2 逻辑电路与Verilog HDL简介

有组合逻辑和时序逻辑两种逻辑电路。**组合逻辑电路的输出只和当前的输入有关，时序逻辑电路的输出不仅和当前的输入也和之前的输入有关**

> *There are two types of the logic circuits: combinational circuits and sequential circuits. A combinational circuit can be defined as one whose outputs are dependent only on the present inputs. A sequential circuit can be defined as one whose outputs depend not only on the present inputs but also on the past history of inputs. it needs flip-plops to record the current state.*

英语

1. *algebra*代数
2. *parentheses*圆括号
3. *radix小数点*，基数
4. *vend*卖方、销售*vending machine*自动售货机
1. *inverter*反相器
2. *latch*门闩，锁存器
3. *flip-flop*触发器

2.1 逻辑门

七种逻辑门

与门、或门、非门、与非门、或非门、异或门、同或门

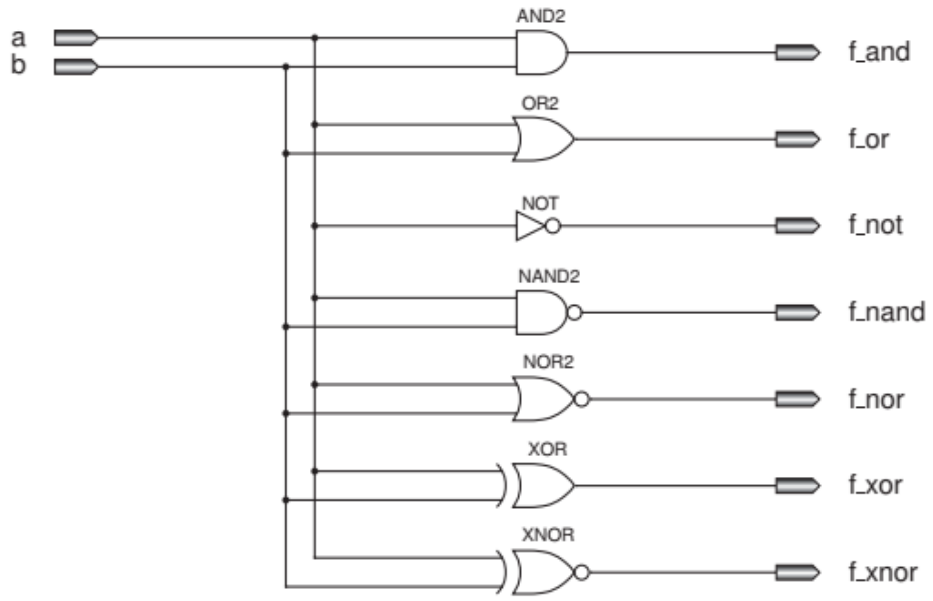


Figure 2.1 Three basic gates and four common gates

设计代码

```
`timescale 1ns / 1ps

module gates7_design_logicGates(
    input a,b,
    output f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor
);
    and and_init(f_and,a,b);
    or or_init(f_or,a,b);
    not not_init(f_not,a);
    nand nand_init(f_nand,a,b);
    nor nor_init(f_nor,a,b);
    xor xor_init(f_xor,a,b);
    xnor xnor_init(f_xnor,a,b);
endmodule
module gates7_design_dataflow(
    input a,b,
    output f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor
);
```

```
    assign f_and=a&b;
    assign f_or=a|b;
    assign f_not=~a;
    assign f_nand=~(a&b);
    assign f_nor=~(a|b);
    assign f_xor=a^b;
    assign f_xnor=~(a^b);
endmodule
```

Verilog

测试代码

```
`timescale 1ns / 1ps

module gates7_sim();
    reg a,b;
    wire f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor;
    wire f_and1,f_or1,f_not1,f_nand1,f_nor1,f_xor1,f_xnor1;
    initial begin
        a=0;b=0;
        #5 a=0;b=1;
        #5 a=1;b=0;
        #5 a=1;b=1;
        #5 $finish;
    end
    gates7_design_logicGates  gates7_design_logicGates_inst (
        .a(a),
        .b(b),
        .f_and(f_and),
        .f_or(f_or),
        .f_not(f_not),
        .f_nand(f_nand),
        .f_nor(f_nor),
```

```
        .f_xor(f_xor),
        .f_xnor(f_xnor)
    );
    gates7_design_dataflow  gates7_design_dataflow_inst (
        .a(a),
        .b(b),
        .f_and(f_and1),
        .f_or(f_or1),
        .f_not(f_not1),
        .f_nand(f_nand1),
        .f_nor(f_nor1),
        .f_xor(f_xor1),
        .f_xnor(f_xnor1)
    );
endmodule
```

Verilog

布尔代数定律和摩尔定律

$$1. A \cdot A = A$$

$$2. A + A = A$$

$$3. A \cdot B = B \cdot A$$

$$4. A + B = B + A$$

$$5. (A \cdot B) \cdot C = A \cdot (B \cdot C)$$

$$6. (A + B) + C = A + (B + C)$$

$$7. A \cdot (B + C) = (A \cdot B) + (A \cdot C)$$

$$8. A + (B \cdot C) = (A + B) \cdot (A + C)$$

$$9. A \cdot (A + B) = A$$

$$10. A + (A \cdot B) = A$$

$$11. A \cdot 0 = 0$$

$$12. A \cdot 1 = A$$

$$13. A + 0 = A$$

$$15. A \cdot barA = 0$$

16. $A + barA = 1$

17. $overlinebarA = A$

18. $overlineA \cdot B = barA + barB$

19. $overlineA + B = barA \cdot barB$

如何设计组合逻辑电路

设计组合逻辑电路的步骤如下：

1. 写真值表
2. 使用 Karnaugh 映射得到每个输出的简化表达式
3. 基于2得到的表达式设计逻辑电路
4. 通过仿真验证电路的正确性

下面以设置两路选择器为例来介绍：

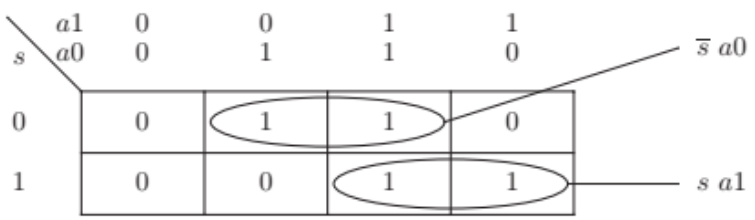
1. 真值表：

Table 2.2 Truth table of a 1-bit 2-to-1 multiplexer

Input			Output	Comment
s	$a1$	$a0$	y	
0	0	0	0	y is the same as $a0$
0	0	1	1	
0	1	0	0	
0	1	1	1	
1	0	0	0	y is the same as $a1$
1	0	1	0	
1	1	0	1	
1	1	1	1	

2. 使用 Karnaugh 映射得到每个输出的简化表达式

Karnaugh map卡诺图如下：



按最大的可能合并其中的“1”得到

$$y = barsa_0 + sa_1$$

3. 基于2得到的表达式设计逻辑电路

按照y=\bar{s} a_0+{s}a_1，设计组合逻辑电路，用到了非门、与门、或门

电路结构如下：

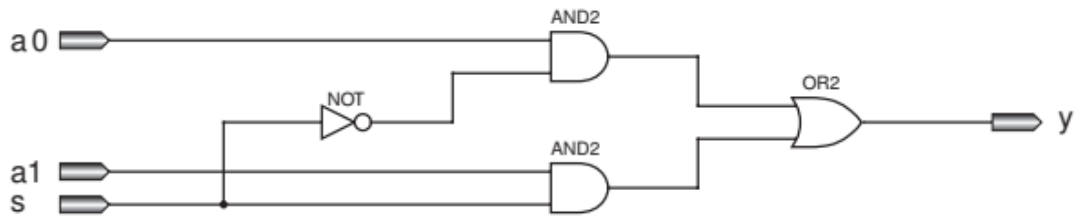


Figure 2.3 Schematic diagram of a 2-to-1 multiplexer

4. 设计Verilog HDL代码——见后2.3 四种Verilog设计风格

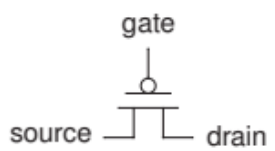
1. *Following ‘timescale, the first 1ns defines time unit and the second 1ns defines the minimum time precision we can use in the simulation.*
2. *Combining with the variable $time, $monitor makes the test bench code shorter*

```
$monitor ("%2d:\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b",
          $time,a,b,f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor);
```

2.2 CMOS门

2.2.1 CMOS反相器

PMOS

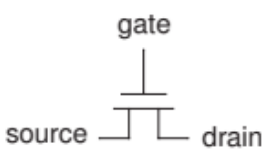


(a) PMOS transistor

当gate为0时，source→drain

For PMOS transistors, as shown in left, if the gate input is a 0, the switch is on, and the voltage of the drain is the same as that of the source; otherwise it is off.

NMOS

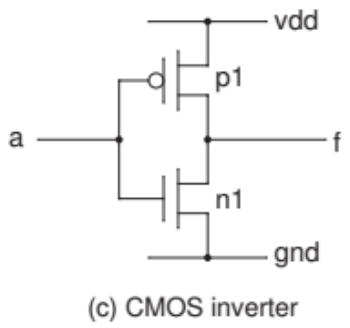


(b) NMOS transistor

当gate为1时，source→drain

For the NMOS, as shown in left, if the input gate is 1, the transistor is on, and the voltage of drain is the same as that of source; otherwise the transistor is off.

CMOS反相器



(c) CMOS inverter

CMOS反相器如左图所示，是PMOS和NMOS的集成

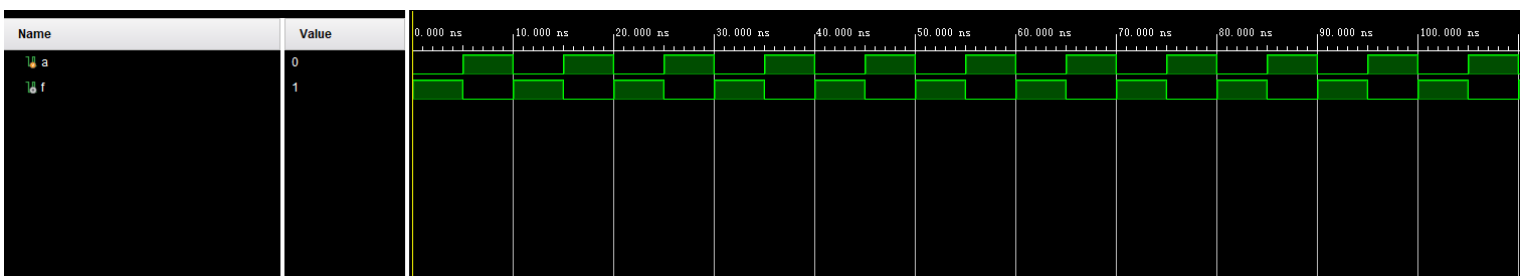
用MOS晶体管实现CMOS反相器

设计代码

```Verilog
`timescale 1ns/1ps

module cmos_inverter(
    input a,
    output f
);
    supply1 vdd;//高电平
    supply0 gnd;//接地端

    // Verilog已实现pmos(drain,source,gate);
    pmos pmos_1(f,vdd,a);
    // Verilog已实现nmos(drain,source,gate);
    nmos nmos_1(f,gnd,a);

endmodule
```

测试代码

```Verilog
`timescale 1ns / 1ps

module cmos_inverter_test();
    reg a;
    wire f;
    initial begin
        a=0;
    end
    always #5 a=~a;

    cmos_inverter  cmos_inverter_inst (
        .a(a),
        .f(f)
    );

    always @(*) begin
        if ($time>=1000) begin
            $finish;
        end
    end
endmodule
```



2.2.2 CMOS与非门和CMOS或非门

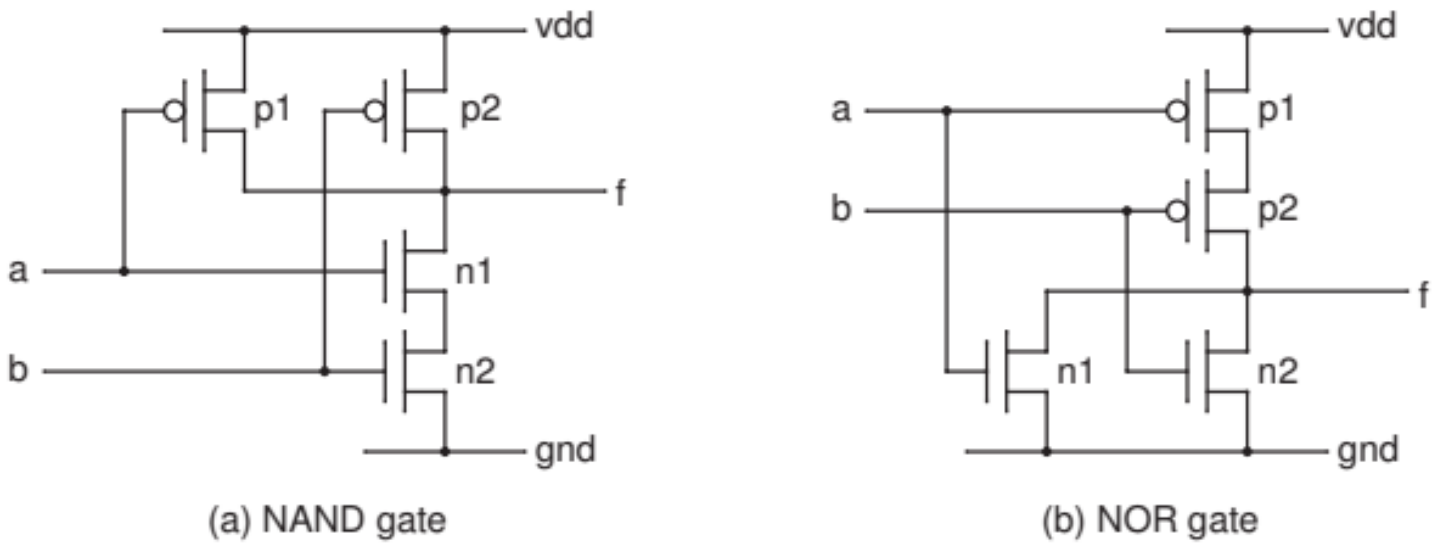


Figure 2.6 Schematic diagram of NAND and NOR gates

根据电路图：

NAND的p1、p2的drain端均是f，source端均是vdd，p1的gate是a，p2的gate是b；n2的gate是b，source是gnd，drain是n1的source，n1的gate是a，n1的drain是f。因此**需要设置一个中间变量，实现n2 drain→n1 source**

NOR的思想同NAND，只不过**中间变量设计在PMOS门间，p1的drain→p2的source**

设计代码

cmos_nand:

```
`timescale 1ns / 1ps

module cmos_nand(
    input a,b,
    output f
);
    supply1 vdd;
    supply0 gnd;
    wire temp;
    pmos pmos1(f,vdd,a);
    pmos pmos2(f,vdd,b);
    nmos nmos1(f,temp,a);
```

```
    nmos nmos2(temp,gnd,b);
endmodule
```
Verilog

cmos_nor:

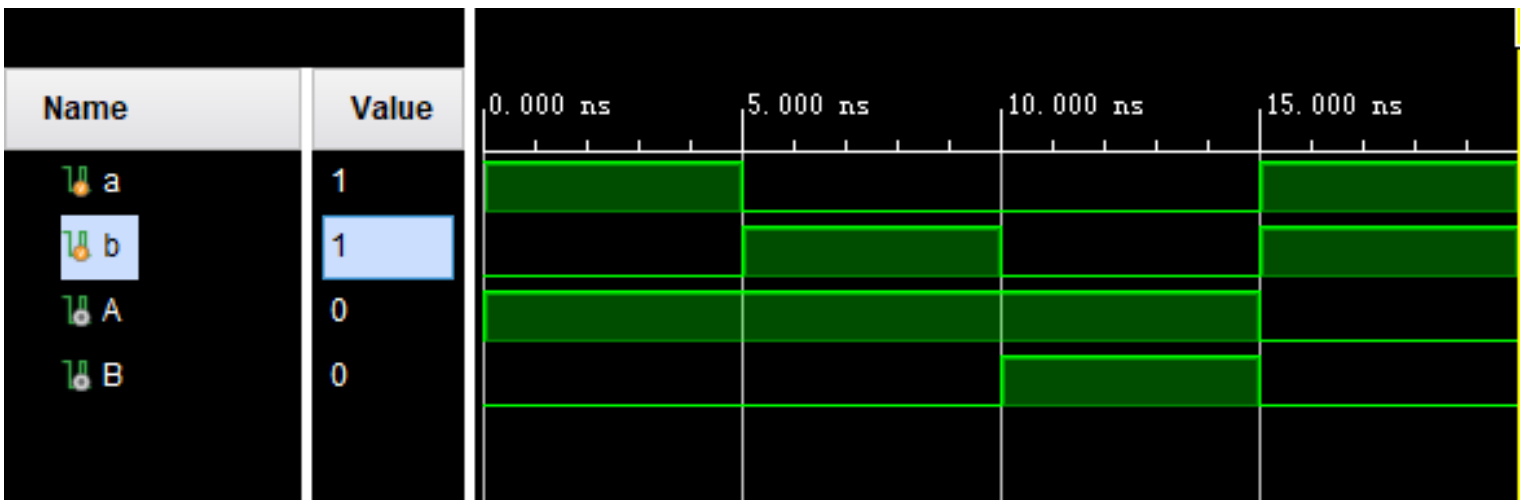
```
`timescale 1ns / 1ps

module cmos_nor(
    input a,b,
    output f
    );
    supply1 vdd;
    supply0 gnd;
    wire temp;
    pmos pmos1(temp,vdd,a);
    pmos pmos2(f,temp,b);
    nmos nmos1(f,gnd,a);
    nmos nmos2(f,gnd,b);
endmodule
```
Verilog

测试代码

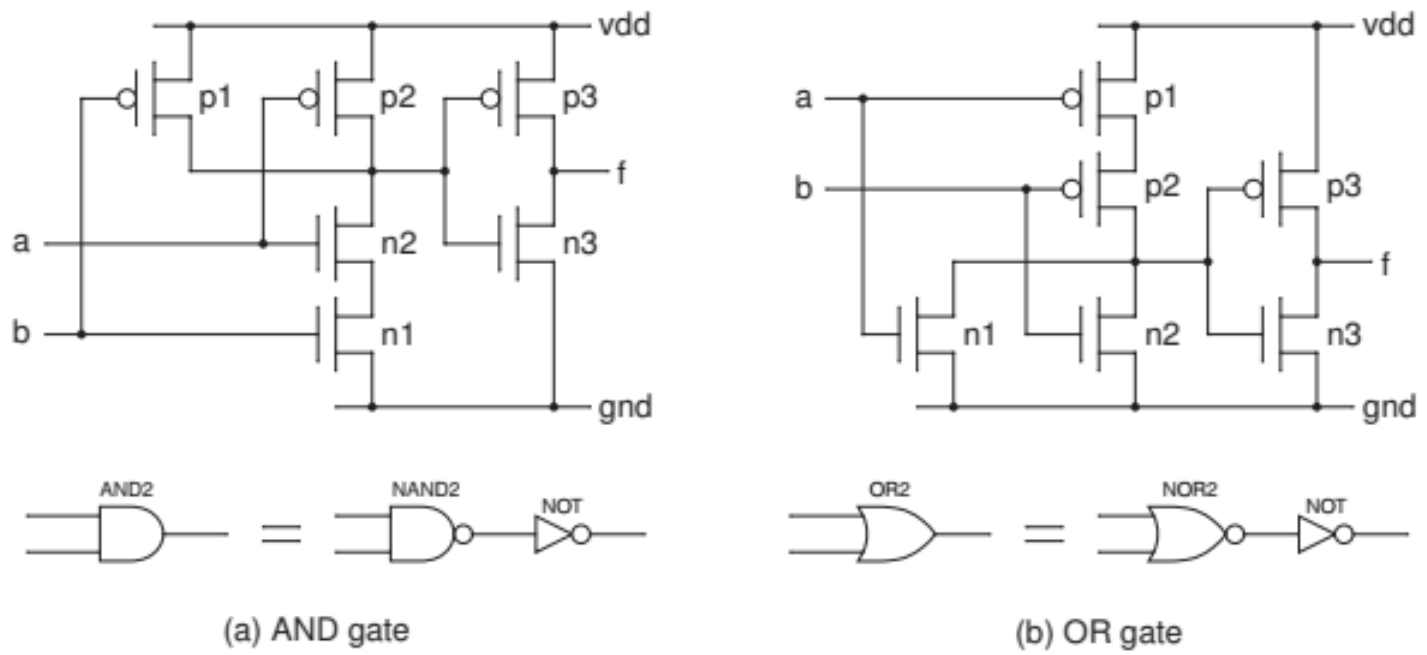
```
`timescale 1ns / 1ps

module cmos_nandOr_test();

    reg a,b;
    wire A,B;

    initial begin
        a=1;b=0;
```

```
        #5 a=0;b=1;
        #5 a=0;b=0;
        #5 a=1;b=1;
        #5 $finish;
    end
    cmos_nand  cmos_nand_inst (
        .a(a),
        .b(b),
        .f(A)
    );
    cmos_nor  cmos_nor_inst (
        .a(a),
        .b(b),
        .f(B)
    );
endmodule
```

Verilog



CMOS与门和或门



CMOS与门是CMOS与非门+CMOS反相器组成，CMOS与非门的输出作为CMOS反相器的输入。CMOS或门是CMOS或非门+CMOS反相器的组成，CMOS或非门的输出作为CMOS反向器的输入

✍因此CMOS与门和或门的实现比CMOS与非门、或非门多了两个晶体管，所以从功耗角度考虑，我们设计电路仅仅使用与非门或者或非门——使用布尔代数定律+摩尔定律转换表达式

> *We know that an AND gate uses two more transistors than a NAND gate. Therefore, if we can design a circuit using only NAND gates, the cost of the circuit will become lower.*

设计代码

```
`timescale 1ns / 1ps

module cmos_and(
    input a,b,
    output f
);
    wire temp;
    cmos_nand cmos_nand_inst(a,b,temp);
    cmos_inverter cmos_inverter_inst(temp,f);
endmodule
```

Verilog

```
`timescale 1ns / 1ps

module cmos_or(
    input a,b,
    output f
);
    wire temp;
    cmos_nor cmos_nor_inst(a,b,temp);
    cmos_inverter cmos_inverter_inst(temp,f);
endmodule
```

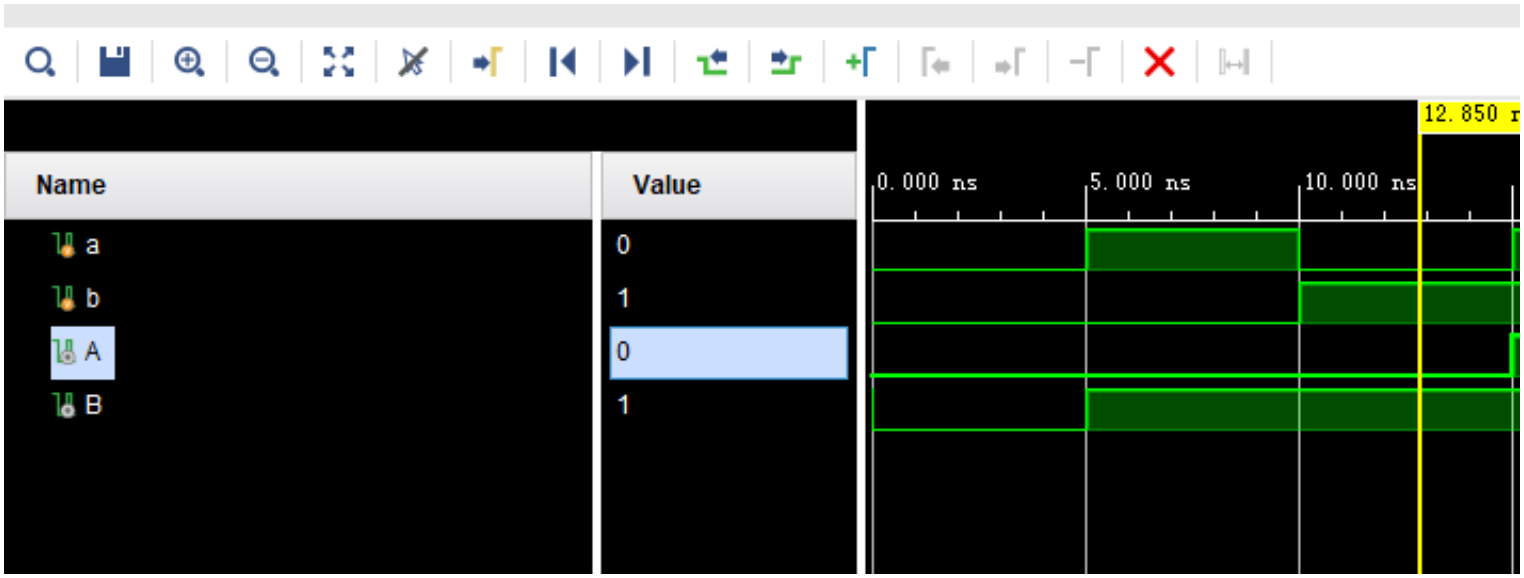
Verilog

测试代码

```
`timescale 1ns / 1ps

module cmos_andOr_test();
    reg a,b;
    wire A,B;
    initial begin
        a=0;b=0;
        #5 a=1;b=0;
        #5 a=0;b=1;
        #5 a=1;b=1;
    end

    cmos_and  cmos_and_inst (
        .a(a),
        .b(b),
        .f(A)
```

```
    );
    cmos_or  cmos_or_inst (
        .a(a),
        .b(b),
        .f(B)
    );

endmodule
```

Verilog



2.3 四种Verilog设计风格

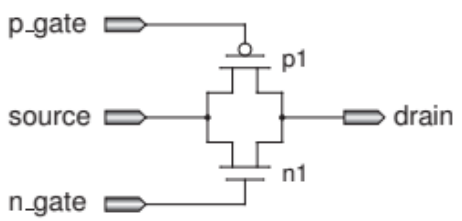
这一部分以二路多选器为例，介绍四种Verilog设计风格

✍一般用dataflow和behavioral两种即可

2.3.1 晶体管风格

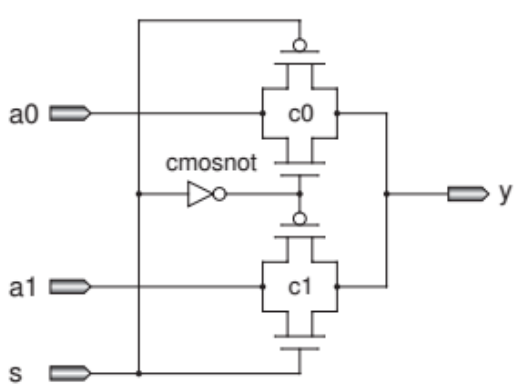
CMOS晶体管

前面介绍了PMOS、NMOS以及用PMOS、NMOS所实现的反相器、与非门、或非门、与门和或门。**这一部分将介绍PMOS、NMOS构成的CMOS，其结构如下**，只有当p_gate为0，n_gate为1时，source→drain



(a) CMOS transistor

CMOS构成的多路选择器



(b) Implementing multiplexer using CMOS transistors

当s为1时，c1gate均有效，a1→y；当s为0时，c0gate均有效，a0→drain

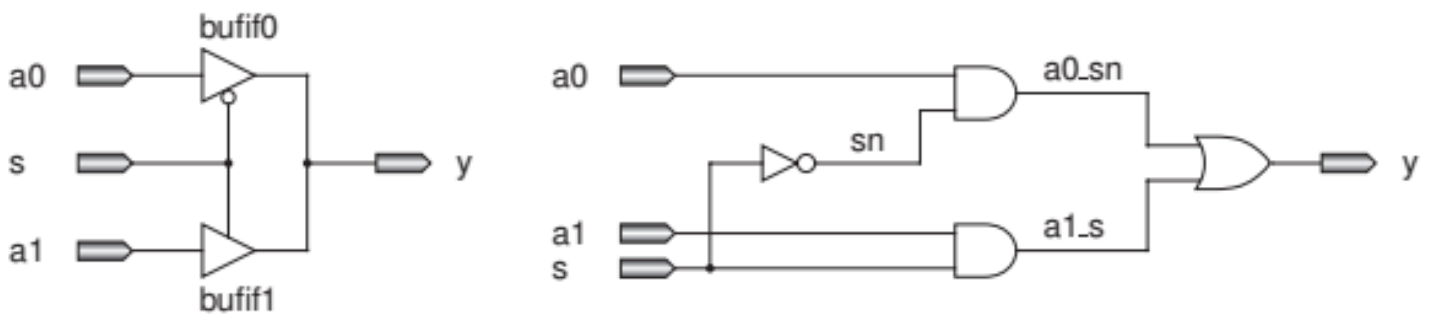
设计代码

```
`timescale 1ns / 1ps

module mux2to1_cmos(
    input a0,a1,s,
    output y
);
    wire temp;
    cmos_inverter cmos_inverter_init(s,temp);
    cmoscmos c0(a0,s,temp,y);
    cmoscmos c1(a1,temp,s,y);
endmodule
module cmoscmos(
    input source,p_gate,n_gate,
    output drain
```

```
);
    pmos p1(drain,source,p_gate);
    nmos n1(drain,source,n_gate);
endmodule
```

Verilog

2.3.2 逻辑门风格



(a) A multiplexer built with tri state gates　　(b) A multiplexer built with AND OR NOT gates

Figure 2.10 Schematic diagram of multiplexer using tri-state and ordinary gates

既可以选择三态门也可以选择与、或、非门，两种方法的电路图如上所示

```
//logic gates style
module mux2to1_bybufif(
    input a0,a1,s,
    output y
);
    //bufif0 0使能的三态门 (drain,source,gate)
    //bufif1 1使能的三态门 (drain,source,gate)
    bufif0 bufif0_init(y,a0,s);
    bufif1 bufif1_init(y,a1,s);
endmodule

module mux2to1_bylogic(
    input a0,a1,s,
```

```
    output y
);
    wire temp,a0_sn,a1_s;
    not not_init(temp,s);
    and and0(a0_sn,a0,temp);
    and and1(a1_s,a1,s);
    or or_init(y,a0_sn,a1_s);
endmodule
```

Verilog

2.3.3数据流风格

数据流风格即直接书写输出的表达式，也可以结合条件表达式

```
//dataflow style 常用
module mux2to1_dataflow(
    input a0,a1,s,
    output y
);
    assign y=s?a1:a0;//y=~s&a0+s&a1;
endmodule
```

Verilog

2.3.4 行为风格

在always块或者function块内，使用类似高级语言的风格

function块的使用方法

```
function function_name(input arg1,input arg2,...);
  begin
     //function_body
     //返回变量function_name
```

```
  end
endfunction
```

Verilog

类似函数的使用方法，**wire变量=function_name(输入列表);**

> *A function has a name. The name is also the output of the function. This output can be assigned to a net or an output with assign outside the function. The input arguments to the function are specified as input inside the function. We can use local variable names for these arguments.*

```
//behavioral style 常用
module mux2to1_byBehavioral(
    input a0,a1,s,
    output reg y
);
//块内使用高级语言的形式，always块或者function块
always @(*) begin
    y=a0;
    if (s) begin
        y=a1;
    end
end

// function sel(input a0,input a1,input s);
//     begin
//         case(s)
//             1'b0:sel=a0;
//             1'b1:sel=a1;
//         endcase
//     end
// endfunction
endmodule
```

Verilog

2.4 组合逻辑电路设计

2.4.1 多路选择器

多路选择器

2.4.2 译码器

n \rightarrow 2^n 译码器

Table 2.3 Truth table of a 3-8 decoder

Input				Output							
ena	*n*[2]	*n*[1]	*n*[0]	*d*[7]	*d*[6]	*d*[5]	*d*[4]	*d*[3]	*d*[2]	*d*[1]	*d*[0]
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0
0	x	x	x	0	0	0	0	0	0	0	0

当ena端有效时，才进行译码；否则输出为0

```
`timescale 1ns / 1ps


//3-8译码器
module decoder_design(
    input ena,
    input [2:0]n,
    output reg[7:0]d
);
```

```
always @(*) begin
    d=0;
    d[n]=ena;
end
endmodule
```

Verilog

```
`timescale 1ns / 1ps

//3-8译码器
module decoder_design(
    input ena,
    input [2:0]n,
    output reg[7:0]d
);
always @(*) begin
    d=0;
    d[n]=ena;
end
endmodule
```

Verilog

2.4.3 编码器

普通编码器

Table 2.4 Truth table of an 8-3 encoder

Input									Output		
ena	*d*[7]	*d*[6]	*d*[5]	*d*[4]	*d*[3]	*d*[2]	*d*[1]	*d*[0]	*n*[2]	*n*[1]	*n*[0]	*g*
1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	1
1	0	0	0	0	0	0	1	0	0	0	1	1
1	0	0	0	0	0	1	0	0	0	1	0	1
1	0	0	0	0	1	0	0	0	0	1	1	1
1	0	0	0	1	0	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0	0	1	0	1	1
1	0	1	0	0	0	0	0	0	1	1	0	1
1	1	0	0	0	0	0	0	0	1	1	1	1
0	x	x	x	x	x	x	x	x	0	0	0	0

ena表示输入是否有效，g表示是否有输入的1

只有当出现input对应的结果时才有对应的输出

```
`timescale 1ns / 1ps

module encode_custom(
    input ena,
    input [7:0]d,
    output reg[2:0]n,
    output reg g
);

//求逻辑表达式是一种方法
always @(*) begin
    n=3'b0;
    g=0;
    if (ena) begin
        g=|d;
        case(d)
            8'b00000001:n=3'b000;
```

```
            8'b00000010:n=3'b001;
            8'b00000100:n=3'b010;
            8'b00001000:n=3'b011;
            8'b00010000:n=3'b100;
            8'b00100000:n=3'b101;
            8'b01000000:n=3'b110;
            8'b10000000:n=3'b111;
        endcase
    end
end
endmodule
```

Verilog

优先编码器

Table 2.5 Truth table of an 8-3 priority encoder

Input									Output			
ena	*d*[7]	*d*[6]	*d*[5]	*d*[4]	*d*[3]	*d*[2]	*d*[1]	*d*[0]	*n*[2]	*n*[1]	*n*[0]	*g*
1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	1
1	0	0	0	0	0	0	1	x	0	0	1	1
1	0	0	0	0	0	1	x	x	0	1	0	1
1	0	0	0	0	1	x	x	x	0	1	1	1
1	0	0	0	1	x	x	x	x	1	0	0	1
1	0	0	1	x	x	x	x	x	1	0	1	1
1	0	1	x	x	x	x	x	x	1	1	0	1
1	1	x	x	x	x	x	x	x	1	1	1	1
0	x	x	x	x	x	x	x	x	0	0	0	0

输入任意，只译码最高位的1对应的编号，那么就从低到高检测d[i]是否为1，若为1，则n=i。也可以使用case（有不定态要用casex）

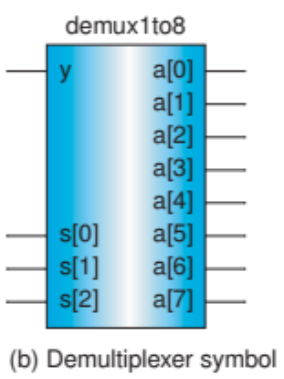
https://blog.csdn.net/Reborn_Lee/article/details/82390445书签：【 Verilog HDL 】case, casez, casex 之干货总结_casez verilog_李锐博恩的博客-CSDN博客

```
`timescale 1ns / 1ps

module encode_priority(
    input ena,
    input [7:0]d,
    output reg[2:0]n,
    output reg g
);

always @(*) begin
    n=3'b0;
    g=0;
    if (ena) begin
        g=|d;
        casex(d) //带有不定态的case用casex
            8'b0000_0001:n=3'b000;
            8'b0000_001x:n=3'b001;
            8'b0000_01xx:n=3'b010;
            8'b0000_1xxx:n=3'b011;
            8'b0001_xxxx:n=3'b100;
            8'b001x_xxxx:n=3'b101;
            8'b01xx_xxxx:n=3'b110;
            8'b1xxx_xxxx:n=3'b111;
        endcase
    end
end

// always @(*) begin:for_loop
//     integer i;
//     n=3'b0;
```

```
//     g=0;
//     if (ena) begin
//         g=|d;
//         for (i = 0; i<=7; i=i+1) begin
//             if (d[i]) begin
//                 n=i;
//             end
//         end
//     end
// end
endmodule
```

Verilog

2.4.4 反多路选择器



(b) Demultiplexer symbol

多路选择器是根据s值将对应编号的a[s]赋值给y

那么反多路选择器就是根据s值，将y赋给对应的a[s]，然后其余a[]任意

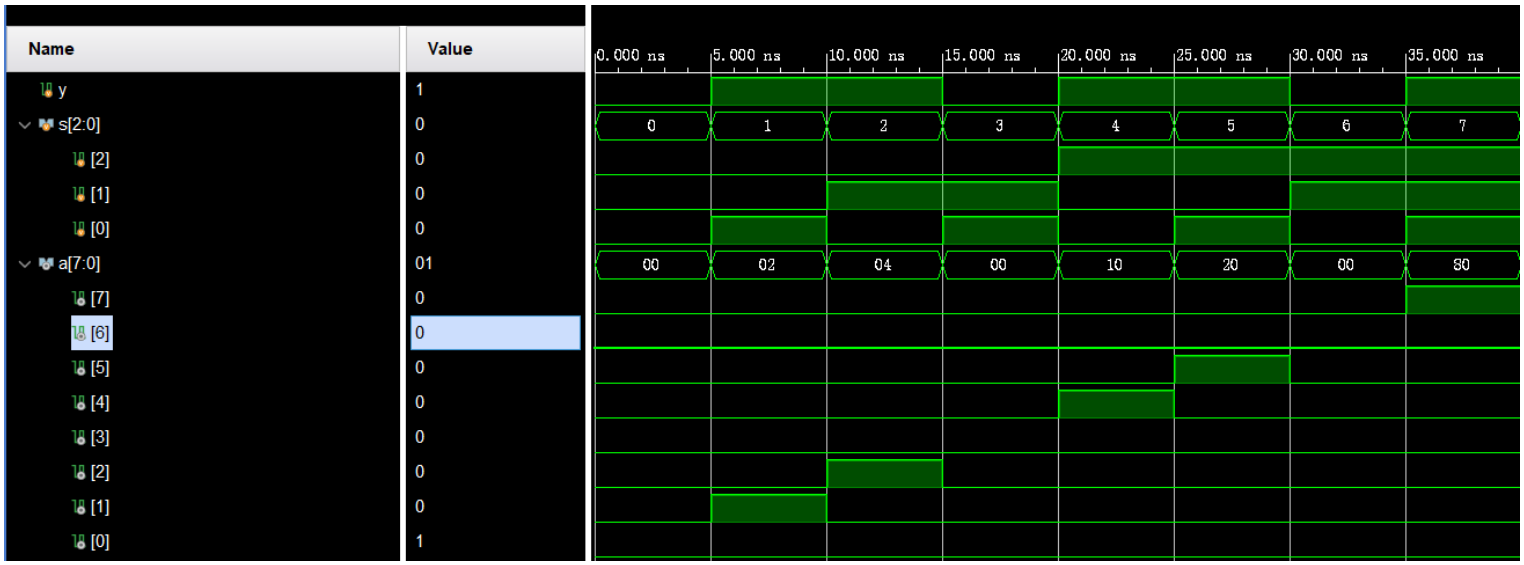
```
`timescale 1ns / 1ps

module demultiplexer_design(
    input y,
    input [2:0]s,
    output reg[7:0]a
);
    always @(*) begin
```

```
        a=8'b0;
        a[s]=y;
    end
endmodule
```

Verilog

```
`timescale 1ns / 1ps

module demultiplexer_test();

    reg y;
    reg [2:0]s;
    wire [7:0]a;
    initial begin
        y=0;s=0;
        #5;y=1;
        #5;y=0;
        #5;y=1;
        #5;y=1;
        #5;y=0;
        #5;y=1;
    end
    always #5 s=s+1;
    demultiplexer_design  demultiplexer_design_inst (
        .y(y),
        .s(s),
        .a(a)
    );
endmodule
```

Verilog



2.4.5筒式移位器

移位包括**左移**[注释1]**<<、逻辑右移>>、算术右移>>>**（在习题会实现带有算术左移的）

一个32位的滚动变位器根据right（表示是左移还是右移)和5位Sa（移位量）输入，对32位操作数移动指定位。此外，还有一个算术输入，用于指示当右输入为1时应进行逻辑变位还是算术变位。**逻辑右移会在空位上插入零，而算术右移会在空位上复制符号位。**

> *A 32-bit barrel shifter shifts a 32-bit input to the left or right by 0 to 31 bits based on a right input and a 5-bit sa (shift amount) input. There is also an arith (arithmetic) input that indicates whether to perform a logical shift or to perform an arithmetic shift when right is a 1. A logical shift right inserts zeroes in the emptied bit positions, and an arithmetic shift right replicates the sign bit in the emptied bit positions.*

下图是一个使用多选器实现32位左移器的传统例子，中间根据sa的5位值设置5个多路选择器，因此实现的移位只能是移16、8、4、2、1位

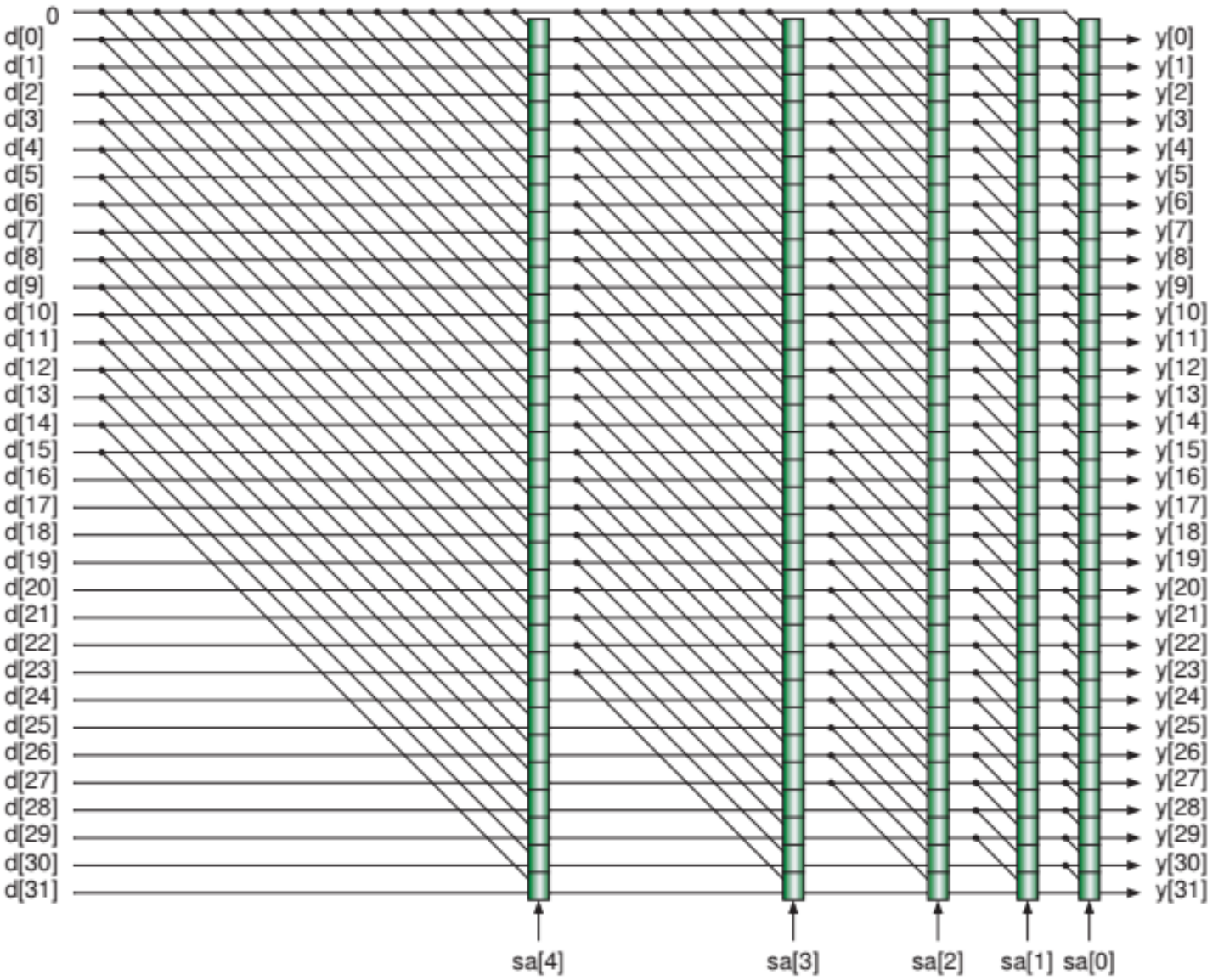


Figure 2.15 Schematic diagram of 32-bit left shifter

按照上图多选器的这种思想

sa[4]=1,移16位，左移{d{15:0},{16{0}}，右移则还需要根据选择器选择带符号还是不带符号（下图是在e的生成中已经选择）{e,d{31:16}}

sa[3]=1,移8位，左移{d{23:0},{8{0}}},右移{e{15:8},d{31:8}}

sa[2]=1.移4位，左移{d{27:0},{4{0}}},右移{e{15:12},d{31:4}}

sa[1]=1,移2位，左移{d{29:0},{4{0}}},右移{e{15:14},d{31:2}}

sa[0]=1,移1位，左移{d{30:0},{1{0}}},右移{e[15],d{31:1}}

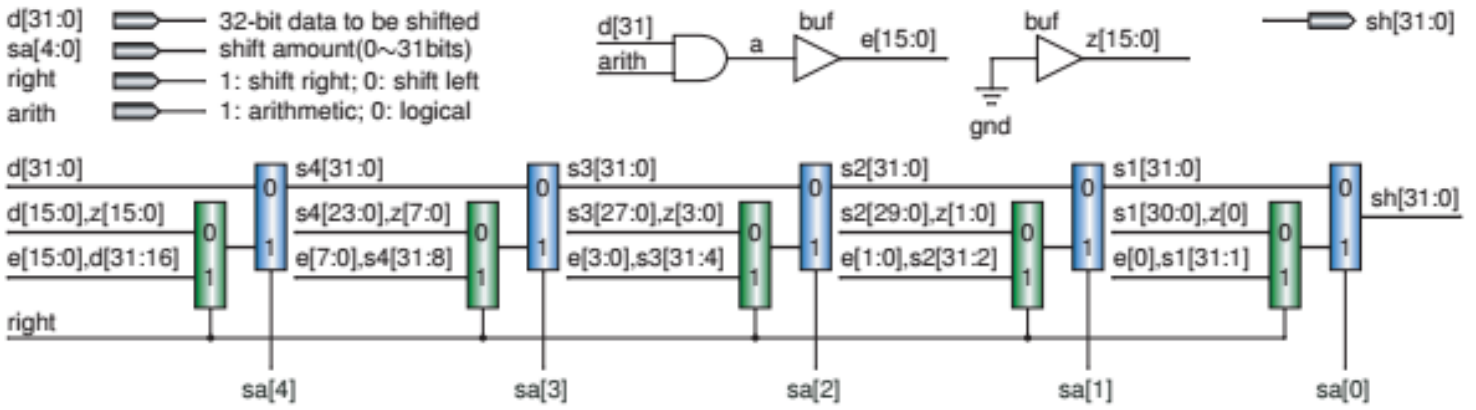


Figure 2.16 Schematic diagram of 32-bit barrel shifter

```
`timescale 1ns / 1ps

module barrel_shift_bydataflow(
    input [31:0]d,
    input [4:0]sa,
    input right,
    input arith,
    output [31:0]sh
);
    wire [15:0]sign={16{d[31]}};

    wire [31:0]right16=arith?{sign,d[31:16]}:{{16{1'b0}},d[31:16]};
    wire [31:0]left16={d[15:0],{16{1'b0}}};
    wire [31:0]res_16=sa[4]?(right?right16:left16):d;

    wire [31:0]right8=arith?{sign[7:0],res_16[31:8]}:{{8{1'b0}},res_16[31:8]};
    wire [31:0]left8={res_16[23:0],{8{1'b0}}};
    wire [31:0]res_8=sa[3]?(right?right8:left8):res_16;

    wire [31:0]right4=arith?{sign[3:0],res_8[31:4]}:{{4{1'b0}},res_8[31:4]};
    wire [31:0]left4={res_8[27:0],{4{1'b0}}};
    wire [31:0]res_4=sa[2]?(right?right4:left4):res_8;
```

```
    wire [31:0]right2=arith?{sign[1:0],res_8[31:2]}:{{2{1'b0}},res_8[31:2]};
    wire [31:0]left2={res_8[29:0],{2{1'b0}}};
    wire [31:0]res_2=sa[1]?(right?right2:left2):res_4;

    wire [31:0]right1=arith?{sign[0],res_8[31:1]}:{{1'b0},res_8[31:1]};
    wire [31:0]left1={res_2[30:0],{1'b0}};
    assign sh=sa[0]?(right?right1:left1):res_2;
endmodule
```

Verilog

此外也可以采用行为风格，直接使用<<、>>和$signed() >>>来实现上述的功能

```
module barrel_shift_byBehavioral(
    input [31:0]d,
    input [4:0]sa,
    input right,
    input arith,
    output reg[31:0]sh
);
always @(*) begin
    sh=d<<sa;
    if (right) begin
        if (arith) begin
            sh=$signed(d)>>>sa;
        end else begin
            sh=d>>sa;
        end
    end
end
endmodule
```

Verilog

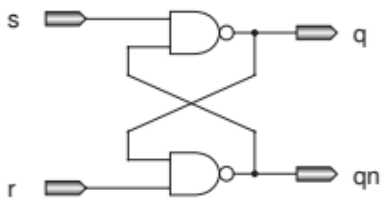
> sh=~right?d<sa:(~arith?d>sa:$signed(d) >>> sa);? 为什么不是算术右移

条件表达式和这个结合一块使用的时候有问题？

2.5 时序逻辑电路设计

2.5.1 D锁存器和D触发器

RS(reser-set)锁存器



(a) RS latch

当s为1，r为0时，qn为1，q为0

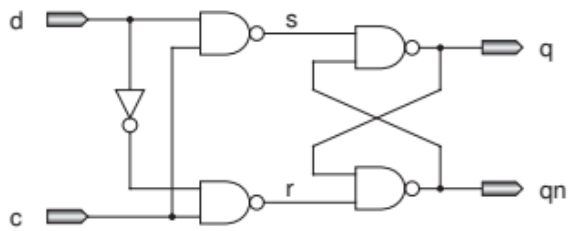
当s为0，r为1时，qn为0，q为1

当s、r均为1时，q不改变

s、r均低有效

```Verilog
`timescale 1ns / 1ps

module rs_latch(
    input s,r,
    output q,qn
);
    nand nand1(q,s,qn);
    nand nand2(qn,r,q);
endmodule
```

D锁存器



(b) D latch

D锁存器在RS锁存器的基础上，增加了两个与非门和一个反相器。增加的这些器件接收d、c的输入，生成s和r。

当c为0时，s和r均是1，q状态不变；当c为1时，d→q

```Verilog
`timescale 1ns / 1ps

module d_latch(
    input d,c,
    output q,qn
);
    wire s,r,temp;
    not not_init(temp,d);
    nand nand1(s,d,c);
    nand nand2(r,temp,c);

    rs_latch  rs_latch_inst (
        .s(s),
        .r(r),
        .q(q),
        .qn(qn)
    );
endmodule
```

D触发器

D锁存器是“**在c有效的时间内，只要d在变化那么q、qn就跟着变化**”　　D触发器是“**只有在clk边沿触发时，q、qn才会跟着d变化**”

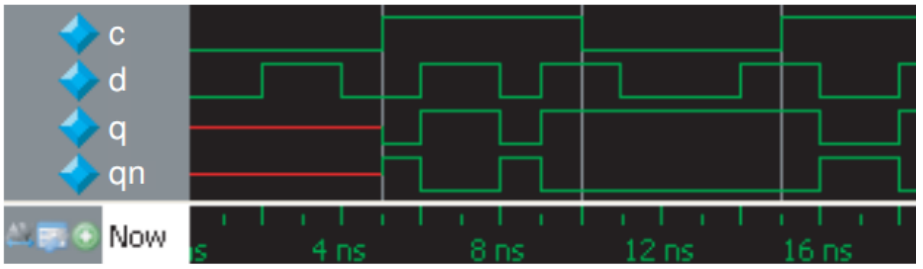


Figure 2.18 Waveform of a D latch

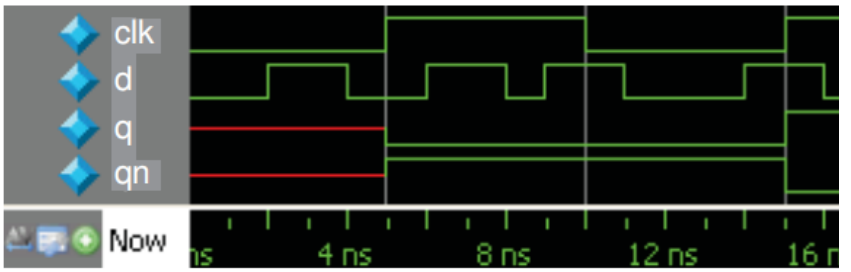


Figure 2.20 Waveform of an academic

✍相比较来讲，D触发器缩小了D锁存器的输出变化区间，仅在边沿触发时输出变化

D触发器结构如下：

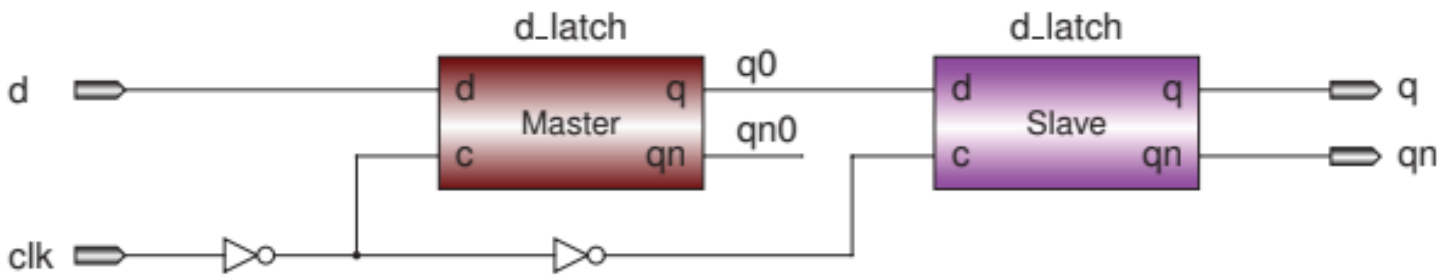


Figure 2.19 Schematic diagram of an academic D flip-flop

根据D触发器的电路图，D触发器由两个D锁存器和两个反相器构成。左边的D锁存器被叫做主锁存器，右边的D锁存器被叫做从锁存器。

当clk为0时，主锁存器的c有效，d→q0，但从锁存器的c无效，q保持；当clk为1时，主锁存器的c无效，q0保持，从锁存器的c有效，q0→q。因此当clk 0→1时，q会跟随d，而当clk为0或clk为1时，q/q0保持

> *The left D latch is called a master latch, and the right one is called a slave latch. When clk (clock) signal is a 0, the output of the master latch (q0) is equal to the input d, and the output of the slave latch does not change. When clk goes to 1 from 0, the output of the slave latch follows q0 but q0 does not change.* ***Therefore, the state of a DFF is altered only when the clock changes from 0 to 1 (rising edge or positive edge).*** *This is the meaning of "edge-triggered."*

```
module dff_design(
    input d,
    input clk,
    output q,
    output qn
```

```
);
    wire clkn,clknn;
    wire q0,qn0;
    not not1(clkn,clk);
    d_latch  d_latch_1 (
        .d(d),
        .c(clkn),
        .q(q0),
        .qn(qn0)
    );
    not not2(clknn,clkn);//为什么要两个非门，不直接接clk->表示从触发器不是受clk上升沿控制的，只是用它的电平
    d_latch  d_latch_2 (
        .d(q0),
        .c(clknn),
        .q(q),
        .qn(qn)
    );
endmodule
```

Verilog

工业版的D触发器

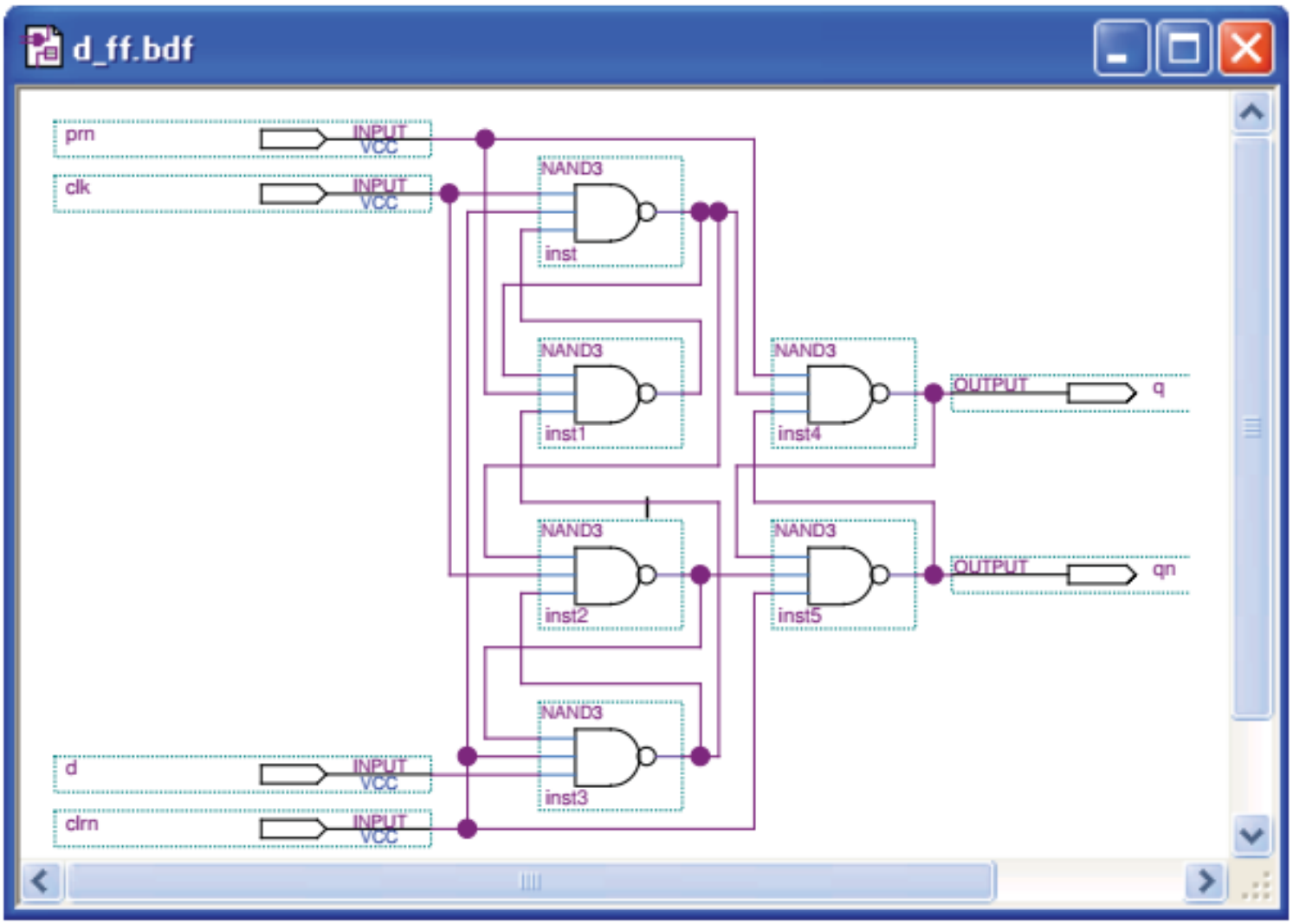


Figure 2.21 Schematic diagram of an industry D flip-flop

prn是预设端口，预设q为1；clrn是清除端口，清除q为0

prn和clrn均低有效,tongs

> *An industry version of the DFF circuit is given in Figure 2.21, where prn and clrn are the preset and clear inputs, respectively. Both the signals are active-low.*

```
module dff_insDesign(
    input d,
    input clk,
    input prn,
    input clrn,
    output q,
    output qn
```

```
);
    wire nand1,nand2,nand3,nand4,nand5,nand6;
    assign nand1=~(clk&clrn&nand2);
    assign nand2=~(nand1&prn&nand4);
    assign nand3=~(nand1&clk&nand4);
    assign nand4=~(nand3&clrn&d);
    assign nand5=~(prn&nand1&nand6);
    assign nand6=~(nand5&nand3&clrn);
    assign q=nand5;
    assign qn=nand6;
endmodule
```

Verilog

带使能端的D触发器

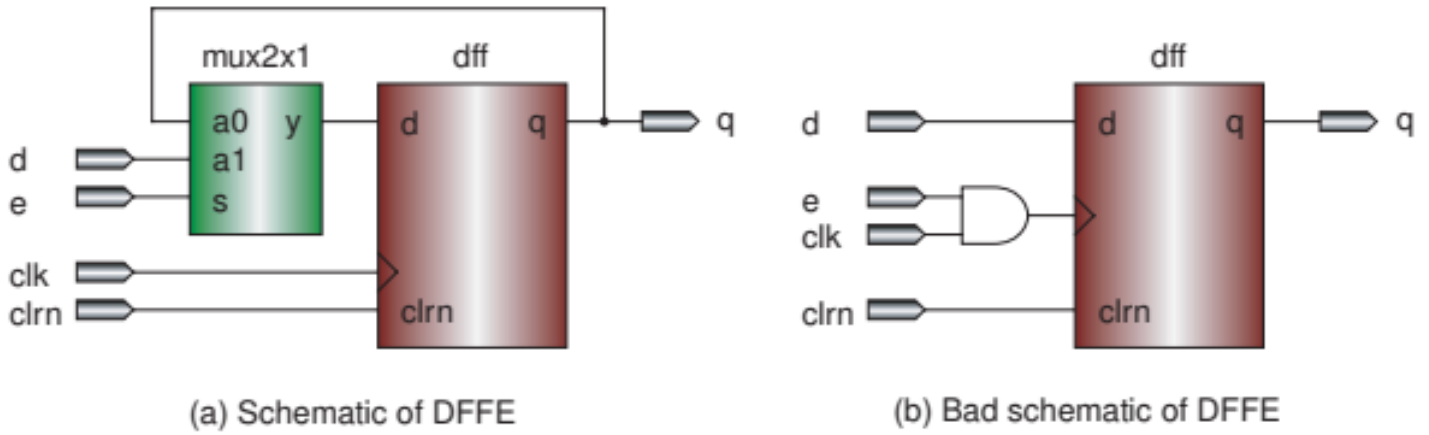


Figure 2.23 Schematic diagram of a D flip-flop with enable control

在原来的DFF的d端加入一个多路选择器，当e有效时，输入的d才能到DFF的d端，否则输入无效q➡DFF的d端

```
`timescale 1ns / 1ps



//坏的设计——en和clk与接clk->改变了时钟

//好的设计如下，在dff的d输入端加一个多路选择器，使能有效时输入d无效时输入q
module dffe_design(
```

```
    input d,e,clk,clrn,prn,
    output q,qn
);
    wire temp=e?d:q;
    dff_insDesign dffe(
        temp,clk,prn,clrn,q,qn
    );
endmodule
```

Verilog

> *if e is a 0, the DFFE does not change even if a new clock edge arrives and/or the d input changes; otherwise, DFFE acts as a DFF. This is implemented with a 2-to-1 multiplexer: the DFFE is updated on every clock rising edge but the current state will be written again (no change) if the enable signal is inactive. Figure 2.23(b) shows a bad design of DFFE that uses an AND gate to prohibit the clock.*——b的做法会让clk受限

行为风格的D触发器

- D触发器flopr

```
module flopr(
    input clk,rst,
    input d,
    output reg q
);
    always @(posedge clk) begin
        if (rst) begin
            q<=0;
        end else begin
            q<=d;
        end
    end
endmodule
```

Verilog

- 只有使能端的D触发器floperr

```Verilog
module floperr(
    input clk,rst,enable,
    input d,
    output reg q
);
    always @(posedge clk) begin
        if (rst) begin
            q<=0;
        end else if (enable) begin
            q<=d;
        end
    end
endmodule
```

- 同步清0的D触发器floprc_sync

```
module floprc_sync(
    input clk,rst,clear,//clear低有效
    input d,
    output reg q
);
    always @(posedge clk) begin
        if (rst) begin
            q<=0;
        end else if (~clear) begin
            q<=0;
        end else begin
            q<=d;
        end
```

```
    end
endmodule
```

Verilog

- 异步清0的D触发器floprc_async

```
module floprc_async(
    input clk,rst,clear,//clear低有效
    input d,
    output reg q
);
    always @(posedge clk or negedge clear) begin
        if (rst) begin
            q<=0;
        end else if (~clear) begin
            q<=0;
        end else begin
            q<=d;
        end
    end

endmodule
```

Verilog

- 带使能端的异步清0的D触发器flopenrc_async

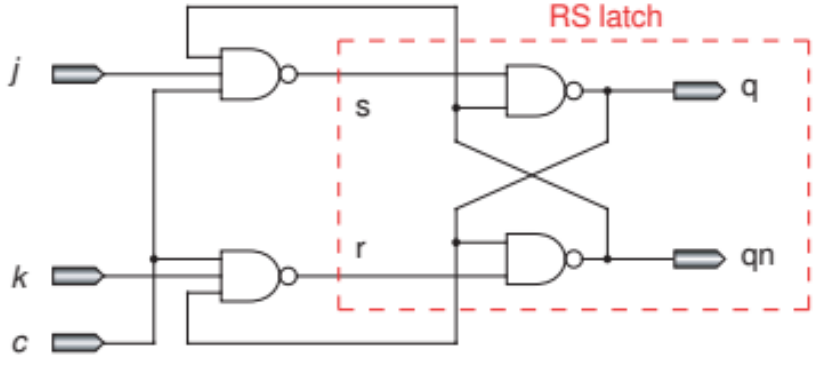
```
module flopenrc_async(
    input clk,rst,enable,clear,//clear低有效
    input d,
    output reg q
);
    always @(posedge clk or negedge clear) begin
        if (rst) begin
```

```
            q<=0;
        end else if (~clear) begin
            q<=0;
        end else if (enable) begin
            q<=d;
        end
    end

endmodule
```

Verilog

2.5.2 JK锁存器和JK触发器

JK锁存器



c	j	k	State
1	0	0	No change
1	0	1	Reset
1	1	0	Set
1	1	1	Toggle
0	x	x	No change

Figure 2.24 Schematic diagram of the JK latch

```
`timescale 1ns / 1ps

module jk_latch(
    input c,
    input j,k,
    output q,qn
);
    wire temp_s=~(j&qn&c);
    wire temp_r=~(c&k&q);
```

```
    rs_latch rs_latch_init(temp_s,temp_r,q,qn);
endmodule
```

Verilog

JK锁存器由RS锁存器和两个三输入的与非门组成，其真值表见上图

输出表达式为$q_n = c(jbarq + barkq) + barcq, q_n$为$nextspaceq$

> *Figure 2.24 shows the circuit of a JK latch where an RS latch is used for storing the state. The truth table is also given in the figure. The output expression is qn = c(j q + kq)+cq, whereqn is the next state of q*

JK锁存器很少被用到，因为当它的三个输入端均为1时执行的切换功能没有一个速度的控制

> *The JK latch is rarely used because it toggles without speed control if all the three inputs are 1.*

JK触发器

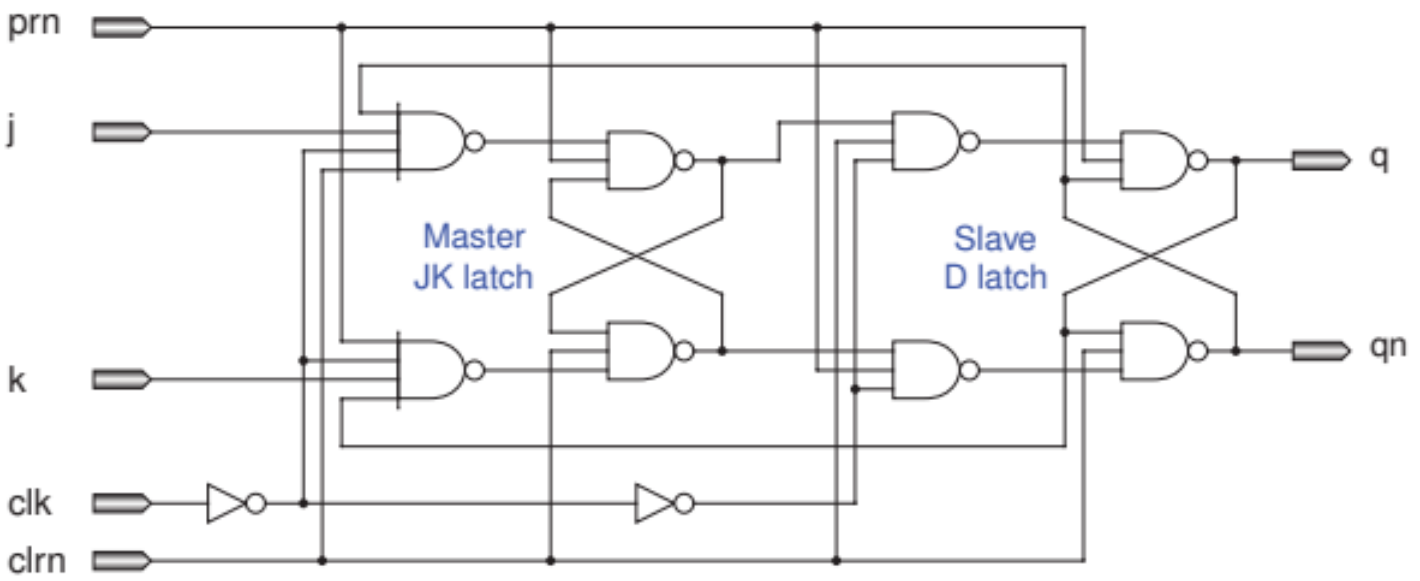


Figure 2.25 Schematic diagram of the JK flip-flop

上图的JK触发器有一个prn预设端口，一个clrn清除端口（均低电平有效）。JK触发器由一个JK锁存器（主锁存器）、一个D锁存器（从锁存器）组成。因为从锁存器的输出又送回到了JK锁存器的输入端，所以**JK触发器在clk上升沿切换状态**

> *Figure 2.25 shows the circuit of a JKFF with a preset and a clear. It consists of a JK latch (master latch) and a D latch (slave latch). This JKFF toggles on the positive edge of clock because the outputs of the D latch, not the outputs of the JK latch, are sent back to the first-level gates. Note that **at least a clear input is required**; otherwise, the initial states of the JK and D latches cannot be determined.*

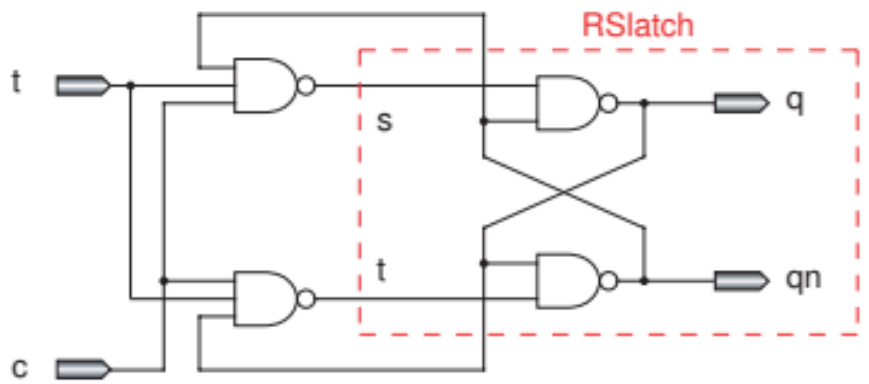
实现方法：用行为风格实现，切换表达式为$q_n = jbarq + barkq$

```
`timescale 1ns / 1ps


//采用Behavioral风格
module jkff_design(
    input prn,clrn,//低有效
    input j,k,clk,
    output reg q,qn
);
    always @(posedge clk or negedge clrn) begin
        if (~clrn) begin
            q<=0;
            qn<=1;
        end else if (prn) begin
            q<=1;
            qn<=0;
        end else begin
            q<=j&qn+~k&q;
            qn<=~q;
        end
    end
endmodule
```

Verilog

2.5.3 T锁存器和T触发器

T锁存器



c	t	State
1	0	No change
1	1	Toggle
0	x	No change

Figure 2.27 Schematic diagram of the T latch

T锁存器由一个RS锁存器和两个与非门组成，其真值表如上图

当c为0时，输出q状态不改变；当c为1时，若t为1则切换状态否则状态不变

输出q的表达式是：$q_n = c(tbarq + bartq) + barcq, q_n spaceisspacenextspaceq$

T锁存器也很少使用，**同JK锁存器也是因为c=1，t=1时的状态切换没有速度控制**

> *Figure 2.27 shows the circuit of a T latch where an RS latch is used for storing the state. When the t and c inputs are both high, the output q toggles. The truth table is also given in the figure. The output expression is qn = c(t q + tq)+cq,whereqn is the next state of q. Similar to the JK latch, the T latch is rarely used.*

```
`timescale 1ns / 1ps

module t_latch(
    input t,c,
    output q,qn
);
    wire s_temp=~(t^c^qn);
    wire t_temp=~(t^c^q);

    rs_latch  rs_latch_inst (
        .s(s_temp),
        .r(t_temp),
        .q(q),
        .qn(qn)
```

```
        );
endmodule
```
Verilog

T触发器

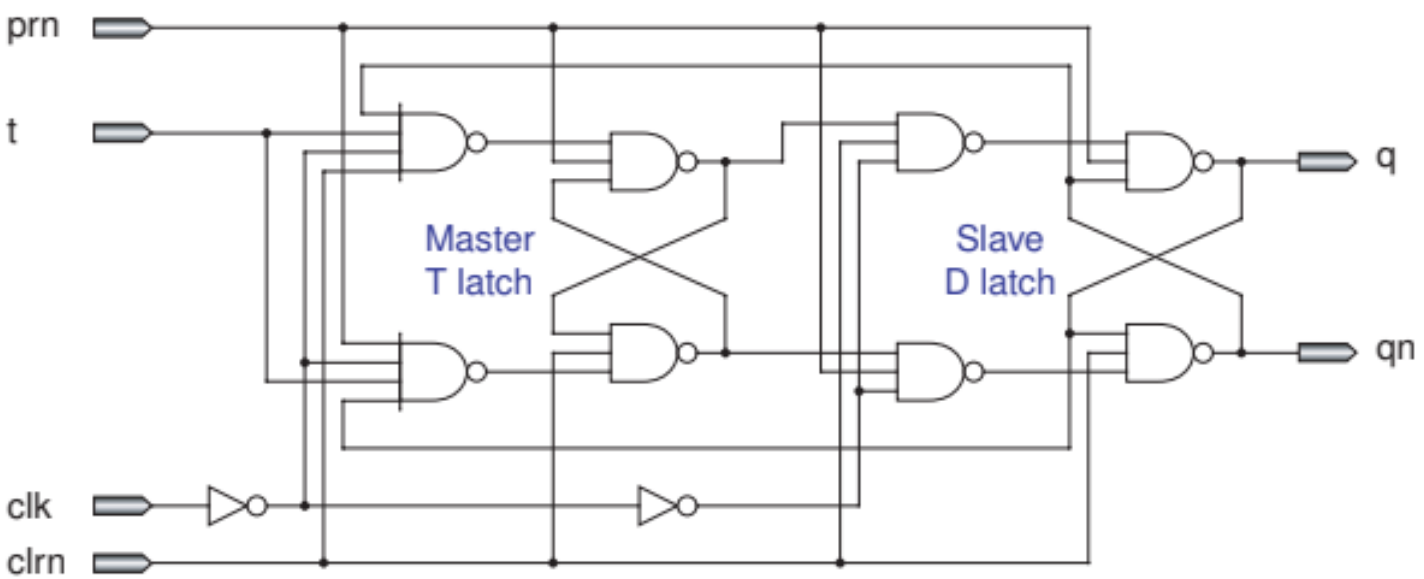


Figure 2.28 Schematic diagram of the T flip-flop

T触发器由一个T锁存器（主锁存器）和一个D锁存器（从锁存器）组成。prn预设端口、clrn清除端口均低有效。**输入t决定是否进行状态的切换——t为1时上升沿切换状态，t为0时状态保持不变**

> *It consists of a T latch (master latch) and a D latch (slave latch). The output does not change until the next positive edge of the clock (clk). The input t determines whether to toggle the output: if t is a 1, the state is toggled in the clock edge; otherwise, the state does not change. prn and clrn are the preset and clear inputs, respectively.*

采用行为风格实现T触发器，输出状态的表达式是$q_n = tbarq + bartq$

```
`timescale 1ns / 1ps

module tff_design(
    input prn,t,
    input clk,clrn,
```

```
    output reg p,pn
);
always @(posedge clk or negedge clrn) begin
    if (~clrn) begin
        p<=0;
        pn<=1;
    end else if (prn) begin
        p<=1;
        pn<=0;
    end else begin
        p<=t&pn|~t&p;
        pn<=~p;
    end
end
endmodule
```

Verilog

2.5.4 移位寄存器

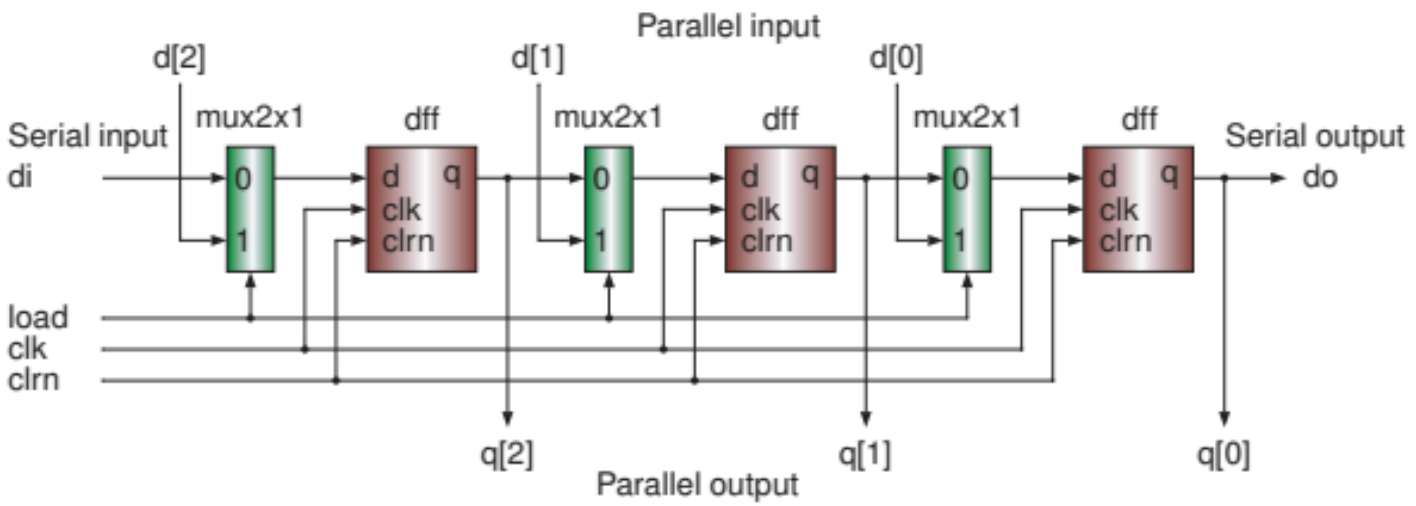


Figure 2.30 Schematic diagram of a shift register

移位寄存器即可以实现串入并出，也可以实现并入串出

> *We can design a shift register, as shown in Figure 2.30. It can convert a serial signal di to a parallel signal q[2:0]. It can also convert a parallel signal d[2:0] to a serial signal do*

load信号用于加载并行输入到DFF，当移位寄存器处于工作时，load=0

> *The load signal is used to load the parallel data d[2:0] to the DFFs (load = 1). mux2x1 is a 1-bit 2-to-1 multiplexer. When shift is performed, the load signal should be 0.*

```Verilog
`timescale 1ns / 1ps

module shift_design(
    input load,
    input clk,
    input clrn,

    input [2:0]d,
    input di,

    output [2:0]q,
    output d_o
);
    wire q1,q2;
    wire d1=load?d[2]:di;
    wire d2=load?d[1]:q1;
    wire d3=load?d[0]:q2;

    floprc_sync f1(clk,0,clrn,d1,q1);
    floprc_sync f2(clk,0,clrn,d2,q2);
    floprc_sync f3(clk,0,clrn,d3,d_o);
    assign q[2]=q1;
    assign q[1]=q2;
    assign q[2]=d_o;
endmodule
```

```
`timescale 1ns / 1ps

module shift_test();
    reg load, clk, clrn;
    reg [2:0]d;
    reg di;

    wire [2:0]q;
    wire d_o;

    initial begin
        clk=0;
        clrn=1;
        load=0;
        #3;clrn=0;//3ns 同步清0
        #3;clrn=1;//6ns
        #8;di=1;//14ns 串入并出
        #10;di=0;//24ns
        #10;di=1;//34ns 101

        #30;d=3'b110;load=1;//64ns 并入串出
        #2;load=0;//66ns load加载完
        //3个周期, 95 0 105 1 115 1
        #54;$finish;
    end
    shift_design  shift_design_inst (
        .load(load),
        .clk(clk),
        .clrn(clrn),
        .d(d),
```

```
        .di(di),
        .q(q),
        .d_o(d_o)
    );
    always #5 clk=~clk;
endmodule
```

Verilog

2.5.5 FIFO缓冲器

串行FIFO

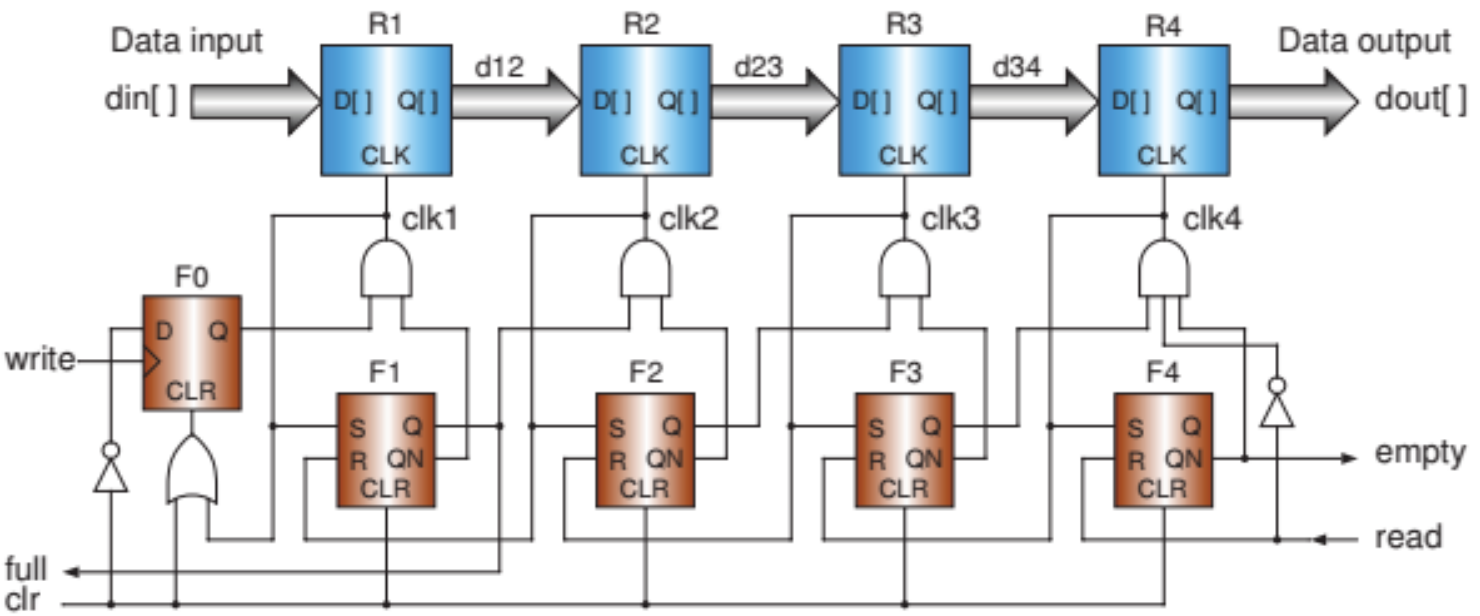


Figure 2.31 Schematic diagram of a FIFO of depth 4

上图为深度为4的FIFO缓冲器的综合电路图，R1、R2、R3、R4为4个DFF，用于保存数据和实现数据的传递。下面的F1、F2、F3、F4为RS锁存器（端口高有效），用于生成Ri的clk脉冲，DFF0是用于为clk1生成一个写信号

> *FIFO (first-in first-out) is a special buffer for queuing data,Figure 2.31 shows a schematic diagram of aFIFOofdepth4.R1, R2, R3,andR4 are four registers (DFFs). The data in the input port enter the R1 register when the write signal is asserted. Then the data are passed to the rightmost empty register. When the read signal is asserted, the data in the R4 register will be read out and other data go to their right registers automatically.*

There are four RS latches: F1, F2, F3,andF4. The inputs of the RS latches are active high: if S (set) is a 1, the latch is set; if R (reset) is a 1, the latch is cleared.

The DFF F0 is used to generate a write pulse for clk1.

开始时，Fi处于清除状态，Q=0，QN=1

> *The RS latches are cleared initially,therefore the outputs of QNs are high.*

为了便于观察，设置**与门存在延迟，延迟为1ns**

> *We set the delay time of the AND gates to 1 ns in the simulation.*

当write信号上升沿有效时，F0可以工作。clr为1则Q输出为0，从而使得clk1=0；clr为0时则Q输出为1，则在#1后，clk1为1；

clk1为1时，会使得F1的S端有效，则Q_F1输出有效、full有效；也会使得F0的CLR端有效，Q_F0=0，则在#2时，clk1为0，clk2为1；

clk2为1时，会使得F2的S端有效，则Q_F2输出有效；也会使得F1的R端有效，QN_F1=1,Q_F1=0；则在#3时，clk1=0(中间异步清0后，Q_F0始终为0，QN_F1保持1)，clk2=0，clk3=1,full=0；

clk3为1时，会使得F3的S端有效，F2的R端有效，则Q_F3=1,QN_F2=1,Q_F2=0,则在#4时clk1=0，clk2=0，clk3=0，clk4=1；

clk4为1时，会使得F3的R有效，F4的S有效，则会设置empty=0，在#5时clk4=0

在下一个write上升沿有效时，Q_F0就会输出1，此时QN_F1=1，因此#1后clk1=1；设置F0清除状态，之后在下一个上升沿到来前始终保持0，F1的S端有效，Q_F1=1，full=1；则在#2时，clk1=0，clk2=1；之后#3时，clk1=0，clk2=0，clk3=1；在#4时QN_F4=0，clk4=0；

当read信号有效时，设置F4的R端有效，QN_F4=1,empty=1;则当read信号无效时，#1，clk4=1，输出dout；并使得F3_QN=1,F4_QN=0，empty=0；在#2时，clk4=0，clk3=1；#3时,clk2=1;#4时，full=0；

✍当F_{i-1}的Q端为1，F_i的QN端为1时，会有clki的上升沿；

clki的1会使得clk(i-1)变成0；

clki的有效可以使得数据从Ri的输入端到输出端

> *If Q of the (i − 1)th latch becomes 1 and Q of the ith latch is 0 (QN is 1), then clki becomes 1 (a rising edge) that sets the ith latch and resets the (i − 1)th latch. clki will go back to 0 once clki + 1 becomes 1. Thus, a pulse of clki is generated that is used to store the data of the (i − 1)th register into the ith register.*

若read无取反到clk4的与门，那么会使得clk4为1，数据会向前传递，导致在read有效后的#1数据被取出，而不是read无效后的#1取出数据

> *The NOT gate in the right side prevents the data in the R4 register from being overwritten when the read signal is asserted*

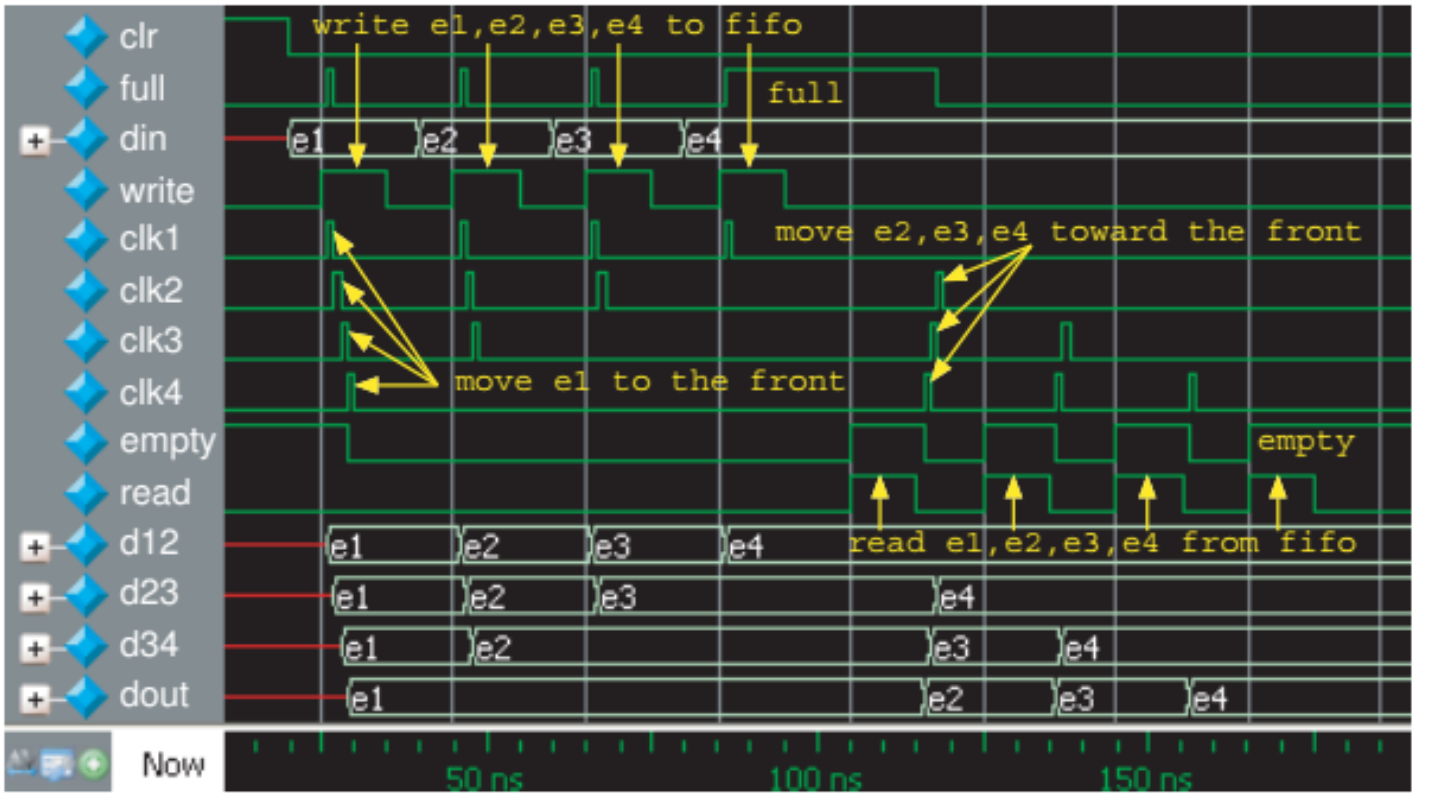


Figure 2.32 Waveforms of FIFO4

```
`timescale 1ns / 1ps

module fifo_serio_design(
    input [31:0]d,
    input clrn,
    input read,write,

    output [31:0]q,
    output full,
    output empty
);
    wire clk1,clk2,clk3,clk4;
    wire [31:0]d12,d23,d34;
```

```
    wire q1,qn1,q2,qn2,q3,qn3,q4,qn4;

    flopr f1(clk1,0,d,d12);
    flopr f2(clk2,0,d12,d23);
    flopr f3(clk3,0,d23,d34);
    flopr f4(clk4,0,d34,q);

    wire temp1;
    floprc_async gen_temp1(write,0,clrn|clk1,~clrn,temp1);
    rs_latch_high gen_q1(clk1,clk2,clrn,q1,qn1);
    //clk1
    assign #1 clk1=temp1&qn1;

    rs_latch_high gen_q2(clk2,clk3,clrn,q2,qn2);
    //clk2
    assign #1 clk2=q1&qn2;

    rs_latch_high gen_q3(clk3,clk4,clrn,q3,qn3);
    //clk3
    assign #1 clk3=q2&qn3;

    rs_latch_high gen_q4(clk4,read,clrn,q4,qn4);
    //clk4
    assign #1 clk4=q3&qn4&~read;

    assign full=q1;
    assign empty=qn4;
endmodule
```

Verilog

```
`timescale 1ns / 1ps
module fifo_serio_test();
```

```
    reg [31:0]d;
    reg clrn;//clrn低有效
    reg read,write;

    wire [31:0]q;
    wire full;
    wire empty;

    initial begin
        clrn=0;write=0;read=0;
        #2;clrn=1;d=32'he1;
        #1;clrn=0;
        #2;write=1;//5ns
        #5;write=0;//10ns
        #3;d=32'he2;
        #2;write=1;//15ns
        #5;write=0;//20ns
        #3;d=32'he3;
        #2;write=1;//25ns
        #5;write=0;//30ns
        #3;d=32'he3;
        #2;write=1;//35ns
        #5;write=0;
        #5;read=1;
        #5;read=0;
        #5;read=1;
        #5;read=0;
        #5;read=1;
        #5;read=0;
        #5;$finish;
    end
    fifo_serio_design  fifo_serio_design_inst (
        .d(d),
```

```
        .clrn(clrn),
        .read(read),
        .write(write),
        .q(q),
        .full(full),
        .empty(empty)
    );
endmodule
```

Verilog

RAM实现的循环FIFO

在上面所实现的FIFO缓冲中并没有地址，这意味着并不能随机访问缓冲中的寄存器，输出只能从最后端寄存器得到。下图展示了一种使用RAM实现的FIFO

> *There is no address in a FIFO buffer, meaning that a register in the FIFO cannot be accessed randomly*

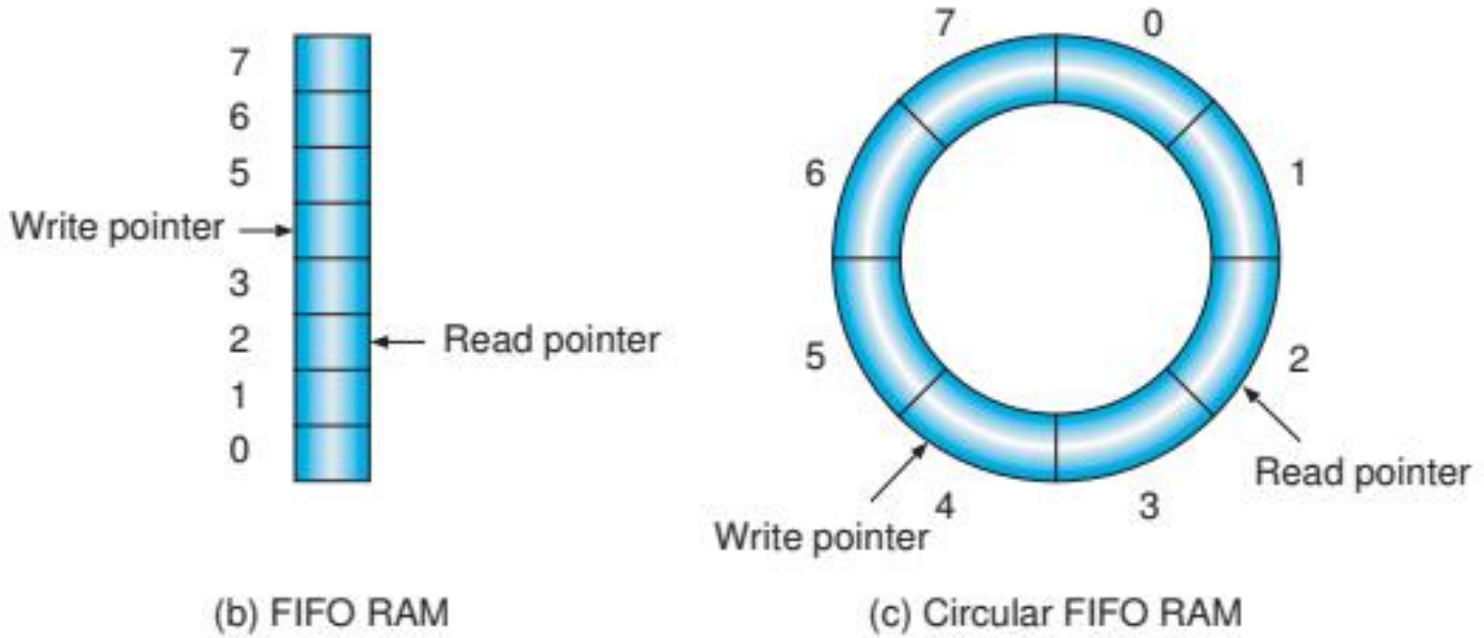


Figure 2.33 A circular FIFO implemented with RAM

如上图所示，设置两个指针——写指针和读指针，作为RAM的地址。初始时写指针=读指针，队列为空。每次写操作均会使得写指针+1（写指针+1需要保证序号在范围内，因此是(写指针+1)%长度）。每次读操作都会使得读指针+1((读指针+1)%长度)

> *We prepare two n-bit internal pointers, the write pointer and the read pointer, as the addresses of the RAM. Each write or read operation will result in an increment of the corresponding pointer. Therefore, the FIFO is actually circulated. The depth of the FIFO is 2n and the next location to the location 2n − 1is 0.*

但是当写满时，读写指针又相等。所以对于写满读空时的读写指针均为0需要区分判断，一般的方法是**为读写指针多设置一个扩展位**，**当读写指针的扩展位和数据位均相同时，表示空；当读写指针数据位相同但是扩展位不同，则表示满**

```
//设置empty和full信号表示缓冲区的空和满
`timescale 1ns/1ps

module fifo_ram_design(
    input clk,//时序控制
    input clrn,//清0 高有效

    input read,
    input write,

    input [31:0]d_in,
    output reg[31:0]d_out,

    output empty,
    output full
);
  reg [31:0]ram[7:0];
  reg [3:0]read_i,write_i;//多设置一个扩展位用于区分 空满时read_i=write_i的情况

  always @(posedge clk or posedge clrn) begin
      if (clrn) begin
          read_i<=4'b0;
          write_i<=4'b0;
      end else begin
          if (write) begin
```

```
            if (~full) begin
                ram[write_i]<=d_in;
                write_i<=write_i+1;
            end else begin
                $display("FIFO buffer is full,don't allow write");
            end
        end else if (read) begin
            if (~empty) begin
                d_out<=ram[read_i];
                read_i<=read_i+1;
            end else begin
                $display("FIFO buffer is empty,don't allow read");
            end
        end
    end
end

  assign empty=(read_i==write_i)?1:0;
  assign full=((write_i[3]^read_i[3])&(write_i[2:0]==read_i[2:0]))?1:0;
endmodule
```

Verilog

```
`timescale 1ns / 1ps

module fifo_ram_test();
    reg clk;//时序控制
    reg clrn;//清0 高有效

    reg read;
    reg write;
```

```
    reg [31:0]d_in;
    wire [31:0]d_out;

    wire empty;
    wire full;

    always #5 clk=~clk;
    initial begin
        clk=0;clrn=0;
        #3;clrn=1;
        #1;clrn=0;//3~4ns 上升沿clrn清0

        #1;read=1;write=0;//5ns 空时读
        #10;read=0;write=1;d_in=32'habe1;//15ns 写第一个数据
        #10;d_in=d_in<<2;
        #10;d_in=d_in<<2;
        #10;d_in=d_in<<2;
        #10;d_in=d_in<<2;
        #10;d_in=d_in<<2;
        #10;d_in=d_in<<2;//写8个数据
        #10;d_in=d_in<<2;//写满写

        #10;read=1;write=0;//开始读
        #80;$finish;//0 10 20 30 40 50 60 70 80
    end

    fifo_ram_design  fifo_ram_design_inst (
        .clk(clk),
        .clrn(clrn),
```

```
        .read(read),
        .write(write),
        .d_in(d_in),
        .d_out(d_out),
        .empty(empty),
        .full(full)
    );
endmodule
```

Verilog

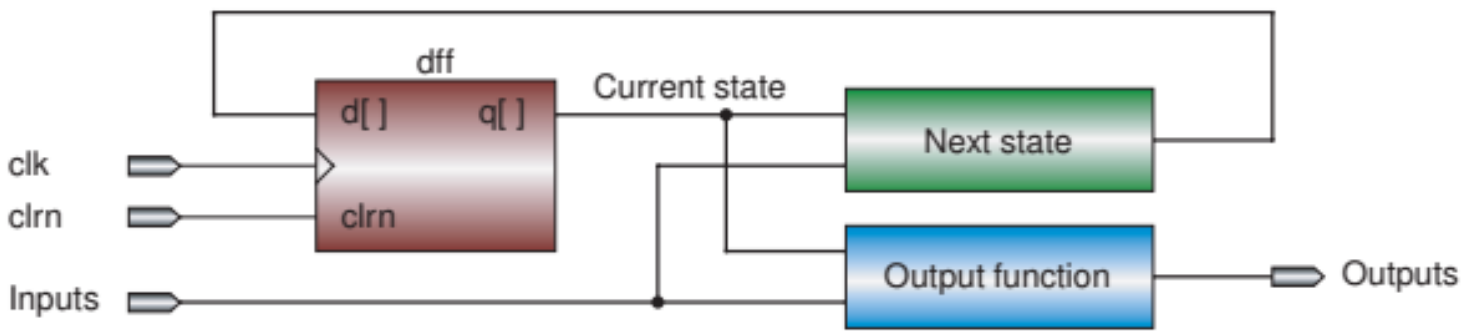
2.5.6 有穷状态机和计数器设计

有穷状态机

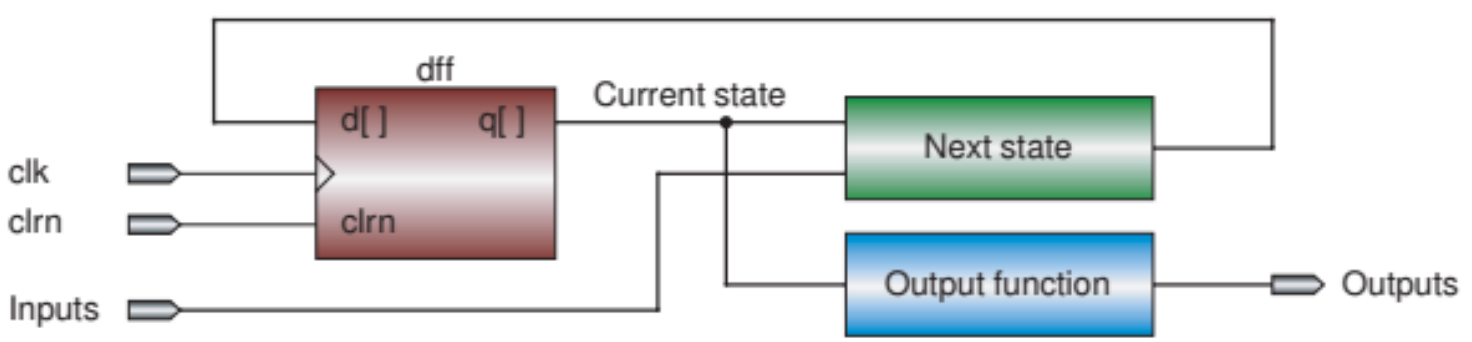
状态机

状态机的类型

状态机可以根据“输出是怎么决定”的分为两种：Moore型和Mealy型，如下图所示，**Moore型状态机的输出之和当前状态有关；Mealy型状态机的输出既和当前状态有关也和输入有关**



(a) Mealy model



(b) Moore model

Figure 2.35 Two models of the general finite state machine

状态机的设计流程

1. 基于陈述的问题绘制状态转移图

Make a state transition diagram based on the problem statement.

2. 根据状态数确定需要的DFF数量，并对状态进行编码

Determine the number of flip-flops and assign binary codes to the states.

N个状态需要$n = lceillog_2Nrceil$个DFF

一个状态的编码可以任意，只要状态之间的编码独立即可

3. 绘制下一个状态真值表
4. 根据绘制的下一个状态的真值表，使用卡诺图化简得到最简的下一个状态的生成表达式
5. 绘制输出真值表
6. 根据输出真值表，使用卡诺图化简得到最简的输出表达式
7. 构建并仿真电路

✍根据设计需求画出状态转移图，确定使用状态机类型，并标注出各种输入输出信号，更有助于编程。一般使用最多的是 Mealy 型 3 段式状态机

状态机设计如下：

- 首先，根据状态机的个数确定状态机编码。利用编码给状态寄存器赋值，代码可读性更好
- 状态机第一段，**时序逻辑，非阻塞赋值<=，传递寄存器的状态**
- 状态机第二段，**组合逻辑，阻塞赋值=，根据当前状态和当前输入，确定下一个状态机的状态**
- 状态机第三代，**时序逻辑，非阻塞赋值<=，因为是 Mealy 型状态机，根据当前状态和当前输入，确定输出信号**

计数器实现

下面使用状态机的形式设计一个计数器，计数值使用7段数码管输出

Figure 2.36 A counter with a seven-segment LED

上升沿更改状态，可以实现升序up计数(u=1, 0→1→2→3→4→5→0)也可以实现down计数(u=0, 0→5→4→3→2→1→0)。7段数码管是共阳极的

1. 基于陈述的问题绘制状态转移图

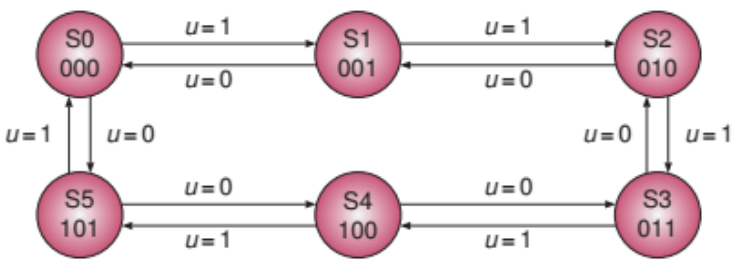


Figure 2.38 State transition diagram of the counter

2. 确定DFF数量并对状态编码

6个状态，需要$[log_2(6)] = 3$个DFF

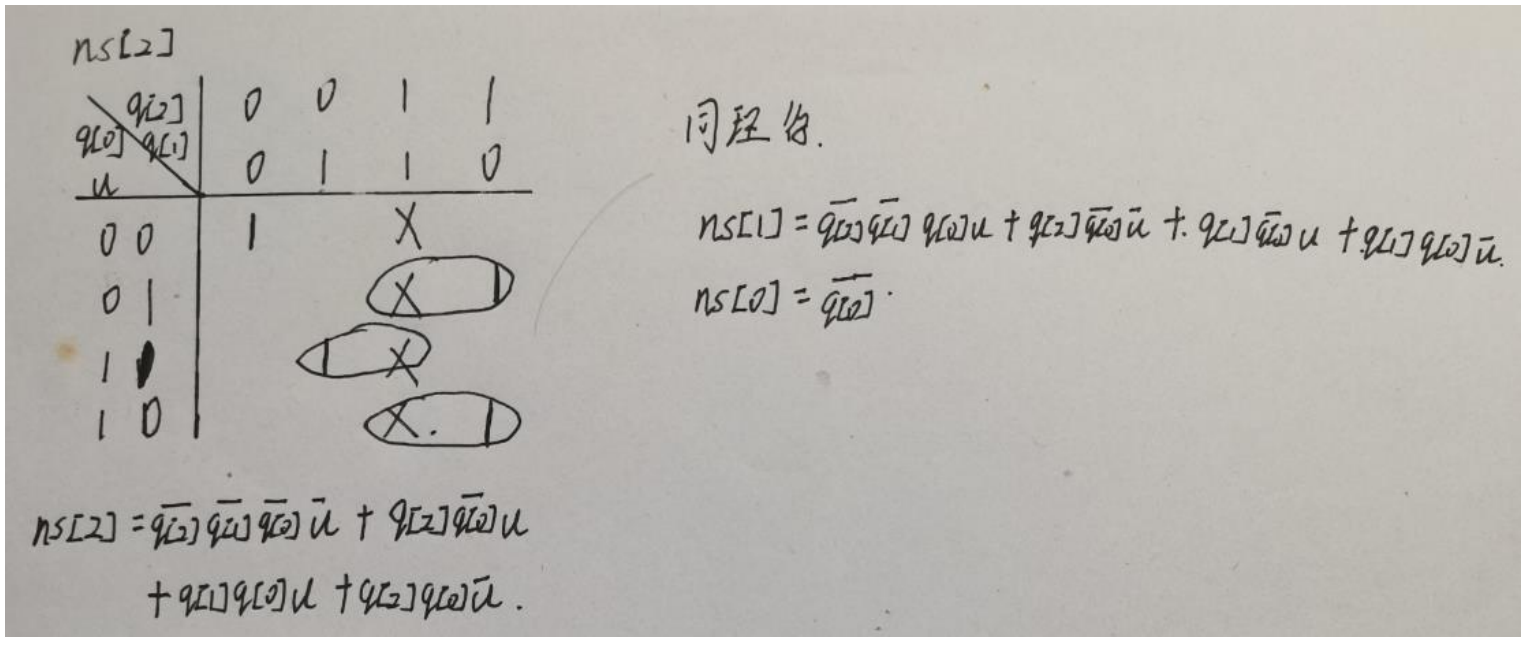
S0→000，S1→001，S2→010，S3→011，S4→100，S5→101

3. 绘制下一个状态真值表

Table 2.6 State transition table of the counter

Current state				Input	Next state			
	$q[2]$	$q[1]$	$q[0]$	u		$ns[2]$	$ns[1]$	$ns[0]$
S0	0	0	0	1	S1	0	0	1
				0	S5	1	0	1
S1	0	0	1	1	S2	0	1	0
				0	S0	0	0	0
S2	0	1	0	1	S3	0	1	1
				0	S1	0	0	1
S3	0	1	1	1	S4	1	0	0
				0	S2	0	1	0
S4	1	0	0	1	S5	1	0	1
				0	S3	0	1	1
S5	1	0	1	1	S0	0	0	0
				0	S4	1	0	0

4. 使用卡诺图对下一个状态真值表化简，得到最简的下一个状态的表达式



5. 绘制输出真值表

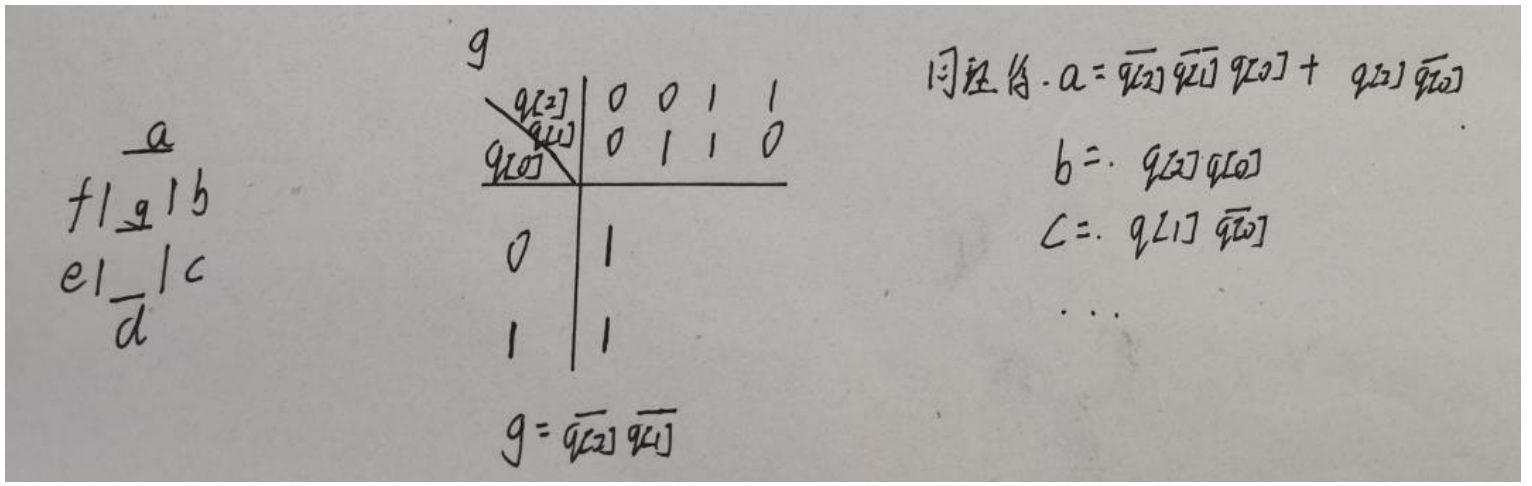
按abcdefg的顺序

0→0000001，1→1001111，2→0010010

3→0000110，4→1001100，5→0100100

$q[2:0]$	g f e d c b a
0 0 0	1 0 0 0 0 0 0
0 0 1	1 1 1 1 0 0 1
0 1 0	0 1 0 0 1 0 0
0 1 1	0 1 1 0 0 0 0
1 0 0	0 0 1 1 0 0 1
1 0 1	0 0 1 0 0 1 0

6. 使用卡诺图对输出真值表化简，得到最简的输出的表达式



$$a = overlineq[2]spaceoverlineq[1]q[0] + q[2]overlineq[0];$$

$$b = q[2]q[0];$$

$$c = q[1]overlineq[0];$$

$$d = overlineq[2]spaceoverlineq[1]q[0] + q[2]overlineq[0] = a;$$

$$e = q[2]overlineq[0] + q[0];$$

$$f = q[1]overlineq[0] + overlineq[2]q[0];$$

$$g = overlineq[2]spaceoverlineq[1].$$

7. 构建电路并仿真

按照三段式Mealy型行为风格构建

```
module counter_designByState(
    input clk,
    input rst,
    input u,
```

```
    output reg[2:0]num,
    output reg[6:0]seg
);

    reg [2:0]cur_state,next_state;
    //第一段 时序逻辑传状态
    always @(posedge clk) begin
        if (rst) begin//同步复位
            cur_state<=0;
        end else begin
            cur_state<=next_state;
        end
    end
    //第二段 组合逻辑决定下一个状态
    always @(*) begin
        next_state[0]=~cur_state[0];

next_state[1]=~cur_state[2]&~cur_state[1]&cur_state[0]&u|cur_state[1]&~cur_state[0]&u|cur_state[1]&cur_state[0]&~u|cur_state[2]&~cur_state[0]&~u;

next_state[2]=~cur_state[2]&~cur_state[1]&~cur_state[0]&~u|cur_state[1]&cur_state[0]&u|cur_state[2]&~cur_state[0]&u|cur_state[2]&cur_state[0]&~u;
    end
    //第三段 时序逻辑决定输出
    always @(posedge clk) begin
        if (rst) begin
            num=0;
        end else begin
            num=cur_state;
        end
        //case与if同级，如果设置if是非阻塞赋值，那么case就会用num的旧值
```

```
        case (num)//共阳极数码管
            3'b000:seg<=7'b0000001;
            3'b001:seg<=7'b1001111;
            3'b010:seg<=7'b0010010;
            3'b011:seg<=7'b0000110;
            3'b100:seg<=7'b1001100;
            3'b101:seg<=7'b0100100;
        endcase
    end
endmodule
```

Verilog

不使用状态机的行为风格实现

```
module counter_designByBehaioral(
    input clk,
    input rst,
    input u,

    output [2:0]num,
    output reg [6:0]seg
);
    reg [2:0]temp;
    assign num=temp;

    always @(posedge clk) begin
        if (rst) begin
            temp=0;
        end else begin
            if(u) begin
                temp=temp+1;
                if (temp==3'b110) begin
                    temp=3'b000;
```

```
            end
        end else begin
            temp=temp-1;
            if (temp==3'b111) begin
                temp=3'b101;
            end
        end
    end
    //case与if同级，如果设置if是非阻塞赋值，那么case就会用temp的旧值
    case (temp)//共阳极数码管
        3'b000:seg<=7'b0000001;
        3'b001:seg<=7'b1001111;
        3'b010:seg<=7'b0010010;
        3'b011:seg<=7'b0000110;
        3'b100:seg<=7'b1001100;
        3'b101:seg<=7'b0100100;
        endcase
    end
endmodule
```

Verilog

```
`timescale 1ns / 1ps

module counter_test();
    reg clk;
    reg rst;
    reg u;

    wire[2:0]num1,num2;
    wire[6:0]seg1,seg2;
```

```
    always #5 clk=~clk;

    initial begin
        clk=0;
        #3;rst=1;//3ns
        #3;rst=0;//6ns
        #8;u=1;//14ns
        //15 25 35 45 55 65
        #60;u=0;
        #60;$finish;
    end

    counter_designByState  counter_designByState_inst (
        .clk(clk),
        .rst(rst),
        .u(u),
        .num(num1),
        .seg(seg1)
    );
    counter_designByBehaioral  counter_designByBehaioral_inst (
        .clk(clk),
        .rst(rst),
        .u(u),
        .num(num2),
        .seg(seg2)
    );
endmodule
```

Verilog

习题

1. *Design the circuits of a three-input CMOS NAND gate and a three-input CMOS NOR gate, and simulate them with the downloaded simulators.*

三输入的与门、或门可以结合二输入的与非门和反相器进行设计

$$overlineabc = overlineoverlineoverlineabc$$

2. *In shift operations, in addition to the logical shift left, logical shift right, and arithmetic shift right, there is also an arithmetic shift left that keeps the sign bit (bit 31) unchanged, as shown in the following example.*

Original data (d):	11111111_00000000_00000000_11111111
Logical shift d to the left by 8 bits:	00000000_00000000_11111111_00000000
Arithmetic shift d to the left by 8 bits:	10000000_00000000_11111111_00000000
Logical shift d to the right by 8 bits:	00000000_11111111_00000000_00000000
Arithmetic shift d to the right by 8 bits:	11111111_11111111_00000000_00000000

Design a shifter that can perform the four types of shift operations described above. The input and output signals are the same as the barrel shifter given in this chapter.

原电路如下图所示：

Figure 2.16 Schematic diagram of 32-bit barrel shifter

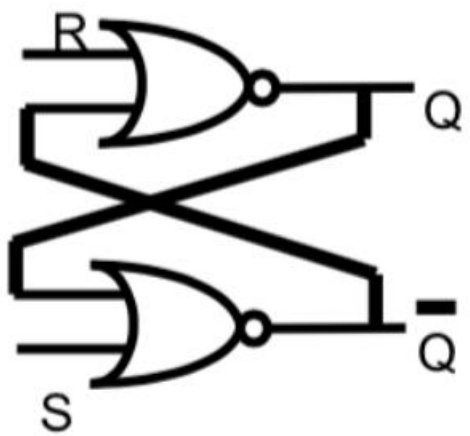
要再实现算术左移，可以在每个绿色多选器的0端再接一个多选器实现算术左移和逻辑左移的选择

3. *Design an active-high RS latch with NOR gates*

高有效的RS锁存器真值表如下：

r	s	q	qn
1	0	0	1
0	1	1	0
0	0	q	qn
1	1	x	x

或非门电路如下：r=1→q=0，r=0，~qn→q，故r与qn或非接q；s与q或非接qn



4. *Explain the reason why in Figure 2.19 two NOT gates were used to generate the control signal (c) for the slave D latch, instead of connecting the clk directly to the input c of the D latch*

?

[注释1] 有的计算机默认左移就是逻辑左移