2

A Brief Introduction to Logic Circuits and Verilog HDL

Logic circuit design is the foundation of computer design. This chapter briefly introduces the basic concept of the logic circuits and Verilog HDL (hardware description language), a language for implementing the circuits.

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. Full adder and multiplexer are two typical examples of combinational circuits. 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. A sequential circuit is used to construct a finite state machine; it needs flip-plops to record the current state. The counter and the control circuit of vending machines are two examples of the sequential circuits.

2.1 Logic Gates

Logic circuits consist of inputs, logic gates, and outputs. There are three basic logic gates: the AND gate, the OR gate, and the NOT gate. The following four gates are not basic gates but they are commonly used in the logic circuit designs: the NAND (NOT-AND) gate, the NOR (NOT-OR) gate, the XOR (Exclusive OR) gate, and the XNOR (Exclusive NOR) gate.

Figure 2.1 shows a logic circuit that consists of two inputs, seven gates as mentioned above, and seven outputs, one for each gate.

Basically, any unique names, except for the keywords of Verilog HDL, can be assigned to the inputs and outputs. The logic expressions of the outputs are given below. Note that the symbol "+" stands for an OR operation.

$$\begin{aligned}
\text{AND Gate (AND):} &\quad f_and &&= a \cdot b = ab \\
\text{OR Gate (OR):} &\quad f_or &&= a + b \\
\text{NOT Gate (NOT):} &\quad f_not &&= \overline{a} \\
\text{NOT-AND Gate (NAND):} &\quad f_nand &&= \overline{ab} \\
\text{NOT-OR Gate (NOR):} &\quad f_nor &&= \overline{a + b} \\
\text{Exclusive OR Gate (XOR):} &\quad f_xor &&= a \oplus b = \overline{a}b + a\overline{b} \\
\text{Exclusive NOR Gate (XNOR):} &\quad f_xnor &&= a \odot b = \overline{a}\overline{b} + ab = \overline{a \oplus b}
\end{aligned}$$

Table 2.1 lists the values of the outputs under all the combinations of the input values. These are the results we expect. Such table is called a truth table.

Computer Principles and Design in Verilog HDL, First Edition. Yamin Li.

Companion Website: www.wiley.com/go/li/verilog

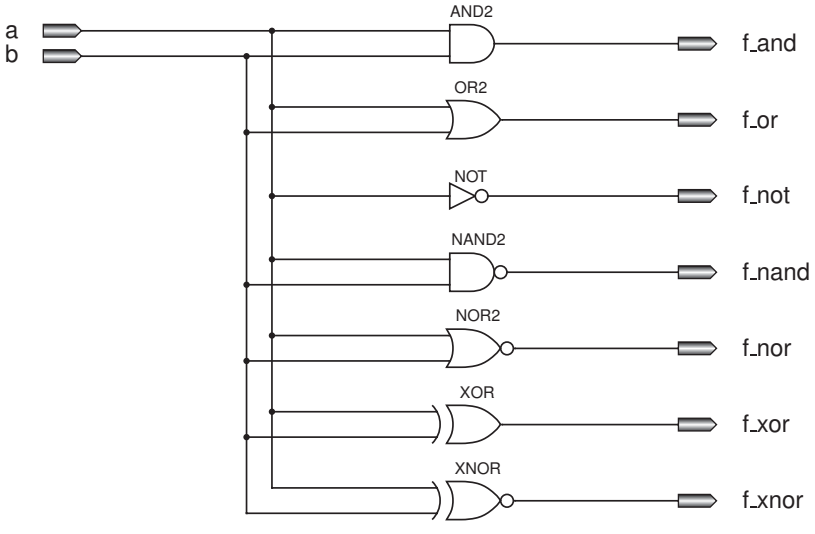


Figure 2.1 Three basic gates and four common gates

Table 2.1 Outputs of seven gates (truth table)

Input		Output						
a	*b*	*f_and*	*f_or*	*f_not*	*f_nand*	*f_nor*	*f_xor*	*f_xnor*
0	0	0	0	1	1	1	0	1
1	0	0	1	0	1	0	1	0
0	1	0	1	1	1	0	1	0
1	1	1	1	0	0	0	0	1

The following are the laws of Boolean algebra. Boolean algebra is a logical algebra of the truth values of 0 and 1, introduced by George Boole (1815–1864). It is ideal for expressing the behavior of logic circuit. The last two equations are called De Morgan's Law, introduced by Augustus De Morgan (1806–1871).

1. $A \cdot A = A$ (Idempotent law)
2. $A + A = A$ (Idempotent law)
3. $A \cdot B = B \cdot A$ (Commutative law)
4. $A + B = B + A$ (Commutative law)
5. $(A \cdot B) \cdot C = A \cdot (B \cdot C)$ (Associative law)
6. $(A + B) + C = A + (B + C)$ (Associative law)
7. $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ (Distributive law)
8. $A + (B \cdot C) = (A + B) \cdot (A + C)$ (Distributive law)
9. $A \cdot (A + B) = A$ (Absorption law)
10. $A + (A \cdot B) = A$ (Absorption law)
11. $A \cdot 0 = 0$ (Annulment law)
12. $A \cdot 1 = A$ (Identity law)
13. $A + 0 = A$ (Identity law)

14. $A + 1 = 1$ (Annulment law)
15. $A \cdot \overline{A} = 0$ (Complement law)
16. $A + \overline{A} = 1$ (Complement law)
17. $\overline{\overline{A}} = A$ (Double negation law)
18. $\overline{A \cdot B} = \overline{A} + \overline{B}$ (De Morgan's Law)
19. $\overline{A + B} = \overline{A} \cdot \overline{B}$ (De Morgan's Law)

Let's take an example to show how to design a combinational circuit. Suppose that we want to design a circuit with three inputs ($a0$, $a1$, and s) and an output (y). The output y will be the same as the input $a0$ if the input s is 0; y is the same as $a1$ otherwise. We call this circuit a 1-bit 2-to-1 multiplexer, denoted as `mux2x1`.

The general steps of designing a combinational circuit are as follows:

1. Write the truth table for each output.
2. Use the Karnaugh map to get a simplified expression for each output.
3. Design the circuit based on these expressions.
4. Verify the correctness of the circuit through logic simulation.

The truth table of the `mux2x1` is shown in Table 2.2. Figure 2.2 shows the Karnaugh map of the output y, from where we get its expression as follows. The expression gives the conditions at which the output y becomes a 1.

$$y = \overline{s}\ a0 + s\ a1$$

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	

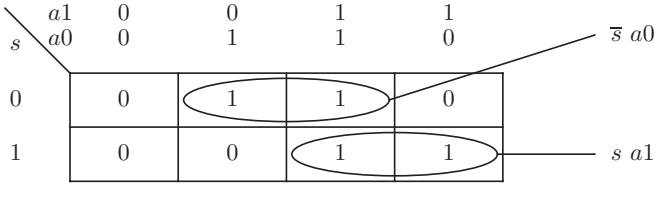


Figure 2.2 Karnaugh map for a 2-to-1 multiplexer

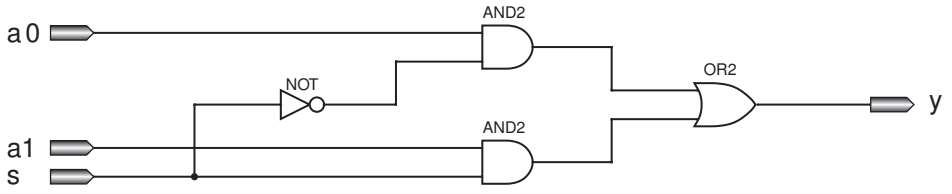


Figure 2.3 Schematic diagram of a 2-to-1 multiplexer

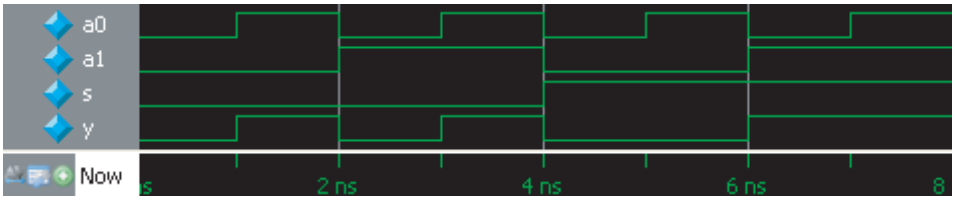


Figure 2.4 Waveform of a 2-to-1 multiplexer

Figure 2.3 shows the circuit schematic diagram and Figure 2.4 is the simulation waveform where the high voltage stands for logic 1 and the low voltage stands for logic 0. From Figure 2.4, we can see that, when s is 0, y is the same as $a0$, and when s is 1, y is the same as $a1$. This is what we wanted.

The logic expression of y can be written as a sum of products directly from the truth table and simplified with the laws of Boolean algebra:

$$\begin{aligned} y &= \overline{s}\ \overline{a1}\ a0 + \overline{s}\ a1\ a0 + s\ a1\ \overline{a0} + s\ a1\ a0 \\ &= \overline{s}\ (\overline{a1} + a1)\ a0 + s\ a1\ (\overline{a0} + a0) \\ &= \overline{s}\ a0 + s\ a1 \end{aligned}$$

2.2 Logic Circuit Design in Verilog HDL

Verilog HDL is a language for digital circuit design. It allows designers to design at various levels of abstraction. The Verilog HDL code listed below implements the circuit in Figure 2.1.

```
module gates7_structural (a,b,f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor);
    input  a, b;                                                    // inputs
    output f_and, f_or, f_not, f_nand, f_nor, f_xor, f_xnor;        // outputs
    and    i1 (f_and,  a, b);                          // and  (out, in1, in2)
    or     i2 (f_or,   a, b);                          // or   (out, in2, in2)
    not    i3 (f_not,  a);                             // not  (out, in)
    nand   i4 (f_nand, a, b);                          // nand (out, in1, in2)
    nor    i5 (f_nor,  a, b);                          // nor  (out, in1, in2)
    xor    i6 (f_xor,  a, b);                          // xor  (out, in1, in2)
    xnor   i7 (f_xnor, a, b);                          // xnor (out, in1, in2)
endmodule
```

The code starts with `module` and ends with `endmodule`; both are reserved keywords. `gates7_structural` is the name of the circuit (module name). The text in the parentheses is the list of inputs and outputs. The `input` and `output` are also keywords that define the input and output signals, respectively. The text starting with `//` is treated as a comment. The rest of the code is the main body that defines the operations of the circuit.

This example uses a low level of abstraction (gate level). The `and`, `or`, `not`, `nand`, `nor`, `xor`, and `xnor` are gate names. Following a gate name is an instance name. The first signal in the parentheses is the output of the gate and the rest of the signals are the inputs. The code is saved as a file. The file name should be same as the module name, and the extension name is `.v`.

To check the correctness of the logic operations, we must simulate the code. Therefore, a test code (called test bench) is required. Below is the code of the test bench. Following `` `timescale ``, the first `1ns` defines time unit and the second `1ns` defines the minimum time precision we can use in the simulation. The module name is `gates7_structural_tb`, and no inputs/outputs are required.

```
`timescale 1ns/1ns
module gates7_structural_tb;
    reg  a,b;
    wire f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor;
    gates7_structural g (a,b,f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor);
    initial begin
           a = 0; b = 0;
        #1 $display (" a=%b",a," b=%b",b," f_and=%b",f_and,
                     " f_or=%b",f_or," f_not=%b",f_not,
                     " f_nand=%b",f_nand," f_nor=%b",f_nor,
                     " f_xor=%b",f_xor," f_xnor=%b",f_xnor);
        #1 a = 1; b = 0;
        #1 $display (" a=%b",a," b=%b",b," f_and=%b",f_and,
                     " f_or=%b",f_or," f_not=%b",f_not,
                     " f_nand=%b",f_nand," f_nor=%b",f_nor,
                     " f_xor=%b",f_xor," f_xnor=%b",f_xnor);
        #1 a = 0; b = 1;
        #1 $display (" a=%b",a," b=%b",b," f_and=%b",f_and,
                     " f_or=%b",f_or," f_not=%b",f_not,
                     " f_nand=%b",f_nand," f_nor=%b",f_nor,
                     " f_xor=%b",f_xor," f_xnor=%b",f_xnor);
        #1 a = 1; b = 1;
        #1 $display (" a=%b",a," b=%b",b," f_and=%b",f_and,
                     " f_or=%b",f_or," f_not=%b",f_not,
                     " f_nand=%b",f_nand," f_nor=%b",f_nor,
                     " f_xor=%b",f_xor," f_xnor=%b",f_xnor);
        #1 a = 0; b = 0;
        #1 $finish;
    end
    initial begin
        $dumpfile("gates7_structural.vcd");
        $dumpvars;
    end
endmodule
```

It consists three parts. The first part invokes the `gates7_structural` module. The signals of `a` and `b` are declared as `reg` (register) type because they will be assigned with different values several times. The second part is an `initial` block that describes the behaviors of signals of `a` and `b`. `#` describes the time delay in the unit of nanoseconds (ns), which was defined in the beginning of the code. The `$display` statements show messages on the display. `%b` indicates the display of a signal in binary format. The last part describes the output file. The format of the output file is `vcd` (value change dump). This `vcd` file can be read by some applications to show the outputs graphically, GTKWave for example. If you do not use the `vcd` file further, this part can be deleted.

A tool (software) that integrates a compiler and a simulator is required. There are many such tools developed by famous companies, such as Cadence, Synopsys, Mentor Graphics, LogicVision, Xilinx, and Altera. There are also some open-source free tools; `Icarus Verilog` is an example. The `iverilog` command in the following text checks the correctness of the grammar and generates an output file `a.out`. Rather than `a.out`, you can designate an output file name with the `-o` option in the `iverilog` command line. The `vvp` command executes `a.out`, which shows the execution results of the test bench.

```
[cpu_verilog]$ iverilog gates7_structural_tb.v gates7_structural.v
[cpu_verilog]$ vvp a.out
VCD info: dumpfile gates7_structural.vcd opened for output.
 a=0 b=0 f_and=0 f_or=0 f_not=1 f_nand=1 f_nor=1 f_xor=0 f_xnor=1
 a=1 b=0 f_and=0 f_or=1 f_not=0 f_nand=1 f_nor=0 f_xor=1 f_xnor=0
 a=0 b=1 f_and=0 f_or=1 f_not=1 f_nand=1 f_nor=0 f_xor=1 f_xnor=0
 a=1 b=1 f_and=1 f_or=1 f_not=0 f_nand=0 f_nor=0 f_xor=0 f_xnor=1
```

The following code also implements the circuit in Figure 2.1. It uses the dataflow style. `assign` is a keyword that assigns a logic operation to an output. The parentheses are required because the NOT operation has the highest priority.

```
module gates7_dataflow (a,b,f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor);
    input  a, b;                                                    // inputs
    output f_and, f_or, f_not, f_nand, f_nor, f_xor, f_xnor;        // outputs
    assign f_and  =   a & b;                            // and
    assign f_or   =   a | b;                            // or
    assign f_not  = ~ a;                                // not
    assign f_nand = ~(a & b);                           // nand = not(a and b)
    assign f_nor  = ~(a | b);                           // nor = not(a or  b)
    assign f_xor  =   a ^ b;                            // xor
    assign f_xnor = ~(a ^ b);                           // xnor = not(a xor b)
endmodule
```

The test bench code is listed below. Instead of using `$display` on every change of the time, we used `$monitor` once to display the output messages. Combining with the variable `$time`, `$monitor` makes the test bench code shorter. The two `always` statements give the behaviors of signals of `a` and `b`.

```
‘timescale 1ns/1ns
module gates7_dataflow_tb;
    reg  a,b;
    wire f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor;
    gates7_dataflow g7 (a,b,f_and,f_or,f_not,f_nand,f_nor,f_xor,f_xnor);
    initial begin
```

```
        $display("time\ta\tb\tand\tor\tnot\tnand\tnor\txor\txnor");
        a = 0;
        b = 0;
        #5 $finish;
    end
    initial begin
        $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);
    end
    always #1 a = !a;
    always #2 b = !b;
endmodule
```

The following text shows the compilation command and the execution results of the test bench code.

```
[cpu_verilog]$ iverilog gates7_dataflow_tb.v gates7_dataflow.v
[cpu_verilog]$ vvp a.out
time    a       b       and     or      not     nand    nor     xor     xnor
 0:     0       0       0       0       1       1       1       0       1
 1:     1       0       0       1       0       1       0       1       0
 2:     0       1       0       1       1       1       0       1       0
 3:     1       1       1       1       0       0       0       0       1
 4:     0       0       0       0       1       1       1       0       1
```

2.3 CMOS Logic Gates

This section describes how to use low-level CMOS (complementary metal–oxide–semiconductor) transistors to design logic gates and gives the gate implementations in Verilog HDL.

2.3.1 CMOS Inverter

In CMOS technology, both PMOS (p-type or p-channel metal–oxide–semiconductor) and NMOS (n-type or n-channel metal–oxide–semiconductor) transistors are used. For PMOS transistors, as shown in Figure 2.5(a), 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. On the other hand, for the NMOS, as shown in Figure 2.5(b), 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. Figure 2.5(c) shows a CMOS inverter (a NOT gate).

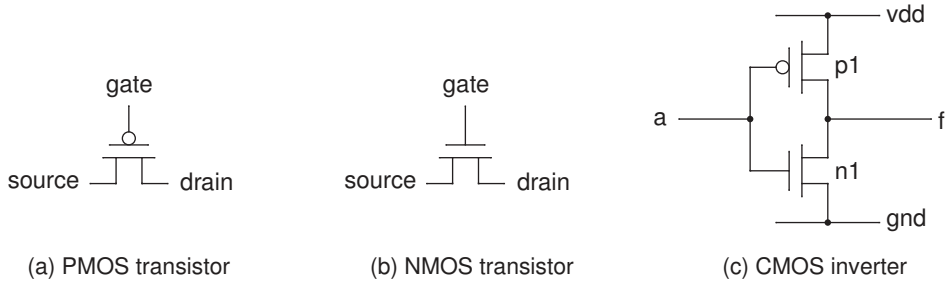


Figure 2.5 Schematic diagram of a CMOS inverter

The following Verilog HDL code implements the CMOS inverter. `supply0`, `supply1`, `pmos`, and `nmos` are keywords that stand for ground, power supply, PMOS transistor, and NMOS transistor, respectively.

```
module cmosnot (f, a);                                    // cmos inverter
    input   a;                                            // input  a
    output  f;                                            // output f = ~a
    supply1 vdd;                                          // logic 1 (power)
    supply0 gnd;                                          // logic 0 (ground)
    // pmos (drain, source, gate);
    pmos p1 (f,     vdd,    a);
    // nmos (drain, source, gate);
    nmos n1 (f,     gnd,    a);
endmodule
```

Below is the code of the test bench.

```
‘timescale 1ns/1ns
module cmosnot_tb;
    reg  a;
    wire f;
    cmosnot cmos_not (f,a);
    initial begin
           a = 0;
        #1 a = 1;
        #1 a = 0;
        #1 $finish;
    end
    initial begin
        $monitor("%2d:\ta = %b\tf = %b",$time,a,f);
    end
endmodule
```

The execution results of the test bench code are shown below, from where we can see that the output `f` is the reverse of the input `a`.

```
[cpu_verilog]$ iverilog cmosnot_tb.v cmosnot.v
[cpu_verilog]$ vvp a.out
 0:     a = 0   f = 1
 1:     a = 1   f = 0
 2:     a = 0   f = 1
```

2.3.2 CMOS NAND and NOR Gates

For a NAND gate, the output is 1 if an input is a 0. Figure 2.6(a) shows the CMOS implementation of a 2-input NAND gate. When both `a` and `b` are 1, `p1` and `p2` are off, and `n1` and `n2` are on, and the output is 0. Otherwise, the output is 1.

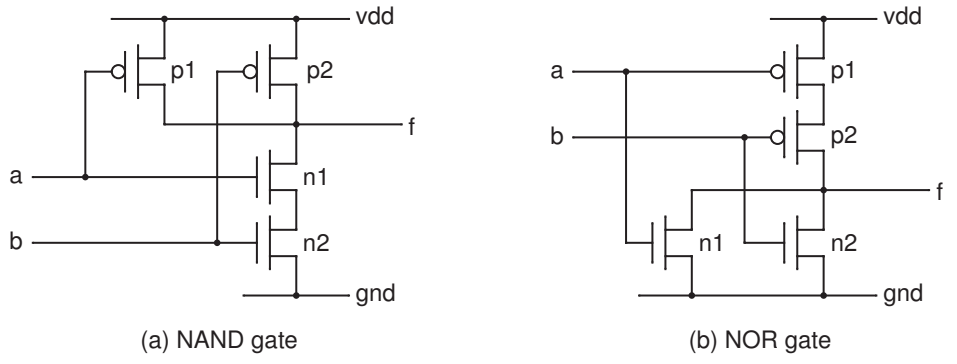


Figure 2.6 Schematic diagram of NAND and NOR gates

Below is the transistor-level Verilog HDL code for the NAND gate. `wire w_n` defines a wire that is used to connect `n1` and `n2` transistors. Basically, every wire in a circuit requires a name in its Verilog HDL implementation code.

```
module cmosnand (f, a, b);                  // cmos nand
    input   a, b;                           // inputs: a, b
    output  f;                              // output: f = ~(a & b)
    supply1 vdd;                            // logic 1 (power)
    supply0 gnd;                            // logic 0 (ground)
    wire    w_n;                            // wire: connects 2 nmos transistors
    // pmos (drain, source, gate);
    pmos p1 (f,     vdd,    a);
    pmos p2 (f,     vdd,    b);
    // nmos (drain, source, gate);
    nmos n1 (f,     w_n,    a);
    nmos n2 (w_n,   gnd,    b);
endmodule
```

For a NOR gate, the output is 0 if an input is a 1. Figure 2.6(b) shows the CMOS implementation of a 2-input NOR gate. When both `a` and `b` are 0, `p1` and `p2` are on, and `n1` and `n2` are off, the output is 1. Otherwise, the output is 0. Below is the transistor-level Verilog HDL code for the NOR gate.

```
module cmosnor (f, a, b);                   // cmos nor
    input   a, b;                           // inputs: a, b
    output  f;                              // output: f = ~(a | b)
    supply1 vdd;                            // logic 1 (power)
    supply0 gnd;                            // logic 0 (ground)
    wire    w_p;                            // wire: connects 2 pmos transistors
    // nmos (drain, source, gate);
    nmos n1 (f,     gnd,    a);
    nmos n2 (f,     gnd,    b);
    // pmos (drain, source, gate);
    pmos p1 (w_p,   vdd,    a);
    pmos p2 (f,     w_p,    b);
endmodule
```

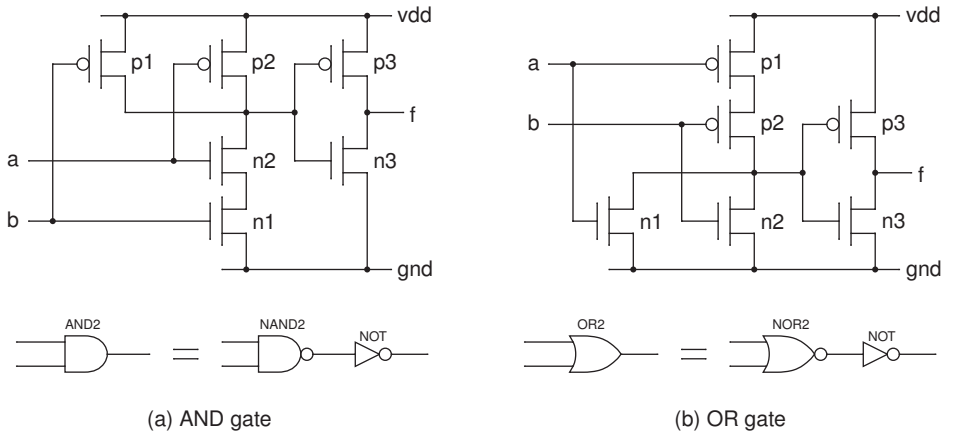


Figure 2.7 Schematic diagram of AND and OR gates

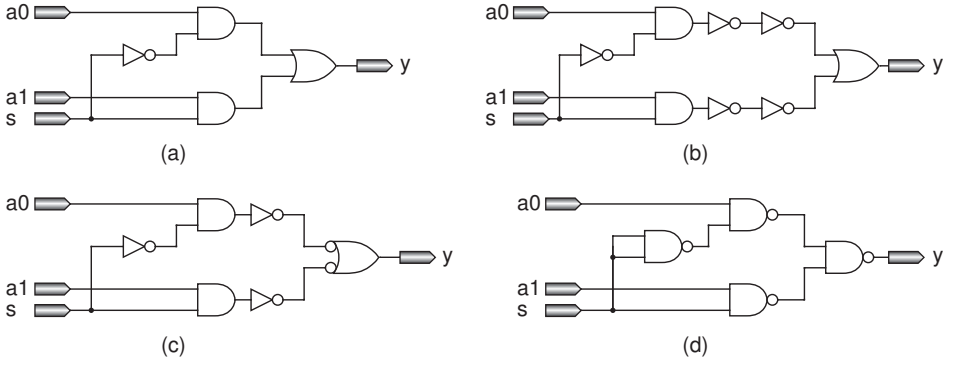


Figure 2.8 Implementing a multiplexer using only NAND gates

Figure 2.7 shows the CMOS implementations of the AND and OR gates. An AND gate is constructed by appending a NOT gate to the output of a NAND gate (Figure 2.7(a)). Similarly, an OR gate is constructed by appending a NOT gate to the output of a NOR gate (Figure 2.7(b)).

From Figure 2.7, 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. Figure 2.8 illustrates how to use only NAND gates to design circuits.

Figure 2.8(a) shows a 2-to-1 multiplexer designed with AND, OR, and NOT gates. Figure 2.8(b) adds two NOT gates to each output of the AND gates. Figure 2.8(c) combines a NOT gate and an OR gate. By applying De Morgan's Law, we get the final circuit of the multiplexer that uses only NAND gates, as shown as in Figure 2.8(d). Some gate array chips contain an array of NAND-only gates.

2.4 Four Levels/Styles of Verilog HDL

Verilog HDL supports a variety of description levels. This section takes the 2-to-1 multiplexer as an example to show the implementations in the following four levels (styles): (i) transistor switch level, (ii) gate level or structural style, (iii) dataflow style, and (iv) behavioral style.

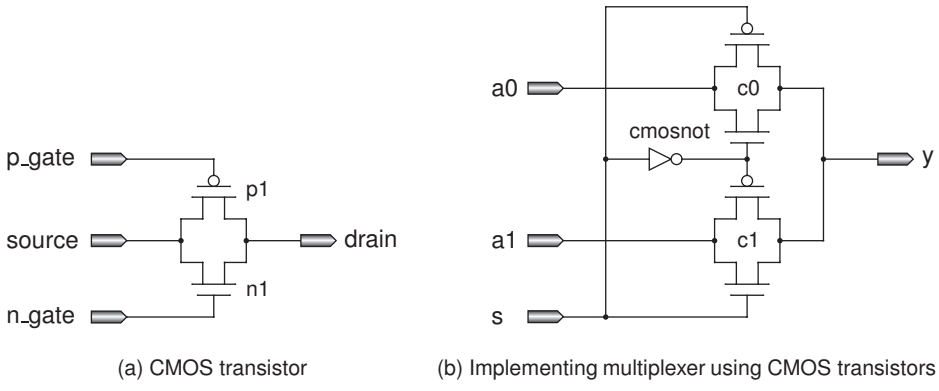


Figure 2.9 Implementing a multiplexer using CMOS transistors

2.4.1 Transistor Switch Level

Figure 2.9 shows the schematic circuit of a 2-to-1 multiplexer designed with CMOS transistors. Figure 2.9(a) is a CMOS transistor that contains a PMOS transistor and an NMOS transistor. If the input `p_gate` is 0 and the input `n_gate` is 1, both transistors of `p1` and `n1` are on and the output `drain` is the same as the input `source`.

The multiplexer shown in Figure 2.9(b) uses two CMOS transistors and a CMOS NOT gate that we already described. When the input `s` is a 0, `c0` is on and `c1` is off; thus the output `y` is the same as the input `a0`. Otherwise, `c0` is off and `c1` is on, and the output `y` is the same as the input `a1`. The Verilog HDL code of the 2-to-1 multiplexer implemented with CMOS transistors is listed below.

```
module mux2x1_cmos (a0, a1, s, y);      // multiplexer using cmos transistors
    input  s, a0, a1;                   // inputs
    output y;                           // output
    wire   sn;                          // internal wire, output of cmosnot
    // cmosnot  (f,  a);                // a cmos invert: (out, in)
    cmosnot inv (sn, s);
    // cmoscmos (drain, source, n_gate, p_gate);
    cmoscmos c0 (y,     a0,     sn,     s);
    cmoscmos c1 (y,     a1,     s,      sn);
endmodule
```

The code listed above invokes `cmosnot` (CMOS NOT gate) and `cmoscmos` (CMOS transistor) modules. The `cmosnot` module was given in the previous section. The `cmoscmos` module and the multiplexer test bench are listed below.

```
module cmoscmos (drain, source, n_gate, p_gate);                // cmos gate
    input  source, n_gate, p_gate;
    output drain;
    pmos p1 (drain, source, p_gate);     // pmos name (drain, source, gate);
    nmos n1 (drain, source, n_gate);     // nmos name (drain, source, gate);
endmodule
```

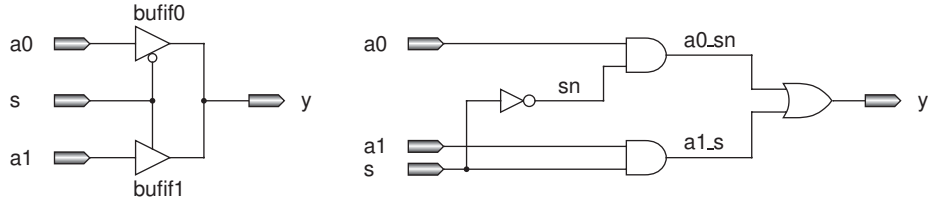
```
‘timescale 1ns/1ns
module mux2x1_cmos_tb;
    reg  s,a0,a1;
    wire y;
    mux2x1_cmos mux2x1 (a0,a1,s,y);
    initial begin
           a0 = 0;
           a1 = 0;
           s  = 0;
           $display("time\ts\ta1\ta0\ty");
           $monitor("%2d:\t%b\t%b\t%b\t%b",$time,s,a1,a0,y);
        #8 $finish;
    end
    always #1 a0 = !a0;
    always #2 a1 = !a1;
    always #4 s  = !s;
endmodule
```

The compilation and execution results are shown below, from where we can see that the multiplexer works correctly.

```
[cpu_verilog]$ iverilog mux2x1_cmos_tb.v mux2x1_cmos.v cmosnot.v cmoscmos.v
[cpu_verilog]$ vvp a.out
time     s        a1       a0       y
 0:      0        0        0        0
 1:      0        0        1        1
 2:      0        1        0        0
 3:      0        1        1        1
 4:      1        0        0        0
 5:      1        0        1        0
 6:      1        1        0        1
 7:      1        1        1        1
 8:      0        0        0        0
```

2.4.2 *Logic Gate Level*

We give two circuits of the multiplexer implemented with logic gates. Figure 2.10(a) shows the circuit of the multiplexer implemented with two tri-state gates. A tri-state gate is a logic gate with three states:



(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

logic 0, logic 1, or high impedance. If the input s is a 0, the output of the tri-state buffer bufif1 is in high impedance and the output y is the same as the input a0. Otherwise, the output of the tri-state buffer bufif0 is in high impedance and the output y is the same as the input a1.

The Verilog HDL code implementing the multiplexer with tri-state gates is listed below. bufif0 and bufif1 are two tri-state gates supported by Verilog HDL.

```
module mux2x1_3s (a0, a1, s, y);        // multiplexer using tri-state gates
    input  s, a0, a1;                   // inputs
    output y;                           // output
    // bufif0 (out, in,  ctl);          // tri-state buffer: ctl==0: out=in;
    bufif0 b0 (y,   a0,  s);            //   ctl == 1: out = high-impedance;
    // bufif1 (out, in,  ctl);          // tri-state buffer: ctl==1: out=in;
    bufif1 b1 (y,   a1,  s);            //   ctl == 0: out = high-impedance;
endmodule
```

Figure 2.10(b) shows the circuit of the multiplexer implemented with ordinary AND, OR, and NOT gates. Its Verilog HDL code is listed below.

```
module mux2x1_gate (a0, a1, s, y);      // multiplexer using ordinary gates
    input  s, a0, a1;                   // inputs
    output y;                           // output
    wire   sn, a0_sn, a1_s;             // internal wires
    not i0 (sn,    s);                  // not (out, in);
    and i1 (a0_sn, a0,    sn);          // and (out, in1, in2);
    and i2 (a1_s,  a1,    s );          // and (out, in1, in2);
    or  i3 (y,     a0_sn, a1_s);        // or  (out, in1, in2);
endmodule
```

2.4.3 *Dataflow Style*

Different from transistor and gate levels, the Verilog HDL code in dataflow style does not use any particular component (transistor or gate). It uses the keyword assign and logic expression to assign a value to a net or an output. A net or an output can be assigned only once in the code. Below is the dataflow style Verilog HDL code that implements the multiplexer.

```
module mux2x1_dataflow1 (a0, a1, s, y);       // multiplexer, dataflow style
    input  s, a0, a1;                         // inputs
    output y;                                 // output
    assign y = ~s & a0 | s & a1;              // logic expression
endmodule
```

Below is another implementation which also uses assign but does not use logic expression. The input s is queried. If it is true (logic 1), the a1 input is assigned to y; otherwise a0 is assigned to y. This code is somewhat in behavioral style.

```
module mux2x1_dataflow2 (a0, a1, s, y);       // multiplexer, dataflow style
    input  s, a0, a1;                         // inputs
    output y;                                 // output
    assign y = s ? a1 : a0;                   // if (s==1) y=a1; else y=a0;
endmodule
```

2.4.4 Behavioral Style

Behavioral style is the highest level of abstraction. The main feature of this style is that the functionality of a circuit is expressed by an algorithmic description. By using the behavioral style, a designer who does not know the details of low-level logic circuit design can design the circuits, just as a programmer can develop software in C or Java although he or she does not know the assembly language programming at all.

The behavioral Verilog HDL code must be inside procedural blocks. There are two types of procedural blocks: `always` and `function`. Inside a block, we can use control statements similar to C programming language, such as `if-else`, `case`, and `for` loop. A signal can be assigned multiple times with different values inside a block. Such signals must be declared as `reg` type outside the procedural blocks. Although `reg` stands for register, we can design the combinational circuits with signals of `reg` type. In a combinational circuit, there is no register.

The following example code of the 2-to-1 multiplexer uses `always` block and the `if-else` statement. As mentioned above, the output `y` is declared as `reg` type (it cannot use `wire` type). All the combinational values of `s` are checked inside the block. This ensures that a combinational circuit will be generated although `y` is declared as a register type.

```
module mux2x1_behavioral_if_else (a0,a1,s,y);        // multiplexer, if else
    input  s, a0, a1;                                // inputs
    output y;                                        // output
    reg    y;                                        // y cannot be a wire
    always @ (s or a0 or a1) begin                   // always block
        if (s) begin                                 // if (s == 1)
            y = a1;                                  //     y = a1;
        end else begin                               // if (s == 0)
            y = a0;                                  //     y = a0;
        end
    end
endmodule
```

The following code uses only the `if` statement but assigns a default value to `y` first. This also ensures that a combinational circuit will be generated.

```
module mux2x1_behavioral_if (a0,a1,s,y);       // multiplexer, default first
    input  s, a0, a1;                          // inputs
    output y;                                  // output
    reg    y;                                  // y cannot be a wire
    always @ (s or a0 or a1) begin             // always block
        y = a0;                                // y = a0;
        if (s) begin                           // if (s == 1)
            y = a1;                            //     y = a1;
        end
    end
endmodule
```

The following code uses the `always` block and the `case` statement. All the combinational cases of `s` are checked inside the block. This ensures that a combinational circuit will be generated.

```
module mux2x1_behavioral_case_all (a0,a1,s,y);     // multiplexer, all cases
    input  s, a0, a1;                              // inputs
    output y;                                      // output
    reg    y;                                      // y cannot be a wire
    always @ (s or a0 or a1) begin                 // always block
        case (s)                                   // cases:
            1'b0: y = a0;                          // if (s == 0) y = a0;
            1'b1: y = a1;                          // if (s == 1) y = a1;
        endcase
    end
endmodule
```

The following code shows how to use `default` in the `case` statement. All the cases that are not specified obviously go to the case of `default`.

```
module mux2x1_behavioral_case_default (a0,a1,s,y);   // multiplexer, default
    input  s, a0, a1;                                // inputs
    output y;                                        // output
    reg    y;                                        // y cannot be a wire
    always @ (s or a0 or a1) begin                   // always block
        case (s)                                     // cases:
            1'b1:    y = a1;                         // if (s == 1) y = a1;
            default: y = a0;                         // other cases y = a0;
        endcase
    end
endmodule
```

The four examples given above use the `always` block. The next code uses a `function` block. 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. The following example uses `case` statement inside the function.

```
module mux2x1_behavioral_function_case_all (a0,a1,s,y);      // multiplexer,
    input  s, a0, a1;             // inputs                  //    function
    output y;                     // output                  //    all cases
    assign y = sel (a0,a1,s);     // call a function with parameters
    function sel;                 // function name (= return value)
        input a,b,c;              // notice the order of the input arguments
        case (c)                  // cases:
            1'b0: sel = a;        // if (c==0) return value = a
            1'b1: sel = b;        // if (c==1) return value = b
        endcase
    endfunction
endmodule
```

The following code also uses the `function` block. Inside the function, the `if-else` statement is used.

```
module mux2x1_behavioral_function_if_else (a0,a1,s,y);       // multiplexer,
    input  s, a0, a1;             // inputs                  //     function
    output y;                     // output                  //     if else
    assign y = sel (a0,a1,s);     // call a function with parameters
    function sel;                 // function name (= return value)
        input a,b,c;              // notice the order of the input arguments
        if (c) sel = b;           // if (c==1) return value = b
        else   sel = a;           // if (c==0) return value = a
    endfunction
endmodule
```

In addition to `always` and `function`, there is also another procedural block: `initial`, which is executed once the simulation starts. This is useful in writing the test benches, which were described in the previous section.

2.5 Combinational Circuit Design

The outputs of combinational circuits are dependent only on the present inputs. This section introduces some common combinational circuits, including 32-bit multiplexers, demultiplexer, decoder, priority encoder, and barrel shifter, which will be used in the design of CPUs. The combinational arithmetic circuits, such as full adder and parallel multiplier, will be given in Chapter 3.

2.5.1 Multiplexer

A multiplexer selects an input from multiple inputs. We already introduced a 1-bit 2-to-1 multiplexer. The following code implements a 32-bit 2-to-1 multiplexer. The `[31:0]` denotes a bus that has 32 bits.

```
module mux2x32 (a0,a1,s,y);                    // multiplexer, 32 bits
    input  [31:0] a0, a1;                      // inputs, 32 bits
    input         s;                           // input,   1 bit
    output [31:0] y;                           // output, 32 bits
    assign        y = s ? a1 : a0;             // if (s==1) y=a1; else y=a0;
endmodule
```

The following code implements a 32-bit 4-to-1 multiplexer. A 4-to-1 multiplexer selects one input from four inputs and needs a 2-bit selection signal. The `assign` statement can also be put in above the function block.

```
module mux4x32 (a0,a1,a2,a3,s,y); // 4-to-1 multiplexer, 32-bit
    input  [31:0] a0, a1, a2, a3; // inputs, 32 bits
    input   [1:0] s;              // input,   2 bits
    output [31:0] y;              // output, 32 bits
    function  [31:0] select;      // function name (= return value, 32 bits)
        input [31:0] a0,a1,a2,a3; // notice the order of the input arguments
        input  [1:0] s;           // notice the order of the input arguments
        case (s)                  // cases:
            2'b00: select = a0;   // if (s==0) return value = a0
            2'b01: select = a1;   // if (s==1) return value = a1
```

```
            2'b10: select = a2;   // if (s==2) return value = a2
            2'b11: select = a3;   // if (s==3) return value = a3
        endcase
    endfunction
    assign y = select(a0,a1,a2,a3,s);   // call the function with parameters
endmodule
```

2.5.2 Decoder

An m-2^m decoder works as follows: suppose there is a 1-bit input *ena*, m-bit input $n[m-1:0]$, and 2^m-bit output $d[2^m-1:0]$. If *ena* = 1, then 1-bit $d[n] = 1$, other bits are 0; if *ena* = 0, all the bits of output $d[2^m-1:0]$ are 0. Table 2.3 is the truth table of a 3-8 decoder.

We can write the logic expression for each output directly from the truth table; for example, $d[3] = ena\ \overline{n[2]}\ n[1]\ n[0]$. Figure 2.11 shows the circuit of the 3-8 decoder. The inputs and outputs of the gate signals can be labeled with wire names so that there is no need to connect them to the input or output pins. Based on the logic expressions, we can write the Verilog HDL code in structural or dataflow style easily (we do not show it here).

Below is a Verilog HDL implementation of the 3-8 decoder in behavioral style. Inside the `always` block, every bit of `d` is assigned with a default value 0. Then a 1 is assigned to the *n*th bit of `d` if *ena* is a 1.

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

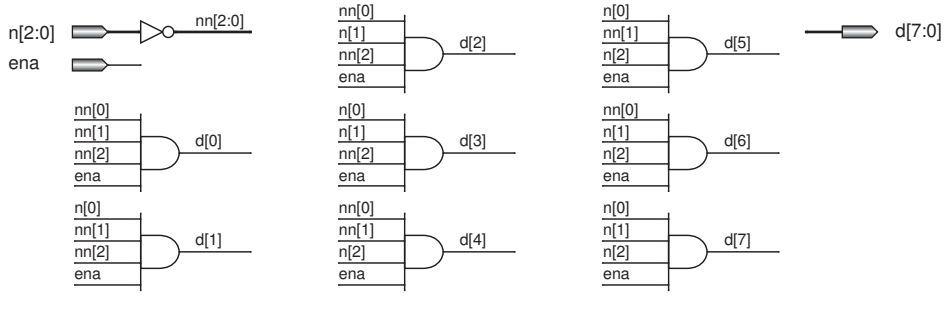


Figure 2.11 Schematic diagram of decoder with enable control

```
module decoder3e (n,ena,d);          // 3-8 decoder with enable
    input  [2:0] n;                  // inputs:  n, 3 bits
    input        ena;                // input:   enable
    output [7:0] d;                  // outputs: d, 2^3 = 8 bits
    reg    [7:0] d;                  // d cannot be a wire
    always @ (ena or n) begin        // always block
        d    = 8'b0;                 // let d = 00000000 first
        d[n] = ena;                  // then let n-th bit of d = ena (0 or 1)
    end
endmodule
```

2.5.3 *Encoder*

An encoder performs the reverse operation of the decoder. It has a maximum of 2^m inputs and m outputs. The symbol and the truth table of an 8-3 encoder are shown in Figure 2.12(b) and Table 2.4, respectively. Figure 2.12(a) is a 3-8 decoder for the comparison with the 8-3 encoder.

As shown in Figure 2.12(b), the input $d[7:0]$ has 8 bits ($m = 3$) and the output $n[2:0]$ has 3 bits. Another output, g, indicates whether there is a 1 in the $d[7:0]$ or not. If we do not add g, we cannot

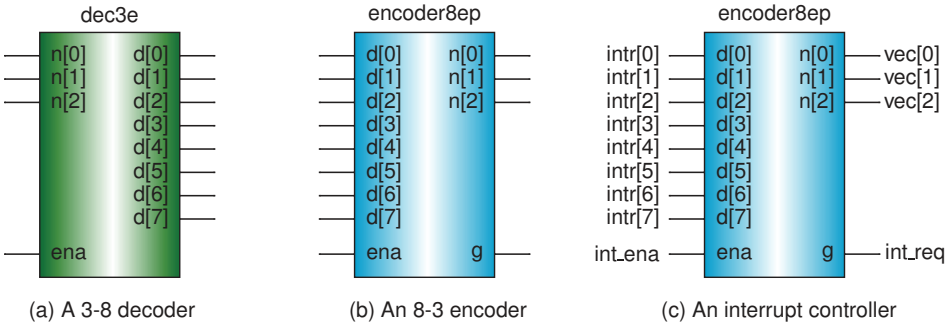


Figure 2.12 Decoder and encoder

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

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

distinguish whether the value of the input $d[7:0]$ is 00000001 or 00000000. The input *ena* is an enable signal. *g* will output a 0 if the *ena* is a 0, irrespective of the $d[7:0]$.

Table 2.4 does not list all combinations of the input $d[7:0]$. This encoder will generate the wrong output code when there is more than one input present at logic 1. For example, if both $d[1]$ and $d[2]$ are 1 at the same time, the resulting output is neither 001 nor 010, but will be 011, indicating $d[3]$ at a logic 1.

To solve this problem, priority encoders are commonly used. Table 2.5 shows the truth table of an 8-3 priority encoder. Each input has an assigned priority. In our priority encoder, the input $d[7]$ has the highest priority and $d[0]$ has the lowest priority. The value x stands for "don't care", meaning that it can be either a 1 or a 0, and has no effect on the output. Table 2.5 lists all combinations of the inputs $d[7:0]$ and *ena*.

From the truth table in Table 2.5, we can express the outputs as follows:

$$
\begin{aligned}
n[2] &= ena\ \ (d[7] + \overline{d[7]}\, d[6] + \overline{d[7]}\, \overline{d[6]}\, d[5] + \overline{d[7]}\, \overline{d[6]}\, \overline{d[5]}\, d[4]) \\
&= ena\ \ (d[7] + d[6] + \overline{d[7]}\, \overline{d[6]}\, (d[5] + d[4])) \\
&= ena\ \ (d[7] + d[6] + \overline{d[7]} + \overline{d[6]}\, (d[5] + d[4])) \\
&= ena\ \ (d[7] + d[6] + d[5] + d[4]) \\
n[1] &= ena\ \ (d[7] + d[6] + \overline{d[5]}\, \overline{d[4]}\, d[3] + \overline{d[5]}\, \overline{d[4]}\, d[2]) \\
n[0] &= ena\ \ (d[7] + \overline{d[6]}\, d[5] + \overline{d[6]}\, \overline{d[4]}\, d[3] + \overline{d[6]}\, \overline{d[4]}\, \overline{d[2]}\, d[1]) \\
g &= ena\ \ (d[7] + d[6] + d[5] + d[4] + d[3] + d[2] + d[1] + d[0])
\end{aligned}
$$

We can draw the logic diagram or write the dataflow style Verilog HDL codes based on the logic expressions of the outputs. Here we give a behavioral style version of the 8-3 priority encoder Verilog HDL codes, as shown below.

```
module encoder8ep (d,ena,n,g);          // 8-3 priority encoder with enable
    input  [7:0] d;                     // input:   d, 8 bits
    input        ena;                   // input:   enable
    output [2:0] n;                     // outputs: n, log_2 8 = 3 bits
    output       g;                     // output:  g = 1 if d is not 0
    assign       g = ena & |d;          // if there is at least a 1 in d
```

```
    assign       n = enc(ena, d);        // call a function enc
    function [2:0] enc;                  // the function enc
        input       e;                   // input of the function
        input [7:0] d;                   // input of the function
        casex ({e,d})                    // cases, x: don't care
            9'b1_1xxxxxxx: enc = 3'd7;   // d[7] has the highest priority
            9'b1_01xxxxxx: enc = 3'd6;   // d[6] is active, ignore d[5:0]
            9'b1_001xxxxx: enc = 3'd5;   // d[5] is active, ignore d[4:0]
            9'b1_0001xxxx: enc = 3'd4;   // d[4] is active, ignore d[3:0]
            9'b1_00001xxx: enc = 3'd3;   // d[3] is active, ignore d[2:0]
            9'b1_000001xx: enc = 3'd2;   // d[2] is active, ignore d[1:0]
            9'b1_0000001x: enc = 3'd1;   // d[1] is active, ignore d[0]
            default:       enc = 3'd0;   // d[0] has the lowest priority
        endcase
    endfunction
endmodule
```

The $|d$ operation in the expression of g is the reduction OR on 8-bit $d[7:0]$:

$$|d = d[7] + d[6] + d[5] + d[4] + d[3] + d[2] + d[1] + d[0]$$

The don't care `x` can be used with `casex` within the function. The underbar separates e and d for easy reading; it will be ignored by Verilog HDL compiler. Figure 2.13 shows the simulation waveform of the Verilog HDL codes given above for the 8-3 priority encoder.

In the design of interrupt controllers, we often use the priority encoder to indicate whether there is an interrupt and which input causes the interrupt request. For example, referring to Figure 2.12(c), eight interrupt requests $intr[7:0]$ can be connected to the inputs $d[7:0]$ of the 8-3 priority encoder; the outputs $n[2:0]$ can be used as the vector $vec[2:0]$ of the interrupt controller; the output g serves as the interrupt request *int_req* to inform the CPU; and the input *ena* can severe as the interrupt enable.

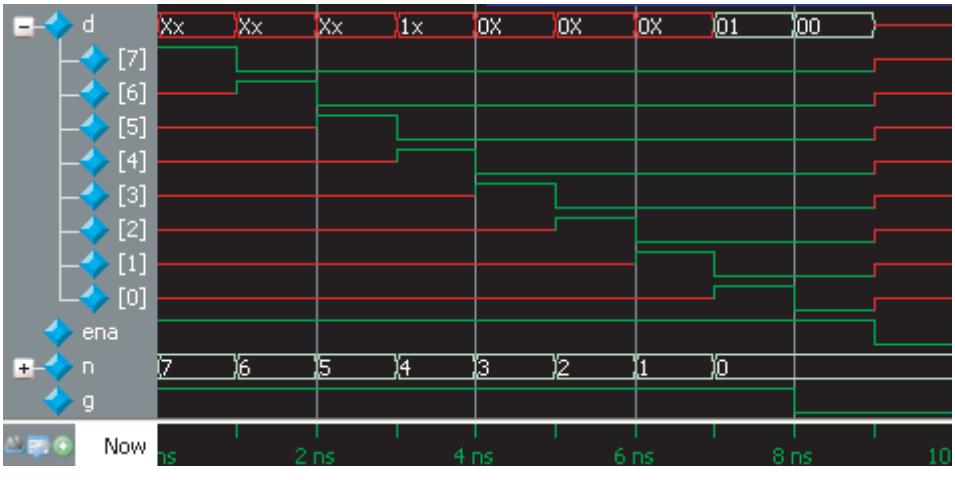


Figure 2.13 Waveform of a 8-3 priority encoder

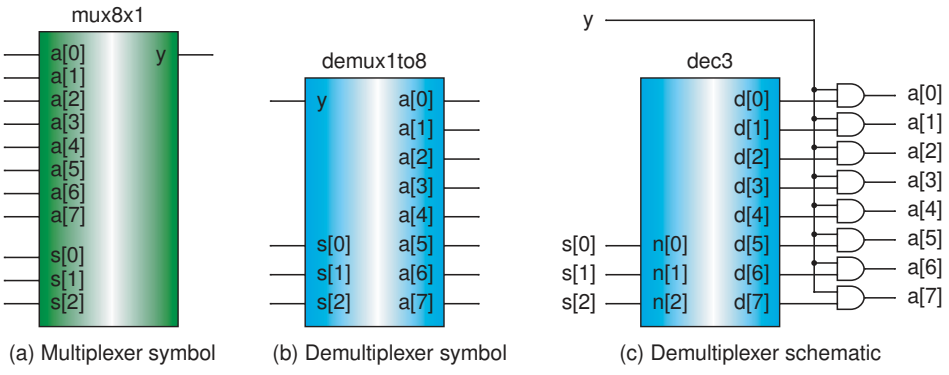


Figure 2.14 A 1-to-8 demultiplexer

2.5.4 *Demultiplexer*

A demultiplexer performs the reverse operation of the multiplexer. It has only a 1-bit data input and an m-bit select signal. The output has 2^m bits. Depending on the select signal, one bit of the output is selected to take the state of the data input.

Figure 2.14(b) shows the symbol of a 1-to-8 demultiplexer. Figure 2.14(a) shows an 8-to-1 multiplexer for comparison with the 1-to-8 demultiplexer. Figure 2.14(c) is the logic schematic diagram of the 1-to-8 demultiplexer. It uses a 3-8 decoder and eight AND gates.

Below is the Verilog HDL code of the 1-to-8 demultiplexer. The shift operation (`1 << s`) implements the decoder. `{8{y}}` replicates `y` to eight bits.

```
module demux1to8 (s,y,a);                  // 1-to-8 demultiplexer
    input  [2:0] s;                        // inputs:  s, dispatch control
    input        y;                        // input:   y, to be dispatched
    output [7:0] a;                        // outputs: only 1 bit's value = y
    assign a = (1 << s) & {8{y}};          // the value of (2^s)th bit = y
endmodule
```

2.5.5 *Barrel Shifter*

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. The following examples show the three kinds of shift by 8 bits on a 32-bit input.

Original data (`d`):	11111111_00000000_00000000_11111111
Shift `d` to the left by 8 bits:	00000000_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

Figure 2.15 shows a traditional implementation of a 32-bit left shifter that uses multiplexers. There are five stages in each of which a 32-bit 2-to-1 multiplexer is used. Each stage performs a shift to the left by a power of 2 (16, 8, 4, 2, 1) bits controlled by a bit of `sa`.

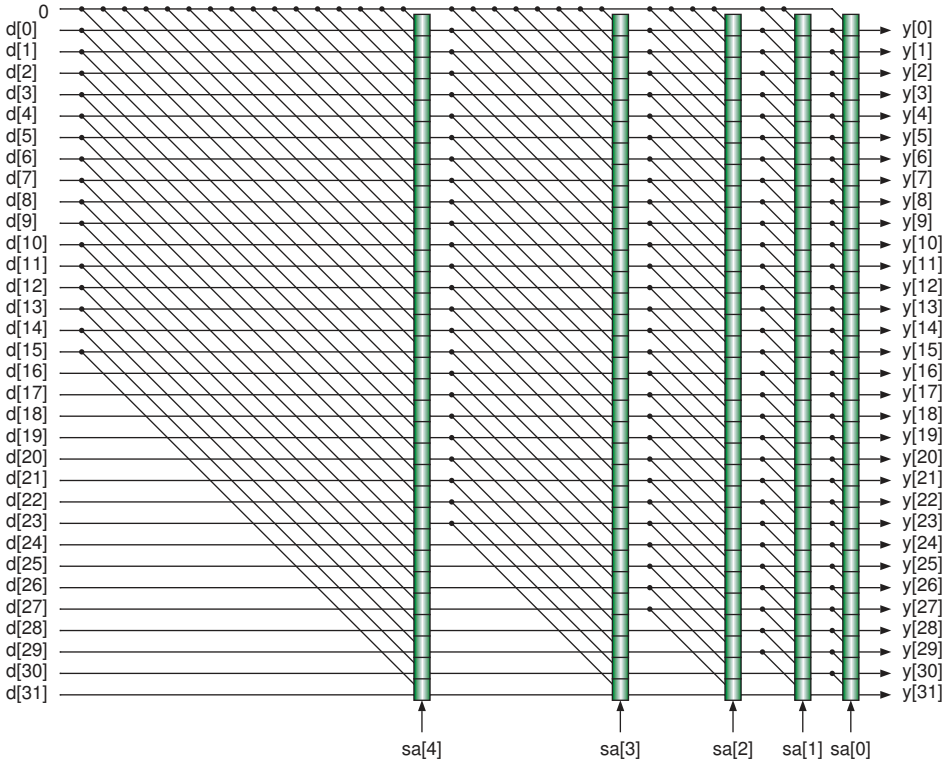


Figure 2.15 Schematic diagram of 32-bit left shifter

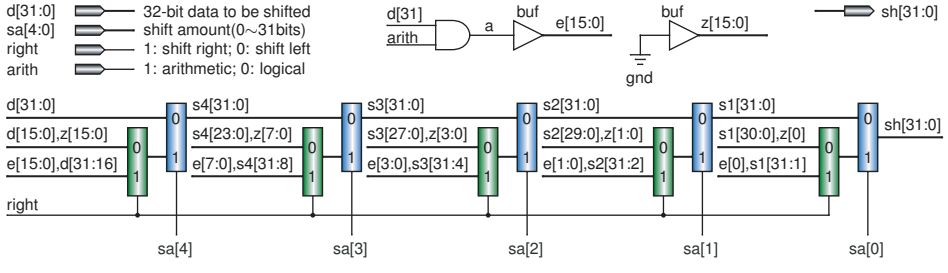


Figure 2.16 Schematic diagram of 32-bit barrel shifter

Figure 2.16 shows an implementation of a 32-bit barrel shifter. Other five multiplexers are used for selecting the result of the left shift or right shift. An AND gate generates the bits `e`, which will be inserted in the emptied bit positions for the right shift. `e` will be zero if `arith` is a 0 (logical shift); otherwise, all the bits of `e` are the same as the sign bit of the input `d` (arithmetic shift). The component named `gnd` outputs a 0, which is used for the left shift. A `buf` component replicates an input bit to multiple output bits, each of which is equal to the input bit.

The Verilog HDL code for the barrel shifter is listed below. It is almost identical to the schematic circuit shown in Figure 2.16.

```
module shift_mux (d,sa,right,arith,sh);          // a barrel shift using muxs
    input  [31:0] d;                             // input: 32-bit data
    input   [4:0] sa;                            // input: shift amount
    input         right;                         // 1: shift right; 0: left
    input         arith;                         // 1: arithmetic; 0: logical
    output [31:0] sh;                            // output: shifted result
    wire   [31:0] t0, t1, t2, t3, t4;            // wires: outputs of muxs
    wire   [31:0] s1, s2, s3, s4;                // wires: outputs of muxs
    wire          a = d[31] & arith;             // a: filling bit
    wire   [15:0] e = {16{a}};                   // replicate a to 16 bits
    parameter     z = 16'b0;                     // a 16 bits zero
    wire   [31:0] sdl4,sdl3,sdl2,sdl1,sdl0;      // left  shifted data
    wire   [31:0] sdr4,sdr3,sdr2,sdr1,sdr0;      // right shifted data
    assign  sdl4  =  {d[15:0],  z};              // shift to left  by 16 bits
    assign  sdl3  =  {s4[23:0], z[7:0]};         // shift to left  by  8 bits
    assign  sdl2  =  {s3[27:0], z[3:0]};         // shift to left  by  4 bits
    assign  sdl1  =  {s2[29:0], z[1:0]};         // shift to left  by  2 bits
    assign  sdl0  =  {s1[30:0], z[0]};           // shift to left  by  1 bit
    assign  sdr4  =  {e,        d[31:16]};       // shift to right by 16 bits
    assign  sdr3  =  {e[7:0],   s4[31:8]};       // shift to right by  8 bits
    assign  sdr2  =  {e[3:0],   s3[31:4]};       // shift to right by  4 bits
    assign  sdr1  =  {e[1:0],   s2[31:2]};       // shift to right by  2 bits
    assign  sdr0  =  {e[0],     s1[31:1]};       // shift to right by  1 bit
    mux2x32 m_right4 (sdl4, sdr4, right, t4);    // select left or right
    mux2x32 m_right3 (sdl3, sdr3, right, t3);    // select left or right
    mux2x32 m_right2 (sdl2, sdr2, right, t2);    // select left or right
    mux2x32 m_right1 (sdl1, sdr1, right, t1);    // select left or right
    mux2x32 m_right0 (sdl0, sdr0, right, t0);    // select left or right
    mux2x32 m_shift4 (d,      t4, sa[4], s4);    // select not_shift or shift
    mux2x32 m_shift3 (s4,     t3, sa[3], s3);    // select not_shift or shift
    mux2x32 m_shift2 (s3,     t2, sa[2], s2);    // select not_shift or shift
    mux2x32 m_shift1 (s2,     t1, sa[1], s1);    // select not_shift or shift
    mux2x32 m_shift0 (s1,     t0, sa[0], sh);    // select not_shift or shift
endmodule
```

Verilog HDL supports the operators of shift left (`<<`), logical shift right (`>>`), and arithmetic shift right (`>>>`). `>>>` must be used together with `$signed()`; see the following behavioral style code that implements the barrel shifter. Note that the condition of the `always` block can be simply `@*`, which is added to the IEEE Verilog HDL standard of the 2001 version.

```
module shift (d,sa,right,arith,sh);      // barrel shift, behavioral style
    input  [31:0] d;                     // input: 32-bit data to be shifted
    input   [4:0] sa;                    // input: shift amount, 5 bits
    input         right;                 // 1: shift right; 0: shift left
    input         arith;                 // 1: arithmetic shift; 0: logical
    output [31:0] sh;                    // output: shifted result
    reg    [31:0] sh;                    // will be combinational
    always @* begin                      // always block
```

```
        if (!right) begin               // if shift left
            sh = d << sa;               //    shift left sa bits
        end else if (!arith) begin      // if shift right logical
            sh = d >> sa;               //    shift right logical sa bits
        end else begin                  // if shift right arithmetic
            sh = $signed(d) >>> sa;     //    shift right arithmetic sa bits
        end
    end
endmodule
```

Through the two implementation versions of the barrel shifter, you may find that the Verilog HDL code in behavioral style is more like a C program, and the Verilog HDL code in structural or dataflow style is like an assembly program.

All the Verilog HDL code examples given above are for designing combinational circuits. There is another type of the logic circuits, sequential circuits, which we will introduce in the next section.

2.6 Sequential Circuit Design

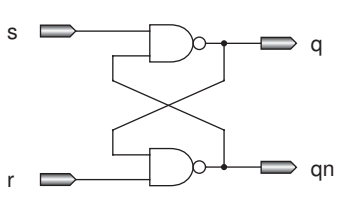
The outputs of sequential circuits are dependent not only on the present inputs but also on the past history of inputs. The components that can store data must be used to record the current state in sequential circuits. The D latch, D flip-flop (DFF), JK flip-flop (JKFF), and T flip-flop (TFF) are examples of such components.

2.6.1 D Latch and D Flip-Flop

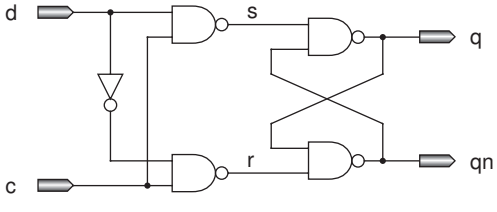
A D latch can store one bit information. It can be implemented with an RS (reset-set) latch. Figure 2.17(a) shows the schematic circuit of an RS latch. When `s` (set, active-low) is a 0 (active) and `r` (reset, active-low) is a 1 (inactive), the output `q` is 1 (set); when `s` is a 1 (inactive) and `r` is a 0 (active), `q` is 0 (reset); when `s` and `r` are both 1 (inactive), `q` does not change (hold).

The schematic circuit of a D latch is shown in Figure 2.17(b). Three gates (two NAND gates and a NOT gate) are added to an RS latch. These additional gates generate the `r` and `s` signals. When the input `c` (control) is a 0, `s` and `r` are both 1, so the state `q` does not change; when `c` is a 1, the output `q` will be equal to the input `d`. Thus we say that a D latch is level-triggered. The Verilog HDL code of the D latch is shown below.

```
module d_latch (c,d,q,qn);                  // d latch
    input  c, d;                            // inputs:  c, d
    output q, qn;                           // outputs: q, qn
```



(a) RS latch



(b) D latch

Figure 2.17 Schematic diagram of a D latch

```
    wire   r, s;                                    // internal wires
    nand nand1 (s,  d, c);                          // nand (out, in1, in2);
    nand nand2 (r, ~d, c);                          // nand (out, in1, in2);
    rs_latch rs (s, r, q, qn);                      // use rs_latch module
endmodule
```

The following Verilog HDL code implements the RS latch that is used in the Verilog HDL code of the D latch.

```
module rs_latch (s,r,q,qn);                         // rs latch
    input  s, r;                                    // inputs:  set, reset
    output q, qn;                                   // outputs: q, qn
    nand nand1 (q,  s, qn);                         // nand (out, in1, in2);
    nand nand2 (qn, r, q);                          // nand (out, in1, in2);
endmodule
```

Figure 2.18 shows the simulation waveform of the D latch from which we can see that, when the input `c` is a 1, the output `q` follows the input `d`, and `q` does not change when the input `c` is a 0. In the beginning (0–5 ns), the state of the D latch is unknown.

Different from a D latch, a DFF is edge-triggered. Figure 2.19 shows an academic version of the DFF circuit. It consists of two D latches and two NOT gates. 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."

Below is the Verilog HDL code of the DFF. This module invokes the module of D latch which was already given above.

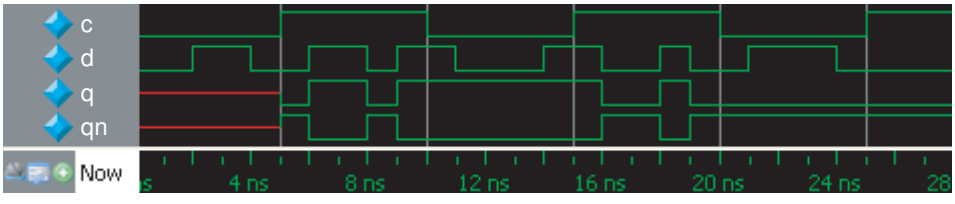


Figure 2.18 Waveform of a D latch

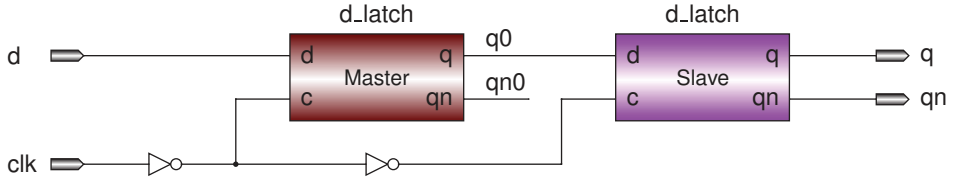


Figure 2.19 Schematic diagram of an academic D flip-flop

```
module d_flip_flop (clk,d,q,qn);                    // dff using 2 d latches
    input  clk, d;                                  // inputs:  clk, d
    output q,  qn;                                  // outputs: q, qn
    wire   q0, qn0;                                 // internal wires
    wire   clkn, clknn;                             // internal wires
    not inv1 (clkn,  clk);                          // inverse of clk
    not inv2 (clknn, clkn);                         // inverse of clkn
    d_latch dlatch1 (clkn,  d,  q0, qn0);           // master d latch
    d_latch dlatch2 (clknn, q0, q,  qn);            // slave  d latch
endmodule
```

Figure 2.20 shows the simulation waveform of the DFF. We can see that the input `d` is stored to the DFF on the positive edge of `clk`.

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.

The following Verilog HDL code was generated from the schematic circuit of Figure 2.21 by Quartus II Web Edition. Some wire names were simplified.

```
module d_ff (prn, clk, d, clrn, q, qn);      // dff generated from schematic
   input  prn, clk, d, clrn;        // prn: preset; clrn: clear (active low)
   output q, qn;                    // outputs q, qn
   wire   wire_0, wire_1;           // internal wire, see figure ch02_fig21
   wire   wire_2, wire_3;           // internal wire, see figure ch02_fig21
   assign wire_0 = ~(wire_1 & prn    & wire_2);              // 3-input nand
   assign wire_1 = ~(clk    & clrn   & wire_0);              // 3-input nand
   assign wire_2 = ~(wire_3 & clrn   & d);                   // 3-input nand
   assign wire_3 = ~(wire_1 & clk    & wire_2);              // 3-input nand
   assign q      = ~(prn    & wire_1 & qn);                  // 3-input nand
   assign qn     = ~(q      & wire_3 & clrn);                // 3-input nand
endmodule
```

The simulation waveform of the industry version of the DFF is shown in Figure 2.22. The state of the DFF is 0 in the beginning because `clrn` is active. Then the state becomes 1 when `prn` is active. The rest shows that the value of `d` is stored into DFF in the positive edge of `clk`.

The DFF stores data on every positive edge of the clock if both the clear and preset inputs are inactive. Figure 2.23(a) shows a DFFE by adding an enable control signal `e` to the DFF: 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.

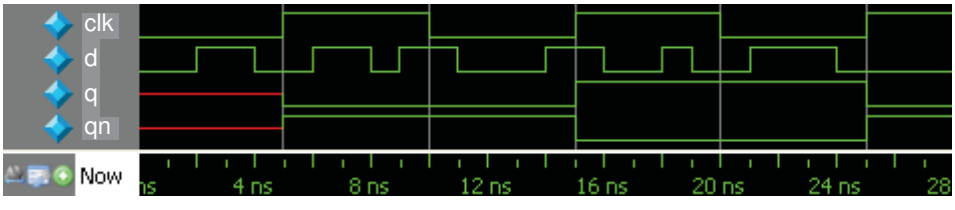


Figure 2.20 Waveform of an academic D flip-flop

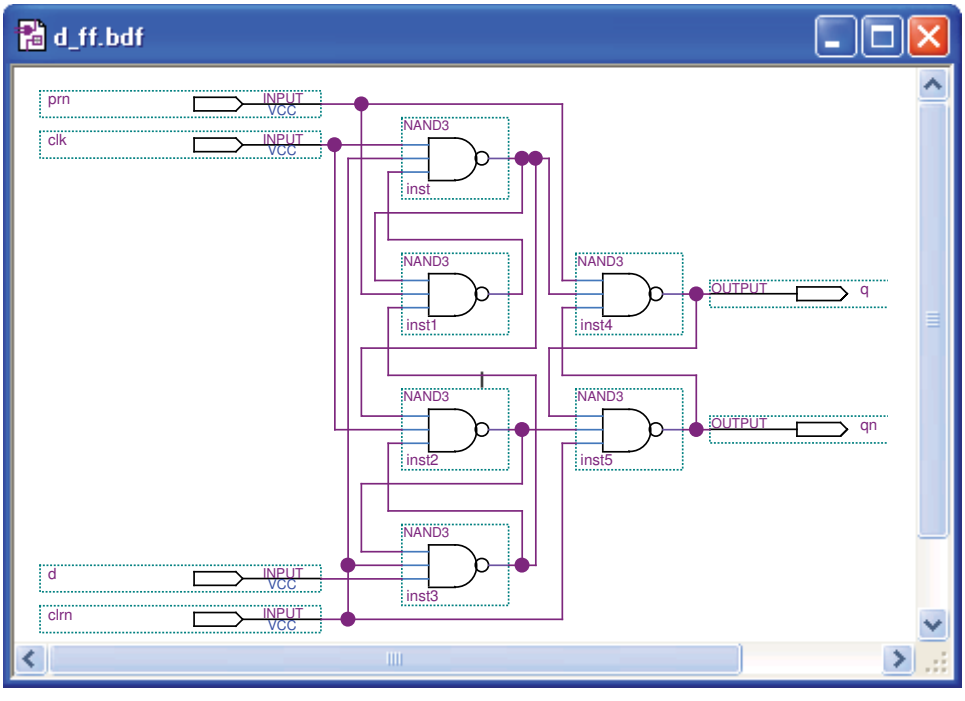


Figure 2.21 Schematic diagram of an industry D flip-flop

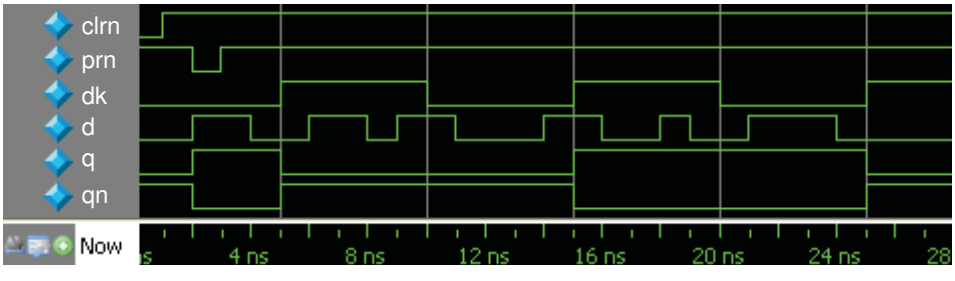
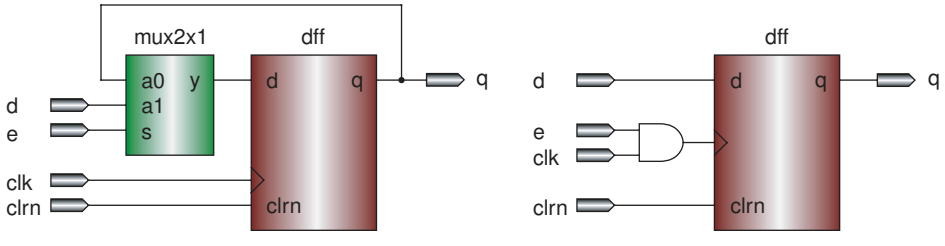


Figure 2.22 Waveform of an industry D flip-flop



(a) Schematic of DFFE (b) Bad schematic of DFFE

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

A DFF with a synchronous clear can be implemented by the following behavioral-style Verilog HDL code. `posedge` is a keyword that means a positive edge. The active-low clear input `clrn` is checked on the positive edge of `clk`: if `clrn` is active, the DFF will be reset to 0; otherwise the input `d` is stored.

```
module d_ff_sync (d,clk,clrn,q);      // dff with synchronous reset (to clk)
    input      d, clk, clrn;          // inputs d, clk, clrn (active low)
    output reg q;                     // output q, register type
    always @ (posedge clk) begin      // always block, posedge: rising edge
        if (!clrn) q <= 0;            // if clrn is asserted, reset dff
        else       q <= d;            // else store d to dff
    end
endmodule
```

The following code implements a DFF with an asynchronous clear. `negedge` is a keyword that means negative edge (falling edge). `clrn` is checked independently of `clk`. In fact, `clrn` is a level-triggered clear, not an edge-triggered clear, because if `clrn` is held active, the output `q` will continue to be held low.

```
module dff (d,clk,clrn,q);            // dff with asynchronous reset
    input      d, clk, clrn;          // inputs d, clk, clrn (active low)
    output reg q;                     // output q, register type
    always @ (posedge clk or negedge clrn) begin // always block, "or"
        if (!clrn) q <= 0;            // if clrn is asserted, reset dff
        else       q <= d;            // else store d to dff
    end
endmodule
```

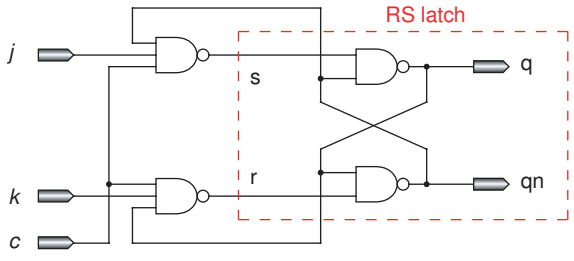
A DFFE that has an enable control signal can be implemented by the following Verilog HDL code: an `if` statement controls the update of the DFFE.

```
module dffe (d,clk,clrn,e,q);         // dff (async) with write enable
    input      d, clk, clrn;          // inputs d, clk, clrn (active low)
    input      e;                     // enable
    output reg q;                     // output q, register type
    always @ (posedge clk or negedge clrn) begin // always block, "or"
        if (!clrn)  q <= 0;           // if clrn is asserted, reset dff
        else if (e) q <= d;           // else if enabled store d to dff
    end
endmodule
```

2.6.2 JK Latch and JK Flip-Flop

The DFF is used most commonly in digital circuit designs. There are two other flip-flops that are also used commonly: JKFF and TFF. 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 $q_n = c\,(j\,\overline{q} + \overline{k}\,q) + \overline{c}\,q$, where q_n is the next state of q.

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



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

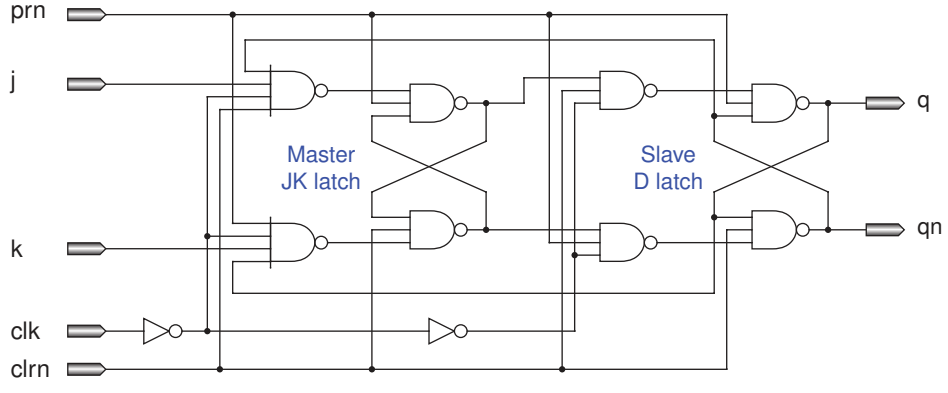


Figure 2.25 Schematic diagram of the JK flip-flop

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.

The following Verilog HDL code uses $q_n = j\overline{q} + \overline{k}q$ to implement the JKFF.

```
module jkff (j,k,clk,clrn,q);                  // jkff with asynchronous reset
    input      j, k, clk, clrn;                // inputs j, k, clock, reset
    output reg q;                              // output q, register type
    always @ (posedge clk or negedge clrn) begin // always block, "or"
        if (!clrn) q <= 0;                     // if clrn is active, reset jkff
        else       q <= j & ~q | ~k & q;       // else update jkff
    end
endmodule
```

Figure 2.26 shows the simulation waveform of the JKFF code.

2.6.3 T Latch and T Flip-Flop

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 $q_n = c(t\overline{q} + \overline{t}q) + \overline{c}\,q$, where q_n is the next state of q. Similar to the JK latch, the T latch is rarely used.

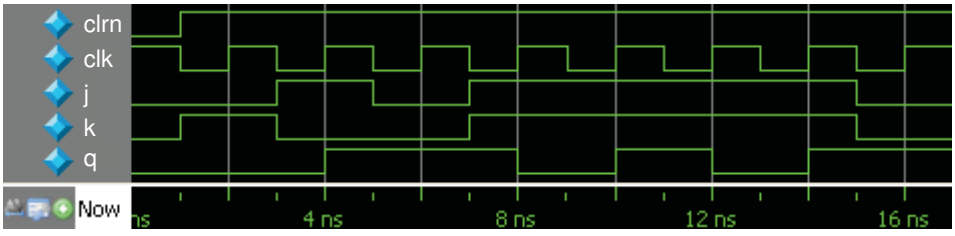
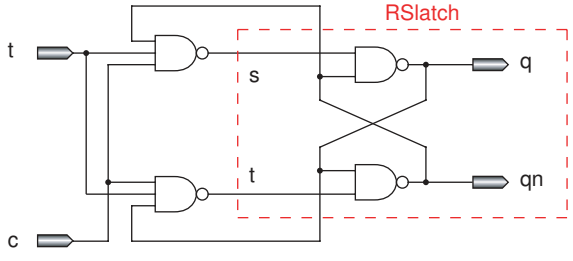


Figure 2.26 Waveform of the JK flip-flop



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

Figure 2.27 Schematic diagram of the T latch

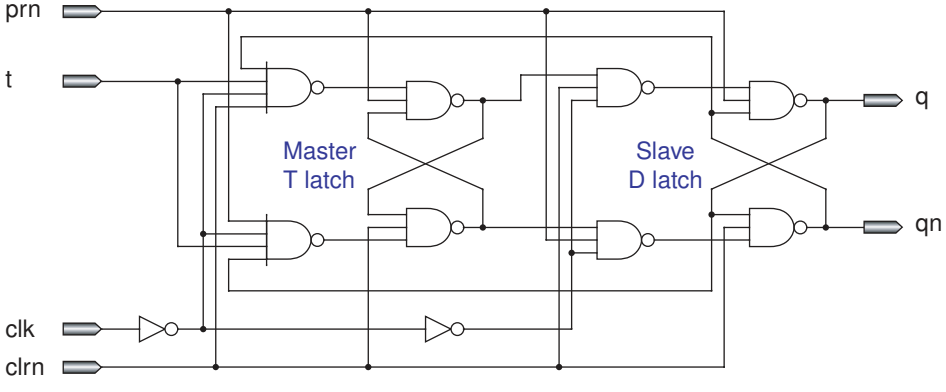


Figure 2.28 Schematic diagram of the T flip-flop

Figure 2.28 shows the circuit of a TFF. 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.

The following Verilog HDL code uses $q_n = t\,\overline{q} + \overline{t}\,q$ to implement the TFF.

```
module tff (t,clk,clrn,q);                    // tff with asynchronous reset
    input      t, clk, clrn;                  // inputs t, clock, reset
    output reg q;                             // output q, register type
```

```
    always @ (posedge clk or negedge clrn) begin // always block, "or"
        if (!clrn) q <= 0;                  // if clrn is active, reset tff
        else       q <= t & ~q | ~t & q;    // else update tff
    end
endmodule
```

Figure 2.29 shows the simulation waveform of the TFF code.

2.6.4 *Shift Register*

Using multiple DFFs, 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`. 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.

2.6.5 *FIFO Buffer*

FIFO (first-in first-out) is a special buffer for queuing data. Figure 2.31 shows a schematic diagram of a FIFO of depth 4. `R1`, `R2`, `R3`, and `R4` 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`, and `F4`. The *i*th latch indicates whether the *i*th register contains data or not. 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. Such a latch can be designed with NOR gates. The RS latches are cleared initially,

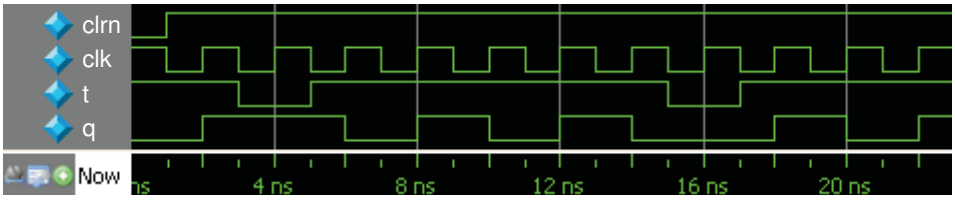


Figure 2.29 Waveform of the T flip-flop

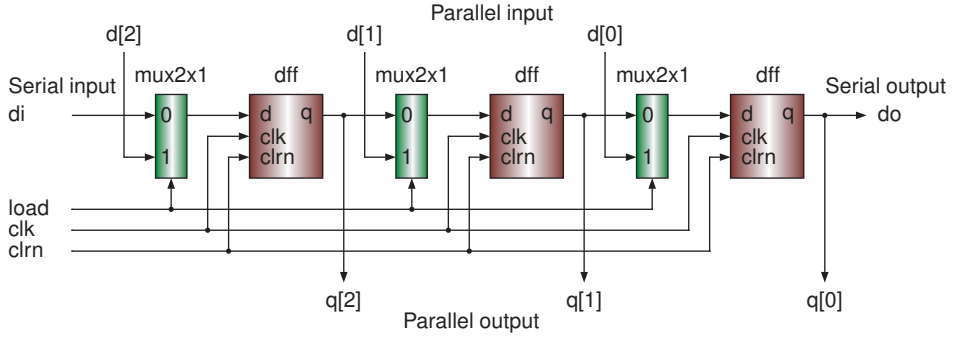


Figure 2.30 Schematic diagram of a shift register

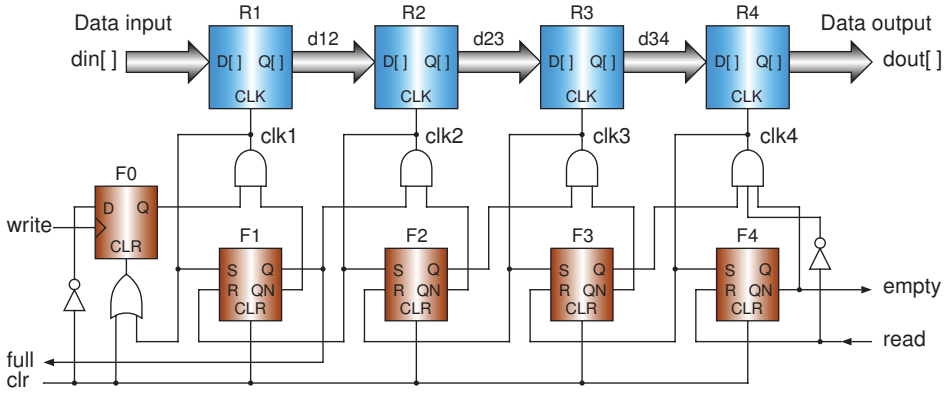


Figure 2.31 Schematic diagram of a FIFO of depth 4

therefore the outputs of QNs are high. These latches and the AND gates are used to generate clock pulse signals for the edge-triggered registers R1, R2, R3, and R4. If Q of the $(i-1)$th latch becomes 1 and Q of the ith latch is 0 (QN is 1), then clk$i$ becomes 1 (a rising edge) that sets the ith latch and resets the $(i-1)$th latch. clk$i$ will go back to 0 once clk$i+1$ becomes 1. Thus, a pulse of clk$i$ is generated that is used to store the data of the $(i-1)$th register into the ith register.

The DFF F0 is used to generate a write pulse for clk1. The NOT gate in the right side prevents the data in the R4 register from being overwritten when the read signal is asserted. A 1 of the empty signal indicates that the FIFO is empty. A 1 of the full signal indicates that the FIFO is full.

Figure 2.32 shows the simulation waveforms of the FIFO circuit. We set the delay time of the AND gates to 1 ns in the simulation. We can see that the data of 0xe1 (0x indicates the data are represented in hexadecimal format) are passed from R1 to R4, and at 86 ns the FIFO is full. The registers R1, R2, R3, and R4 hold the data of 0xe1, 0xe2, 0xe3, and 0xe4, respectively. These data are read out sequentially from the output port, resulting in the FIFO becoming empty.

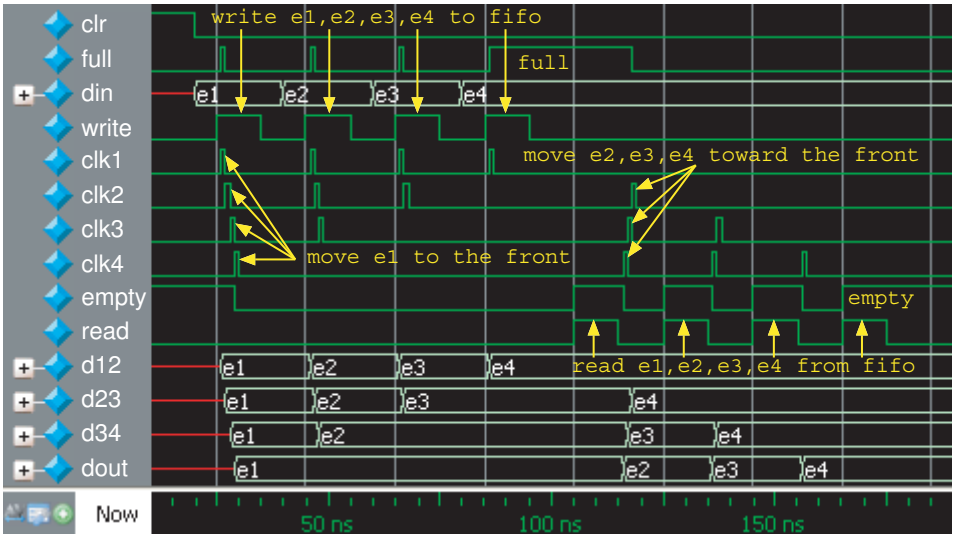


Figure 2.32 Waveforms of FIFO4

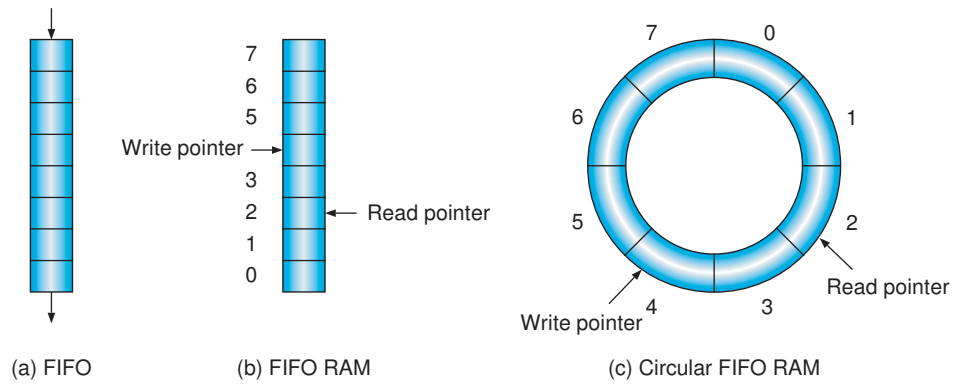


Figure 2.33 A circular FIFO implemented with RAM

There is no address in a FIFO buffer, meaning that a register in the FIFO cannot be accessed randomly. FIFOs are used in many places, such as cache block replacement circuits, the keyboard scan code buffer, and network communication buffers. Figure 2.33 shows another FIFO implementation in which a traditional RAM (random access memory) is used for queuing the data.

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 2^n and the next location to the location $2^n - 1$ is 0.

The following Verilog HDL code implements such a RAM-based FIFO. A 1 of the `ready` signal indicates that the FIFO is not empty. `overflow` means that the FIFO is full and the data written last are lost.

```
module fifo (clk,clrn,read,write,data_in,data_out,ready,overflow);   // fifo
    input        clk, clrn;                          // clock and reset
    input        read;                               // fifo read, active high
    input        write;                              // fifo write, active high
    input  [7:0] data_in;                            // fifo data input
    output [7:0] data_out;                           // fifo data output
    output       ready;                              // fifo has data
    output reg   overflow;                           // fifo overflow flag
    reg    [7:0] fifo_buff [7:0];                    // fifo buffer of depth 8
    reg    [2:0] write_pointer;                      // fifo write pointer
    reg    [2:0] read_pointer;                       // fifo read pointer
    always @ (posedge clk or negedge clrn) begin
        if (!clrn) begin
            write_pointer <= 0;                      // clear write pointer
            read_pointer  <= 0;                      // clear read pointer
            overflow      <= 0;                      // clear overflow flag
        end else begin
            if (write) begin
                if ((write_pointer + 3'b1) != read_pointer) begin
                    fifo_buff[write_pointer] <= data_in;   // push data
                    write_pointer <= write_pointer + 3'd1; // pointer++
```

```
                end else begin
                    overflow <= 1;                          // overflow
                end
            end
            if (read && ready) begin
                read_pointer <= read_pointer + 3'd1;        // pointer++, pop
                overflow <= 0;                              // clear overflow
            end
        end
    end
    assign ready = (write_pointer != read_pointer);         // has data
    assign data_out = fifo_buff[read_pointer];              // data output
endmodule
```

Figure 2.34 shows the simulation waveforms of the RAM-based FIFO. The data of 0xe1, 0xe2, … , and 0xf3 are written to the FIFO, but the data read from the FIFO are 0xe1, 0xe2, … , 0xe9, 0xf1, 0xf2, and 0xf3. 0xea, 0xeb, … , 0xf0 are lost as a result of the FIFO overflow. Although there are eight locations in the FIFO, the code given above results in that the FIFO can hold only seven data at the same time.

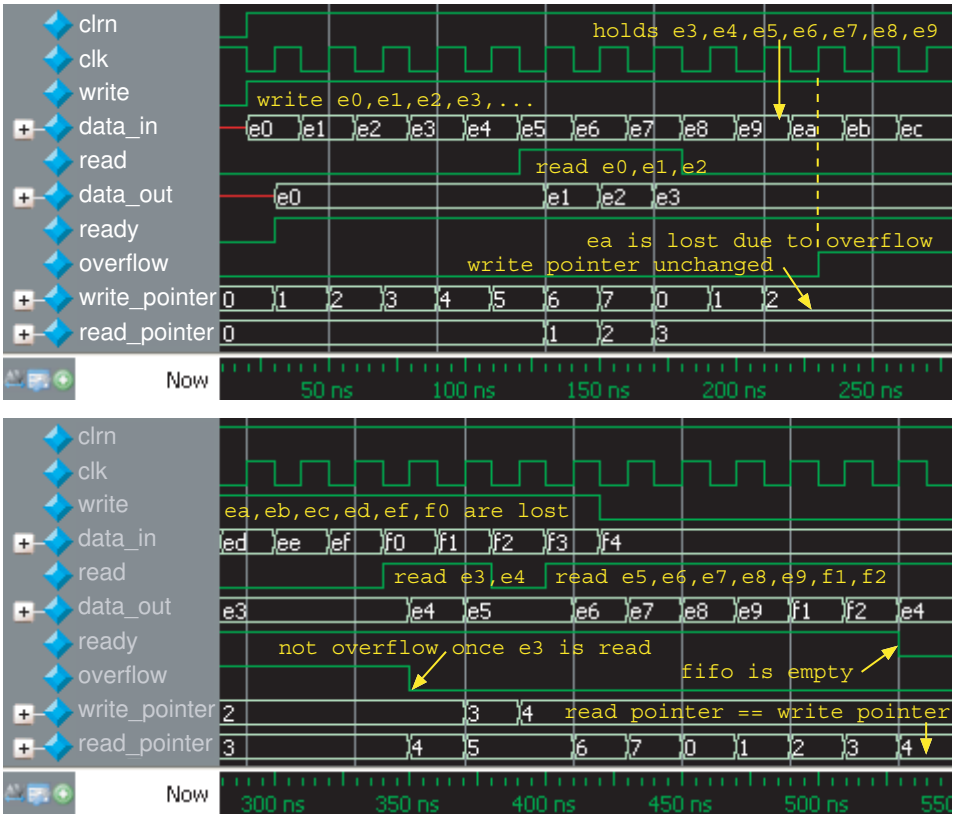


Figure 2.34 Waveforms of RAM-based FIFO

2.6.6 *Finite State Machine and Counter Design*

The flip-flops we have introduced can be used to design the circuit of finite state machines. A finite state machine is a way of modeling a system in which the system's outputs will depend on not only the current inputs but also the past history of inputs. It defines a finite set of states and behaviors, and how the system transits from one state to another when certain conditions are true.

A finite state machine can be implemented in software but here we focus on the hardware circuit design of the finite state machine. There are two models of the finite state machine: the Mealy model and the Moore model, as shown as in Figure 2.35.

There are a finite number of states that can be implemented with DFFs. Suppose that the number of states is N; then $n = \lceil \log_2 N \rceil$ flip-flops are needed. The module of next state is a combinational circuit that determines the next state based on the current state and inputs. The next state will be written into DFFs on the clock edge. The module of output function is also a combinational circuit that generates the outputs based on the current state (Moore model) or the combination of the current state and the current inputs (Mealy model). The general steps of designing a sequential circuit are described below.

1. Make a state transition diagram based on the problem statement.
2. Determine the number of flip-flops and assign binary codes to the states.
3. Fill a truth table for the next state signals.
4. Write the logic expression for each of the next state signals. Karnaugh maps may be used to get simplified expressions.
5. Fill a truth table for the output signals.
6. Write logic expressions of the output signals. Karnaugh maps may be used to get simplified expressions.
7. Build and simulate the circuit.

Let's take an example to show how to design a sequential circuit. Suppose we design a radix of six up/down counters with a seven-segment LED (light-emitting diode), as shown as in Figure 2.36. The state of the counter changes on the positive edge of the clock. If the input `u` is a 1, the counter value

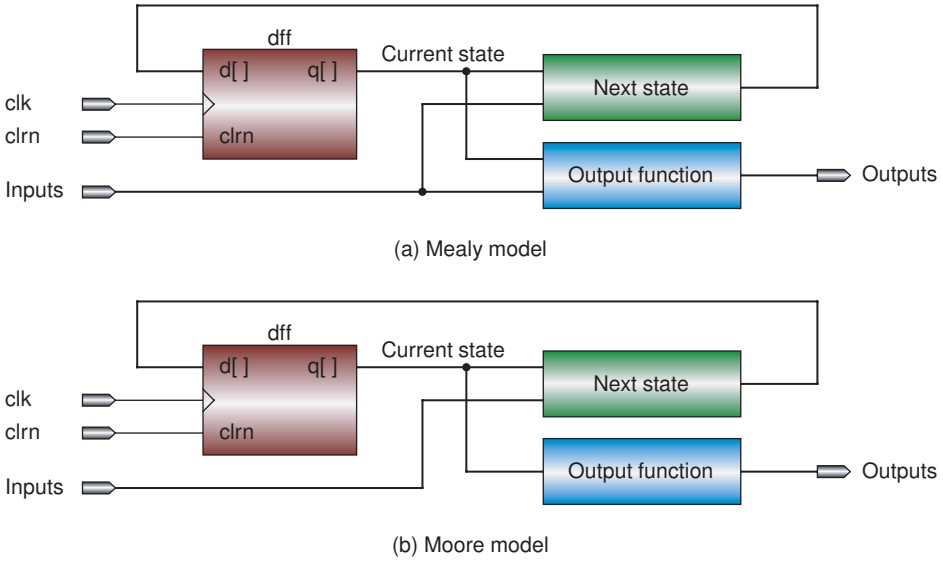


Figure 2.35 Two models of the general finite state machine

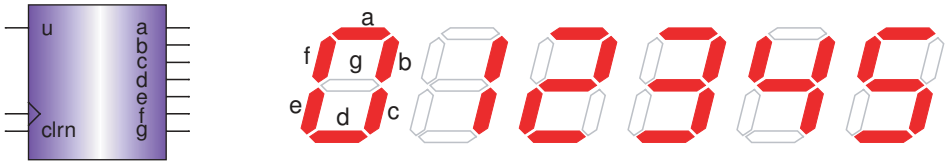


Figure 2.36 A counter with a seven-segment LED

will change in the sequence 0, 1, 2, 3, 4, 5, 0, 1, 2, ... If `u` is a 0, the counter value will change in the sequence 0, 5, 4, 3, 2, 1, 0, 5, 4, ... There are seven output signals, with each connecting to a segment of LED. A segment of the LED will be on if its control signal is a 0 (active-low).

Obviously, there are six states, thus three DFFs are needed. The circuit of the counter is shown in Figure 2.37. The module of `dff3` contains three DFFs. The other module is a combinational circuit that generates signals of the next state (`ns[2:0]`) and LED control signals (`a`, `b`, `c`, `d`, `e`, `f`, and `g`). The current state (the outputs of `dff3`) is denoted with `q[2:0]`.

Figure 2.38 shows the state transition diagram. The arrowed lines indicate the transitions of the states under the condition of the input. Figure 2.38 also shows that a 3-bit unique code is assigned to a state. Any code can be assigned to any state as long as all the codes are unique.

Table 2.6 is the truth table for the next state. Figure 2.39 shows the Karnaugh maps for each of the next state signals. From the Karnaugh maps, we can get the following expressions of the next state signals.

$$ns[0] = \overline{q[0]};$$

$$ns[1] = \overline{q[2]}\,\overline{q[1]}\,q[0]u + q[1]\,\overline{q[0]}u + q[1]\,q[0]\,\overline{u} + q[2]\,\overline{q[0]}\,\overline{u};$$

$$ns[2] = \overline{q[2]}\,\overline{q[1]}\,\overline{q[0]}\,\overline{u} + q[1]\,q[0]\,u + q[2]\,\overline{q[0]}\,u + q[2]\,q[0]\,\overline{u}.$$

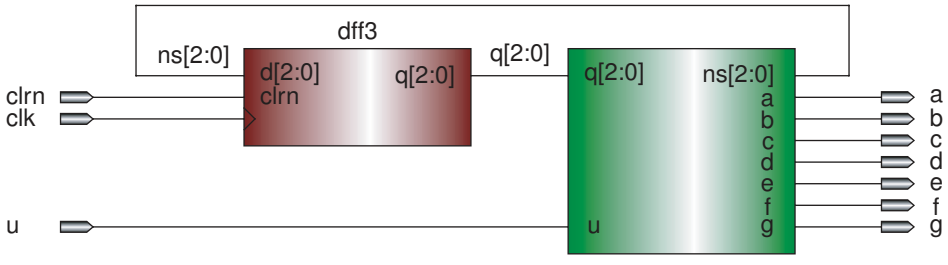


Figure 2.37 Block diagram of a counter with a seven-segment LED

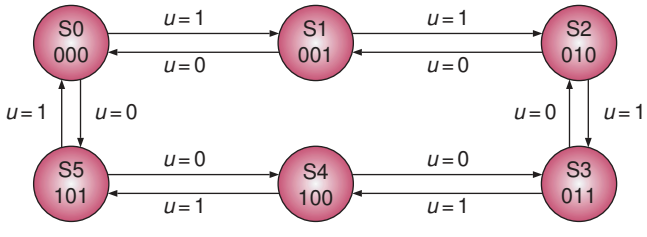


Figure 2.38 State transition diagram of the counter

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

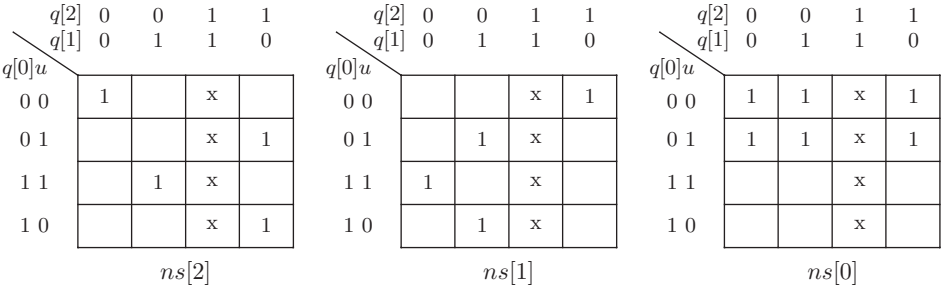


Figure 2.39 Karnaugh map for next state of the counter

Figure 2.40 shows the truth table and Karnaugh maps of the output signals. We get the following expressions of the output signals:

$$a = \overline{q[2]}\ \overline{q[1]}\ q[0] + q[2]\ \overline{q[0]};$$
$$b = q[2]\ q[0];$$
$$c = q[1]\ \overline{q[0]};$$
$$d = \overline{q[2]}\ \overline{q[1]}\ q[0] + q[2]\ \overline{q[0]} = a;$$
$$e = q[2]\ \overline{q[0]} + q[0];$$
$$f = q[1]\ \overline{q[0]} + \overline{q[2]}\ q[0];$$
$$g = \overline{q[2]}\ \overline{q[1]}.$$

Now we can build the circuits of the up/down counter. Figure 2.41 shows the schematic diagram of the 3-bit DFFs. Figure 2.42 shows the schematic diagram of the next state for the counter. Figure 2.43 shows the schematic diagram of the output function for the counter. And Figure 2.44 shows the top schematic

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

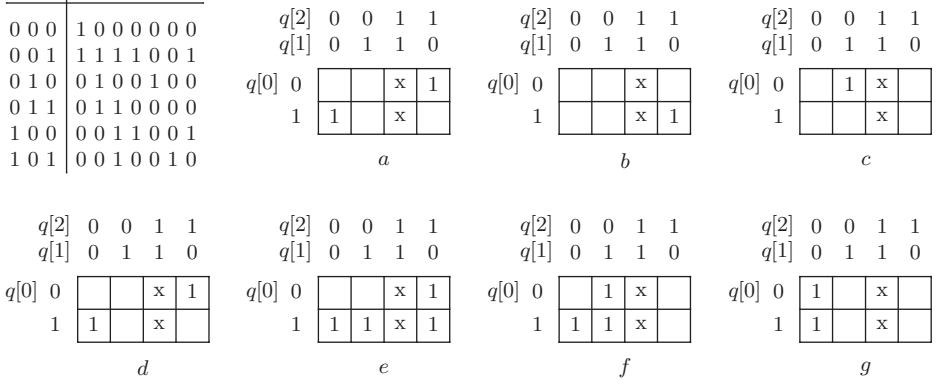


Figure 2.40 Karnaugh map for the output function of the counter

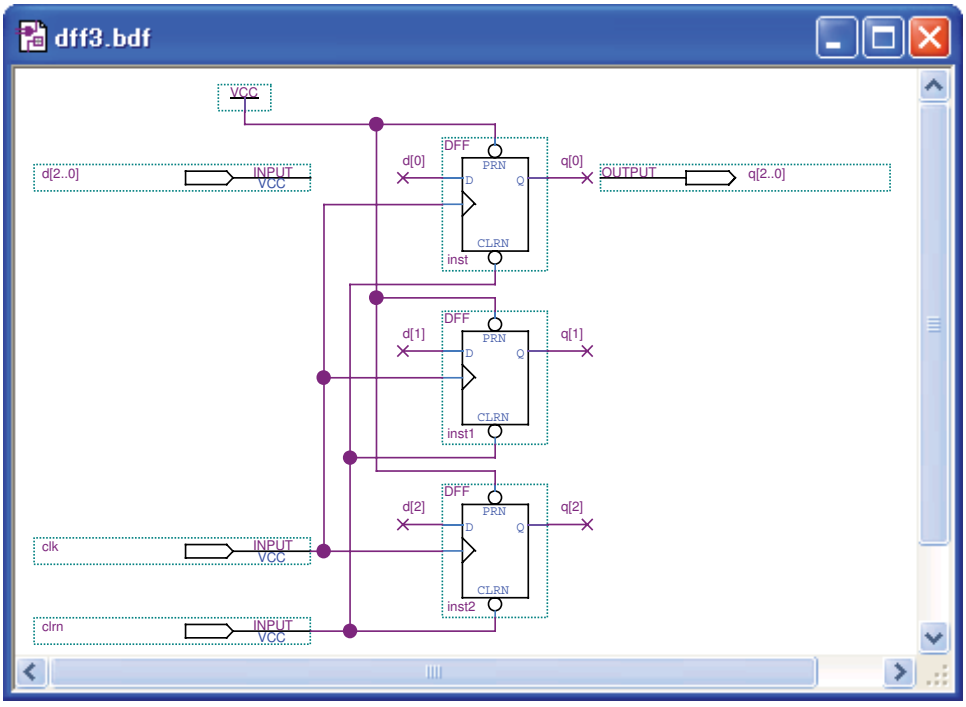


Figure 2.41 Schematic diagram of 3-bit D flip-flops

diagram of the counter with a seven-segment LED. It consists of three parts: (i) `dff3`, the 3-bit DFFs, (ii) `next_state`, the next state, and (iii) `output_function`, the output function.

Figure 2.45 shows the simulation waveform of the up/down counter. The first half shows the counting up, and the second half shows the counting down. The counter changes state on the clock rising edge. The LED control signals are also shown in the figure.

The purpose of describing the details of the counter design is not only for designing a counter but also for understanding the general procedure of sequential circuit designs. If we only want to design a radix of a six up/down counter with the seven-segment LED, we can implement it with the following behavioral-style Verilog HDL code where `%` is the operator of the modulo.

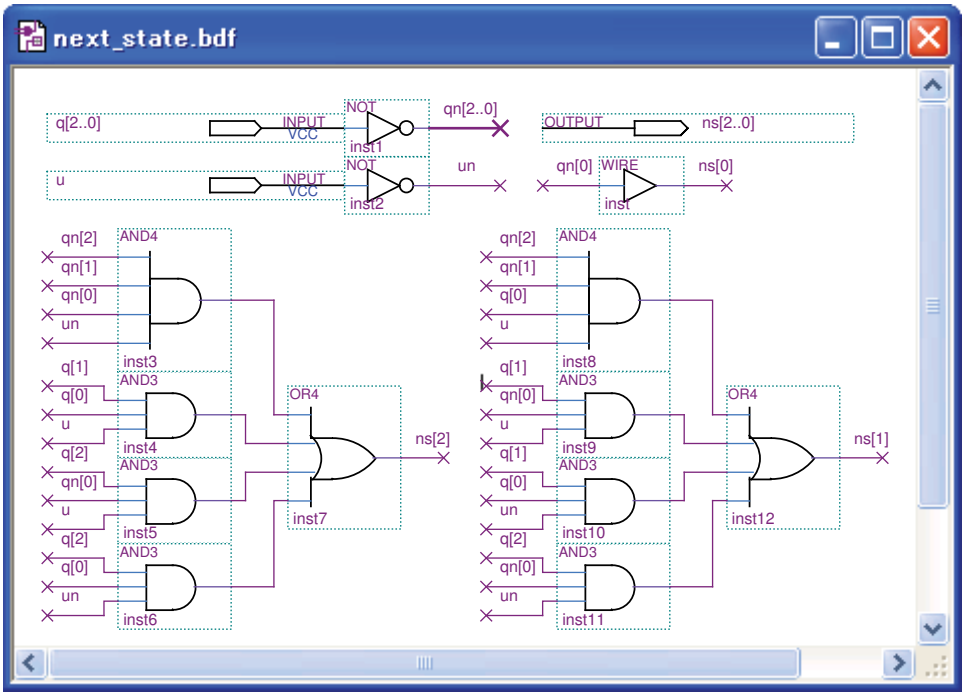


Figure 2.42 Schematic diagram of next state for the counter

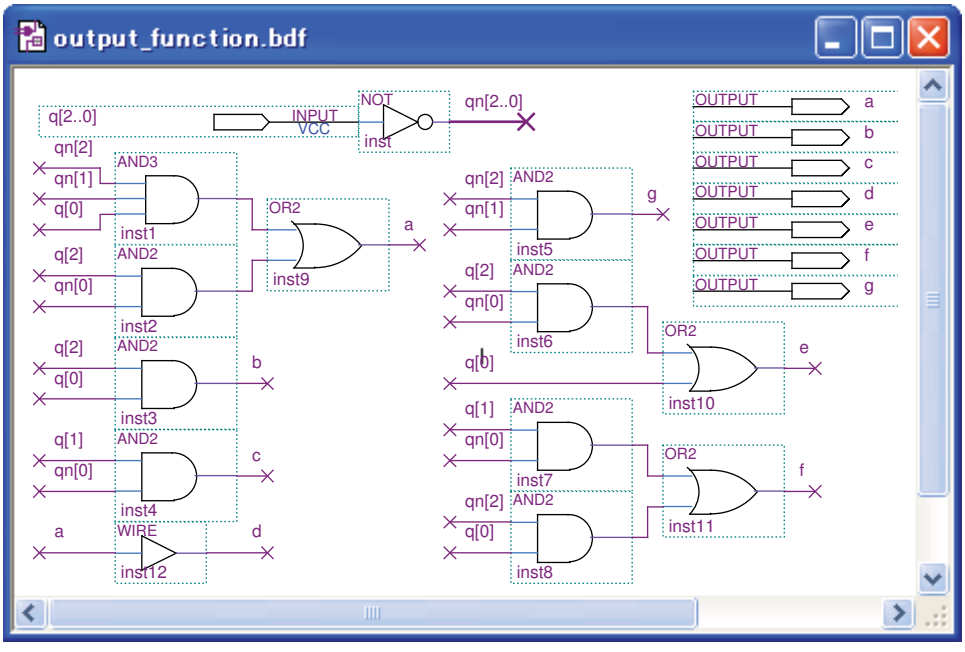


Figure 2.43 Schematic diagram of output function for the counter

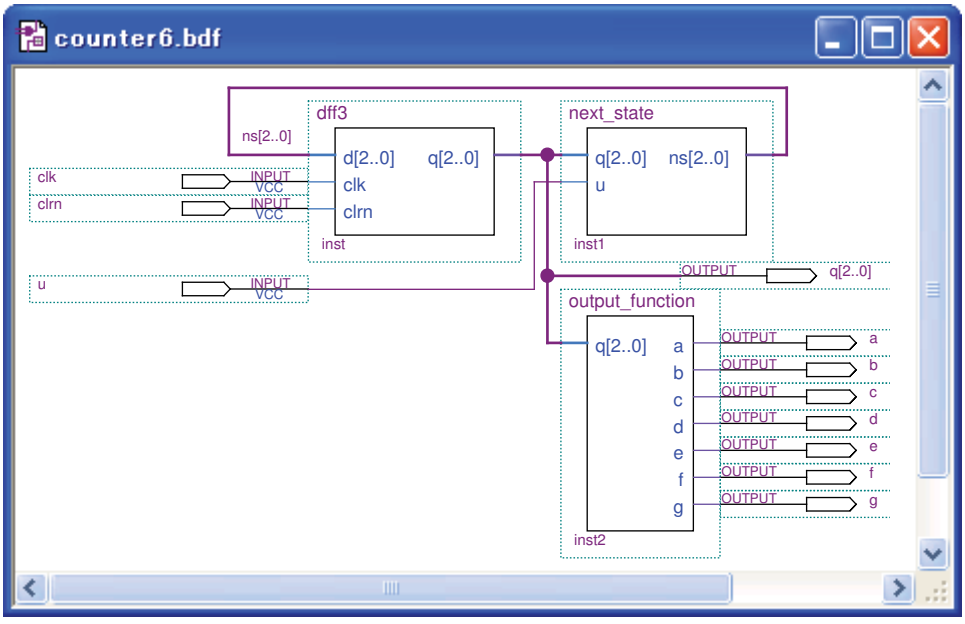


Figure 2.44 Schematic diagram of the counter with a seven-segment LED

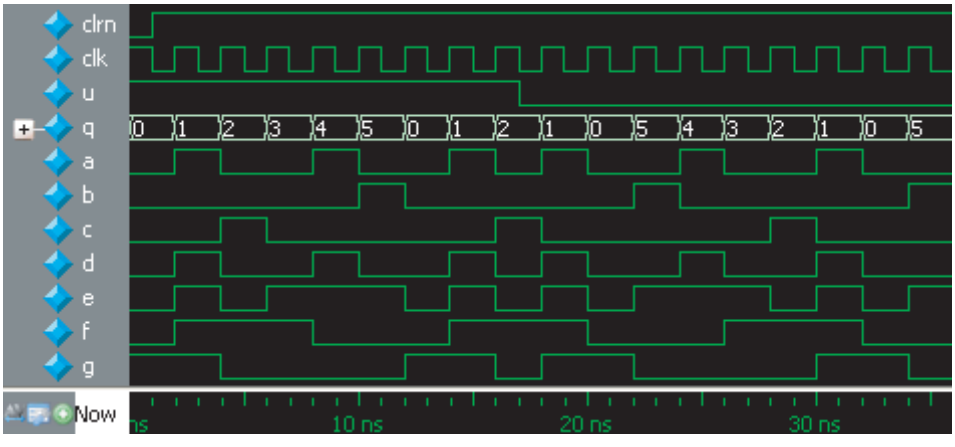


Figure 2.45 Waveform of the up/down counter

```
module counter_6 (u,clk,clrn,q,a,b,c,d,e,f,g);   // a counter with 7-seg LED
    input        clk, clrn;              // clk, clear (active low)
    input        u;                      // u==1: count up; u==0: count down
    output [2:0] q;                      // 3-bit counter output
    output       a, b, c, d, e, f, g;    // seven-segment LED control
    reg    [2:0] q;                      // register type
    always @ (posedge clk or negedge clrn) begin
```

```
        if (!clrn) q <= 0;               // if clrn is asserted, counter=0
        else if (u) q <= (q + 1) % 6;    // if counter up, q++
        else if (q != 0) q <= q - 1;     // else           q-
             else        q <= 3'd5;
    end
    assign {g,f,e,d,c,b,a} = seg7(q);    // call function to get LED control
    function  [6:0] seg7;                // the function, 7-bit return value
        input [2:0] q;                   // input argument
        case (q)                         // cases:
            3'd0 : seg7 = 7'b1000000;    // 0's LED control, 0: light on
            3'd1 : seg7 = 7'b1111001;    // 1's LED control, 1: light off
            3'd2 : seg7 = 7'b0100100;    // 2's LED control
            3'd3 : seg7 = 7'b0110000;    // 3's LED control
            3'd4 : seg7 = 7'b0011001;    // 4's LED control
            3'd5 : seg7 = 7'b0010010;    // 5's LED control
            default: seg7 = 7'b1111111;  // default: all segments light off
        endcase
    endfunction
endmodule
```

We will give another example of the sequential circuit in Chapter 7, where we describe how to design a multiple-cycle CPU.

Exercises

2.1 Try to execute the following Java program (`logic.java`) on your machine and explain the output results.

```
class logic {
    public static void main(String[] args) {
        int a = 0x5;
        int b = 0xc;
        int f_and  =   a & b;
        int f_or   =   a | b;
        int f_not  =  ~a;
        int f_nand = ~(a & b);
        int f_nor  = ~(a | b);
        int f_xor  =   a ^ b;
        int f_xnor = ~(a ^ b);
        System.out.format("a      = 0x%08x\n", a);
        System.out.format("b      = 0x%08x\n", b);
        System.out.format("f_and  = 0x%08x\n", f_and);
        System.out.format("f_or   = 0x%08x\n", f_or);
        System.out.format("f_not  = 0x%08x\n", f_not);
        System.out.format("f_nand = 0x%08x\n", f_nand);
        System.out.format("f_nor  = 0x%08x\n", f_nor);
```

```
        System.out.format("f_xor  = 0x%08x\n", f_xor);
        System.out.format("f_xnor = 0x%08x\n", f_xnor);
    }
}
```

Recommendation: use a command line mode to compile and execute the program:

```
$ javac logic.java
$ java logic
```

2.2 Suppose that we have a code fragment in C or Java as shown below.

```
if (((a == b) || (c >= d)) && (e != f)) {
    // action x :(
} else {
    // action y :)
}
```

Rewrite the `if` condition so that it becomes the following format (exchanged the `then` and `else` clauses). Hint: use De Morgan's Law.

```
if (_________________________________) {
    // action y :)
} else {
    // action x :(
}
```

2.3 Download and install the Icarus Verilog or any other Verilog HDL simulators.

2.4 Write the test bench codes to simulate CMOS NAND and NOR gates.

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

2.6 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.

2.7 Design an active-high RS latch with NOR gates.

2.8 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.

2.9 Explain the reason why the DFFE shown in Figure 2.23(b) is a bad design.

2.10 Try to understand the following code and run it on your FPGA (field-programmable gate array) board.

```
module minute_second (clk,m1,m0,s1,s0,dots);            // what's this?
    input        clk;                                   // 50MHz
    output [6:0] m1, m0;
    output [6:0] s1, s0;
    output [3:0] dots;

    reg          sec_clk = 1;
    reg   [24:0] clk_cnt = 0;
    always @ (posedge clk) begin
        if (clk_cnt == 25'd24999999) begin
            clk_cnt <= 0;
            sec_clk <= ~sec_clk;
        end else begin
            clk_cnt <= clk_cnt + 1;
        end
    end

    reg [2:0] min1 = 0, sec1 = 0;
    reg [3:0] min0 = 0, sec0 = 0;
    always @ (posedge sec_clk) begin
        if (sec0 == 4'd9) begin
            sec0 <= 0;
            if (sec1 == 3'd5) begin
                sec1 <= 0;
                if (min0 == 4'd9) begin
                    min0 <= 0;
                    if (min1 == 3'd5) begin
                        min1 <= 0;
                    end else begin
                        min1 <= min1 + 1;
                    end
                end else begin
                    min0 <= min0 + 1;
                end
            end else begin
                sec1 <= sec1 + 1;
            end
        end else begin
            sec0 <= sec0 + 1;
        end
```

```
    end

    assign m1 = seg7({1'b0,min1});
    assign s1 = seg7({1'b0,sec1});
    assign m0 = seg7(min0);
    assign s0 = seg7(sec0);
    assign dots = {1'b1,sec_clk,2'b11};

    //   0
    // 5   1
    //   6
    // 4   2
    //   3
    function [6:0] seg7;
        input [3:0] q;
        case (q)
            4'd0 : seg7 = 7'b1000000;
            4'd1 : seg7 = 7'b1111001;
            4'd2 : seg7 = 7'b0100100;
            4'd3 : seg7 = 7'b0110000;
            4'd4 : seg7 = 7'b0011001;
            4'd5 : seg7 = 7'b0010010;
            4'd6 : seg7 = 7'b0000010;
            4'd7 : seg7 = 7'b1111000;
            4'd8 : seg7 = 7'b0000000;
            4'd9 : seg7 = 7'b0010000;
            default: seg7 = 7'b1111111;
        endcase
    endfunction
endmodule
```

2.11 Still using eight registers, can you revise the `fifo.v` code given in this chapter so that the FIFO can hold eight data at the same time?

2.12 Design a control circuit for a vending machine of yours.