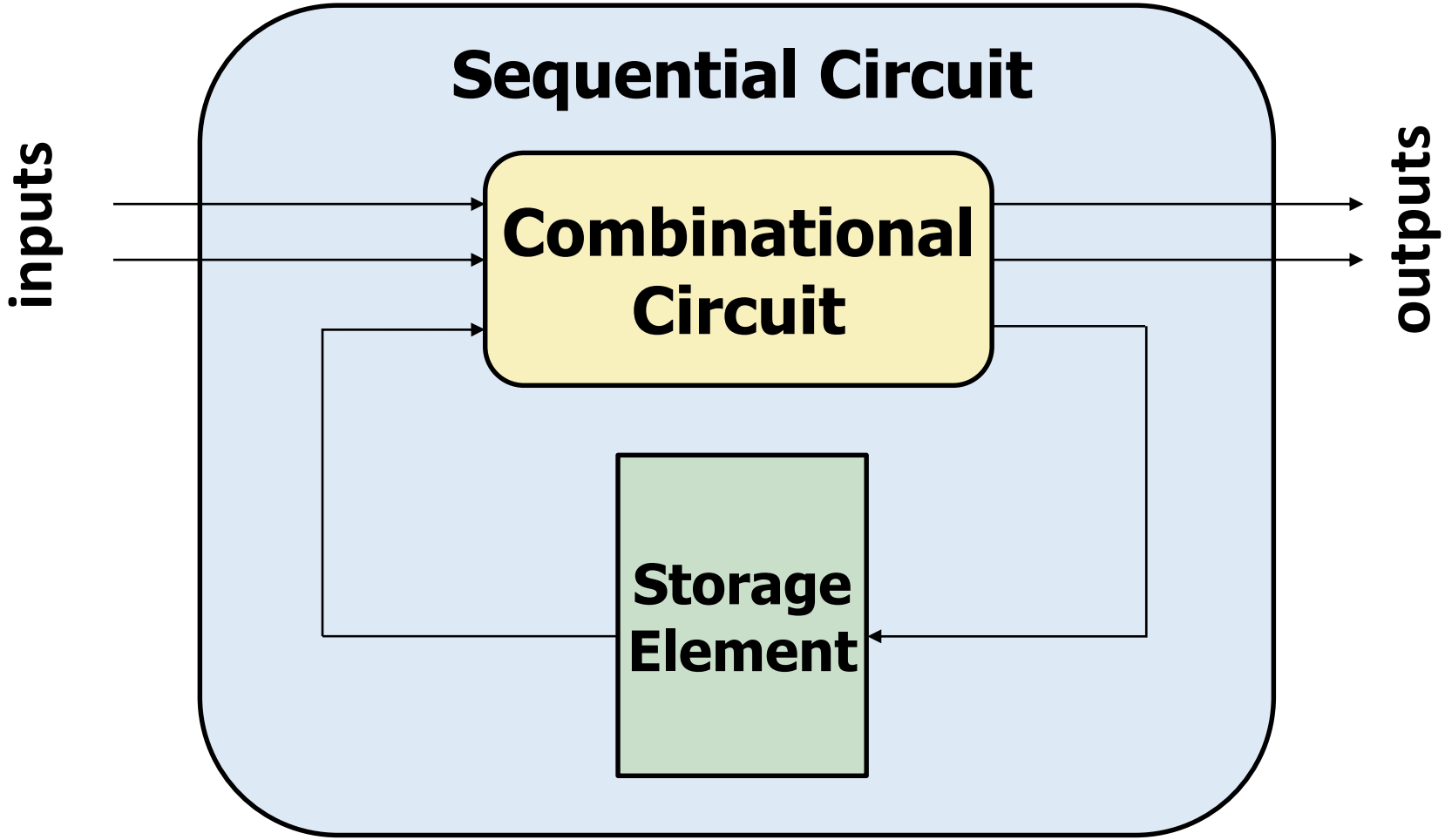
```
// If the parameter is not given, the default (8) is assumed
   mux2 i_mux (d0, d1, s, out);

// The same module with 12-bit bus width:
   mux2 #(12) i_mux_b (d0, d1, s, out);

// A more verbose version:
   mux2 #(.width(12)) i_mux_b (.d0(d0), .d1(d1),
                               .s(s), .out(out));
```

延时的指定

- 在程序中可以指定延时。
- 延时是不可综合的，但是在仿真中非常有用
- 用来仿真硬件的模型
- 1ns代码中时间的单位，1ps仿真的粒度

```
‘timescale 1ns/1ps
module simple (input a, output z1, z2);

assign #5 z1 = ~a; // inverted output after 5ns
assign #9 z2 = a;  // output after 9ns

endmodule
```

时序逻辑电路描述


Sequential Circuit
inputs
Combinational Circuit
outputs
Storage Element

Verilog时序逻辑编程

- 定义具有记忆单元的模块
  - 锁存器latch，触发器flipflop，状态机FSM
- 时序逻辑通过时钟来“触发”
  - 锁存器电平触发
  - 触发器时钟边沿触发，我们仅使用边沿触发
- 时序逻辑编程需要用到
  - always过程语句
  - posedge/negedge触发条件

always过程块

```
always @ (sensitivity list)
      statement;
```

每当敏感度列表中的事件发生时。
语句就会被执行

触发器（基于触发器的寄存器）1

```
module flop(input             clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                  // pronounced “q gets d”

endmodule
```

- posedge 在时钟上升沿响应（处在敏感列表中）
- always过程块内部的语句会在时钟上升沿的时候得到执行
- 时钟上升沿来临时，d被拷贝入q

触发器（基于触发器的寄存器）2

```
module flop(input              clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                    // pronounced “q gets d”

endmodule
```

- assign 语句不会在always过程块内部使用
- <= 非阻塞赋值

触发器（基于触发器的寄存器）3

```
module flop(input             clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;                  // pronounced “q gets d”

endmodule
```

- 赋值变量需要被声明为reg
- reg这个名字不一定意味着这个值是一个寄存器（可以是，但不一定是）。

赋值语句

- 持续赋值语句（continuous assignments）

- assign为持续赋值语句，主要用于对wire型变量的赋值。
- 比如：assign c=a&b;
- 在上面的赋值中，a,b,c三个变量皆为wire型变量，a和b信号的任何变化，都将随时反映到c上来

过程赋值语句

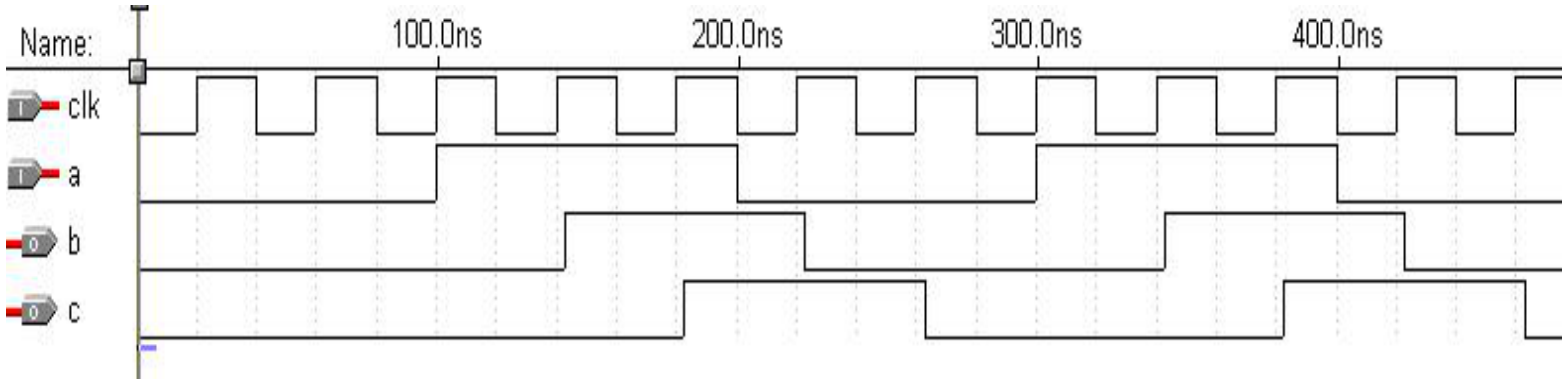
- 过程赋值语句多用于对reg型变量进行赋值。
- （1）非阻塞（non_blocking）赋值方式
- 赋值符号为”<=“，比如：b<=a; 非阻塞赋值在整个过程块结束时才完成赋值操作，即b的值并不是立刻就改变的。
- （2）阻塞（blocking）赋值方式
- 赋值符号为”=”，如：b=a; 阻塞赋值语句在该语句结束时就立即完成赋值操作，即b的值在该条语句结束后立刻改变。如果在一个块语句中，有多条阻塞赋值语句，那么在前面的赋值语句没有完成之前，后面的语句不能被执行，仿佛被阻塞了（blocking）一样，因此被称为是阻塞赋值方式。

阻塞赋值与非阻塞赋值

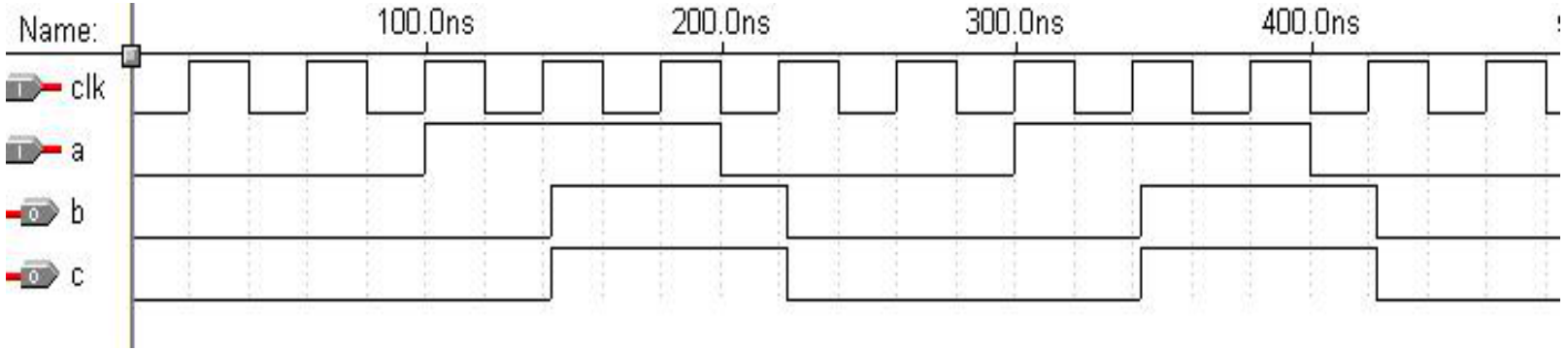
```
module non_block(c,b,a,clk);
  output c,b;
  input clk,a;
  reg c,b;
  always @(posedge clk)
  begin
    b<=a;
    c<=b;
  end
endmodule
```

```
module block(c,b,a,clk);
  output c,b;
  input clk,a;
  reg c,b;
  always @(posedge clk)
  begin
    b=a;
    c=b;
  end
endmodule
```

阻塞赋值与非阻塞赋值的仿真波形

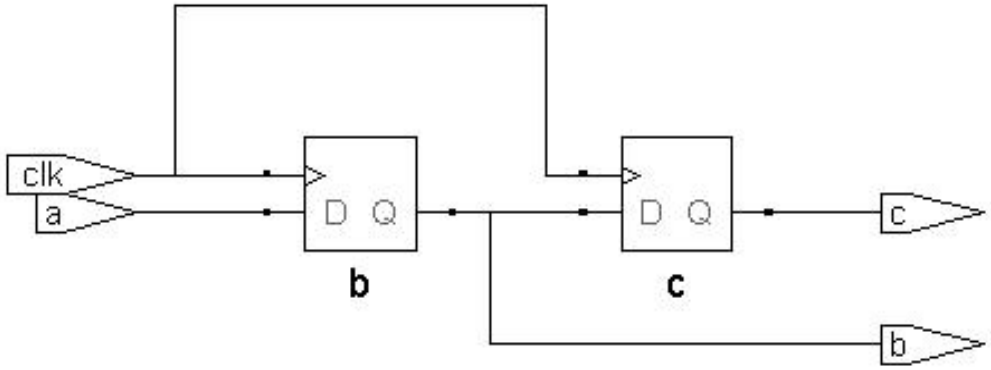


非阻塞赋值仿真波形图

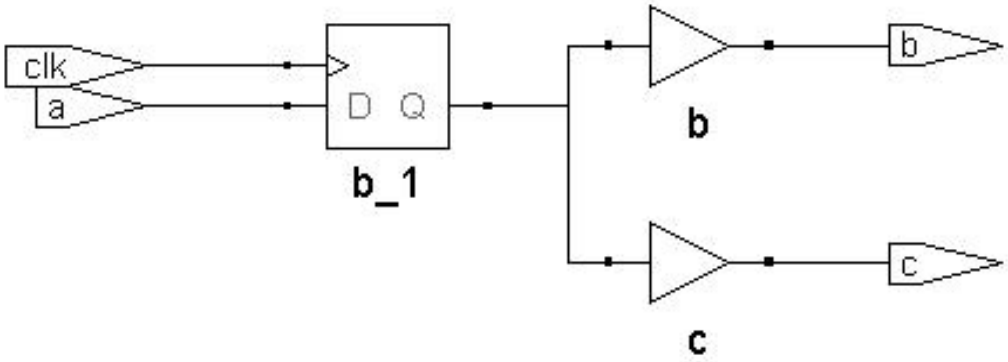


阻塞赋值仿真波形图

阻塞赋值和非阻塞赋值的综合结果



非阻塞赋值综合结果



阻塞赋值综合结果

赋值语句的遵从规则

- 在时序逻辑上使用非阻塞赋值
- 在组合逻辑上使用阻塞赋值
- 在组合逻辑上也可以使用连续赋值语句assign

- 在always过程块中使用阻塞赋值和非阻塞赋值需要仔细考虑不同赋值的特征

顺序执行与并发执行

- 两个或者多个always过程块，assign持续赋值语句，实例元件调用等操作都是同时执行的。
- 在always模块内部，其语句如果是非阻塞赋值，也是并发执行的；而如果是阻塞赋值，则语句是按照指定的顺序执行的，语句的书写顺序对程序执行结果有着直接的影响。

顺序执行的例子

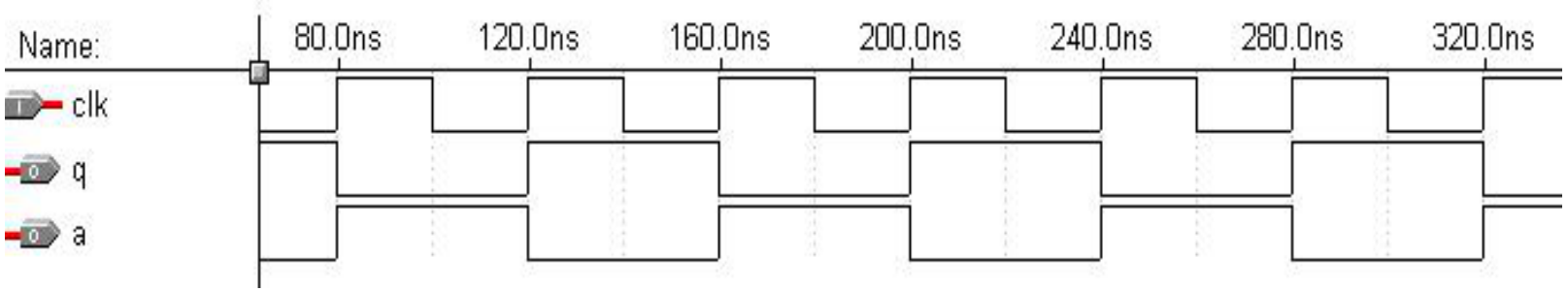
- 顺序执行模块1

```
module serial1(q,a,clk);
output q,a;
input clk;
reg q,a;
always @(posedge clk)
    begin
        q=~q;
        a=~q;
    end
endmodule
```

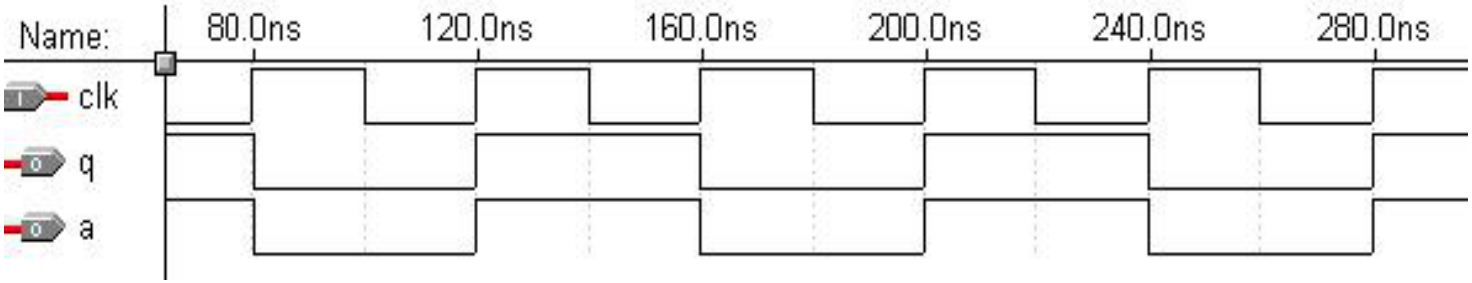
- 顺序执行模块2

```
module serial2(q,a,clk);
output q,a;
input clk;
reg q,a;
always@(posedge clk)
    begin
        a=~q;
        q=~q;
    end
endmodule
```

顺序执行的时序效果

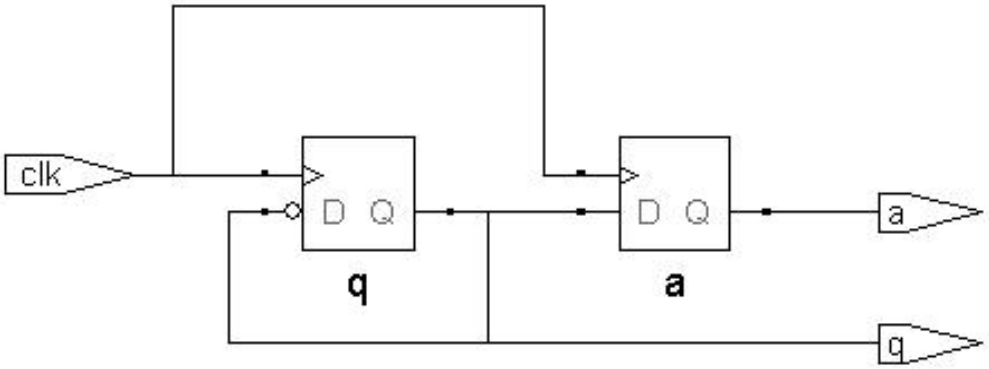


顺序执行模块1仿真波形图

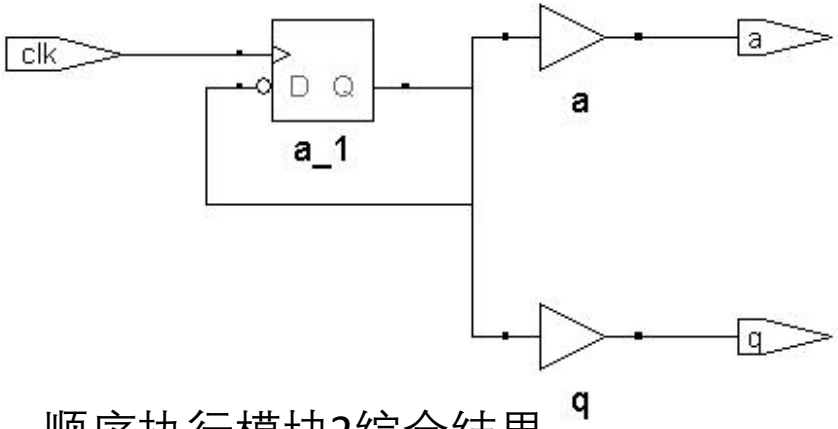


顺序执行模块2仿真波形图

顺序执行模块的综合结果



顺序执行模块1综合结果



顺序执行模块2综合结果

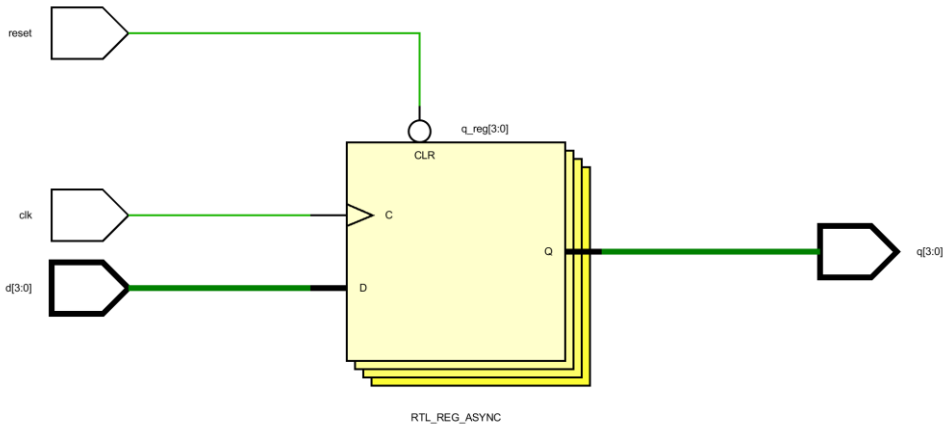
同步复位与异步复位

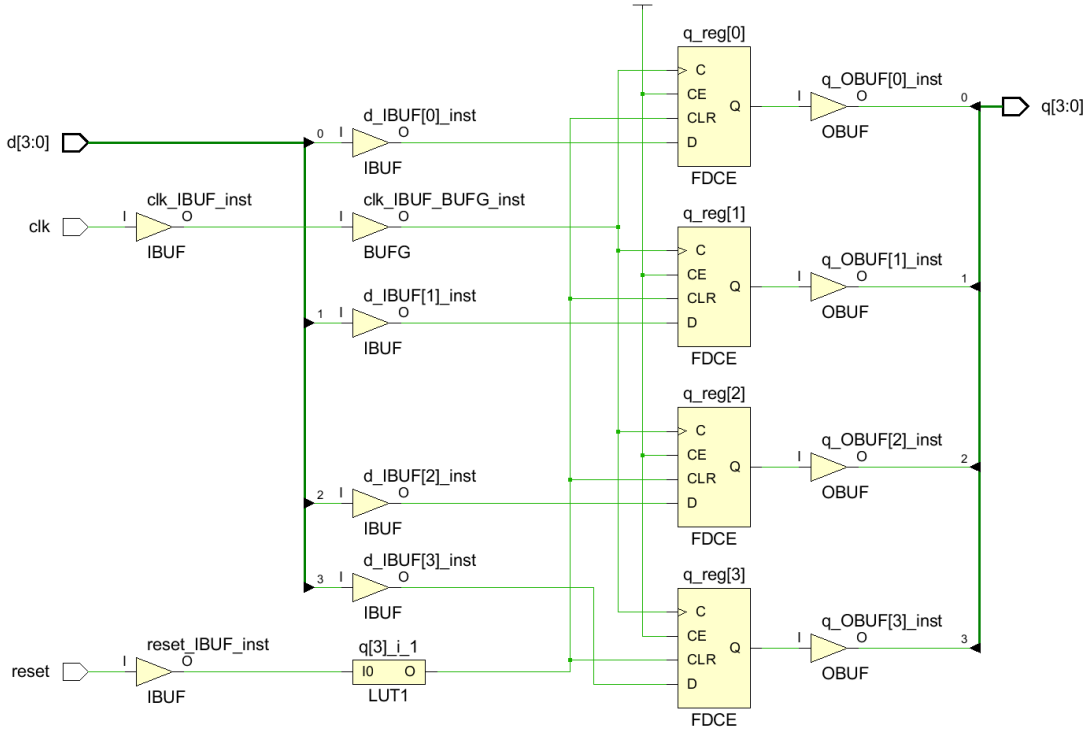
- 复位信号用于将硬件初始化到一个已知状态
  - 通常在系统启动时激活（上电时）
- **异步复位**
  - 复位信号的采样独立于时钟
  - 复位信号具有最高的优先权
  - 对毛刺敏感，可能会有亚稳态的问题
- **同步复位**
  - **复位信号是相对于时钟进行采样的**
  - 复位应该有足够长的激活时间，以便在时钟边沿采样成功。
  - 完全同步电路的结果

异步重置的触发器1

```
module flop_ar (input             clk,
                input             reset,
                input      [3:0] d,
                output reg [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
       if (reset == 0) q <= 0;   // when reset
       else            q <= d;   // when clk
    end
endmodule
```

- 在这个例子中：有两个事件可以触发这个过程语句
  - 在clk上有一个上升沿
  - 复位时的下降沿


reset
q_reg[3:0]
CLR
clk
C
Q
q[3:0]
d[3:0]
D
RTL_REG_ASYNC


d[3:0]
clk
reset
d_IBUF[0]_inst
IBUF
clk_IBUF_inst
IBUF
clk_IBUF_BUFG_inst
BUFG
d_IBUF[1]_inst
IBUF
d_IBUF[2]_inst
IBUF
d_IBUF[3]_inst
IBUF
reset_IBUF_inst
IBUF
q[3]_i_1
LUT1
q_reg[0]
FDCE
q_reg[1]
FDCE
q_reg[2]
FDCE
q_reg[3]
FDCE
q_OBUF[0]_inst
OBUF
q_OBUF[1]_inst
OBUF
q_OBUF[2]_inst
OBUF
q_OBUF[3]_inst
OBUF
q[3:0]

异步重置的触发器2

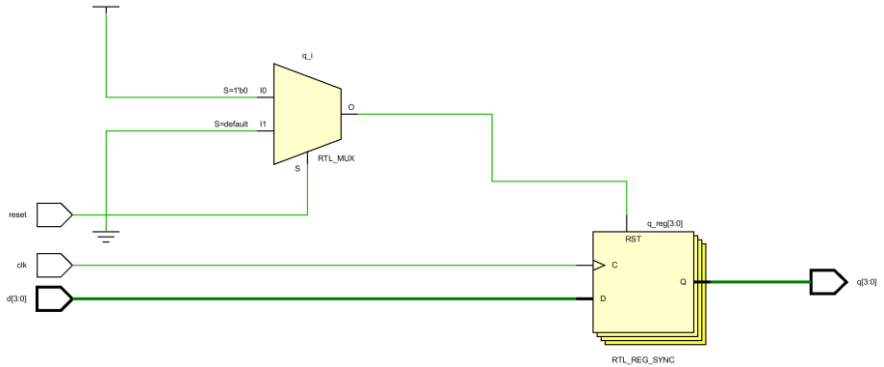
```
module flop_ar (input             clk,
                input             reset,
                input      [3:0] d,
                output reg [3:0] q);

  always @ (posedge clk, negedge reset)
    begin
      if (reset == 0) q <= 0;    // when reset
      else            q <= d;    // when clk
    end
endmodule
```

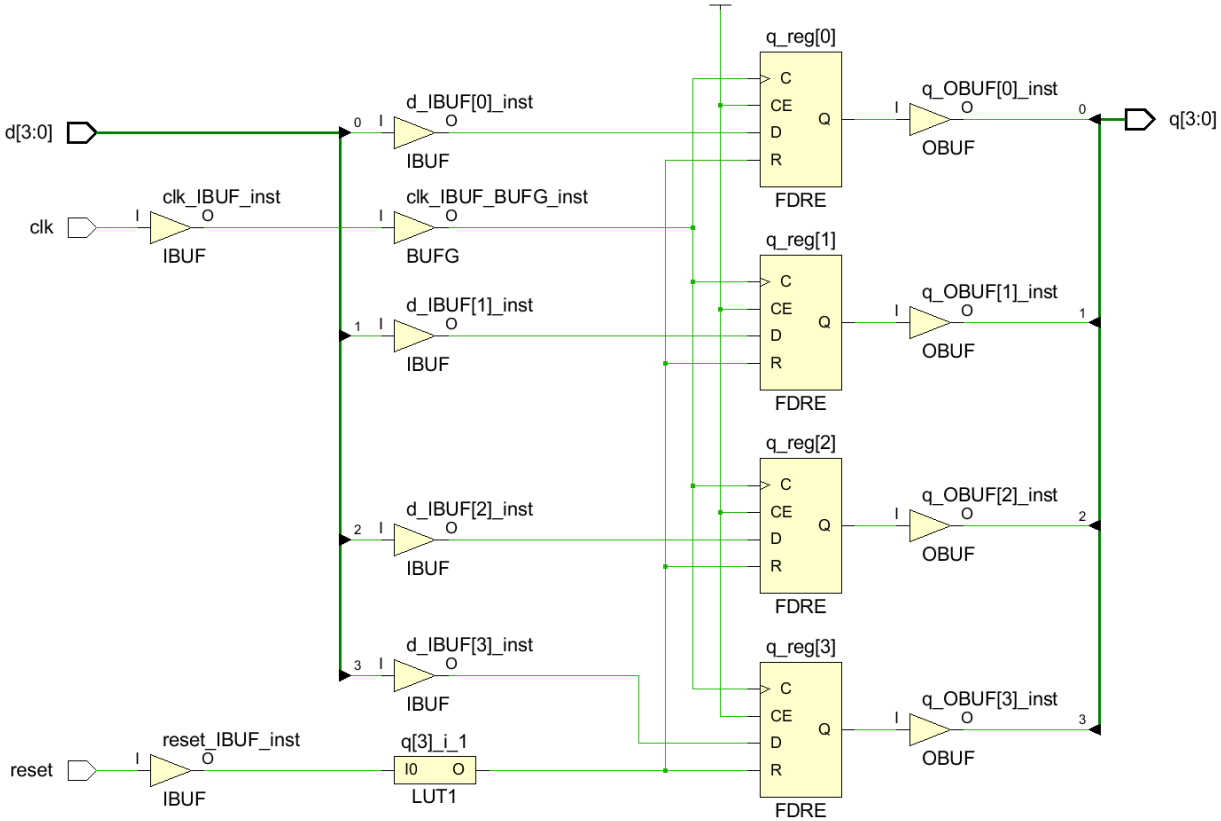
- 首先检查复位：如果复位为0，q被设置为0。
  - 这是一个异步复位，因为复位可以独立于时钟发生（在复位信号的负边沿）
- 如果没有复位，那么常规赋值就会生效

同步重置的触发器1

```
module flop_sr (input             clk,
                input             reset,
                input      [3:0] d,
                output reg [3:0] q);

  always @ (posedge clk)
    begin
       if (reset == 0) q <= 0;   // when reset
       else            q <= d;   // when clk
    end
endmodule
```

- 该过程只对时钟正边沿敏感
  - 复位只在时钟上升时发生，这是一个同步复位


q_i
S=1'b0
I0
S=default
I1
O
RTL_MUX
S
reset
clk
d[3:0]
q_reg[3:0]
RST
C
D
Q
q[3:0]
RTL_REG_SYNC


d[3:0]
clk
reset
d_IBUF[0]_inst
clk_IBUF_inst
clk_IBUF_BUFG_inst
d_IBUF[1]_inst
d_IBUF[2]_inst
d_IBUF[3]_inst
reset_IBUF_inst
q[3]_i_1
IBUF
BUFG
LUT1
I0
O
q_reg[0]
q_reg[1]
q_reg[2]
q_reg[3]
C
CE
D
R
Q
FDRE
q_OBUF[0]_inst
q_OBUF[1]_inst
q_OBUF[2]_inst
q_OBUF[3]_inst
OBUF
q[3:0]

always过程块

```
module example (input             clk,
                input      [3:0] d,
                output reg [3:0] q);

  wire [3:0] normal;          // standard wire
  reg  [3:0] special;         // assigned in always

  always @ (posedge clk)
    special <= d;             // first FF array

  assign normal = ~special; // simple assignment

  always @ (posedge clk)
    q <= normal;              // second FF array
endmodule
```

可以有多个always过程块

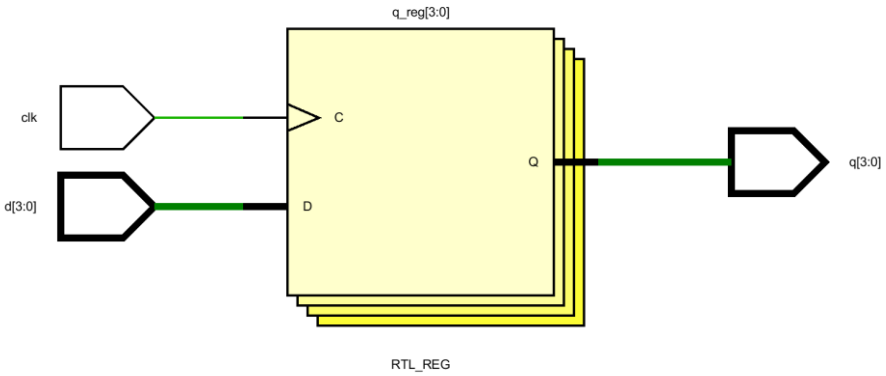
不允许在不同的always过程块中给相同的信号赋值

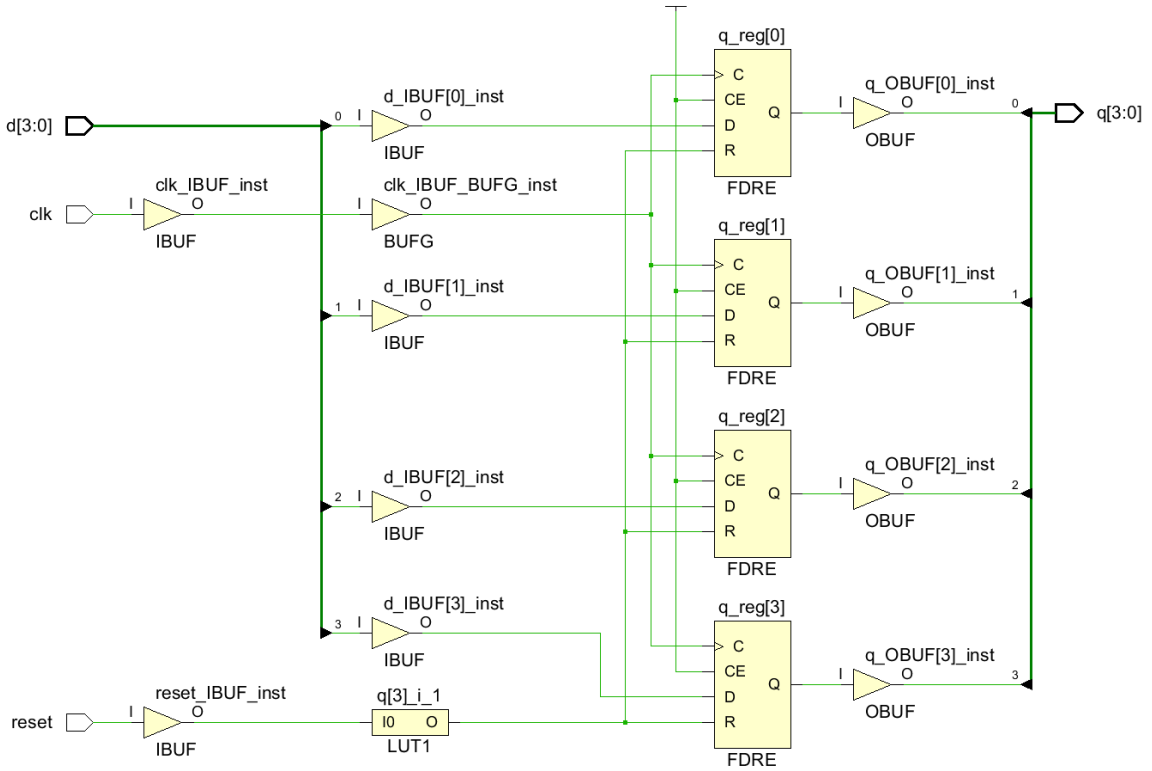
为什么?

always过程块的记忆

```
module flop (input             clk,
             input      [3:0] d,
             output reg [3:0] q);

  always @ (posedge clk)
    begin
        q <= d;   // when clk rises copy d to q
    end
endmodule
```

这个过程语句描述的是时钟正边沿时候的动作
如果时钟信号不是正边沿（高电平1，低电平0，或者负边沿）怎么办?
→ 保持不变：这就是过程块记忆实现时序逻辑

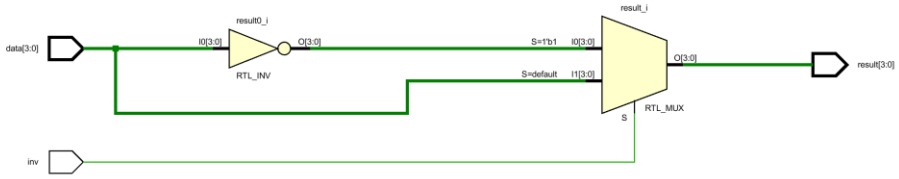

q_reg[3:0]
clk
C
d[3:0]
D
Q
q[3:0]
RTL_REG

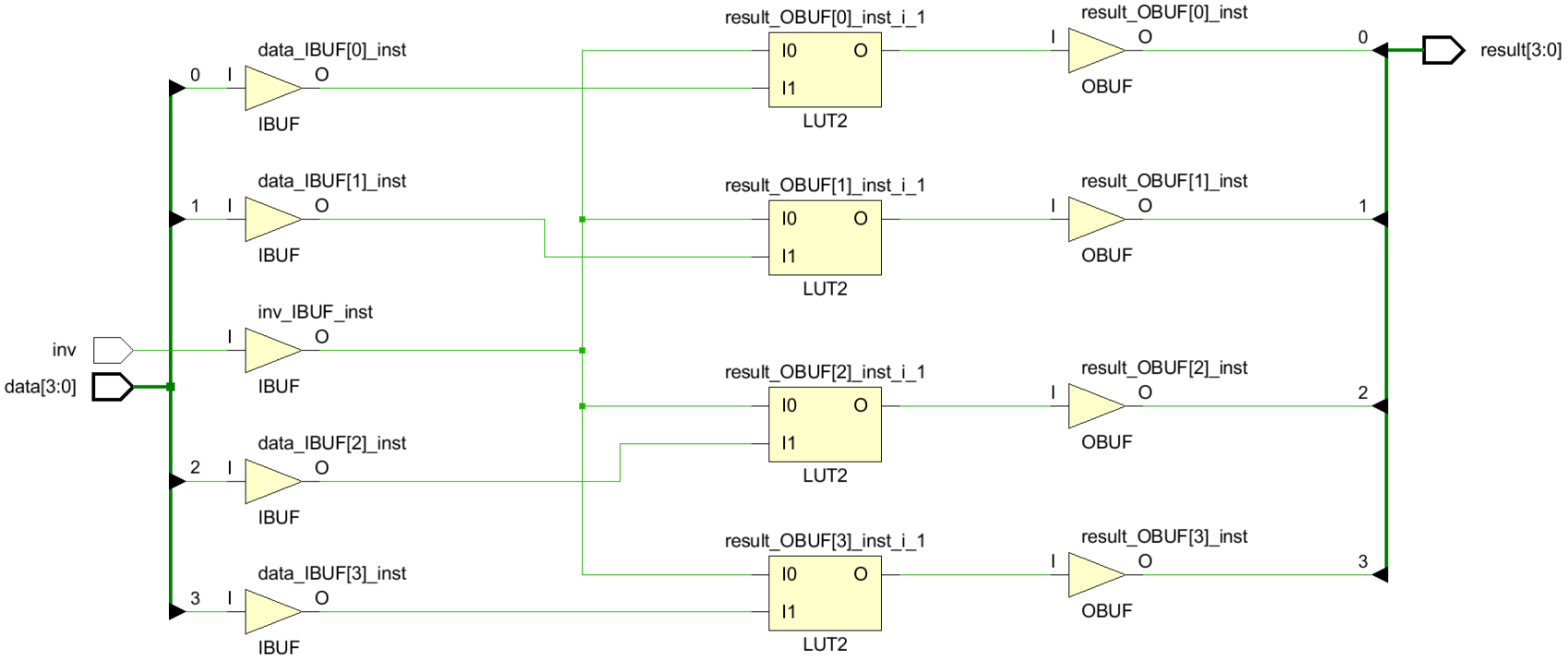

d[3:0]
clk
reset
d_IBUF[0]_inst
clk_IBUF_inst
clk_IBUF_BUFG_inst
d_IBUF[1]_inst
d_IBUF[2]_inst
d_IBUF[3]_inst
reset_IBUF_inst
q[3]_i_1
IBUF
BUFG
LUT1
q_reg[0]
q_reg[1]
q_reg[2]
q_reg[3]
FDRE
q_OBUF[0]_inst
q_OBUF[1]_inst
q_OBUF[2]_inst
q_OBUF[3]_inst
OBUF
q[3:0]

不带有记忆效应的always过程块

```
module comb (input             inv,
             input      [3:0] data,
             output reg [3:0] result);

  always @ (inv, data)       // trigger with inv, data
    if (inv) result <= ~data;// result is inverted data
    else     result <= data; // result is data

endmodule
```

上述过程块敏感信号是inv和data的电平，穷尽了所有的情况，没有记忆，是组合逻辑过程块

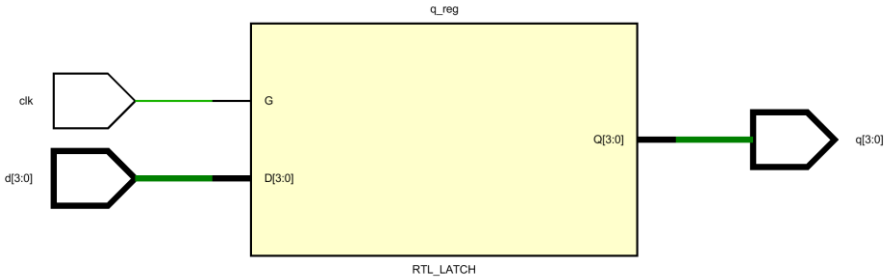

data[3:0]
result0_i
I0[3:0]
O[3:0]
RTL_INV
result_i
S=1'b1
I0[3:0]
S=default
I1[3:0]
O[3:0]
RTL_MUX
S
result[3:0]
inv

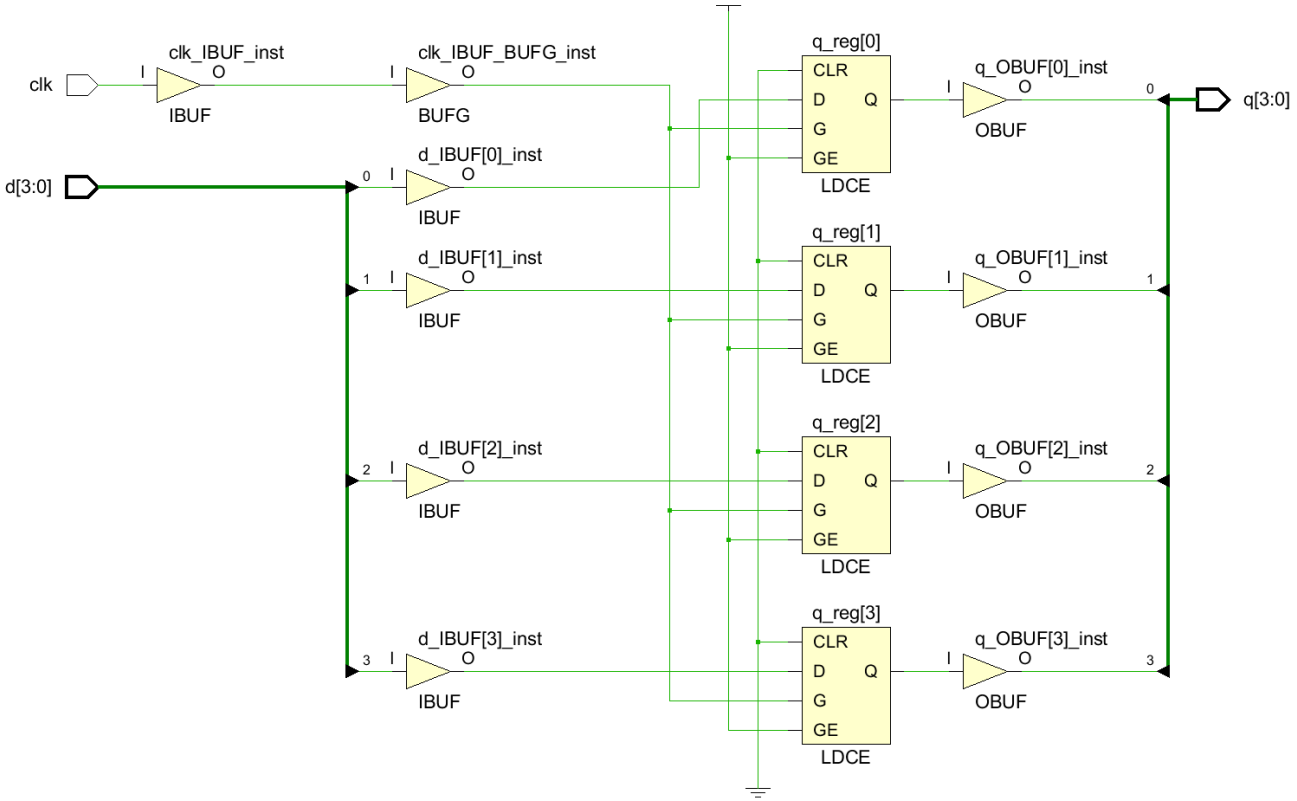

data_IBUF[0]_inst
data_IBUF[1]_inst
inv_IBUF_inst
data_IBUF[2]_inst
data_IBUF[3]_inst
IBUF
result_OBUF[0]_inst_i_1
result_OBUF[1]_inst_i_1
result_OBUF[2]_inst_i_1
result_OBUF[3]_inst_i_1
I0
I1
O
LUT2
result_OBUF[0]_inst
result_OBUF[1]_inst
result_OBUF[2]_inst
result_OBUF[3]_inst
OBUF
result[3:0]
inv
data[3:0]

锁存器

```
module latch (input             clk,
              input      [3:0] d,
              output reg [3:0] q);

  always @ (clk, d)
    if (clk) q <= d;       // latch is transparent when
                           // clock is 1
endmodule
```

这是电平响应，只规定了clk高电平时候的动作
else部分确实，代表这部分q值保持不变，被综合为锁存器


q_reg
clk
G
d[3:0]
D[3:0]
Q[3:0]
q[3:0]
RTL_LATCH


clk
clk_IBUF_inst
IBUF
clk_IBUF_BUFG_inst
BUFG
d[3:0]
d_IBUF[0]_inst
d_IBUF[1]_inst
d_IBUF[2]_inst
d_IBUF[3]_inst
q_reg[0]
q_reg[1]
q_reg[2]
q_reg[3]
CLR
D
G
GE
Q
LDCE
q_OBUF[0]_inst
q_OBUF[1]_inst
q_OBUF[2]_inst
q_OBUF[3]_inst
OBUF
q[3:0]

SystemVerilog描述always过程块不同属性

```
module flop (input       clk,
             input       [3:0] d,
             output reg [3:0] q);
  always_ff @ (posedge clk)
    begin
        q <= d;
    end
endmodule
```

```
module latch (input  clk,
              input [3:0] d,
          output reg[3:0] q);
  always_latch @ (clk, d)
    if (clk) q <= d;
endmodule
```

```
module comb (input             inv,
             input       [3:0] data,
             output reg [3:0] result);
  always_comb @ (inv, data)          // 没有敏感信号列表
    if (inv) result <= ~data;
    else     result <= data;
endmodule
```

在SystemVerilog中指定过程块的属性可以让系统帮助我们检查，预防出错
在我们的实验中不要出现latch，也不要同时响应时钟的上边沿和下边沿
（FPGA中无此器件）

使用always会不小心生成锁存器

```
wire enable, data;
reg out_a, out_b;

always @ (*) begin
        out_a = 1'b0;
        if(enable) begin
                out_a = data;
                out_b = data;
        end
end
```

No assignment for ~enable

Sequential

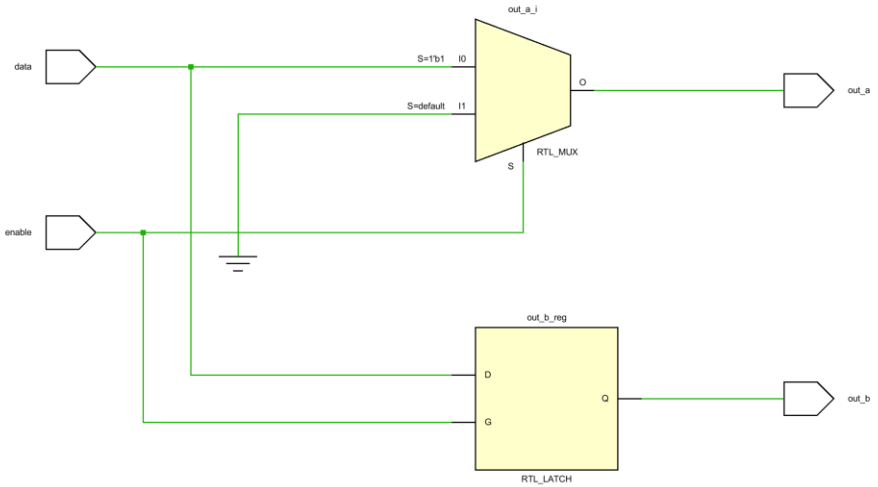
```
wire enable, data;
reg out_a, out_b;

always @ (data) begin
        out_a = 1'b0;
        out_b = 1'b0;
        if(enable) begin
                out_a = data;
                out_b = data;
        end
end
```

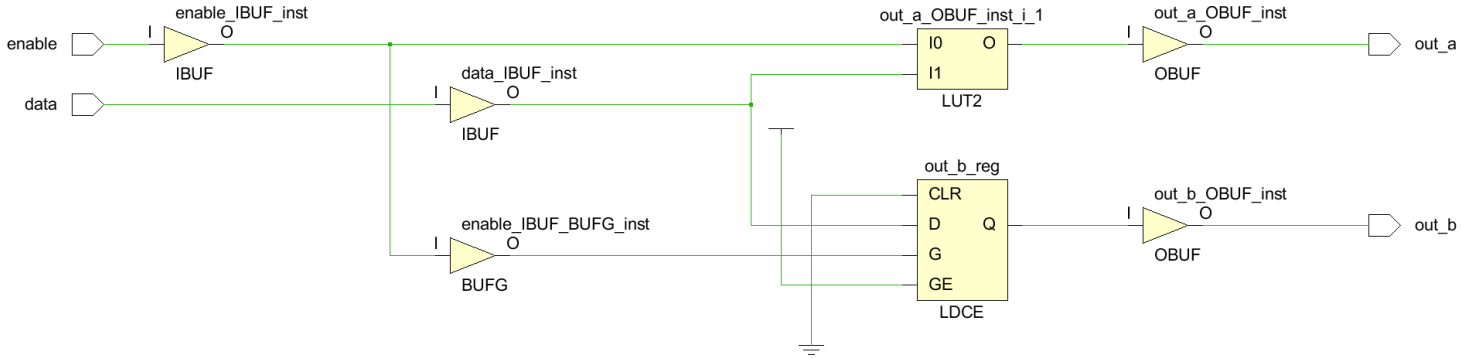
Not in the sensitivity list

Sequential

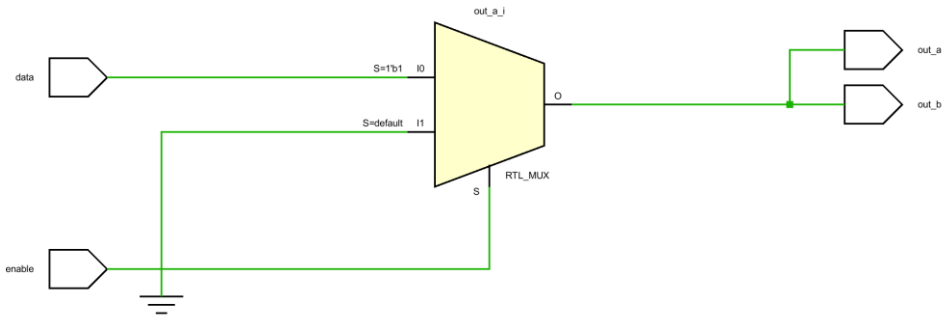
使用SystemVerilog就会收到检查错误的消息
确定自己实现组合逻辑，请使用always_comb

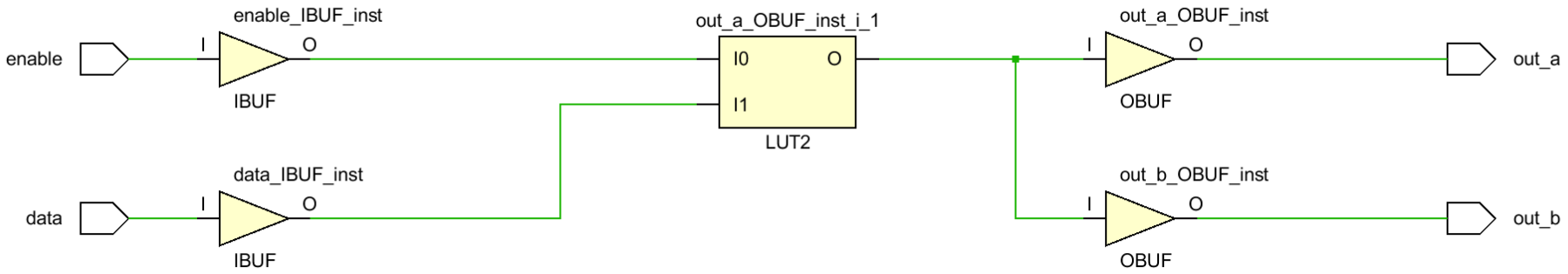
```
wire enable, data;
reg out_a, out_b;

always @ (*) begin
        out_a = 1'b0;
        if(enable) begin
                out_a = data;
                out_b = data;
        end
end
```





```
wire enable, data;
reg out_a, out_b;

always @ (data) begin
        out_a = 1'b0;
        out_b = 1'b0;
        if(enable) begin
                out_a = data;
                out_b = data;
        end
end
```





case语句

```
module sevensegment (input      [3:0] data,
                     output reg [6:0] segments);

  always @ ( * )                    // * is short for all signals
  always_comb
    case (data)                     // case statement
      4'd0: segments = 7'b111_1110;  // when data is 0
      4'd1: segments = 7'b011_0000;  // when data is 1
      4'd2: segments = 7'b110_1101;
      4'd3: segments = 7'b111_1001;
      4'd4: segments = 7'b011_0011;
      4'd5: segments = 7'b101_1011;
      // etc etc
      default: segments = 7'b000_0000; // required
    endcase

endmodule
```

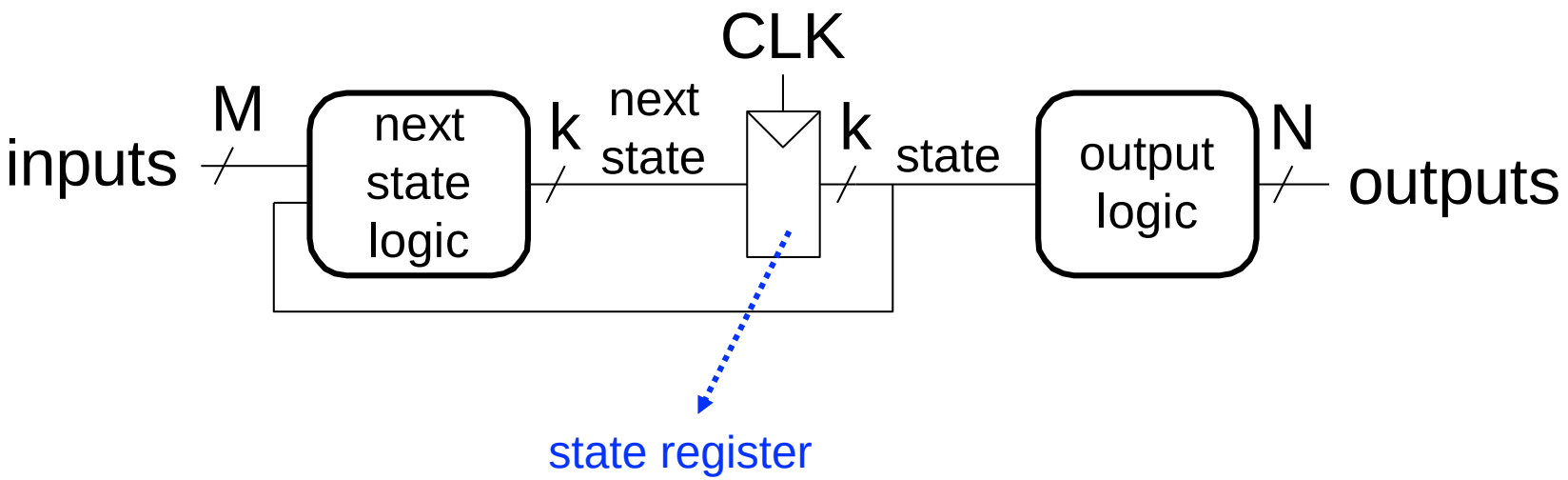
casex，casez使用通配符

```
always_comb
  begin
    {int2, int1, int0} = 3'b000;
    casez (irq)
      3'b1?? : int2 = 1'b1;
      3'b?1? : int1 = 1'b1;
      3'b??1 : int0 = 1'b1;
      default: {int2, int1, int0} = 3'b000;
    endcase
end
```

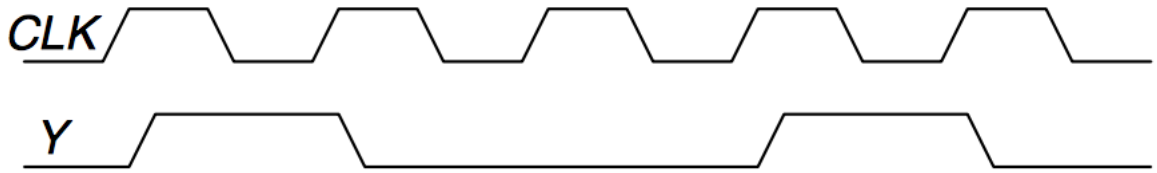
代码中禁止使用casex，通配的话用casez就好

FSM编程

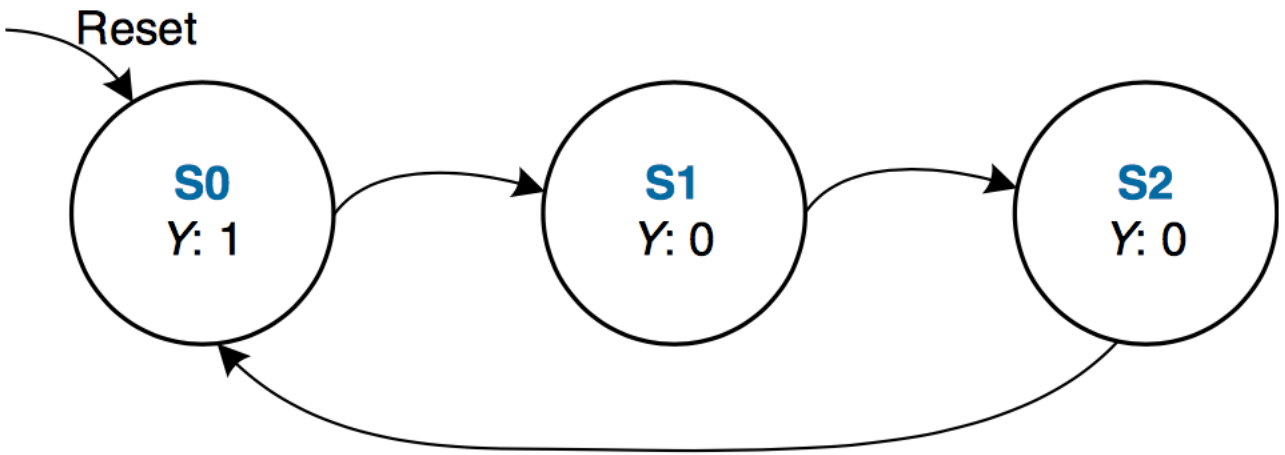
- 每个有限状态机包含三个部分
  - 现态逻辑（时序逻辑）
  - 次态逻辑（组合逻辑）
  - 输出逻辑（组合逻辑）



FSM举例：三分频率1



在每3个时钟周期中，输出Y为高电平。换句话说，该输出将时钟的频率除以3。



FSM举例：三分频率2

```
module divideby3FSM (input clk, input reset, output q);
   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00; parameter S1 = 2'b01; parameter S2 = 2'b10;

   always @ (posedge clk, posedge reset) // state register
      if (reset) state <= S0;
      else       state <= nextstate;

   always @ (*)                          // next state logic
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase
   assign q = (state == S0);             // output logic
endmodule
```

状态编码

- 常用的编码方式
- 顺序编码
- 格雷编码
- 约翰逊编码
- 一位热码

	State Variables		
State	**One-Hot Code**	**Binary Code**	**Gray Code**
S0	00001	000	000
S1	00010	001	001
S2	00100	010	011
S3	01000	011	010
S4	10000	100	110

Table 1:An example of state Encoding for a 4 state Machine

SystemVerilog

- SystemVerilog 中，一切都可以是 logic
- 不必显式区分 wire 和 reg
  - wire确实是连线，而reg不一定被综合成寄存器
- 在不同的环境下自动被推断为寄存器或组合逻辑
- 可以(几乎在)所有场合代替原有的 wire 和 reg

例子

```
logic [3:0] counter, counter_next;
logic counter_wrap;

// combinational logic
assign counter_wrap = counter == 4'b1111;

// also combinational logic
always_comb begin
  counter_next = counter + 1;
end

// sequential logic
always_ff @(posedge clk) begin
if (reset) begin
    counter <= 'b0;
  end
  else begin
    counter <= counter + 1;
  end
end
```

例子（错误）

```
logic not_latch, must_be_seq;

always_comb begin
  if (some_cond) begin
    not_latch = 1'b1;
  end
  // latch (no else) in always_comb -- synthesis error
end

always_ff @(clk) begin
  must_be_seq <= 1'b1;
  // no trigger edge in always_ff -- synthesis error
end
```

unique 与 priority

- Verilog 的 case 语句语义比较复杂，需要注解提示综合器(参见:full case, parallel case)，并且容易产生 latch。if 也可能会忘记书写分支，导致 latch， 或者产生意料之外的多个命中。
- SystemVerilog 中，新增了三个关键词 unique, unique0, priority，可以搭配 case 和 if 使用:
  - unique:只可能命中一项。仿真时如果命中多个或者没有命中将产生警告。
  - unique0:只可能命中至多一项(危险!)。仿真时如果命中多个将产生警告。
  - priority:可能命中多项，此时语句书写顺序作为匹配优先级。仿真时如果没有命中将产生警告。
- 注意:上述几个关键词并不会避免产生 latch，推荐总是将组合逻辑放置在 always_comb 中让综合器进行检查。

unique 与 priority(例子)

- 注意:如果需要通配符 ?，请使用 casez，不要使用 casex。此外，可综合的代码的 case 中不应当出现 X 和 Z。否则，可能出现非预期的结果。

```
// 4-bit priority decoder
// p: 4 bits input, d: 2 bits output
always_comb begin
  priority casez (p) begin
    4'b1???: {valid, d} = 3'b111;
    4'b01??: {valid, d} = 3'b110;
    4'b001?: {valid, d} = 3'b101;
    4'b0001: {valid, d} = 3'b100;
    default: {valid, d} = 3'b000;
  end
end
```

谢谢