```
      elsif (clk'event and clk='1') then
         if wr_en='1' then
            array_reg(to_integer(unsigned(w_addr))) <= w_data;
         end if;
      end if;
   end process;
   -- read port
   r_data <= array_reg(to_integer(unsigned(r_addr)));
end arch;
```

The code includes several new features. First, since no built-in two-dimensional array is defined in the std_logic_1164 package a user-defined array-of-array data type, reg_file_type, is introduced. It is first defined by a type statement and is then used by the array_reg signal. Second, a signal is used as an index to access an element in the array, as in array_reg(..w_addr..). Although the description is very abstract, Xilinx software recognizes this language construct and can derive the correct implementation accordingly. The array_reg(...) <= ... and ... <= array_reg(...) statements infer decoding and multiplexing logic, respectively.

Some applications may need to retrieve multiple data words at the same time. This can be done by adding an additional read port:

```
   r_data2 <= array_reg(to_integer(unsigned(r_addr_2)));
```

4.2.4 Storage components in a Spartan-3 device$^{Xilinx\ specific}$

In a Spartan-3 device, each logic cell contains a D FF with asynchronous reset and synchronous enable. These D FFs basically constitute the register of Figure 4.2. Since a logic cell also contains a four-input LUT, it will be wasteful if the cell is just used simply as 1 bit of a massive storage. The Spartan-3 device also has distributed RAM (random access memory) and block RAM modules, and they can be used for larger storage requirements. These modules can be configured for synchronous operation, and their characteristics are somewhat like a restricted version of the register file. The configuration and inference of these modules are discussed in Chapter 11.

4.3 SIMPLE DESIGN EXAMPLES

We illustrate the construction of several simple, representative sequential circuits in this section.

4.3.1 Shift register

Free-running shift register A free-running shift register shifts its content to the left or right by one position in each clock cycle. There is no other control signal. The code for an N-bit free-running shift-right register is shown in Listing 4.7.

Listing 4.7 Free-running shift register

```
library ieee;
use ieee.std_logic_1164.all;
entity free_run_shift_reg is
```

```
   generic(N: integer := 8);
   port(
      clk, reset: in std_logic;
      s_in: in std_logic;
      s_out: out std_logic
   );
end free_run_shift_reg;

architecture arch of free_run_shift_reg is
   signal r_reg: std_logic_vector(N-1 downto 0);
   signal r_next: std_logic_vector(N-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic (shift right 1 bit)
   r_next <= s_in & r_reg(N-1 downto 1);
   -- output
   s_out <= r_reg(0);
end arch;
```

The next-state logic is a 1-bit shifter, which shifts r_reg right one position and inserts the serial input, s_in, to the MSB. Since the 1-bit shifter involves only reconnection of the input and output signals, no real logic is needed. Its propagation delay represents the smallest possible T_{comb}, and the corresponding f_{max} represents the highest clock rate that can be achieved for a given device technology.

Universal shift register A universal shift register can load parallel data, shift its content left or right, or remain in the same state. It can perform parallel-to-serial operation (first loading parallel input and then shifting) or serial-to-parallel operation (first shifting and then retrieving parallel output). The desired operation is specified by a 2-bit control signal, ctrl. The code is shown in Listing 4.8.

Listing 4.8 Universal shift register

```
library ieee;
use ieee.std_logic_1164.all;
entity univ_shift_reg is
   generic(N: integer := 8);
   port(
      clk, reset: in std_logic;
      ctrl: in std_logic_vector(1 downto 0);
      d: in std_logic_vector(N-1 downto 0);
      q: out std_logic_vector(N-1 downto 0)
   );
end univ_shift_reg;

architecture arch of univ_shift_reg is
```

```
   signal r_reg: std_logic_vector(N-1 downto 0);
   signal r_next: std_logic_vector(N-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   with ctrl select
    r_next <=
      r_reg                        when "00", --no op
      r_reg(N-2 downto 0) & d(0)   when "01", --shift left;
      d(N-1) & r_reg(N-1 downto 1) when "10", --shift right;
      d                            when others; -- load
   -- output
   q <= r_reg;
end arch;
```

The next-state logic uses a 4-to-1 multiplexer to select the desired next value of the register. Note that the LSB and MSB of d (i.e., d(0) and d(N-1)) are used as serial input for the shift-left and shift-right operations.

In a Xilinx Spartan-3 device, a logic cell's 4-input LUT is implemented by a 16-by-1 SRAM. The same SRAM can also be configured as a cascading chain of sixteen 1-bit SRAM cells, which resembles a 16-bit shift register. This can be used to construct certain forms of shift register and leads to very efficient implementation. **Xilinx specific**

4.3.2 Binary counter and variant

Free-running binary counter A free-running binary counter circulates through a binary sequence repeatedly. For example, a 4-bit binary counter counts from "0000", "0001", . . ., to "1111" and wraps around. The code for a parameterized N-bit free-running binary counter is shown in Listing 4.9.

Listing 4.9 Free-running binary counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity free_run_bin_counter is
   generic(N: integer := 8);
   port(
      clk, reset: in std_logic;
      max_tick: out std_logic;
      q: out std_logic_vector(N-1 downto 0)
   );
end free_run_bin_counter;

architecture arch of free_run_bin_counter is
```

Table 4.1 Function table of a universal binary counter

syn_clr	load	en	up	q*	**Operation**
1	–	–	–	00 ··· 00	synchronous clear
0	1	–	–	d	parallel load
0	0	1	1	q+1	count up
0	0	1	0	q-1	count down
0	0	0	–	q	pause

```
   signal r_reg: unsigned(N-1 downto 0);
   signal r_next: unsigned(N-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
   max_tick <= '1' when r_reg=(2**N-1) else '0';
end arch;
```

The next-state logic is an incrementor, which adds 1 to the register's current value. By definition of the + operator in the IEEE numeric_std package, the operation implicitly wraps around after the r_reg reaches "1...1". The circuit also consists of an output status signal, max_tick, which is asserted when the counter reaches the maximal value, "1...1" (which is equal to $2^N - 1$).

The max_tick signal represents a special type of signal that is asserted for a single clock cycle. In this book, we call this type of signal a *tick* and use the suffix _tick to indicate a signal with this property. It is commonly used to interface with the enable signal of other sequential circuits.

Universal binary counter A universal binary counter is more versatile. It can count up or down, pause, be loaded with a specific value, or be synchronously cleared. Its functions are summarized in Table 4.1. Note the difference between the reset and syn_clr signals. The former is asynchronous and should only be used for system initialization. The latter is sampled at the rising edge of the clock and can be used in normal synchronous design. The code for this counter is shown in Listing 4.10.

Listing 4.10 Universal binary counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity univ_bin_counter is
```

```
    generic(N: integer := 8);
    port(
       clk, reset: in std_logic;
       syn_clr, load, en, up: in std_logic;
       d: in std_logic_vector(N-1 downto 0);
       max_tick, min_tick: out std_logic;
       q: out std_logic_vector(N-1 downto 0)
    );
end univ_bin_counter;

architecture arch of univ_bin_counter is
    signal r_reg: unsigned(N-1 downto 0);
    signal r_next: unsigned(N-1 downto 0);
begin
    -- register
    process(clk,reset)
    begin
       if (reset='1') then
          r_reg <= (others=>'0');
       elsif (clk'event and clk='1') then
          r_reg <= r_next;
       end if;
    end process;
    -- next-state logic
    r_next <= (others=>'0') when syn_clr='1' else
              unsigned(d)    when load='1' else
              r_reg + 1      when en ='1' and up='1' else
              r_reg - 1      when en ='1' and up='0' else
              r_reg;
    -- output logic
    q <= std_logic_vector(r_reg);
    max_tick <= '1' when r_reg=(2**N-1) else '0';
    min_tick <= '1' when r_reg=0 else '0';
end arch;
```

The next-state logic follows the function table and uses a conditional signal assignment to prioritize the desired operations.

Mod-m counter A mod-m counter counts from 0 to $m - 1$ and wraps around. A parameterized mod-m counter is shown in Listing 4.11. It has two generics. One is M, which specifies the limit, m, and the other is N, which specifies the number of bits needed and should be equal to $\lceil \log_2 M \rceil$. The code is shown in Listing 4.11, and the default value is for a mod-10 counter.

Listing 4.11 Mod-m counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mod_m_counter is
    generic(
       N: integer := 4;       -- number of bits
       M: integer := 10       -- mod-M
    );
```

```
   port(
      clk, reset: in std_logic;
      max_tick: out std_logic;
      q: out std_logic_vector(N-1 downto 0)
   );
end mod_m_counter;

architecture arch of mod_m_counter is
   signal r_reg: unsigned(N-1 downto 0);
   signal r_next: unsigned(N-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= (others=>'0') when r_reg=(M-1) else
             r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
   max_tick <= '1' when r_reg=(M-1) else '0';
end arch;
```

The next-state logic is constructed by a conditional signal assignment statement. If the counter reaches M-1, the new value is cleared to 0. Otherwise, it is incremented by 1.

Inclusion of the N parameter in the code is somewhat redundant since its value depends on M. A more elegant way is to define a function that calculates N from M automatically. In VHDL, this can be done by creating a user-defined *function* in a *package* and invoking the package before the entity declaration. This is beyond the scope of this book and the details may be found in the references cited in the Bibliographic section.

4.4 TESTBENCH FOR SEQUENTIAL CIRCUITS

A testbench is a program that mimics a physical lab bench, as discussed in Section 1.4. Developing a comprehensive testbench is beyond the scope of this book. We discuss a simple testbench for the previous universal binary counter in this section. It can serve as a template for other sequential circuits. The code for the testbench is shown in Listing 4.12.

Listing 4.12 Testbench for a universal binary counter

```
library ieee;
use ieee.std_logic_1164.all;

entity bin_counter_tb is
end bin_counter_tb;

architecture arch of bin_counter_tb is
```

```
   constant THREE: integer := 3;
   constant T: time := 20 ns; -- clk period
   signal clk, reset: std_logic;
   signal syn_clr, load, en, up: std_logic;
   signal d: std_logic_vector(THREE-1 downto 0);
   signal max_tick, min_tick: std_logic;
   signal q: std_logic_vector(THREE-1 downto 0);
begin
   --*************************
   -- instantiation
   --*************************
   counter_unit: entity work.univ_bin_counter(arch)
      generic map(N=>THREE)
      port map(clk=>clk, reset=>reset, syn_clr=>syn_clr,
               load=>load, en=>en, up=>up, d=>d,
               max_tick=>max_tick, min_tick=>min_tick, q=>q);

   --*************************
   -- clock
   --*************************
   -- 20 ns clock running forever
   process
   begin
      clk <= '0';
      wait for T/2;
      clk <= '1';
      wait for T/2;
   end process;
   --*************************
   -- reset
   --*************************
   -- reset asserted for T/2
   reset <= '1', '0' after T/2;

   --*************************
   -- other stimulus
   --*************************
   process
   begin
      --*************************
      -- initial input
      --*************************
      syn_clr <= '0';
      load <= '0';
      en <= '0';
      up <= '1';  -- count up
      d <= (others=>'0');
      wait until falling_edge(clk);
      wait until falling_edge(clk);
      --*************************
      -- test load
      --*************************
      load <= '1';
```

```
d <= "011";
wait until falling_edge(clk);
load <= '0';
-- pause 2 clocks
wait until falling_edge(clk);
wait until falling_edge(clk);
--************************
-- test syn_clear
--************************
syn_clr <= '1';  -- clear
wait until falling_edge(clk);
syn_clr <= '0';
--************************
-- test up counter and pause
--************************
en <= '1'; -- count
up <= '1';
for i in 1 to 10 loop -- count 10 clocks
   wait until falling_edge(clk);
end loop;
en <='0';
wait until falling_edge(clk);
wait until falling_edge(clk);
en <='1';
wait until falling_edge(clk);
wait until falling_edge(clk);
--************************
-- test down counter
--************************
up <= '0';
for i in 1 to 10 loop -- run 10 clocks
   wait until falling_edge(clk);
end loop;
--************************
-- other wait conditions
--************************
-- continue until q=2
wait until q="010";
wait until falling_edge(clk);
up <= '1';
-- continue until min_tick changes value
wait on min_tick;
wait until falling_edge(clk);
up <= '0';
wait for 4*T;  -- wait for 80 ns
en <= '0';
wait for 4*T;
--************************
-- terminate simulation
--************************
assert false
   report "Simulation Completed"
 severity failure;
```

```
   end process ;
end arch;
```

The code consists of a component instantiation statement, which creates an instance of a 3-bit counter, and three segments, which generate a stimulus for clock, reset, and regular inputs. Since operation of a synchronous system is synchronized by a clock signal, we define a constant with the built-in data type `time` for the clock period:

```
constant T: time := 20 ns; -- clk period
```

The clock generation is specified by a process:

```
process
begin
   clk <= '0';
   wait for T/2;
   clk <= '1';
   wait for T/2;
end process;
```

The `clk` signal is assigned between '0' and '1' alternatively, and each value lasts for half a period. Note that the process has no sensitivity list and repeats itself forever.

The reset stimulus involves one statement,

```
reset <= '1', '0' after T/2;
```

It indicates that the `reset` signal is set to '1' initially and changed to '0' after half a period. The statement represents the "power-on" condition, in which the `reset` signal is asserted momentarily to clear the system to the initial state. Note that, by default, the `'U'` value (for uninitialized), not `'0'`, is assigned to a signal with the `std_logic` type. Using a short reset pulse is a good mechanism to perform system initialization.

The last process statement generates a stimulus for other input signals. We first test the load and clear operations and then exercise counting in both directions. The final **assert** `false` statement forces the simulator to terminate simulation, as discussed in Section 2.7.

For a synchronous system with positive edge-triggered FFs, an input signal must be stable around the rising edge of the clock signal to satisfy the setup and hold time constraints. One easy way to achieve this is to change an input signal's value during the '1'-to-'0' transition of the `clk` signal. The `falling_edge` function of the `std_logic_1164` package checks this condition, and we can use it in a wait statement:

```
wait until falling_edge(clk);
```

Note that each statement represents a new falling edge, which corresponds to the advancement of one clock cycle. In our template, we generally use this statement to specify the progress of time. For multiple clock cycles, we can use a loop statement:

```
for i in 1 to 10 loop -- count 10 clocks
   wait until falling_edge(clk);
end loop;
```

There are other useful forms of wait statements, as shown at the end of the process. We can wait until a special condition, such as "when q is equal to 2",

```
wait until q="010";
```

or wait until a signal changes, such as