counter, an extra XOR gate and an $n - 1$ input NOR gate connected to all shift-register outputs except X0 accomplishes the same thing.

The states of an LFSR counter are not visited in binary counting order. However, LFSR counters are typically used in applications where this characteristic is an advantage. A major application of LFSR counters is in generating test inputs for logic circuits. In most cases, the “pseudo-random” counting sequence of an LFSR counter is more likely than a binary counting sequence to detect errors. LFSRs are also used in the encoding and decoding circuits for certain error-detecting and error-correcting codes, including CRC codes, which we introduced in Section 2.15.4.

In data communications, LFSR counters are often used to “scramble” and “descramble” the data patterns transmitted by high-speed modems and network interfaces, including 100 Mbps Ethernet. This is done by XORing the LFSR’s output with the user data stream. Even when the user data stream contains a long run of 0s or 1s, combining it with the LFSR’s pseudo-random output improves the DC balance of the transmitted signal and creates a rich set of transitions that allows clocking information to be recovered more easily at the receiver.

8.5.9 Shift Registers in ABEL and PLDs

General-purpose shift registers can be specified quite easily in ABEL and fit nicely into typical sequential PLDs. For example, Figure 8-70 and Table 8-23 show how to realize a function similar to that of a 74x194 universal shift register using a 16V8. Notice that one of the I/O pins of the 16V8, pin 12, is used as an input.

The 16V8 realization of the ‘194 differs from the real ’194 in just one way—in the function of the CLR_L input. In the real ’194, CLR_L is an asynchronous input, while in the 16V8 it is sampled along with other inputs at the rising edge of CLK.

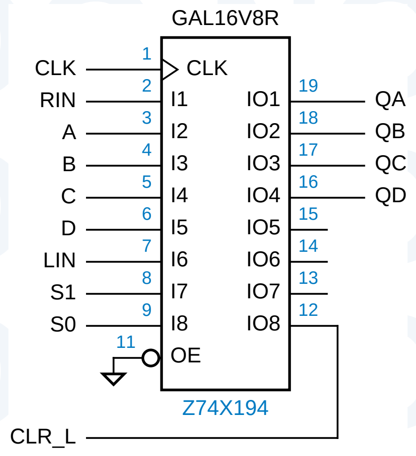

Figure 8-70
PLD realizations of a 74x194-like universal shift register with synchronous clear.

Table 8-23 ABEL program for a 4-bit universal shift register.

```
module Z74x194
title '4-bit Universal Shift Register'
Z74X194 device 'P16V8R';

" Input pins
CLK, RIN, A, B, C, D, LIN          pin 1, 2, 3, 4, 5, 6, 7;
S1, S0, CLR_L                      pin 8, 9, 12;

" Output pins
QA, QB, QC, QD                     pin 19, 18, 17, 16 istype 'reg';

" Set definitions
INPUT   = [  A,  B,  C, D  ];
LEFTIN  = [ QB, QC, QD, LIN];
RIGHTIN = [RIN, QA, QB, QC ];
OUT     = [ QA, QB, QC, QD ];

CTRL  = [S1,S0];
HOLD  = (CTRL == [0,0]);
RIGHT = (CTRL == [0,1]);
LEFT  = (CTRL == [1,0]);
LOAD  = (CTRL == [1,1]);

equations
OUT.CLK = CLK;

OUT := !CLR_L & (
           HOLD & OUT
         # RIGHT & RIGHTIN
         # LEFT & LEFTIN
         # LOAD & INPUT);

end Z74x194
```

If you really need to provide an asynchronous clear input, you can use the 22V10, which provides a single product line to control the reset inputs of all of its flip-flops. This requires only a few changes in the original program (see Exercise 8.50).

The flexibility of ABEL can be used to create shift registers circuits with more or different functionality. For example, Table 8-24 defines an 8-bit shift register that can be cleared, loaded with a single 1 in any bit position, shifted left, shifted right, or held. The operation to be performed at each clock tick is specified by a 4-bit operation code, OP[3:0]. Despite the large number of "WHEN" cases, the circuit can be synthesized with only five product terms per output..

Table 8-24 ABEL program for a multi-function shift register.

```
module shifty
title '8-bit shift register with decoded load'

" Inputs and Outputs
CLK, OP3..OP0              pin;
Q7..Q0                     pin istype 'reg';

" Definitions

Q = [Q7..Q0];
OP = [OP3..OP0];

HOLD = (OP == 0);
CLEAR = (OP == 1);
LEFT = (OP == 2);
RIGHT = (OP == 3);
NOP = (OP >= 4) & (OP < 8);
LOADQ0 = (OP == 8);
LOADQ1 = (OP == 9);
LOADQ2 = (OP == 10);
LOADQ3 = (OP == 11);
LOADQ4 = (OP == 12);
LOADQ5 = (OP == 13);
LOADQ6 = (OP == 14);
LOADQ7 = (OP == 15);

Equations

Q.CLK = CLK;

WHEN HOLD THEN Q := Q;
ELSE WHEN CLEAR THEN Q := 0;
ELSE WHEN LEFT THEN Q := [Q6..Q0, Q7];
ELSE WHEN RIGHT THEN Q := [Q0, Q7..Q1];
ELSE WHEN LOADQ0 THEN Q := 1;
ELSE WHEN LOADQ1 THEN Q := 2;
ELSE WHEN LOADQ2 THEN Q := 4;
ELSE WHEN LOADQ3 THEN Q := 8;
ELSE WHEN LOADQ4 THEN Q := 16;
ELSE WHEN LOADQ5 THEN Q := 32;
ELSE WHEN LOADQ6 THEN Q := 64;
ELSE WHEN LOADQ7 THEN Q := 128;
ELSE Q := Q;

end shifty
```

Table 8-25 Program for an 8-bit ring counter.

```
module Ring8
title '8-bit Ring Counter'

" Inputs and Outputs
MCLK, CNTEN, RESTART                 pin;
S0..S7                               pin istype 'reg';

equations

[S0..S7].CLK = MCLK;

S0 := CNTEN & !S0 & !S1 & !S2 & !S3 & !S4 & !S5 & !S6  " Self-sync
    # !CNTEN & S0                                      " Hold
    # RESTART;                                         " Start with one 1
[S1..S7] := !RESTART & ( !CNTEN & [S1..S7]     " Shift
                        # CNTEN & [S0..S6] );  " Hold
end Ring8
```

ABEL can be used readily to specify shift register counters of the various types that we introduced in previous subsections. For example, Table 8-25 is the program for an 8-bit ring counter. We've used the extra capability of the PLD to add two functions not present in our previous MSI designs: counting occurs only if CNTEN is asserted, and the next state is forced to S0 if RESTART is asserted.

Ring counters are often used to generate multiphase clocks or enable signals in digital systems, and the requirements in different systems are many and varied. The ability to reprogram the counter's behavior easily is a distinct advantage of an HDL-based design.

Figure 8-71 shows a set of clock or enable signals that might be required in a digital system with six distinct phases of operation. Each phase lasts for two ticks of a master clock signal, MCLK, during which the corresponding active-low phase-enable signal Pi_L is asserted. We can obtain this sort of timing from a ring counter if we provide an extra flip-flop to count the two ticks of each phase, so that a shift occurs on the *second* tick of each phase.

The timing generator can be built with a few inputs and outputs of a PLD. Three control inputs are provided, with the following behavior:

RESET	When this input is asserted, no outputs are asserted. The counter always goes to the first tick of phase 1 after RESET is negated.
RUN	When asserted, this input allows the counter to advance to the second tick of the current phase, or to the first tick of the next phase; otherwise, the current tick of the current phase is extended.
RESTART	Asserting this input causes the counter to go back to the first tick of phase 1, even if RUN is not asserted.

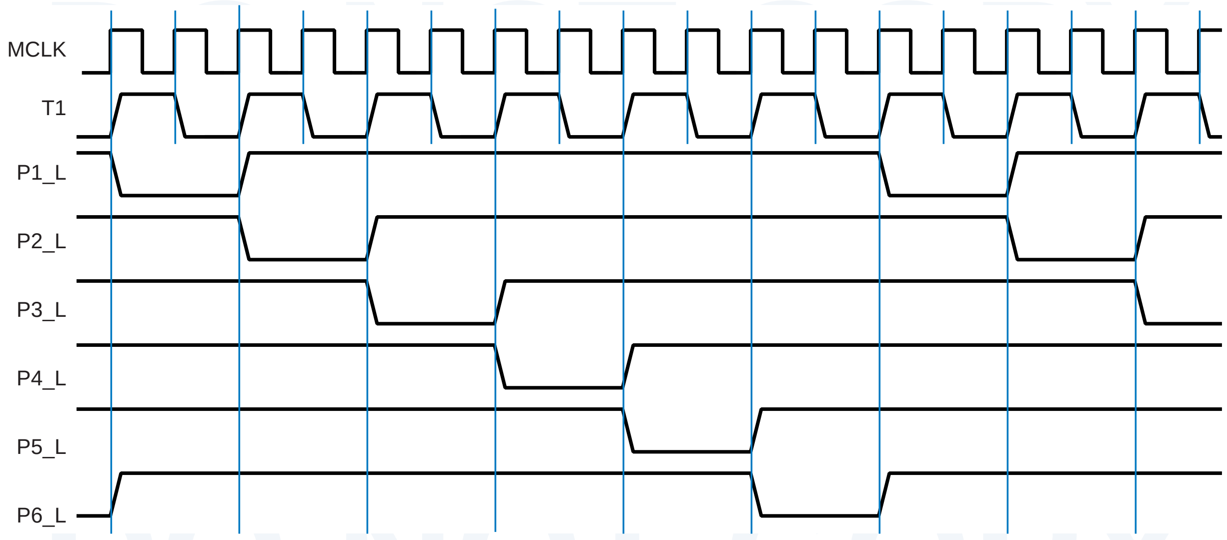

Figure 8-71 Six-phase timing waveforms required in a certain digital system.

Table 8-26 Program for a six-phase waveform generator.

```
module TIMEGEN6
title 'Six-phase Master Timing Generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART                    pin;
T1, P1_L, P2_L, P3_L, P4_L, P5_L, P6_L     pin istype 'reg';

" State definitions
PHASES = [P1_L, P2_L, P3_L, P4_L, P5_L, P6_L];
NEXTPH = [P6_L, P1_L, P2_L, P3_L, P4_L, P5_L];
SRESET = [1, 1, 1, 1, 1, 1];
P1 =     [0, 1, 1, 1, 1, 1];

equations
T1.CLK = MCLK; PHASES.CLK = MCLK;

WHEN RESET THEN {T1 := 1; PHASES := SRESET;}
ELSE WHEN (PHASES==SRESET) # RESTART THEN {T1 := 1; PHASES := P1;}
ELSE WHEN RUN & T1 THEN {T1 := 0; PHASES := PHASES;}
ELSE WHEN RUN & !T1 THEN {T1 := 1; PHASES := NEXTPH;}
ELSE {T1 := T1; PHASES := PHASES;}

end TIMEGEN6
```

Table 8-26 is a program that creates the required behavior. Notice the use of sets to specify the ring counter's behavior very concisely, with the RESET, RESTART, and RUN having the specified behavior in any counter phase.

Table 8-27 Alternate program for the waveform generator.

```
module TIMEGN6A
title 'Six-phase Master Timing Generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART                  pin;
T1, P1_L, P2_L, P3_L, P4_L, P5_L, P6_L     pin istype 'reg';

" State definitions
TSTATE = [T1, P1_L, P2_L, P3_L, P4_L, P5_L, P6_L];
SRESET = [1, 1, 1, 1, 1, 1, 1];
P1F =    [1, 0, 1, 1, 1, 1, 1];
P1S =    [0, 0, 1, 1, 1, 1, 1];
P2F =    [1, 1, 0, 1, 1, 1, 1];
P2S =    [0, 1, 0, 1, 1, 1, 1];
P3F =    [1, 1, 1, 0, 1, 1, 1];
P3S =    [0, 1, 1, 0, 1, 1, 1];
P4F =    [1, 1, 1, 1, 0, 1, 1];
P4S =    [0, 1, 1, 1, 0, 1, 1];
P5F =    [1, 1, 1, 1, 1, 0, 1];
P5S =    [0, 1, 1, 1, 1, 0, 1];
P6F =    [1, 1, 1, 1, 1, 1, 0];
P6S =    [0, 1, 1, 1, 1, 1, 0];

equations
TSTATE.CLK = MCLK;
WHEN RESET THEN TSTATE := SRESET;

state_diagram TSTATE

state SRESET: IF RESET THEN SRESET ELSE P1F;

state P1F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P1S ELSE P1F;

state P1S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P2F ELSE P1S;

state P2F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P2S ELSE P2F;

state P2S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P3F ELSE P2S;

state P3F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P3S ELSE P3F;

state P3S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P4F ELSE P3S;

state P4F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P4S ELSE P4F;

state P4S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P5F ELSE P4S;
```

Table 8-27 (continued) Alternate program for the waveform generator.

```
state P5F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P5S ELSE P5F;

state P5S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P6F ELSE P5S;

state P6F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P6S ELSE P6F;

state P6S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P1F ELSE P6S;

end TIMEGN6A
```

The same timing-generator behavior in Figure 8-71 can be specified using a state-machine design approach, as shown in Table 8-27. This ABEL program, though longer, generates the same external behavior during normal operation as the previous one, and from a certain point of view it may be easier to understand. However, its realization requires 8 to 20 AND terms per output, compared to only 3 to 5 per output for the original ring-counter version. This is a good example of how you can improve circuit efficiency and performance by adapting a standard, simple structure to a “custom” design problem, rather than grinding out a brute-force state machine.

RELIABLE RESET

Notice in Table 8-27 that TSTATE is assigned a value in the equations section of the program, as well as being used in the state_diagram section. This was done very a very specific purpose, to ensure that the program goes to the SRESET state from any undefined state, as explained below.

ABEL augments the on-set for an output each time the output appears on the left-hand side of an equation, as we explained for combinational outputs on page 252. In the case of registered outputs, ABEL also augments the on-set of each state variable in the state vector for each “state” definition in a state_diagram. For each state-variable output, all of the input combinations that cause that output to be 1 in each state are added to the output’s on-set.

The state machine in Table 8-27 has 2^7 or 128 states in total, of which only 13 are explicitly defined and have a transition into SRESET. Nevertheless, the WHEN equation ensures that anytime that RESET is asserted, the machine goes to the SRESET state. This is true regardless of the state definitions in the state_diagram. When RESET is asserted, the all-1s state encoding of SRESET is, in effect, ORed with the next state, if any, specified by the state_diagram. This approach to reliable reset would not be possible if SRESET were encoded as all 0s, for example.

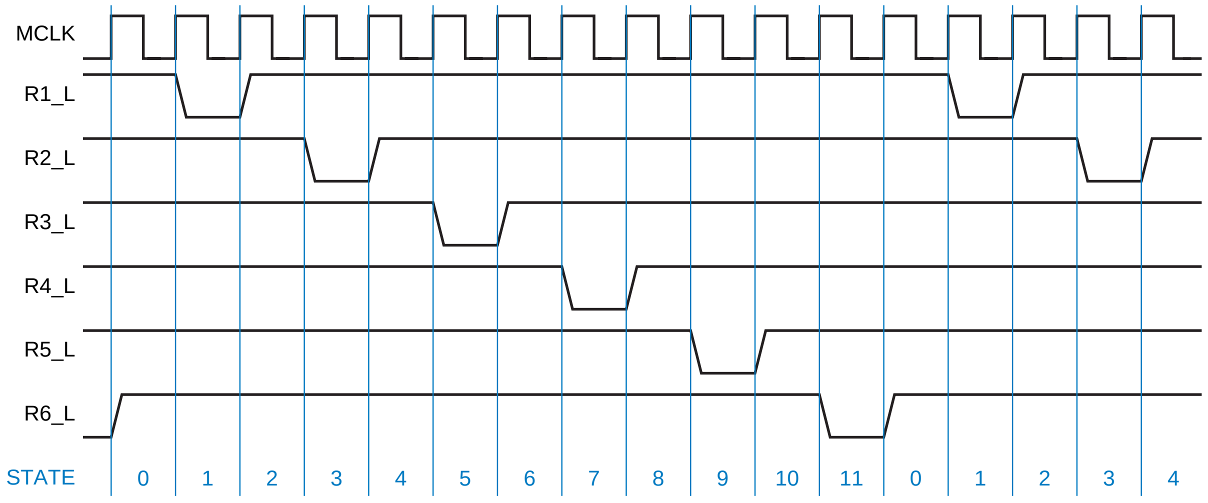

Figure 8-72 Modified timing waveforms for a digital system.

Now let's look at a variation of the previous timing waveforms that might be required in a different system. Figure 8-72 is similar to the previous waveforms, except that each phase output Ri is asserted for only one clock tick per phase. This change has a subtle but important effect on the design approach.

In the original design, we used a six-bit ring counter and one auxiliary state bit T1 to keep track of the two states within each phase. With the new waveforms, this is not posible. In the states between active-low pulses (STATE = 0, 2, 4, etc. in Figure 8-72), the phase outputs are all negated, so they can no longer be used to figure out which state should be visited next. Something else is needed to keep track of the state.

There are many different ways to solve this problem. One idea is to start with the original design in Table 8-26, but use the phase outputs P1_L, P2_L,

Table 8-28 Additions to Table 8-26 for a modified six-phase waveform generator.

```
module TIMEG12K
...
R1_L, R2_L, R3_L, R4_L, R5_L, R6_L         pin istype 'com';
...
OUTPUTS = [R1_L, R2_L, R3_L, R4_L, R5_L, R6_L];

equations
...
!OUTPUTS = !PHASES & !T1;

end TIMEG12K
```

and so on as internal states only. Then, each phase output Ri_L can be defined as a Moore-type combinational output that is asserted when the corresponding Pi_L is asserted *and* we are in the second tick of a phase. The additional ABEL code to support this first approach is shown in Table 8-28.

This first approach is easy, and it works just fine if the Pi_L signals are going to be used only as enables or other control inputs. However, it's a bad idea if these signals are going to be used as clocks, because they may have glitches, as we'll now explain. The Pi_L and T1 signals are all outputs from flip-flops clocked by the same master clock MCLK. Although these signals change at approximately the same time, their timing is never quite exact. One output may change sooner than another; this is called *output timing skew*. For example, suppose that on the transition from state 1 to 2 in Figure 8-71, P2_L goes LOW before T1 goes HIGH. In this case, a short glitch could appear on the R2_L output.

output timing skew

To get glitch-free outputs, we should design the circuit so that each phase output is the a registered output. One way to do this is to build a 12-bit ring counter, and only use alternate outputs to yield the desired waveforms; an ABEL program using this approach is shown in Table 8-29.

Table 8-29 ABEL program for a modified six-phase waveform generator.

```
module TIMEG12
title 'Modified six-phase Master Timing Generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART                    pin;
P1_L, P2_L, P3_L, P4_L, P5_L, P6_L           pin istype 'reg';
P1A, P2A, P3A, P4A, P5A, P6A                 pin istype 'reg';

" State definitions
PHASES = [P1A, P1_L, P2A, P2_L, P3A, P3_L, P4A, P4_L, P5A, P5_L, P6A, P6_L];
NEXTPH = [P6_L, P1A, P1_L, P2A, P2_L, P3A, P3_L, P4A, P4_L, P5A, P5_L, P6A];
SRESET = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
P1 =     [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];

equations

PHASES.CLK = MCLK;

WHEN RESET THEN PHASES := SRESET;
ELSE WHEN RESTART # (PHASES == SRESET) THEN PHASES := P1;
ELSE WHEN RUN THEN PHASES := NEXTPH;
ELSE PHASES := PHASES;

end TIMEG12
```

Table 8-30 Counter-based program for six-phase waveform generator.

```
module TIMEG12A
title 'Counter-based six-phase master timing generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART               pin;
P1_L, P2_L, P3_L, P4_L, P5_L, P6_L      pin istype 'reg';
CNT3..CNT0                              pin istype 'reg';

" Definitions
CNT = [CNT3..CNT0];
P_L = [P1_L, P2_L, P3_L, P4_L, P5_L, P6_L];

equations

CNT.CLK = MCLK;  P_L.CLK = MCLK;

WHEN RESET THEN CNT := 15
ELSE WHEN RESTART THEN CNT := 0
ELSE WHEN (RUN & (CNT < 11)) THEN CNT := CNT + 1
ELSE WHEN RUN THEN CNT := 0
ELSE CNT := CNT;

P1_L := !(CNT == 0);
P2_L := !(CNT == 2);
P3_L := !(CNT == 4);
P4_L := !(CNT == 6);
P5_L := !(CNT == 8);
P6_L := !(CNT == 10);

end TIMEG12A
```

Still another approach is to recognize that since the waveforms cycle through 12 states, we can build a modulo-12 binary counter and decode the states of that counter. An ABEL program using this approach is shown in Table 8-30. The states of the counter correspond to the “STATE” values shown in Figure 8-72. Since the phase outputs are registered, they are glitch-free. Note that they are decoded one cycle early, to account for the one-tick decoding delay. Also, during reset, the counter is forced to state 15 rather than 0, so that the P1_L output is not asserted during reset.

8.5.10 Shift Registers in VHDL

Shift registers can be specified structurally or behaviorally in VHDL; we’ll look at a few behavioral descriptions and applications. Table 8-31 is the function table for an 8-bit shift register with an extended set of functions. In addition to the hold, load, and shift functions of the 74x194 and 74x299, it performs circular

Table 8-31 Function table for an extended-function 8-bit shift register.

	Inputs			*Next state*							
Function	***S2***	***S1***	***S0***	***Q7****	***Q6****	***Q5****	***Q4****	***Q3****	***Q2****	***Q1****	***Q0****
Hold	0	0	0	Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0
Load	0	0	1	D7	D6	D5	D4	D3	D2	D1	D0
Shift right	0	1	0	RIN	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Shift left	0	1	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	LIN
Shift circular right	1	0	0	Q0	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Shift circular left	1	0	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	Q7
Shift arithmetic right	1	1	0	Q7	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Shift arithmetic left	1	1	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	0

circular shift
arithmetic shift

and arithmetic shift operations. In the *circular shift* operations, the bit that "falls off" one end during a shift is fed back into the other end. In the *arithmetic shift* operations, the edge input is set up for multiplication or division by 2; for a left shift, the right input is 0, and for a right shift, the leftmost (sign) bit is replicated.

A behavioral VHDL program for the extended-function shift register is shown in Table 8-32. As in previous examples, we define a process and use the `event` attribute on the `CLK` signal to obtain the desired edge-triggered behavior. Several other features of this program are worth noting:

- An internal signal, `IQ`, is used for what eventually becomes the `Q` output, so it can be both read and written by process statements. Alternatively, we could have defined the `Q` output as type "`buffer`".
- The `CLR` input is asychronous; because it's in the process sensitivity list, it is tested whenever it changes. And the `IF` statement is structured so that `CLR` takes precedence over any other condition.
- A `CASE` statement is used to define the operation of the shift register for the eight possible values of the select inputs `S(2 downto 0)`.
- In the `CASE` statement, the "`when others`" case is required to prevent the compiler from complaining about approximately 2^{32} uncovered cases!
- The "`null`" statement indicates that no action is taken in certain cases. In case 1, note that no action is required; the default is for a signal to hold its value unless otherwise stated.
- In most of the cases, the concatenation operator "`&`" is used to construct an 8-bit array from a 7-bit subset of `IQ` and one other bit.

Table 8-32 VHDL program for an extended-function 8-bit shift register.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity Vshftreg is
    port (
        CLK, CLR, RIN, LIN: in STD_LOGIC;
        S: in STD_LOGIC_VECTOR (2 downto 0); -- function select
        D: in STD_LOGIC_VECTOR (7 downto 0); -- data in
        Q: out STD_LOGIC_VECTOR (7 downto 0) -- data out
    );
end Vshftreg;

architecture Vshftreg_arch of Vshftreg is
signal IQ: STD_LOGIC_VECTOR (7 downto 0);
begin
process (CLK, CLR, IQ)
  begin
    if (CLR='1') then IQ <= (others=>'0'); -- Asynchronous clear
    elsif (CLK'event and CLK='1') then
      case CONV_INTEGER(S) is
        when 0 => null;                        -- Hold
        when 1 => IQ <= D;                     -- Load
        when 2 => IQ <= RIN & IQ(7 downto 1);  -- Shift right
        when 3 => IQ <= IQ(6 downto 0) & LIN;  -- Shift left
        when 4 => IQ <= IQ(0) & IQ(7 downto 1); -- Shift circular right
        when 5 => IQ <= IQ(6 downto 0) & IQ(7); -- Shift circular left
        when 6 => IQ <= IQ(7) & IQ(7 downto 1); -- Shift arithmetic right
        when 7 => IQ <= IQ(6 downto 0) & '0';  -- Shift arithmetic left
        when others => null;
      end case;
    end if;
    Q <= IQ;
  end process;
end Vshftreg_arch;
```

- Because of VHDL's strong requirements for type matching, we used the `CONV_INTEGER` function in the `IEEE.std_logic_unsigned` library to convert the `STD_LOGIC_VECTOR` select input `S` to an integer in the `CASE` statement. Alternatively, we could have written each case label as a `STD_LOGIC_VECTOR` (e.g., `('0','1','1')` instead of integer 3).

One application of shift registers is in ring counters, as in the six-phase waveform generator that we described on page 636 with the waveforms in Figure 8-71. A VHDL program that provides the corresponding behavior is

Table 8-33 VHDL program for a six-phase waveform generator.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vtimegn6 is
    port (
        MCLK, RESET, RUN, RESTART: in STD_LOGIC; -- clock, control inputs
        P_L: out STD_LOGIC_VECTOR (1 to 6)       -- active-low phase outputs
    );
end Vtimegn6;

architecture Vtimegn6_arch of Vtimegn6 is
signal IP: STD_LOGIC_VECTOR (1 to 6); -- internal active-high phase signals
signal T1: STD_LOGIC;                 -- first tick within phase
begin
process (MCLK, IP)
  begin
    if (MCLK'event and MCLK='1') then
      if (RESET='1') then
        T1 <= '1'; IP <= ('0','0','0','0','0','0');
      elsif ((IP=('0','0','0','0','0','0')) or (RESTART='1')) then
        T1 <= '1'; IP <= ('1','0','0','0','0','0');
      elsif (RUN='1') then
        T1 <= not T1;
        if (T1='0') then IP <= IP(6) & IP(1 to 5); end if;
      end if;
    end if;
    P_L <= not IP;
  end process;
end Vtimegn6_arch;
```

shown in Table 8-33. As in the previous VHDL example, an internal active-high signal vector, IP, is used for reading and writing what eventually becomes the circuit's output; this internal signal is conveniently inverted in the last statement to obtain the required active-low output signal vector. The rest of the program is straightforward, but notice that it has three levels of nested IF statements.

A possible modification to the preceding application is to produce output waveforms that are asserted only during the second tick of each two-tick phase; such waveforms were shown in Figure 8-72 on page 640. One way to do this is to create a 12-bit ring counter, and use only alternate outputs. In the VHDL realization, only the six phase outputs, P_L(1 to 6), would appear in the entity definition. The additional six signals, which we name NEXTP(1 to 6), are local to the architecture definition. Figure 8-73 shows the relationship of these signals for shift-register operation, and Table 8-34 is the VHDL program.

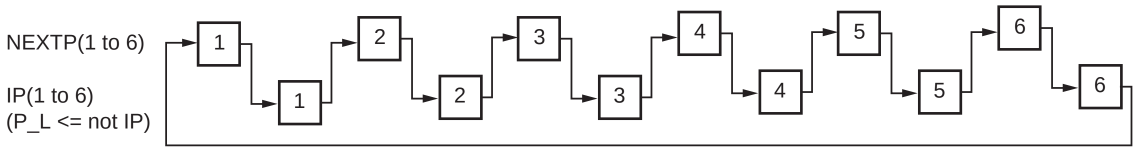

Figure 8-73 Shifting sequence for waveform generator 12-bit ring counter.

Table 8-34 VHDL program for a modified six-phase waveform generator.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vtimeg12 is
    port (
        MCLK, RESET, RUN, RESTART: in STD_LOGIC; -- clock, control inputs
        P_L: out STD_LOGIC_VECTOR (1 to 6)       -- active-low phase outputs
    );
end Vtimeg12;

architecture Vtimeg12_arch of Vtimeg12 is
signal IP, NEXTP: STD_LOGIC_VECTOR (1 to 6); -- internal active-high phase signals
begin
process (MCLK, IP, NEXTP)
  variable TEMP: STD_LOGIC_VECTOR (1 to 6);  -- temporary for signal shift
  constant IDLE: STD_LOGIC_VECTOR (1 to 6) := ('0','0','0','0','0','0');
  constant FIRSTP: STD_LOGIC_VECTOR (1 to 6) := ('1','0','0','0','0','0');
  begin
    if (MCLK'event and MCLK='1') then
      if (RESET='1') then IP <= IDLE;  NEXTP <= IDLE;
      elsif (RESTART='1') or (IP=IDLE and NEXTP=IDLE) then IP <= IDLE;  NEXTP <= FIRSTP;
      elsif (RUN='1') then
        if (IP=IDLE) and (NEXTP=IDLE) then NEXTP <= FIRSTP;
        else TEMP := IP;  IP <= NEXTP;  NEXTP <= TEMP(6) & TEMP(1 to 5);
        end if;
      end if;
    end if;
    P_L <= not IP;
  end process;
end Vtimeg12_arch;
```

As in the previous program, a 6-bit active-high signal, IP, is declared in the architecture body and used for reading and writing what eventually becomes the circuit’s active-low output, P_L. The additional 6-bit signal, NEXTP, holds the remaining six bits of state. Constants IDLE and FIRSTP are used to improve the program’s readability.

Notice that a six-bit variable, `TEMP`, is used just as a temporary place to hold the old value of `IP` when shifting occurs—`IP` is loaded with `NEXTP`, and `NEXTP` is loaded with the shifted, old value of `IP`. Because the assignment statements in a process are executed *sequentially*, we couldn't get away with just writing "`IP <= NEXTP; NEXTP <= IP(6) & IP(1 to 5);`". If we did that, then `NEXTP` would pick up the *new* value of `IP`, not the old. Notice also that since `TEMP` is a variable, not a signal, its value is not preserved between process invocations. Thus, the compiler does not synthesize any flip-flops to hold `TEMP`'s value.

*8.6 Iterative versus Sequential Circuits

We introduced iterative circuits in Section 5.9.2. The function of an *n*-module iterative circuit can be performed by a sequential circuit that uses just one copy of the module but requires *n* steps (clock ticks) to obtain the result. This is an excellent example of a space/time tradeoff in digital design.

As shown in Figure 8-74, flip-flops are used in the sequential-circuit version to store the cascading outputs at the end of each step; the flip-flop outputs are used as the cascading inputs at the beginning of the next step. The flip-flops must be initialized to the boundary-input values before the first clock tick, and they contain the boundary-output values after the *n*th tick.

Since an iterative circuit is a combinational circuit, all of its primary and boundary inputs may be applied simultaneously, and its primary and boundary outputs are all available after a combinational delay. In the sequential-circuit version, the primary inputs must be delivered sequentially, one per clock tick, and the primary outputs are produced with similar timing. Therefore, serial-out shift registers are often used to provide the inputs, and serial-in shift registers are used to collect the outputs. For this reason, the sequential-circuit version of an "iterative widget" is often called a "serial widget."

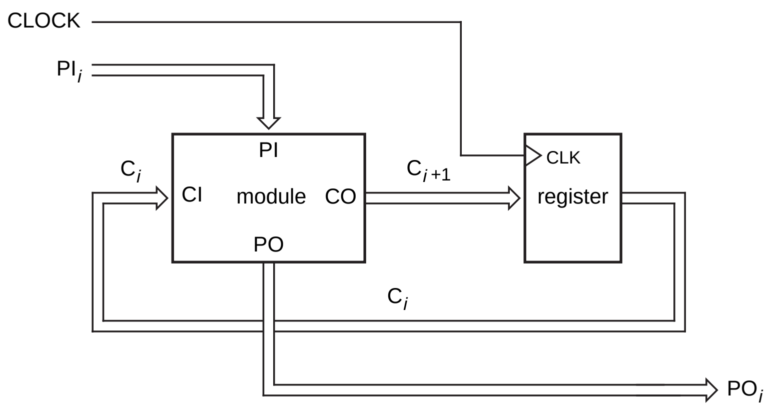

Figure 8-74 General structure of the sequential-circuit version of an iterative circuit.