

# FDU 计算机组成与体系结构 3. 硬件描述语言

---

本文参考以下教材：

- Digital Design and Computer Architecture (D. M. Harris, S. L. Harris 2rd) Chapter 4
- 数字设计和计算机体系结构 (D. M. Harris, S. L. Harris 2rd 陈俊颖译) 第 4 章

欢迎批评指正！

## 3.1 Introduction

---

### 3.1.1 模块 (Modules)

包含输入和输出的硬件块称为**模块** (module)

描述模块功能的主要形式有两种：**行为模型** (behavioral) 和**结构模型** (structural)  
前者描述一个模块做什么，后者层次化地描述一个模块怎样由更简单的部件构造。

模块体现的是**模块化** (modularity) 的设计思想。

它定义了由输入和输出组成的接口，并执行特定功能，  
而且其编码内容对于调用该模块的其他模块来说并不重要，只需要它执行自己的功能。

### 3.1.2 硬件描述语言 (HDL, Hardware Description Languages)

两种主流的硬件描述语言分别是 SystemVerilog 和 VHDL。

这两种语言都足以描述任何硬件系统。

不过前者的语句更简洁、灵活，因而更受工业界青睐。

Verilog 于 1990 年成为一个公开的标准，

并于 2005 年改进为 SystemVerilog 标准，其文件名通常以 `.sv` 结束。

### 3.1.3 模拟和综合 (Simulation and Synthesis)

硬件描述语言 (HDL) 的主要目的是逻辑模拟和综合。

- **(1) 模拟 (Simulation)**

在系统建立前进行逻辑模拟是很必要的，这样能以很小的代价诊断出硬件设计中的**漏洞** (bug)  
在模拟阶段，给模块增加输入，并检查输出以便验证模块设计是否正确。

- **(2) 综合 (Synthesis)**

逻辑综合将 HDL 代码转换成**网表** (netlist)，以描述硬件 (例如逻辑门以其连线)

逻辑综合可以进行优化以便减少硬件的数量。

它可以是一个文本文件，也可以以原理图的形式绘制出来，使电路可视化。

我们将把 HDL 代码分成**可综合模块** (synthesizable module) 和**测试程序** (testbench)。

可综合模块用于描述硬件，

测试程序包含将输入应用于模块的代码，检测输出结果是否正确，并输出期望结果与实际结果的差别。

测试程序代码只用以模拟，而不能用于综合。

初学者学习 HDL 的最好方法就是在例子中学习。

我们不会严格地定义所有的 HDL 语法，因为那是十分无聊的，

而且这样会使大家把 HDL 看作一个编程语言，而不是一个对硬件的描述。

## 3.2 组合逻辑 (Combinational Logic)

组合逻辑的输出只依赖于当前输入。

### 3.2.1 位运算符 (Bitwise Operators)

**位运算符**对单位信号和多位总线进行操作。

以 4 位反向器 (inverter) `inv.sv` 为例：

```
module inv(input logic [3:0] a,
          output logic [3:0] y);
    assign y = ~a;
endmodule
```

`a[3:0]` 代表一根 4 位的总线，  
它以**小端顺序**排序 (little-endian)，最低有效位 `a[0]` 具有最低地址 0。  
实际上，总线的**字节顺序** (endianness) 是任意选择的，  
我们也可以将总线命名为 `a[4:1]` 或 `a[0:3]` (**大端顺序**, big-endian)  
不过在本门课程里，我们将一直沿用**小端顺序**，即对于 `N` 位总线，记其顺序为 `[N-1:0]`

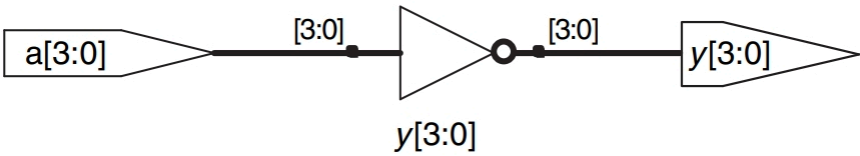


Figure 4.3 `inv` synthesized circuit

位运算符	描述
<code>&amp;</code>	按位与 (Bitwise AND)
<code> </code>	按位或 (Bitwise OR)
<code>^</code>	按位异或 (Bitwise XOR) (二者不同则为 1)
<code>~</code>	按位取反 (Bitwise NOT)
<code>~&amp;</code>	按位与非 (Bitwise NAND)
<code>~ </code>	按位或非 (Bitwise NOR)
<code>~^</code> / <code>^~</code>	按位同或非 (Bitwise XNOR) (二者相同则为 1)

```

module gates(input logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5, y6);

/* six 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
    assign y6 = a ~^ b;     // XNOR

endmodule

```

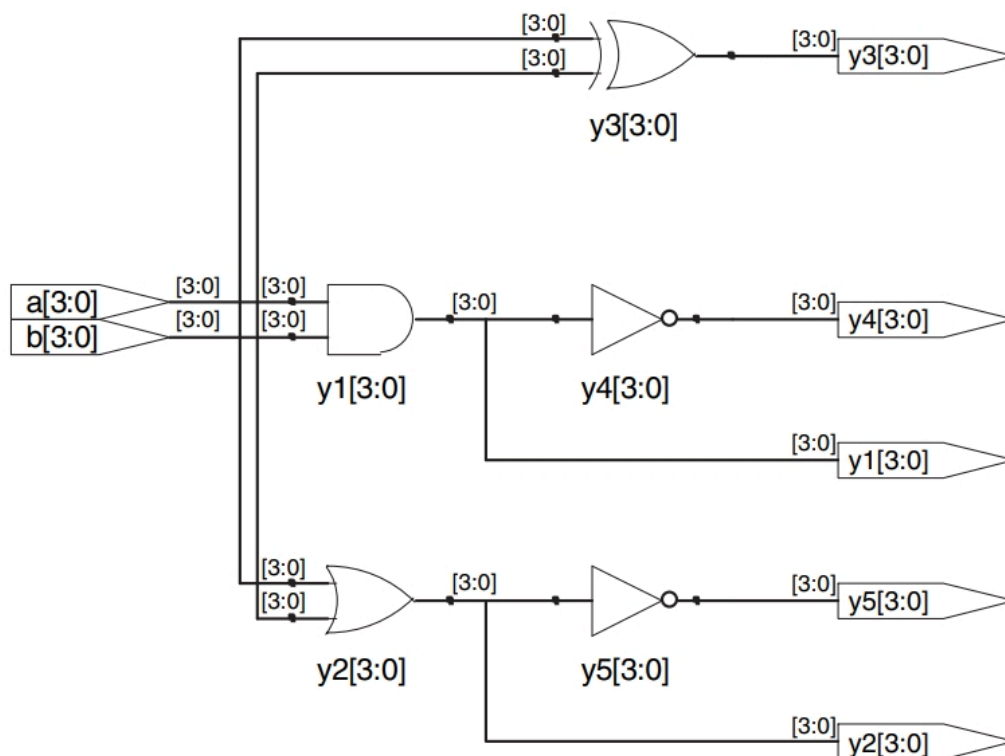
`assign out = in1 op in2;` 称为**连续赋值语句** (continuous assignment statement)

它以分号 `;` 结尾，用于描述组合逻辑。

一旦等号右边的输入值改变，等号左边的输出也会立即做出相应的改变。

HDL 赋值语句是**并行执行的**，因此赋值语句在模块中出现的顺序并不重要，这与传统的编程语言 (例如 C) 不同。

(下图尚未加入 `y6` 的修改)



**Figure 4.4** gates synthesized circuit

### 3.2.2 注释和空白 (Comments and White Space)

3.2.1 中 `gates` 的例子展示了如何注释。

它与 C 和 Java 一样，多行注释 `/* ... */`，单行注释 `//`

SystemVerilog 对空白 (即空格、Tab键、换行) 并不敏感，因此我们可以合理使用缩进排版和换行以增加代码可读性。

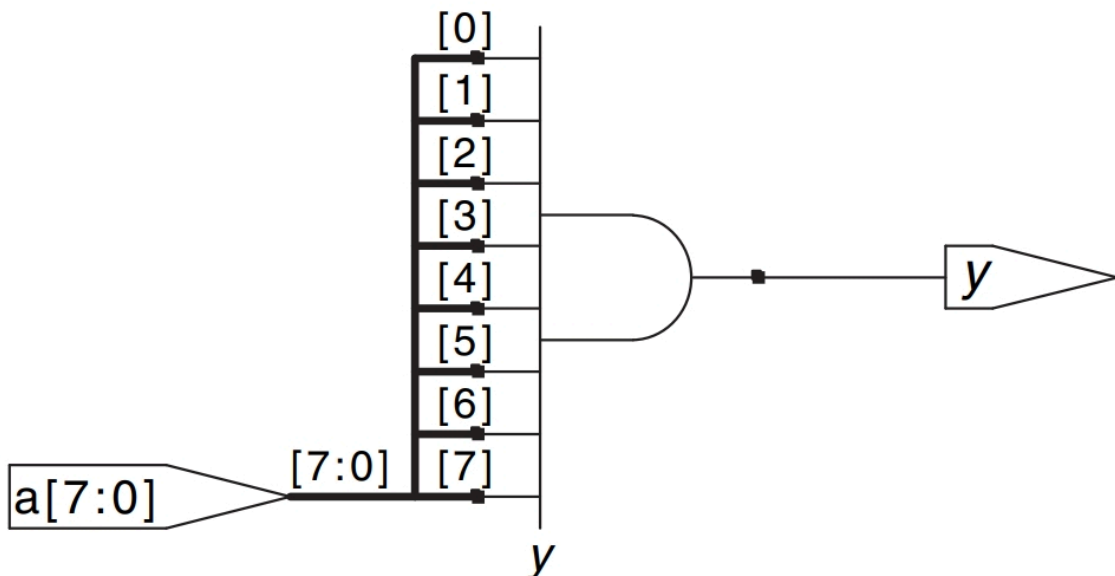
SystemVerilog 是**大小写敏感的** (case sensitive),  
但我们不建议只通过大小写的不同来区分不同的信号, 这容易造成混乱.  
注意在给信号和模块命名时, 使用大写字母和下划线, 并且不能以数字开头.

### 3.2.3 缩位运算符 (Reduction Operators)

缩位运算符表示作用在一条总线上的多输入门.

例如下面的例子描述了一个 8 输入与门:

```
module and8(input logic [7:0] a,  
            output logic 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
```



**Figure 4.5** and8 synthesized circuit

或门、异或门、与非门、或非门和同或门也有类似的缩位运算符.

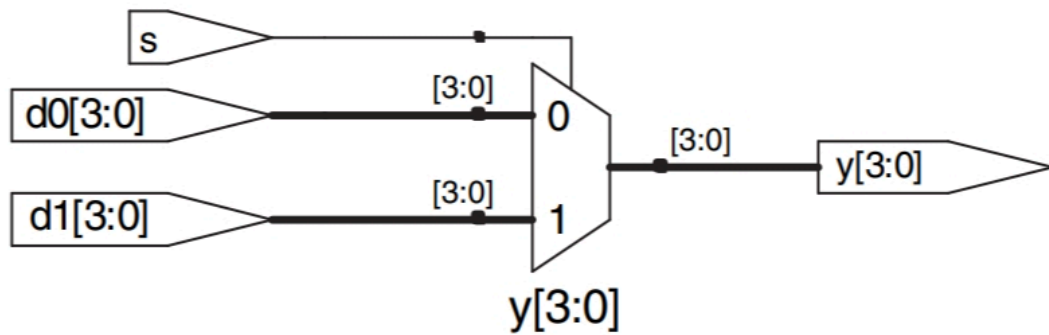
特别地, 当多输入异或门进行奇偶校验时, 如果奇数个输入为 `TRUE` 则返回 `TRUE`.

### 3.2.4 条件赋值 (Conditional Assignment)

**条件赋值**根据称为**条件**的输入, 在所有可选项中选择一个输出.

例如 2 : 1 复用器:

```
module mux2(input logic [3:0] d0, d1,  
            input logic s,  
            output logic [3:0] y);  
    assign y = s ? d1 : d0;  
endmodule
```



**Figure 4.6** mux2 synthesized circuit

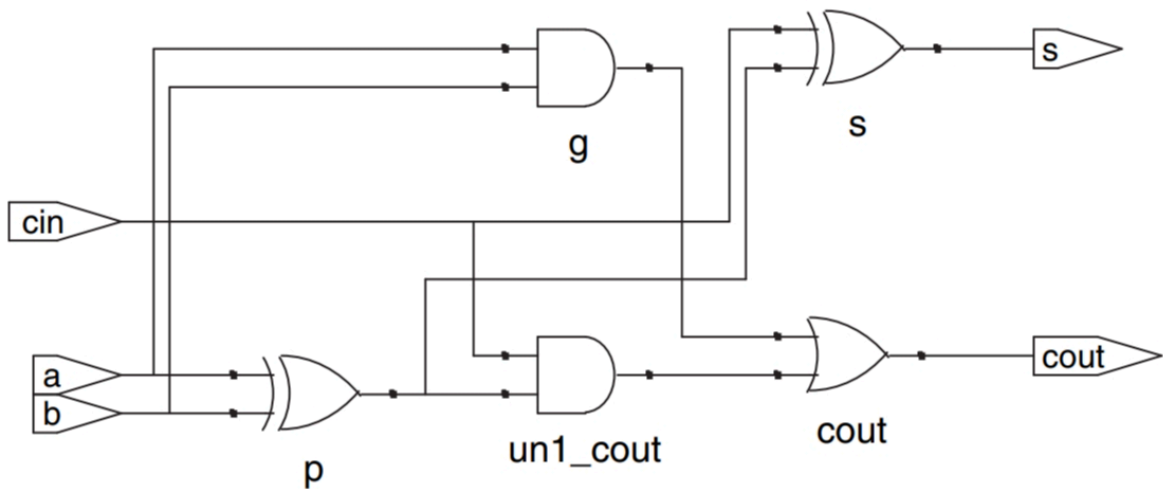
### 3.2.5 内部变量 (Internal Variables)

通常来说，把一个复杂功能分为几个中间过程来完成会更方便。

例如全加器 `fulladder.sv`：

```
module fulladder(input logic a, b, cin,
                 output logic s, cout);
    logic p, g;
    assign p = a ^ b;
    assign g = a & b;
    assign s = p ^ cin;
    assign cout = g | (p & cin);
endmodule
```

在 SystemVerilog 中，内部变量通常用 `logic` 变量类型声明。



**Figure 4.8** fulladder synthesized circuit

### 3.2.6 优先级 (Precedence)

Op	Meaning
<code>~</code>	NOT (非)
<code>*</code> , <code>/</code> , <code>%</code>	MUL, DIV, MOD (乘除模)
<code>+</code> , <code>-</code>	PLUS, MINUS (加减)
<code>&lt;&lt;</code> , <code>&gt;&gt;</code>	Logical Left/Right Shift (逻辑左移/逻辑右移)
<code>&gt;&gt;&gt;</code>	Arithmetic Right Shift (算术右移)
<code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code>	Relative Comparison (相对比较)
<code>==</code> , <code>!=</code>	Equality Comparison (相等比较)
<code>&amp;</code> , <code>~&amp;</code>	AND, NAND
<code>^</code> , <code>~^</code>	XOR, XNOR
<code> </code> , <code>~ </code>	OR, NOR
<code>?:</code>	Conditional

### 3.2.7 数字 (Numbers)

数字可以采用二进制 (b)、八进制 (o)、十进制 (d) 或者十六进制 (h) 来表示。  
我们还可以指定数字的位数，即在数字的开头将插入一些 0 来满足数字大小的要求。  
数字中间出现的下划线会被忽略，我们可以使用下划线将一些长的数字分成多个部分，从而加强可读性。

SystemVerilog 中声明常量的格式是 `N'Bvalue`

其中 `N` 是位数，`B` 指定基数，而 `value` 是值。

若没有出现基数，则默认是十进制；

若没有给出位数，则默认为当前表达式的位数，0 会自动填补在数字的前面以达到满位；

例如，给定 6 位总线 `w`，`assign w = 'b11` 会给 `w` 赋值 `000011`

然而有一个例外，我们规定 `'0` 和 `'1` 分别将全 0 和全 1 赋值给一条总线。

Numbers	Bits	Base	Value	Stored
3'b101	3	2	5	101
'b11	?	2	3	000...011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00...0101010

### 3.2.8 Z 和 X

SystemVerilog 信号值有 0, 1, *z*, *x* 四种.

在需要时, 以 *z* 或 *x* 开始的 SystemVerilog 常量都会用 *z* 或者 *x* 填充, 并填满位数.

*z* 表示**浮空值** (floating value), 而 *x* 表示**无效的逻辑电平** (invalid logic level)

在模拟时看到 *x* 和 *z* 值, 基本已经说明出现了漏洞或不正确的代码.

在综合后的电路中, 它们表示浮空的门输入、未初始化的状态或内容,

*x* 可能被电路随机地解释为 0 或 1, 导致不可预测的行为.

下表列出了 4 个可能信号值的与门真值表:

(注意, 即使有些输入是未知的, 与门也可以决定输出)

&	A			
B	0	1	Z	X
0	0	0	0	0
1	0	1	X	X
Z	0	X	X	X
X	0	X	X	X

### 3.2.9 位混合 (Bit Swizzling)

有时我们连接不同信号 (或者其子集) 来构成总线, 这些操作称为**位混合**

例如使用位混合操作给 *y* 赋 9 位值  $c_2c_1d_0d_0d_0c_0101$ :

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

其中 *{}* 操作符用于连接总线, *{3{d[0]}}* 表示 *d[0]* 的 3 个拷贝.

如果 *y* 的长度大于 9 位, 则将在较高位填充 0

### 3.2.10 延迟 (Delays)

HDL 语句可以与任意单位的延迟相关联。

这对于在模拟预测电路工作速度 (若指定了有意义的延迟)

和调试需要知道原因和后果时 (如果模拟结果中所有信号同时改变, 则推断一个错误输出的根源就非常棘手)

就显得很有用。

在综合时这些延迟会被忽略, 其门延迟并不由 HDL 代码中的数字决定。

下例给  $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$  的原始功能上增加了延迟。

假定反相器、3 输入与门、3 输入或门的延迟分别为 1ns, 2ns, 4ns

```
'timescale 1ns/1ps
module example(input logic a, b, c,
               output logic y);
    logic ab, bb, cb, n1, n2, n3;
    assign #1 {ab, bb, cb} = ~{a, b, c};
    assign #2 n1 = ab & bb & cb;
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3;
endmodule
```

其中时间尺度指令 `'timescale 1ns/1ps` 规定了这个文件中:

每个时间单位为 1ns, 而模拟精度为 1ps

如果文件中没有给出时间尺度指令, 则使用默认的单位 and 精度 (一般两者都是 1ns)

# 符号用于说明延迟单位的数量, 它可以放在 `assign` 语句中。

## 3.3 时序逻辑 (Sequential Logic)

HDL 综合器可以识别某种**风格** (idiom), 然后把它们转换成特定的时序电路。

其他的编码风格虽然可以正确地模拟, 但可能会把明显或不明显的错误综合到电路中。

### 3.3.1 寄存器 (Registers)

寄存器使用正边沿触发的 D 触发器。

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

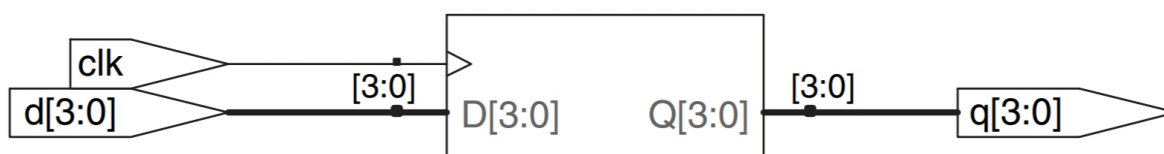


Figure 4.14 flop synthesized circuit

一般来说, SystemVerilog 的 `always` 语句具有以下形式:  
(敏感信息列表 `sensitivity list` 中的多个信号用 `,` 或 `or` 分隔)

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

只有当敏感信号列表中的信号说明事件发生时, 才执行 `statement`  
在这个例子中, 在下一个 `clk` 的上升沿到来前, `q` 都保持原来的值, 即使 `d` 在中途发生改变.  
这与组合逻辑中的 `assign` 连续赋值语句不同.  
在 `always` 语句中, **非阻塞赋值** `<=` (这时可以看作一个普通的 `=` 号) 代替了 `assign`.  
(我们在后面做更详尽的讨论)

在后面我们会看到,

`always` 语句可以用来表示触发器、锁存器或组合逻辑,  
具体功能取决于敏感信号列表和执行语句.

正因为这样的灵活性, 所以容易在不经意间制造出错误的硬件电路.

SystemVerilog 引入 `always_ff`, `always_latch` 和 `always_comb` 的记号来降低错误风险.

`always_ff` 的行为与 `always` 一样, 但只能用于表示触发器,  
如果它被用于表示其他器件, 则编程工具会生成相应的警告信息.

### 3.3.2 可复位寄存器 (Resettable Registers)

当模拟开始或者电路首次通电时, 触发器或者寄存器的输出是**未知的** (SystemVerilog 中由 `x` 表示)

一般来说, 应该使用复位寄存器, 这样在上电时可以把系统置于**已知状态**.

复位可以是**同步的** (synchronous), 也可以是**异步的** (asynchronous).

异步复位立即生效, 而同步复位只能在时钟的下一个上沿时才能生效.

- **异步复位 (asynchronous reset):**

```
module flopr(input logic      clk,
             input logic      reset,
             input logic [3:0] d,
             output logic [3:0] q);
    // asynchronous reset
    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 4'b0;
        else q <= d;
endmodule
```

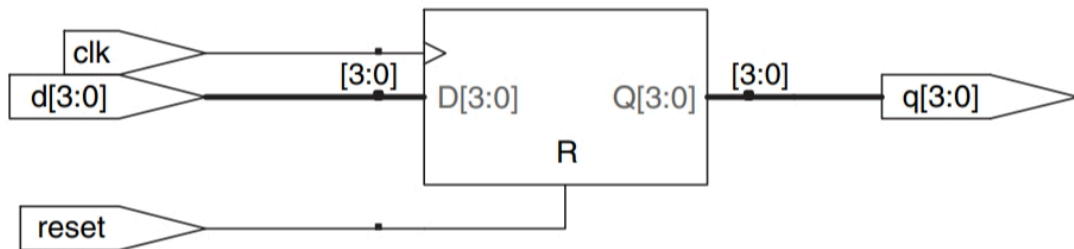
`posedge reset` 保证了异步复位触发器会立即响应 `reset` 的上升沿,  
但同步复位触发器只在时钟 `clk` 的上升沿时响应 `reset`

- **同步复位 (synchronous reset):**

```

module flopr(input logic clk,
             input logic reset,
             input logic [3:0] d,
             output logic [3:0] q);
    // synchronous reset
    always_ff @(posedge clk)
        if (reset) q <= 4'b0;
        else q <= d;
endmodule

```



(a)



(b)

**Figure 4.15** flopr synthesized circuit (a) asynchronous reset, (b) synchronous reset

### 3.3.3 带使能端的寄存器 (Enabled Registers)

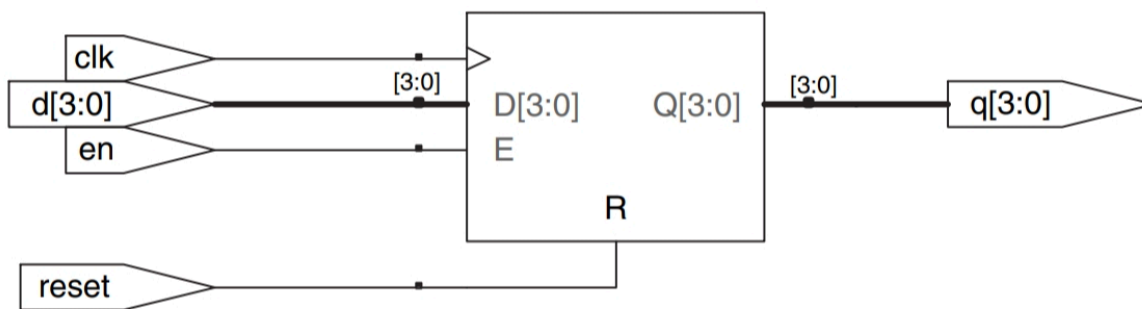
只有在使能有效时，带使能端寄存器才响应时钟。

下面的代码描述了一个异步复位使能寄存器：

```

module flopenr(input logic clk,
               input logic reset,
               input logic en,
               input logic [3:0] d,
               output logic [3:0] q);
    // asynchronous reset
    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 4'b0;
        else if (en) q <= d;
endmodule

```



**Figure 4.16** flopenr synthesized circuit

### 3.3.4 多寄存器 (Multiple Registers)

一条单独的 always 语句可以用于描述多个硬件。

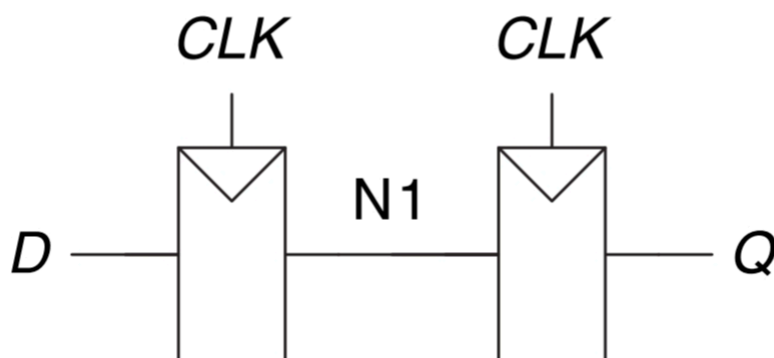
例如下面代码描述了**同步器** (synchronizer),

它由两个背靠背连接的触发器组成，用于减少或消除由于信号在不同时钟域之间传输时引起的亚稳态问题。

在 `clk` 的上升沿，将 `d` 复制到 `n1`，然后在下一个时钟周期，将 `n1` 复制到 `q`

```
module sync(input logic clk,
            input logic d,
            output logic q);
    logic n1;
    always_ff @(posedge clk)
    begin
        n1 <= d; // nonblocking
        q <= n1; // nonblocking
    end
endmodule
```

这里的 `begin/end` 结构是必要的，这与 C 语言中的 `{ }` 类似。



**Figure 4.17** Synchronizer circuit

### 3.3.5 锁存器 (Latches)

当时钟为 **high** 时，D 锁存器是透明的，允许数据从输入流向输出；

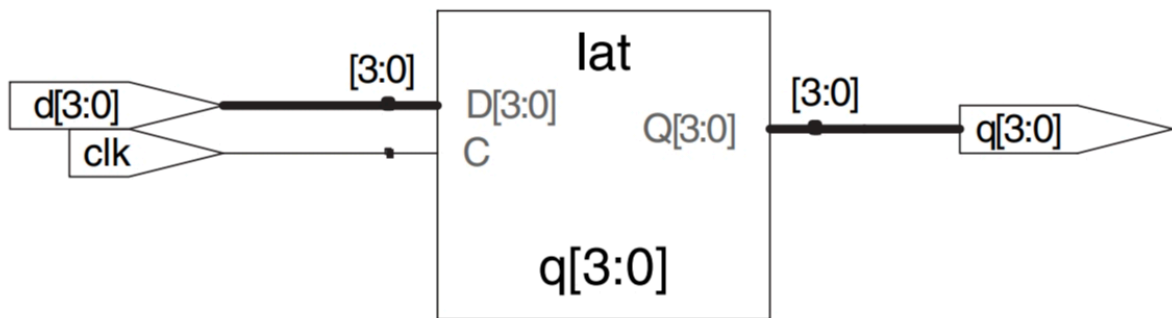
当时钟为 **low** 时，D 锁存器变为不透明的，保持原来的状态。

下面的代码描述了一个 D 锁存器：

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

`always_latch` 等同于 `always @(clk, d)`，它是 SystemVerilog 描述锁存器的首选风格。

如果 `always_latch` 模块不表示锁存器，则 SystemVerilog 将生成相应的警告。



**Figure 4.19** latch synthesized circuit

## 3.4 更多组合逻辑 (More Combinational Logic)

`always` 语句可以用于描述组合逻辑：

此时敏感信号列表包含对所有输入的响应，正文描述每一种可能输入组合的输出值。

`=` 在 `always` 语句中称为**阻塞赋值** (blocking assignment)，适用于组合逻辑；

一组阻塞赋值语句将以其在代码中出现的顺序来计算；

`<=` 在 `always` 语句中称为**非阻塞赋值** (non-blocking assignment)，适用于时序逻辑；

一组非阻塞赋值语句并行地计算 (因此也称为**并发赋值**)；

不要将 `assign` 的连续赋值与上述两种赋值混为一谈。

`assign` 语句不能出现在 `always` 语句内部，而且是并发计算。

下面的代码描述了一个 4 位反向器：

```
module inv(input logic [3:0] a,
           output logic [3:0] y);
    always_comb
        y = ~a;
endmodule
```

在这个例子中, `always_comb` 等价于 `always @(a)`  
但前者更好, 因为它避免了在 `always` 语句中由于信号改名或添加信号所带来的错误.  
总之, `always_comb` 等价于 `always @(*)`,  
它自动地依赖于所有在块内使用的变量, 而不需要手动指定依赖的信号.  
如果 `always_comb` 模块中的代码不是组合逻辑, 则 SystemVerilog 将产生警告信息.

### 3.4.1 case 语句

在 SystemVerilog 中, `case` 语句必须出现在 `always` 语句中.

`case` 句基于输入值执行不同的动作.

如果所有可能的输入组合都被定义, 则 `case` 语句就表示组合逻辑;  
否则, 它就表示时序逻辑, 因为输出会保持为未定义情况下的原来值.  
因此, 我们在用 `case` 语句表示组合逻辑时,  
总是应当加入 `default` 分支, 以确保所有可能的输入组合都被考虑到.

下面的代码描述了一个 3 : 8 译码器:

```
module decoder3_8(input logic [2:0] a,
                 output logic [7:0] y);
    always_comb
        case(a)
            3'b000: y = 8'b00000001;
            3'b001: y = 8'b00000010;
            3'b010: y = 8'b00000100;
            3'b011: y = 8'b00001000;
            3'b100: y = 8'b00010000;
            3'b101: y = 8'b00100000;
            3'b110: y = 8'b01000000;
            3'b111: y = 8'b10000000;
            default: y = 8'bxxxxxxxx;
        endcase
endmodule
```

就本例的逻辑综合而言, `default` 语句不是必须的, 因为我们已经定义了所有可能的输入组合.  
但是在模拟中需要考虑周全, 以防出现某个输入为  $x$  或  $z$  的情况.

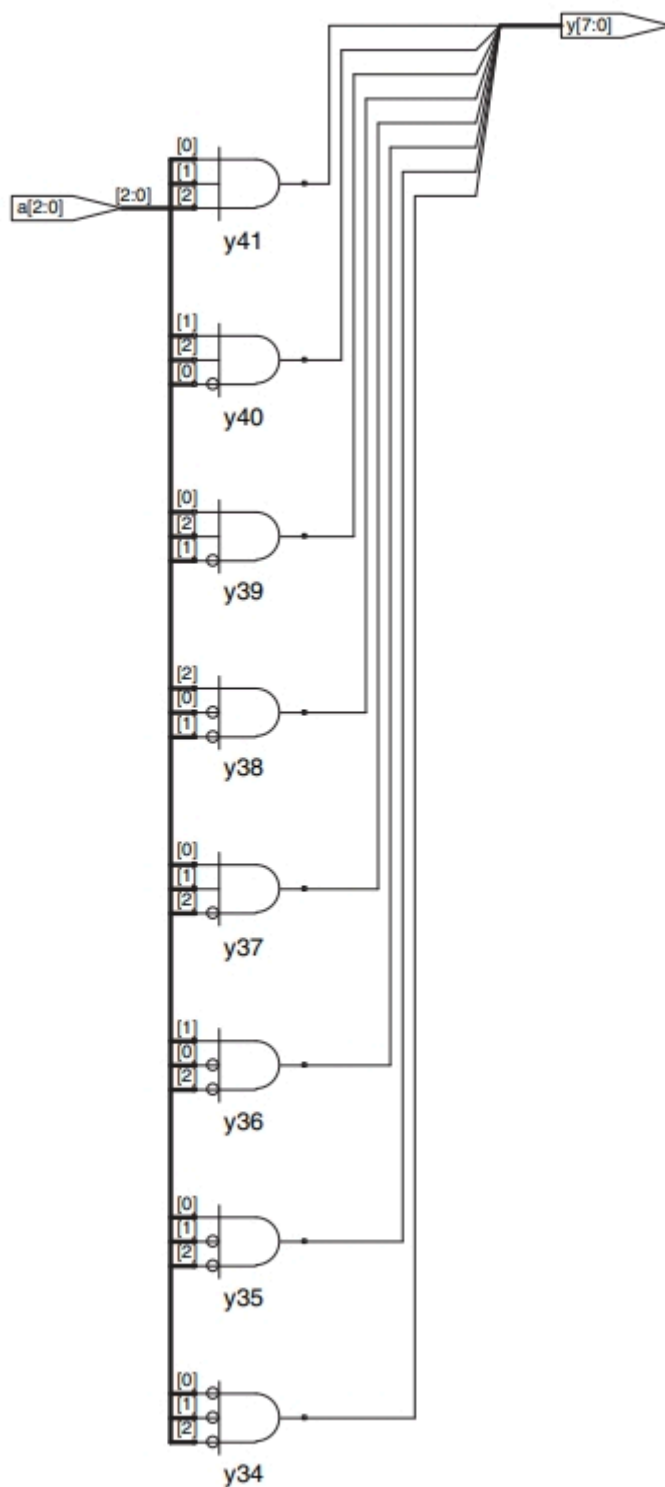


Figure 4.21 decoder3\_8 synthesized circuit

### 3.4.2 if 语句

在 SystemVerilog 中, `if/else` 语句必须出现在 `always` 语句中.

只有当所有可能的输入组合都处理了, 这条语句才表示组合逻辑;

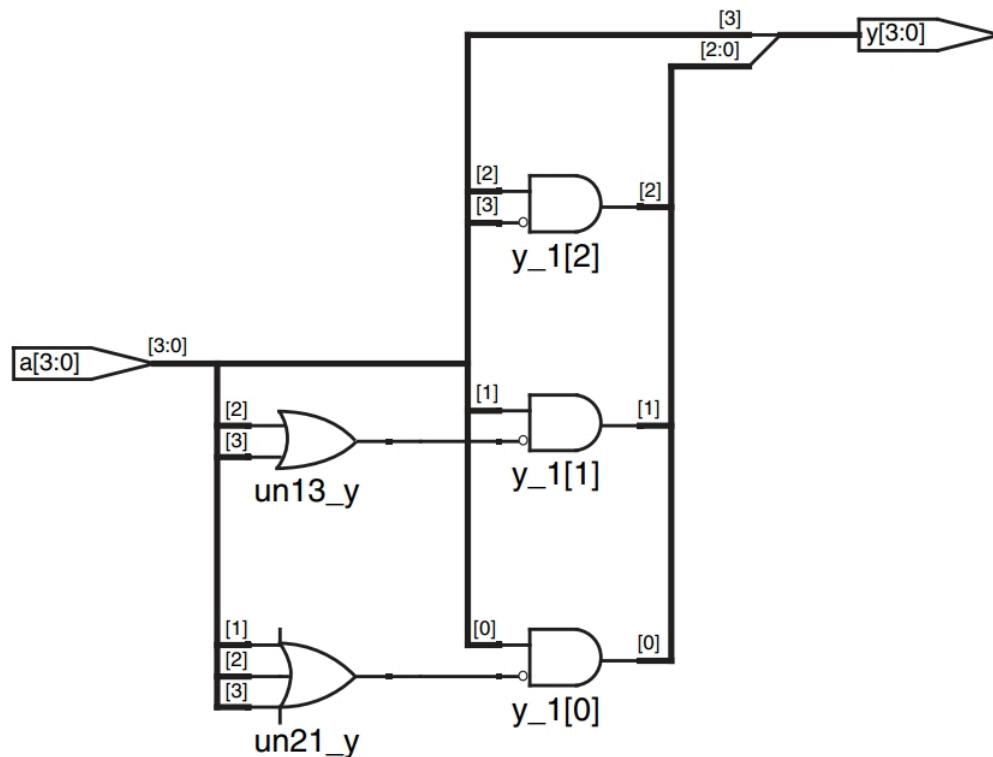
否则, 就产生时序逻辑 (类似 D 锁存器)

下面的代码描述了一个  $N$  输入优先电路, 它将对于的最高有效输入为 `TRUE` 的位设置为输出 `TRUE` :

```

module priorityckt(input logic [3:0] a,
                  output logic [3:0] y);
always_comb
    if (a[3]) y <= 4'b1000;
    else if (a[2]) y <= 4'b0100;
    else if (a[1]) y <= 4'b0010;
    else if (a[0]) y <= 4'b0001;
    else y <= 4'b0000;
endmodule

```



**Figure 4.22** priorityckt synthesized circuit

### 3.4.3 带有无关项的真值表 (Truth Tables with Don't Cares)

真值表中可能包含无关项从而提供更多逻辑简化可能.

`casez` 语句的作用与 `case` 语句一样, 当它能识别作为无关项的 ?

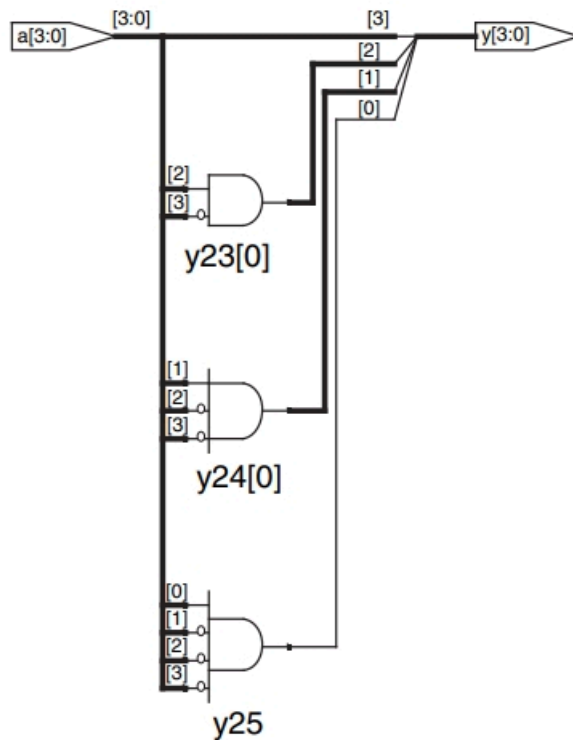
下面的代码使用无关项重新描述了 3.4.2 中 *N* 输入优先电路:

(综合出的电路与 3.4.2 中的稍有不同, 但它们是逻辑等价的)

```

module priority_casez(input logic [3:0] a,
                    output logic [3:0] y);
always_comb
    casez(a)
        4'b1??? : y <= 4'b1000;
        4'b01?? : y <= 4'b0100;
        4'b001? : y <= 4'b0010;
        4'b0001 : y <= 4'b0001;
        default : y <= 4'b0000;
    endcase
endmodule

```



**Figure 4.23** priority\_casex synthesized circuit

### 3.4.4 阻塞赋值和非阻塞赋值 (Blocking and Nonblocking Assignments)

下面我们给出使用不同赋值类型的准则。

如果不遵照这些准则使用，编写的代码就可能在模拟时正确，但综合到不正确的硬件。

- (1) 使用 `always_ff @(posedge clk)` 和非阻塞赋值 `<=` 描述同步时序逻辑：

```
always_ff @(posedge clk)
begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
end
```

- (2) 使用连续赋值 `assign` 描述简单组合逻辑：

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

- (3) 使用 `always_comb` 和阻塞赋值 `=` 描述复杂组合逻辑：

```
always_comb
begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
end
```

- (4) 不要多于 1 个 `always` 语句或 `assign` 语句中对同一个信号重复赋值。

## 3.5 数据类型 (Data Types)

在 SystemVerilog 出现之前, Verilog 主要使用两种类型: `reg` 和 `wire`。

尽管名字如此, 但 `reg` 信号不一定与寄存器相关联。

这对初学者来说是一个巨大的混淆根源。

SystemVerilog 引入 `logic` 类型来消除歧义。

在 Verilog 中, 如果信号出现在 `always` 模块中 `<=` 或 `=` 的左边,

那么它必须声明为 `reg`; 否则, 它应该声明为 `wire`。

因此, 一个 `reg` 信号可能是一个触发器、锁存器或者组合逻辑的输出,

取决于 `always` 模块的敏感信号列表和语句。

输入和输出端口默认为 `wire` 类型, 除非它们的类型被明确定义为 `reg`。

下面的代码展示了如何使用传统的 Verilog 来描述触发器:

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

其中 `clk` 和 `d` 默认为 `wire` 类型,

而 `q` 被明确定义为 `reg` 类型, 因为它出现在 `always` 模块中 `<=` 的左边。

SystemVerilog 引入 `logic` 类型来消除歧义。

`logic` 类型是 `reg` 类型的同义词, 以避免误导用户认为它是一个触发器。

而且, SystemVerilog 放宽了 `assign` 语句和分层端口实例的规则,

所以 `logic` 变量可以在 `always` 模块的外面使用,

而传统语法要求在 `always` 模块的外面使用 `wire` 变量。

因此, 几乎所有的 SystemVerilog 变量都可以是 `logic` 类型,

但也有例外, 如果信号有多个驱动源 (例如三态总线), 则必须声明为 `net` 类型。

当 `logic` 信号不小心连接到多驱动源时, 这个规则允许 SystemVerilog 生成错误信息而不是生成  $x$  值。

`net` 最常用的类型是 `wire` 和 `tri`, 分别用于单信号源驱动和多信号源驱动。

SystemVerilog 中 `wire` 类型已被废弃。

当 `tri` 变量被一个或多个信号源驱动为某个值时, 它就呈现那个值。

当 `tri` 变量未被驱动时, 它呈现浮空值  $z$ ;

当 `tri` 变量被多个信号源驱动为不同的值  $(0, 1, x)$  时, 它呈现不确定值  $x$ ;

其他较少使用的 `net` 类型 (它们可在任何使用 `tri` 类型的地方作为 `tri` 的替代):

Net Type	No Driver	Conflicting Drivers
tri	z	x
triereg	previous value	x
triand	z	0 if there are any 0
trior	z	1 if there are any 1
tri0	0	x
tri1	1	x

## 3.6 参数化模块 (Parameterized Modules)

目前为止，所有模块的输入和输出的宽度都是固定的。

通过使用参数化模块，我们可以灵活地改变位宽度。

下面的代码描述了参数化的  $N$  位 2 : 1 复用器：

```
module mux2 #(parameter width = 8)
    (input logic [width-1:0] d0, d1,
     input logic s,
     output logic [width-1:0] y);
    assign y = s ? d1 : d0;
endmodule
```

SystemVerilog 允许在输入、输出之前使用  `#(parameter ...)` 语句定义参数。

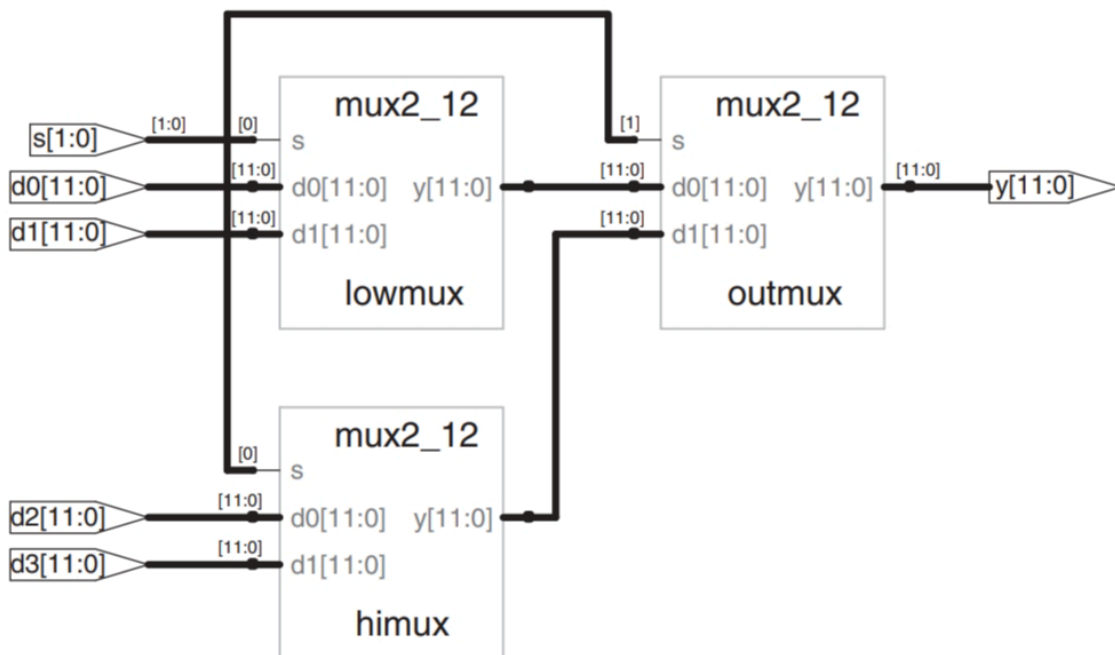
在本例中， `#(parameter width = 8)` 定义了一个默认值为 8 的参数 `width`

输入、输出的位数依赖于这个参数。

下面的代码使用  `mux2` 模块构建 12 位 4 : 1 复用器：

```
module mux4_12 (input logic [11:0] d0, d1, d2, d3,
               input logic [1:0] s,
               output logic [11:0] y);
    logic [11:0] low, hi;
    mux2 #(12) lowmux(d0, d1, s[0], low);
    mux2 #(12) himux (d2, d3, s[0], hi);
    mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

注意不要将参数设置  `#(12)` 与延迟 (参见 3.2.10)  `#12` 混淆。



**Figure 4.29** mux4\_12 synthesized circuit

HDL 还提供 `generate` 语句产生基于参数值的可变数量的硬件。它支持 `for` 循环和 `if` 语句来确定产生多少和什么类型的硬件。使用 `generate` 语句必须注意，它很容易不经意地生成大量的硬件。

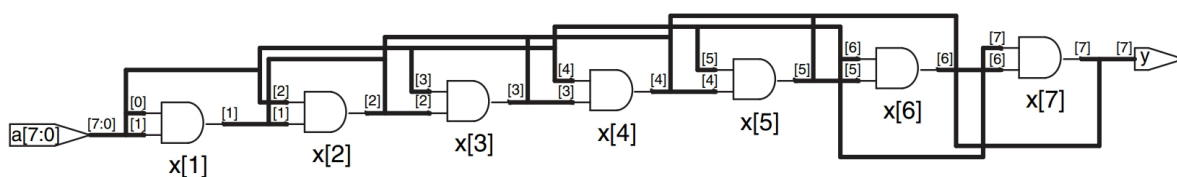
下面的代码描述了如何使用 `generate` 语句产生一个由 2 输入与门级联构成的  $N$  输入 AND 功能：

```
module andN #(parameter width = 8)
    (input logic [width-1:0] a,
     output logic y);
    genvar i;
    logic [width-1:0] x;

    generate
        assign x[0] = a[0];
        for(i=1; i<width; i=i+1) begin: forloop
            assign x[i] = a[i] & x[i-1];
        end
    endgenerate

    assign y = x[width-1];
endmodule
```

在 `generate for` 循环中的 `begin` 后面必须有 `:` 和一个任意的标识 (本例中是 `forloop`)



**Figure 4.30** andN synthesized circuit

## 3.7 测试程序 (Testbench)

**测试程序**是用于测试**被测设备** (DUT, Device Under Test) 的硬件描述语言模块。测试程序包含了向被测设备提供输入的语句，以便检查是否产生理想的正确输出。

输入和期待的输出模式称为**测试向量** (test vector)

我们可以将测试向量放在一个单独的文件中，  
测试程序简单地从文件中读取测试向量，向被测设备输入测试向量，  
检查输出值是否与期待的输出值一致，  
重复这个过程直到测试向量文件的结尾。

考虑计算  $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$  的 `sillyfunction` 模块。

下面的代码给出了一个自检测测试程序：

```
module testbench();
    logic a, b, c, y;

    // instantiate device under test
    sillyfunction dut(a, b, c, y);

    // apply inputs one at a time
    // checking results
    initial begin
        a = 0; b = 0; c = 0; #10;
        assert (y === 1) else $error("000 failed.");
        c = 1; #10;
        assert (y === 0) else $error("001 failed.");
        b = 1; c = 0; #10;
        assert (y === 0) else $error("010 failed.");
        c = 1; #10;
        assert (y === 0) else $error("011 failed.");
        a = 1; b = 0; c = 0; #10;
        assert (y === 1) else $error("100 failed.");
        c = 1; #10;
        assert (y === 1) else $error("101 failed.");
        b = 1; c = 0; #10;
        assert (y === 0) else $error("110 failed.");
        c = 1; #10;
        assert (y === 0) else $error("111 failed.");
    end
endmodule
```

其中 `initial` 语句只能在测试程序上用于模拟，而不能综合为实际硬件的模块。

在模拟开始时，`initial` 语句执行该段内的语句。

在本例中，它首先提供输入模式 `000`，然后等 10 个时间单位；

然后提供 `001`，等待 10 个时间单位，以此类推，直到提供了所有 8 个可能的输入。

`assert` 语句检查特定条件是否成立，如果不成立，则执行 `else` 语句。

`else` 语句中的 `$error` 系统任务用于输出描述 `assert` 错误的错误信息。

在综合过程中，`assert` 语句将被忽略。

在 SystemVerilog 中，可以在不包括  $x$  和  $z$  值的信号之间使用 `==` 或 `!=` 进行比较。

而测试程序分别使用 `===` 和 `!==` 运算符判断相等或不相等，以对包含  $x$  和  $z$  的运算数正确操作。

### 带测试文件的测试程序：

`example.tv` 是包含二进制格式输入和期待输出的文本文件：

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

下面的代码给出了一个带测试文件的测试程序：

(在时钟的上升沿向被测设备提供新的输入，在时钟的下降沿检查输出)

```
module testbench2();
    logic clk, reset;
    logic a, b, c, y, yexpected;
    logic [31:0] vectornum, errors;
    logic [3:0] testvectors[10000:0];

    // instantiate device under test
    sillyfunction dut(a, b, c, y);

    /* generate clock
       没有敏感信号列表的always语句产生一个时钟，这样它会连续不断地重复运行。
       虽然本例是组合逻辑测试，不需要时钟信号和复位信号，
       但它们也被包含在代码中，因为它们在测试时序 DUT 时是很重要的。 */
    always
    begin
        clk = 1; #5; clk = 0; #5;
    end

    /* at start of test, load vectors and pulse reset
       在模拟的开始，它从文本文件example.tv中读取测试向量，提供两个周期的reset脉冲
       $readmem将二进制数字文件读入testvectors数组中
       $readmemh与之类似，但它读取十六进制数字的文件 */
    initial
    begin
        $readmemb("example.tv", testvectors);
        vectornum = 0; errors = 0;
        reset = 1; #27; reset = 0;
    end

    /* apply test vectors on rising edge of clk
       在时钟的上升沿后等待一个时间单位（以防止时钟和数据同时改变造成的混乱）
       然后根据当前测试向量中的4位设置成3位输入（a,b,c）和期望的输出（yexpected） */
    always @(posedge clk)
    begin
        #1; {a, b, c, yexpected} = testvectors[vectornum];
    end

    /* check results on falling edge of clk
       测试程序将期望的输出yexpected与生产的输出y比较
```

如果它们不相等，则输出一条错误信息

`%b,%d,%h` 分别表示以二进制、十进制、十六进制输出值

这个过程重复到`testvector`数组中没有更多可用的测试向量

`$finish` 结束模拟 `*/`

```
always @(negedge clk)
    if (~reset) begin // skip during reset
        if (y != yexpected) begin // check result
            $display("Error: inputs = %b", {a, b, c});
            $display(" outputs = %b (%b expected)", y, yexpected);
            errors = errors + 1;
        end
        vectornum = vectornum + 1;
        if (testvectors[vectornum] === 4'bx) begin
            $display("%d tests completed with %d errors", vectornum, errors);
            $finish;
        end
    end
endmodule
```

THE END