Sequential Circuits – Circuits with Memory

The main categories and common uses are:

1) No clock – latches
   - Nand latch – debounce switches
   - Latch an alarm siren (light beam burglar alarm in Digital 1 course.

2) Clocked
   - Clock level – Flip flips and RAM
   - Clock edge – Flip flops and RAM
   - Clock edge is the most common in FPGAs
   - Flip flop timing requirements (setup, hold, propagation delay) are very important in FPGAs

Describe an active-LOW input S-R latch with inputs named SBAR, RBAR, and one output named Q. It should follow the function table of a NAND latch (see Figure 5-6) and the invalid input combination should produce $Q = 1$.

SBAR	RBAR	Q
0	0	illegal
0	1	1
1	0	0
1	1	no change

1. Q is defined as an OUT, even though it is fed back in the circuit. VHDL does not allow ports of mode OUT to be read within the hardware description code. Outputs with BUFFER mode can be fed back in.
2. A PROCESS describes what happens when the values in the sensitivity list (SBAR, RBAR) change state.
3. The clause after IF will determine which output state occurs when both inputs are active (invalid state). In this code the SET command rules.
4. In VHDL, data latching (storage) is implied by intentionally leaving out the ELSE choice in an IF statement. The compiler will "understand" that when neither of the control inputs is active (LOW) the output will not change, which results in the current data bit being stored.

```
ENTITY fig5_67 IS
PORT (sbar, rbar          :IN BIT;
      q                   :OUT BIT);
END fig5_67;

ARCHITECTURE behavior OF fig5_67 IS
BEGIN
PROCESS (sbar, rbar)
BEGIN
   IF sbar = '0' THEN q <= '1';      -- set or illegal command
   ELSIF rbar = '0' THEN q <= '0';   -- reset
   END IF;                           -- hold implied
END PROCESS;
END behavior;
```

FIGURE 5-67 A NAND latch using VHDL.

REGULAR SEQUENTIAL CIRCUIT

4.1 INTRODUCTION

A sequential circuit is a circuit with *memory*, which forms the *internal state* of the circuit. Unlike a combinational circuit, in which the output is a function of input only, the output of a sequential circuit is a function of the input and the internal state. The *synchronous design methodology* is the most commonly used practice in designing a sequential circuit. In this methodology, all storage elements are controlled (i.e., synchronized) by a global clock signal and the data is sampled and stored at the rising or falling edge of the clock signal. It allows designers to separate the storage components from the circuit and greatly simplifies the development process. This methodology is the most important principle in developing a large, complex digital system and is the foundation of most synthesis, verification, and testing algorithms. All of the designs in the book follow this methodology.

56. SEQUENTIAL CIRCUITS

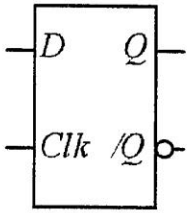
Wait Until can only be used with a clock.
Input D would not be in sensitivity list.

The D flip-flop description introduces several new concepts. First, notice the **WAIT UNTIL** statement. Since there is no sensitivity list on the **PROCESS**, the **WAIT UNTIL** controls the assignment of *internal_Q*. Also notice the term *clock'EVENT*. It literally means that the clock *has* changed – either a rising edge or a falling edge has occurred. Since the *clock'EVENT* is anded with a *clock = '1'* test, the entire **WAIT UNTIL** can be interpreted as "Wait for a rising edge of the clock."

```
ENTITY dff1 IS

    PORT( D              : IN  STD_LOGIC;
          clock          : IN  STD_LOGIC;
          Qhigh, Qlow    : OUT STD_LOGIC );

END dff1;
```

Signals **assigned** in a **clocked** process create D FFs

```
ARCHITECTURE arch OF dff1 IS
   SIGNAL internal_Q : STD_LOGIC;
BEGIN

   Qhigh <=      internal_Q;
   Qlow  <= not  internal_Q;

   PROCESS
   BEGIN
   WAIT UNTIL ( clock'EVENT and clock = '1' );

      internal_Q <= D;

   END PROCESS;

END arch;
```



Left side of <= is assigned

Q: Since *internal_Q* does not change until the rising edge of the clock, is this a synchronous or an asynchronous circuit?

57. INTERNAL SIGNALS

From the previous question, notice the declaration of a local signal, *internal_Q*, in the **ARCHITECTURE**. This is needed as a hold value between the output *Qhigh* and input to the inverter that outputs *Qlow*. Remember, that an OUT signal can *only* appear on the left-hand side of an assignment operator, "<=". In this example, *Qhigh* had to be inverted to generate *Qlow*, so an internal signal was needed as an intermediary value holder.

Q: Can you rewrite the description of the D flip-flop to eliminate the *internal_Q* signal?

Ans: Yes, the code to the right illustrates one possibility. Note that in this description the value of *Qhigh* is *not* used to define *Qlow*.

```
ARCHITECTURE arch OF dff1 IS
BEGIN

   PROCESS
   BEGIN
     WAIT UNTIL ( clock'EVENT and clock = '1' );

     Qhigh <=     D;
     Qlow  <= not D;

   END PROCESS;

END arch;
```

59. Combining Synchronous and Asynchronous Functionality

Notice in the previous questions' VHDL code that the *Qhigh* and *Qlow* signals were synchronously set, while no reset capabilities were described. This code illustrates how to add an asynchronous clear while maintaining the same synchronous functionality of the previous flip-flops.

Q: Why does this example use a sensitivity list instead of the WAIT UNTIL construct?

Wait Until is for clock only, no sensitivity.

Ans: In this case, the **PROCESS** needs to be activated on a rising edge of *clk* or on a

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff2 IS

   PORT( D, clk       : IN  STD_LOGIC;
         clear        : IN  STD_LOGIC;
         Qhigh, Qlow  : OUT STD_LOGIC);

END dff2;

ARCHITECTURE arch OF dff2 IS
   SIGNAL internal_Q : STD_LOGIC;
BEGIN

   Qhigh <=     internal_Q;
   Qlow  <= not internal_Q;

   PROCESS( clk , clear )
   BEGIN

      IF( clear = '1' ) THEN
         internal_Q <= '0';
      ELSIF( clk'EVENT and clk='1' ) THEN
         internal_Q <= D;
      END IF;

   END PROCESS;
```

60. More on Synchronous vs. Asynchronous

The reset functionality can also be made synchronous.

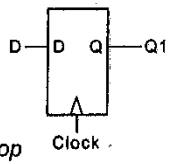
Q: How would you change the code from the previous question to implement a synchronous clear?

Ans: One possible answer is shown to the right. Note that nothing is changed unless the clock undergoes a rising edge. Again, the order of the IF forces the reset to supersede the assignment of internal_Q.

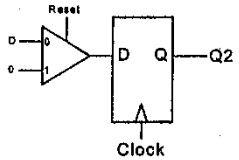
```
ARCHITECTURE arch OF dff2 IS
   SIGNAL internal_Q : STD_LOGIC;
BEGIN

   Qhigh <=     internal_Q;
   Qlow  <= not internal_Q;

   PROCESS
   BEGIN
   WAIT UNTIL ( clock'EVENT and clock = '1' );

      IF( clear = '1' ) THEN
         internal_Q <= '0';
      ELSE
         internal_Q <= D;
      END IF;

   END PROCESS;

END arch;
```

Put clear under clock event.

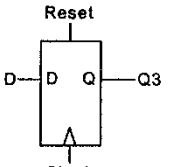
```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;

ENTITY DFFs IS
      PORT(  D, Clock, Reset, Enable   : IN     STD_LOGIC;
             Q1, Q2, Q3, Q4            : OUT  STD_LOGIC  );
END DFFs;

ARCHITECTURE behavior OF DFFs IS
BEGIN


PROCESS                                  -- Positive edge triggered D flip-flop
BEGIN                                    -- If WAIT is used no sensitivity list is used
      WAIT UNTIL ( Clock 'EVENT AND Clock = '1' );
         Q1 <= D;
END PROCESS;
```

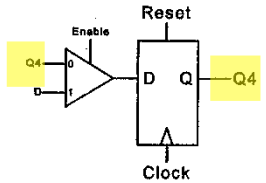



```
PROCESS                                  -- Positive edge triggered D flip-flop
BEGIN                                    --      with synchronous reset
      WAIT UNTIL ( Clock 'EVENT AND Clock = '1' );
            IF reset = '1'  THEN
               Q2 <= '0';
            ELSE
               Q2 <= D;
            END IF;
END PROCESS;
```




```
PROCESS (Reset,Clock)                    -- Positive edge triggered D flip-flop
BEGIN                                    --      with asynchronous reset
      IF reset = '1' THEN
         Q3 <= '0';
      ELSIF ( clock 'EVENT AND clock = '1' ) THEN
         Q3 <= D;
      END IF;
END PROCESS;

PROCESS (Reset,Clock)                    -- Positive edge triggered D flip-flop
BEGIN                                    --      with asynchronous reset and
                                         --      enable
      IF reset = '1' THEN
         Q4 <= '0';
      ELSIF ( clock 'EVENT AND clock = '1' ) THEN
         IF Enable = '1' THEN
            Q4 <= D;
         END IF;
      END IF;
   END PROCESS;
END behavior;
```





```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY counter1 IS

   PORT( clock, reset: IN  STD_LOGIC;
         direction   : IN  STD_LOGIC;
         count       : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 ));

END counter1;

ARCHITECTURE arch OF counter1 IS
   SIGNAL internal_count : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
BEGIN
   count <=  internal_count;

   PROCESS( clock, reset )
   BEGIN

      IF( reset = '1' ) THEN

         internal_count <= "00000000";

      ELSIF( clock'EVENT and clock = '1' ) THEN

         IF( direction = '0' ) THEN
            internal_count <= internal_count - 1;
         ELSE
            internal_count <= internal_count + 1;
         END IF;

      END IF;

   END PROCESS;

END arch;
```

ieee.numeric_std.all;

UNSIGNED (7 DOWNTO 0);

std_logic_vector(internal_count);

This code describes an up/down counter. If the direction signal is a '0' it counts down; otherwise, it counts up. Notice that additional **USE** statements have been added. These define the "+" and "-" operators for the **STD_LOGIC_VECTOR** types.

Q: Which statement causes the need for signal **internal_count**?

SIGNAL rules summary (see NandLand.com, "Variables vs Signals")

1. Signals can be used both inside and outside processes.
2. A signal can be used in multiple processes,
   but only assigned in one of them.
3. Signals are defined in the ARCHITECTURE before BEGIN.
4. Signals are assigned using <= .
5. Signals take the assignment immediately if it is a
   combinatorial circuit, but after the clock if it is a
   sequential circuit because it generates a FF.
6. Signals used in a process end up with the last value assigned to them
   (see Chu Ch. 3 p. 47).

VARIABLE rules summary (NandLand.com, “Variable”)

1. Can only be used inside process.
2. Are **local** – not usable outside that process.
3. Defined after PROCESS and before BEGIN.
4. Assigned using **:=** , which acts **immediately**.
5. If something uses the value of a variable before the assignment, it will be different than the value after the assignment.
6. **In a clocked segment, if a variable is used before it is assigned a value, it will be latched. Conversely, if it is assigned a value before it is used, it will not be latched.**
7. Variables can be tricky, and can be hard for Quartus to synthesize.

Variables A variable is a concept found in a traditional programming language. It can be thought of as a "symbolic memory location" where a value can be stored and modified. There is no direct mapping between a variable and a hardware part. A variable can only be declared and used in a process and is local to that process (the exception is a shared variable, which is difficult to use and is not discussed). The main application of a variable is to describe the abstract behavior of a system.

The syntax of variable declaration is similar to that of signal declaration:

```
variable variable_name, variable_name, ... : data_type
```

An optional initial value can be assigned to variables as well.

The simplified syntax of variable assignment is

```
variable_name := value_expression;
```

Note that no timing information is associated with a variable, and thus only a value, not a waveform, can be assigned to a variable. Since there is no delay, the assignment is known as an *immediate assignment* and the notion := is used. We examine variables in detail when the VHDL process is discussed in Chapter 5.

5.3 VARIABLE ASSIGNMENT STATEMENT

The behavior of the variable assignment is just like that of a regular variable assignment used in a traditional programming language. For example, consider the code segment

```
signal a, b, y: std_logic;
. . .
process(a,b)
  variable tmp: std_logic;
begin
   tmp := '0';
   tmp := tmp or a;
   tmp := tmp or b;
   y <= tmp;
end process;
```

Combinatorial Circuit

tmp gets these values in order: 0, a, a OR b; so y ends up with a OR b.

But this does not implement a two-input or-gate circuit,
it is just a value stored somewhere.

Although the behavior of a variable is easy to understand, mapping it into hardware is difficult. For example, to realize the previous process in hardware, we have to rename the variables tmp0, tmp1 and tmp2 and change the process to

```
process(a,b)
  variable tmp0, tmp1, tmp2: std_logic;
begin
   tmp0 := '0';
   tmp1 := tmp0 or a;
   tmp2 := tmp1 or b;
   y <= tmp2;
end process;
```

For synthesis purposes, we can now interpret the variables as signals or nets. The corresponding diagram is shown in Figure 5.2. Because of the lack of clear hardware mapping, we should try to use signals in code in general and resort to variables only for the characteristics that cannot be described by signals.

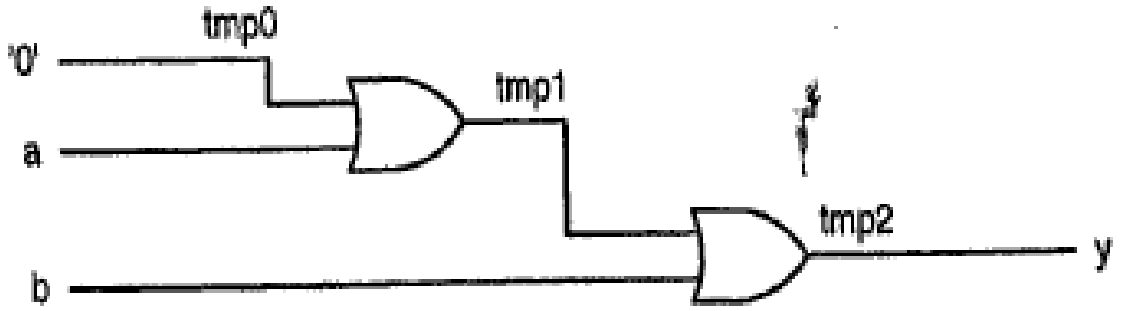


Figure 5.2 Conceptual implementation of simple variable assignments.

For comparison purposes, let us repeat the previous segment by replacing the variables with signals:

```
signal a, b, y, tmp: std_logic;
. . .
process(a,b,tmp)
begin
    tmp <= '0';
    tmp <= tmp or a;
    tmp <= tmp or b;
    y <= tmp;
end process;
```

Note that the signals have to be "global" and declared outside the process, and the tmp signal has to be included in the sensitivity list. This code is the same as

```
process(a,b,tmp)
begin
    tmp <= tmp or b;
    y <= tmp;
end process;
```

Because a signal in a process takes the last value assigned to it.
See p. 47 in Chu FPGA text.

This code implies a combinational loop with an or gate, as shown in Figure 5.3.

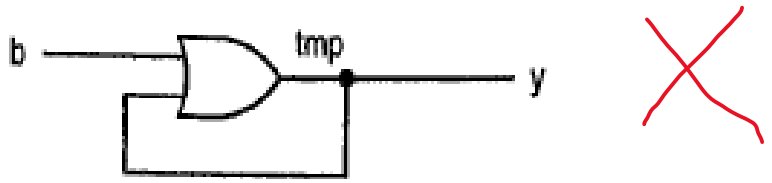


Figure 5.3 Conceptual implementation of erroneous signal assignments.

8.8 USE OF VARIABLES IN SEQUENTIAL CIRCUIT DESCRIPTION

We have learned how to infer an FF or a register from a signal. It is done by using the `clk'event and clk='1'` condition to indicate the rising edge of the clock signal. Any signal assigned under this condition is required to keep its previous value, and thus an FF or a register is inferred accordingly. Assigned is left hand side of <=

A variable can also be assigned under the `clk'event and clk='1'` condition, but its implication is different because a variable is local to the process and its value is not needed outside the process. If a variable is *assigned a value before it is used*, it will get the value every time when the process is invoked and there is no need to keep its previous value. Thus, no memory element is inferred. On the other hand, if a variable is *used before it is assigned a value*, it will use the value from the previous process execution. The variable has to memorize the value between the process invocations, and thus an FF or a register will be inferred.

Since using a variable to infer memory is more error-prone, we generally prefer to use a signal for this task. The major use of variables is to obtain an intermediate value inside the `clk'event and clk='1'` branch without introducing an unintended register. This can best be explained by an example. Let us consider a simple circuit that performs an operation `a and b` and stores the result into an FF at the rising edge of the clock. We use three outputs to illustrate the effect of different coding attempts. The VHDL code is shown in Listing 8.24.

Listing 8.24 Using a variable to infer an FF

```
library ieee;
use ieee.std_logic_1164.all;
entity varaible_ff_demo is
    port(
        a,b,clk: in std_logic;
        q1,q2,q3: out std_logic
    );
end varaible_ff_demo;

architecture arch of varaible_ff_demo is
    signal tmp_sig1: std_logic;
begin

    -- attempt 1
    process(clk)
    begin
        if (clk'event and clk='1') then
            tmp_sig1 <= a and b;
            q1 <= tmp_sig1;
        end if;
    end process;
```

```
    -- attempt 2
    process(clk)
        variable   tmp_var2: std_logic;
    begin
        if (clk'event and clk='1') then
            tmp_var2 := a and b; 1
            q2 <= tmp_var2;      2
        end if;
    end process;
    -- attempt 3
    process(clk)
        variable   tmp_var3: std_logic;
    begin
        if (clk'event and clk='1') then
            q3 <= tmp_var3;      2
            tmp_var3 := a and b; 1
        end if;
    end process;
end arch;
```

In the first attempt, we try to use the `tmp_sig1` signal for the temporary result. However, since the `tmp_sig1` signal is inside the `clk'event and clk='1'` branch, an unintended D FF is inferred. The two statements

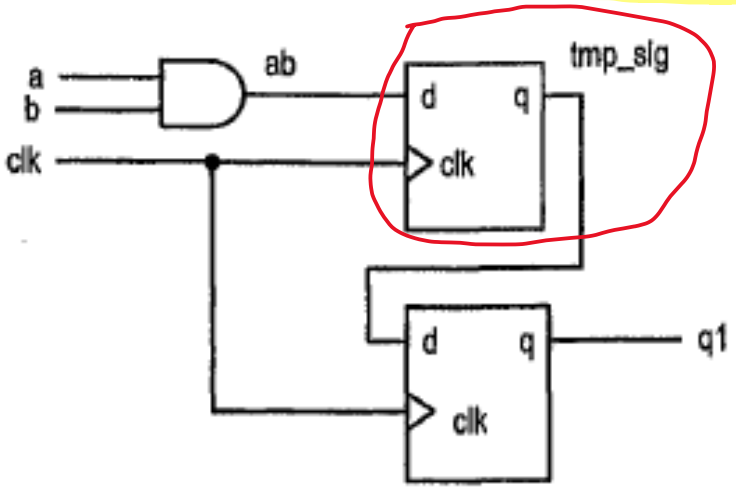
```
tmp_sig1 <= a and b;
q1 <= tmp_sig1;
```

tmp_sig1 is assigned a value (left hand side of <=)

are interpreted as follows. At the rising edge of the `clk` signal, the value of `a and b` will be sampled and stored into an FF named `tmp_sig1`, and the old value (not current value of `a and b`) from the `tmp_sig1` signal will be stored into an FF named `q1`. The diagram is shown in Figure 8.19(a).

The value of `a and b` is delayed by the unintended buffer, and thus this description fails to meet the specification. Since both statements are signal assignment statements, we will obtain the same result if we switch the order of the two statements.

concurrent



(a) Block diagram of first attempt

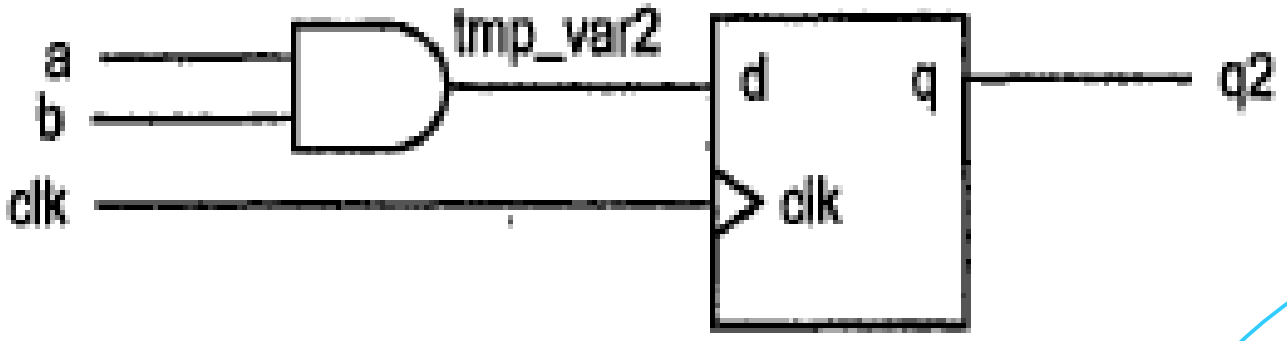
The second attempt uses a variable, `tmp_var2`, for the temporary result and the statements become

```
tmp_var2 := a and b;
q2 <= tmp_var2;
```

Assigned first

Used second

Note that the tmp_var2 variable is first assigned a value and then used in the next statement. Thus, no memory element is inferred and the circuit meets the specification. The diagram is shown in Figure 8.19(b).



(b) Block diagram of second attempt

The third attempt uses a variable, `tmp_var3`, for the temporary result. It is similar to the second process except that the order of the two statements is reversed:

```
q3 <= tmp_var3;
tmp_var3 := a and b;
```

Used first

Assigned second

In this code, the `tmp_var3` variable is first used before it is assigned a value. According to the VHDL definition, the value of `tmp_var3` from the previous process invocation will be used. An FF will be inferred to store the previous value. Thus, the circuit described by the third attempt is the same as that of the first attempt, which contains an unwanted buffer.

8.10 SYNTHESIS GUIDELINES

- Strictly follow the synchronous design methodology; i.e., all registers in a system should be synchronized by a common global clock signal.
- Isolate the memory components from the VHDL description and code them in a separate segment. One-segment coding style is not advisable.
- The memory components should be coded clearly so that a predesigned cell can be inferred from the device library.
- Avoid synthesizing a memory component from scratch. RAM, ROM, FIFO, etc.
- Asynchronous reset, if used, should be only for system initialization. It should not be used to clear the registers during regular operation. Timing problems.
- Unless there is a compelling reason, a variable should not be used to infer a memory component.

```
1   ENTITY fig7_41 IS
2   PORT (
3             clock   :IN BIT;
4             q       :OUT BIT_VECTOR(2 DOWNTO 0)
5        );
6   END fig7_41 ;
7
8   ARCHITECTURE a OF fig7_41     IS
9   BEGIN
10     PROCESS (clock)                                    -- respond to clk input
11     VARIABLE count: BIT_VECTOR(2 DOWNTO 0);    -- create a 3-bit register
12     BEGIN
13        IF (clock = '1' AND clock'EVENT) THEN    -- rising edge trigger
14           CASE count IS
15  --          Present                Next
16  ------------------------------------------------------------------
17              WHEN "000"     =>     count := "001";
18              WHEN "001"     =>     count := "010";
19              WHEN "010"     =>     count := "011";
20              WHEN "011"     =>     count := "100";
21              WHEN "100"     =>     count := "000";
22              WHEN OTHERS    =>     count := "000";
23           END CASE;
24        END IF;
25        q <= count;                -- assign register to output pins
26     END PROCESS;
27  END a;
```

Not a good reason to use a variable since used it to make a FF. Should use a signal for the FF.

variable is used before assigned a value so it makes a FF

variable is local to process, so do this INSIDE process

FIGURE 7-41 VHDL MOD-5 counter.

4.2.2 Register

A register is a collection of D FFs that are controlled by the same clock and reset signals. Like a D FF, a register can have an optional asynchronous reset signal and a synchronous enable signal. The code is identical to that of a D FF except that the array data type, std_logic_vector, is needed for the relevant input and output signals. For example, an 8-bit register with asynchronous reset is shown in Listing 4.5.

Listing 4.5 Register

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_reset is
   port(
      clk, reset: in std_logic;
      d: in std_logic_vector(7 downto 0);
      q: out std_logic_vector(7 downto 0)
   );
end reg_reset;
architecture arch of reg_reset is
begin
   process(clk,reset)
   begin
      if (reset='1') then
         q <=(others=>'0');
      elsif (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end arch;
```

(uses =>)

The (**others => '0'**) means "00000…",
a variable number of zeroes, which is needed when the signal is defined using a generic number of bits.

4.2.3 Register file

A register file is a collection of registers with one input port and one or more output ports. The write address signal, w_addr, specifies where to store data, and the read address signal, r_addr, specifies where to retrieve data. The register file is generally used as fast, temporary storage. The code for a parameterized 2^W-by-B register file is shown in Listing 4.6. Two generics are defined in this design. The W generic specifies the number of address bits, which implies that there are 2^W words in the file, and the B generic specifies the number of bits in a word.

Similar to static RAM, but no read pin.

Listing 4.6 Parameterized register file

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity reg_file is
    generic(
        B: integer:=8; -- number of bits
        W: integer:=2  -- number of address bits
    );
    port(
        clk, reset: in std_logic;
        wr_en: in std_logic;
        w_addr, r_addr: in std_logic_vector (W-1 downto 0);
        w_data: in std_logic_vector (B-1 downto 0);
        r_data: out std_logic_vector (B-1 downto 0)
    );
end reg_file;
```

Need numeric for conversions between std_logic pins and integers.

Don't need read pin: data always on output

```
architecture arch of reg_file is
    type reg_file_type is array (2**W-1 downto 0) of
         std_logic_vector(B-1 downto 0);
    signal array_reg: reg_file_type;
begin
    process(clk,reset)
    begin
        if (reset='1') then
            array_reg <= (others=>(others=>'0'));
        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;
```

$2^W - 1$ 4-byte memory

array (element #)

StdLogicVector integer

Integers are used for array element #

4 elements 8 bits

StdLogic pin

ALWAYS output the data from the read address (no read pin)

array (element #)

StdLogic pin

StdLogicVector integer

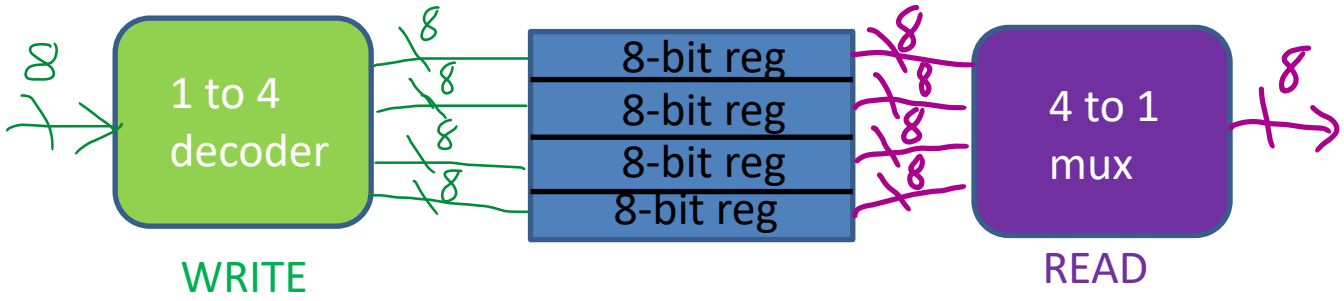
The code includes several new features. First, since no built-in two-dimensional array is defined in the `std_logic_1164` package a user-defined array-of-array data type, `reg_file_type`, is introduced. It is first defined by a type statement and is then used by the `array_reg` signal. Second, a signal is used with 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.

write

read

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)));
```



The (others => '0') on the previous slide means "00000…", a variable number of zeroes, which is needed when the signal is defined using a generic number of bits.

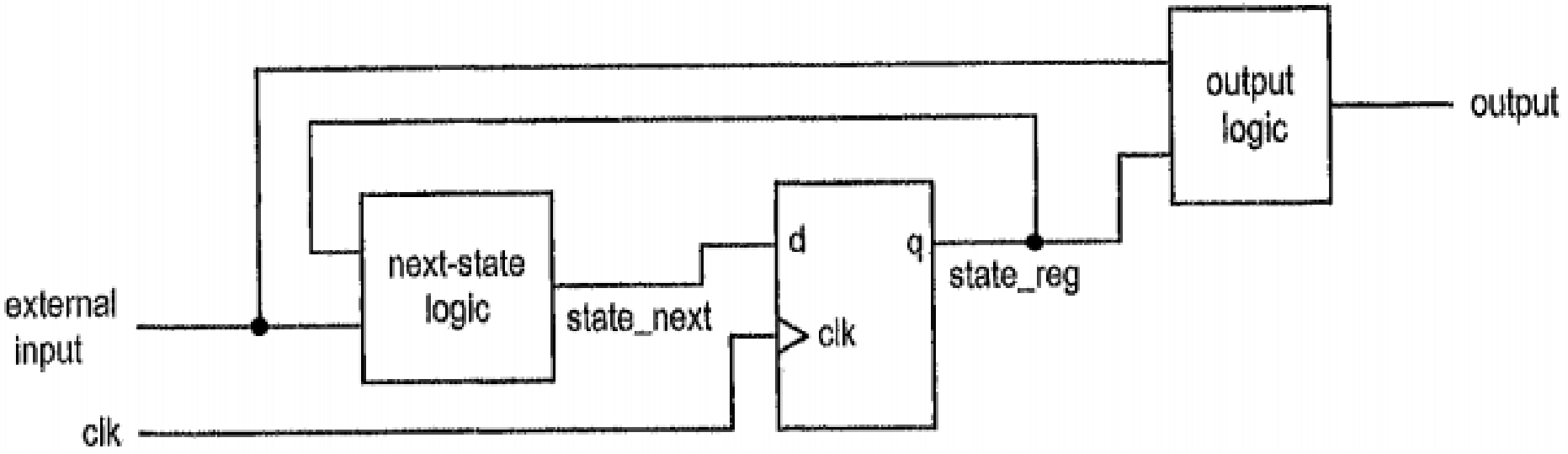


Figure 4.2 Block diagram of a synchronous system.

4.1.2 Synchronous system

Block diagram The block diagram of a synchronous system is shown in Figure 4.2. It consists of the following parts:

- *State register*: a collection of D FFs controlled by the same clock signal
- *Next-state logic*: combinational logic that uses the external input and internal state (i.e., the output of register) to determine the new value of the register
- *Output logic*: combinational logic that generates the output signal

4.1.3 Code development

Our code development follows the basic block diagram in Figure 4.2. The key is to separate the memory component (i.e., the register) from the system. Once the register is isolated, the remaining portion is a pure combinational circuit, and the coding and analysis schemes discussed in previous chapters can be applied accordingly. While this approach may make the code a little bit more cumbersome at times, it helps us to better visualize the circuit architecture and avoid unintended memory and subtle mistakes.

Based on the characteristics of the next-state logic, we divide sequential circuits into three categories:

- *Regular sequential circuit*. The state transitions in the circuit exhibit a "regular" pattern, as in a counter or shift register. The next-state logic is constructed primarily by a predesigned, "regular" component, such as an incrementor or shifter.
- *FSM*. The state transitions in the circuit do not exhibit a simple, repetitive pattern. The next-state logic is constructed by "random logic" and synthesized from scratch. It should be called a random sequential circuit, but is commonly known as an FSM (*finite state machine*).
- *FSMD*. The circuit consists of a regular sequential circuit and an FSM. The two parts are known as a *data path* and a *control path*, and the complete circuit is known as an FSMD (*FSM with data path*). This type of circuit is used to implement an algorithm represented by *register-transfer* (RT) methodology, which describes system operation by a sequence of data transfers and manipulations among registers.

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;
```

See earlier slide on section 4.1.2

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.

Regular Sequential Circuit

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;
```

Carry Out when counter rolls over

```
architecture arch of free_run_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 <= r_reg + 1;
    -- output logic
    q <= std_logic_vector(r_reg);
    max_tick <= '1' when r_reg=(2**N-1) else '0';
end arch;
```

See earlier slide on section 4.1.2

Use numeric (unsigned) for math

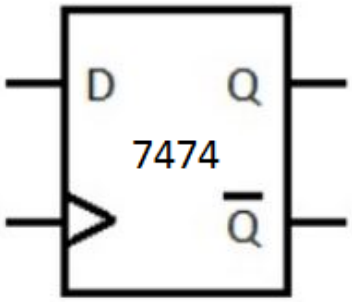
The next-state logic is an incrementor, which adds 1 to the register's current value. By definition of the + operator in the IEEE `numeric_std` package, the operation implicitly wraps around after the `r_reg` reaches "1...1". The circuit also consists of an output status signal, `max_tick`, which is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

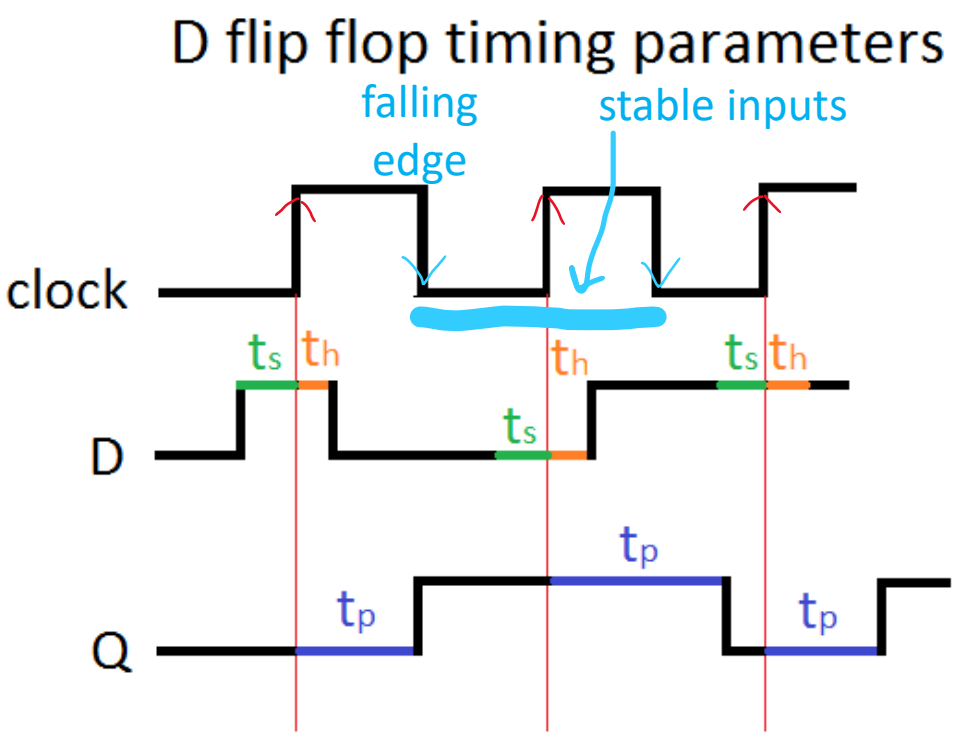
The `max_tick` signal represents a special type of signal that is asserted for a single clock cycle. In this book, we call this type of signal a *tick* and use the suffix `_tick` to indicate a signal with this property. It is commonly used to interface with the enable signal of other sequential circuits.

Most texts call this tick signal the "carry out" when it is for a counter. It is called a tick because is only lasts for one clock cycle, which allows it to be used to trigger the start of a "downstream" component.

MODELSIM TEST BENCH for Sequential Circuits

- The test bench for a sequential circuit is more complicated than one for a combinatorial circuit.
- A sequential circuit has clock-edge flip flops, so the test bench uses a continuous clock signal.
- The flips flops have timing constraints that must be satisfied, such as setup and hold times.
- If the flip flops use the rising edge of the clock, the inputs to the flip flops should be changed on the falling edge to allow plenty of setup time.
- FSM inputs should also be changed on the falling edge for the same reason.





t_p - propagation delay = 25 ns (LH) or 40 ns (HL) for 7474

t_s - setup time = 20 ns for 7474

t_h - hold time = 5 ns for 7474

This counter will be used in the ModelSim sequential testbench.

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

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

The design of this counter is usually a lab assignment.

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);
```

Uses the actual counter as a **component**. These are simulation signals that get connected to the component.

Creating a 3-bit counter

Keyword **time** is a data type (for simulations)

```
begin
    ---*************************
    --- instantiation of the counter component
    ---*************************
    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;
```

Not a real clock, this is simulation!
Can't use "wait for" for **synthesis.**
Test benches are compiled and
run in Model Sim.
20 ns --- 50 MHz
(same as Altera DE2 real clock)

```
-----------------------------
--- reset  Active high
-----------------------------
--- 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);
```

Counter component uses **rising** edge clock, so **change** input signals on **falling** edge so counter FFs have plenty of setup time.

doesn't need to be enabled

```
--*************************
-- 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;
```

www.NandLand.com

Wait UNTIL – condition is true

After these wait statements, make sure input does not change on rising edge of clock.

Wait ON – signal changes

Wait FOR - time

Use these!

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

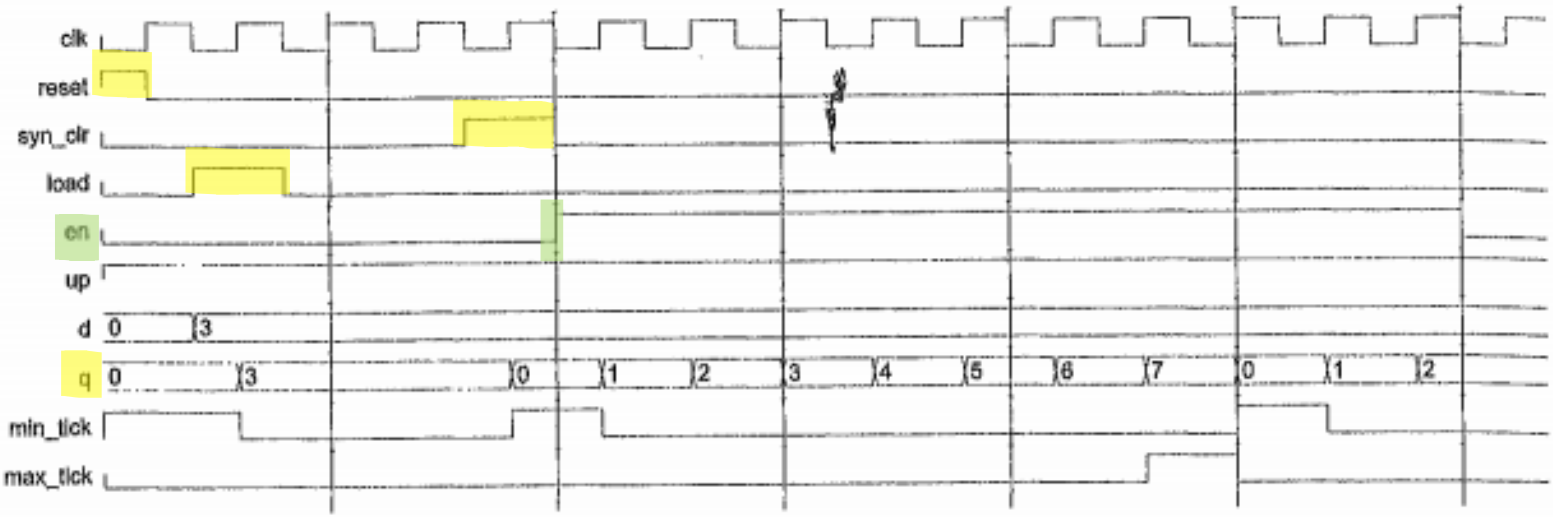


Figure 4.4 Testbench waveform.

```
wait on min_tick;
```

or wait for an absolute time, such as

```
wait for 4*T;  -- wait for 4 clock periods
```

If an input signal is modified after these statements, we need to make sure that the input change does not occur at the rising edge of the clock. An additional

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

statement should be added when needed.

We can compile the code and perform simulation. Part of the simulated waveform is shown in Figure 4.4.

8.7 ALTERNATIVE ONE-SEGMENT CODING STYLE

So far, all VHDL coding follows the basic block diagram of Figure 8.5 and separates the memory elements from the rest of the logic. Alternatively, we can describe the memory elements and the next-state logic in a single process segment. For a simple circuit, this style appears to be more compact. However, it becomes involved and error-prone for more complex circuits. In this section, we use some earlier examples to illustrate the one-segment VHDL description and the problems associated with this style.

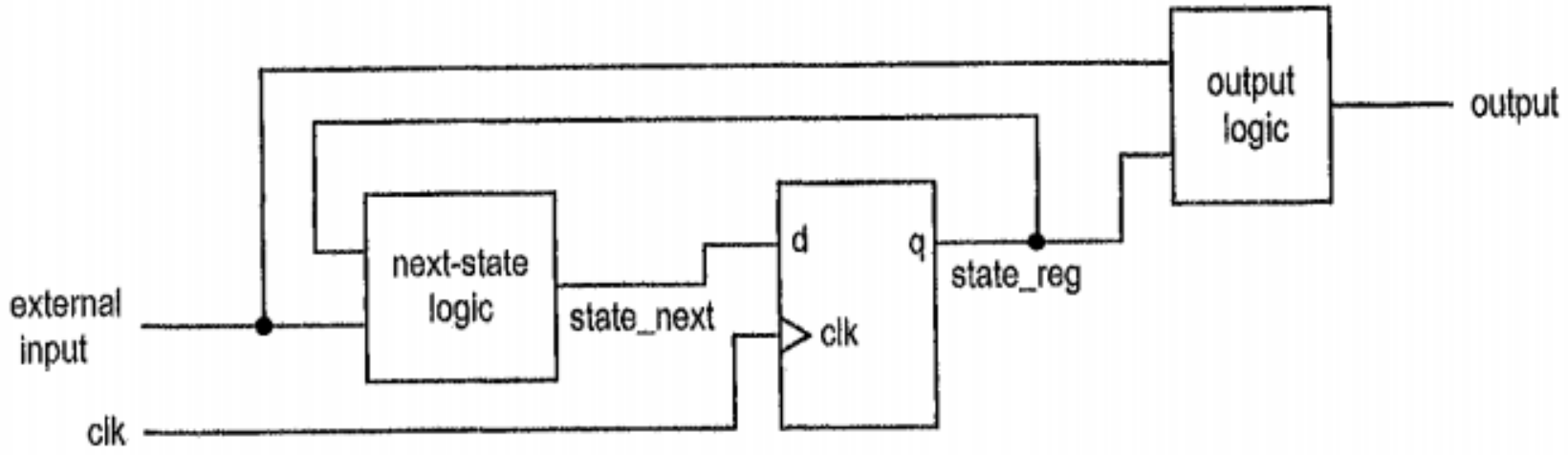


Figure 8.5 Block diagram of a synchronous system.

Free-running binary counter Consider the 4-bit free-running binary counter in Listing 8.12. The first attempt to convert it to a single-segment style is shown in Listing 8.20.

Listing 8.20 Incorrect one-segment description of a free-running binary counter

```
architecture not_work_one_seg_glitch_arch
                          of binary_counter4_pulse is
   signal r_reg: unsigned(3 downto 0);
begin
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_reg + 1;
         if r_reg="1111" then
            max_pulse <= '1';
         else
            max_pulse <= '0';
         end if;
      end if;
   end process;
   q <= std_logic_vector(r_reg);
end not_work_one_seg_glitch_arch;
```

The output logic does not function as we expected. Because the statement

```
if r_reg="1111" then
   max_pulse <= '1';
else
   max_pulse <= '0';
end if;
```

is inside the `clk'event and clk='1'` branch, a 1-bit register is inferred for the `max_pulse` signal. The register works as a buffer and delays the output by one clock cycle, and thus the `max_pulse` signal will be asserted when `r_reg="0000"`. The block diagram of this code is shown in *Figure 8.18*.

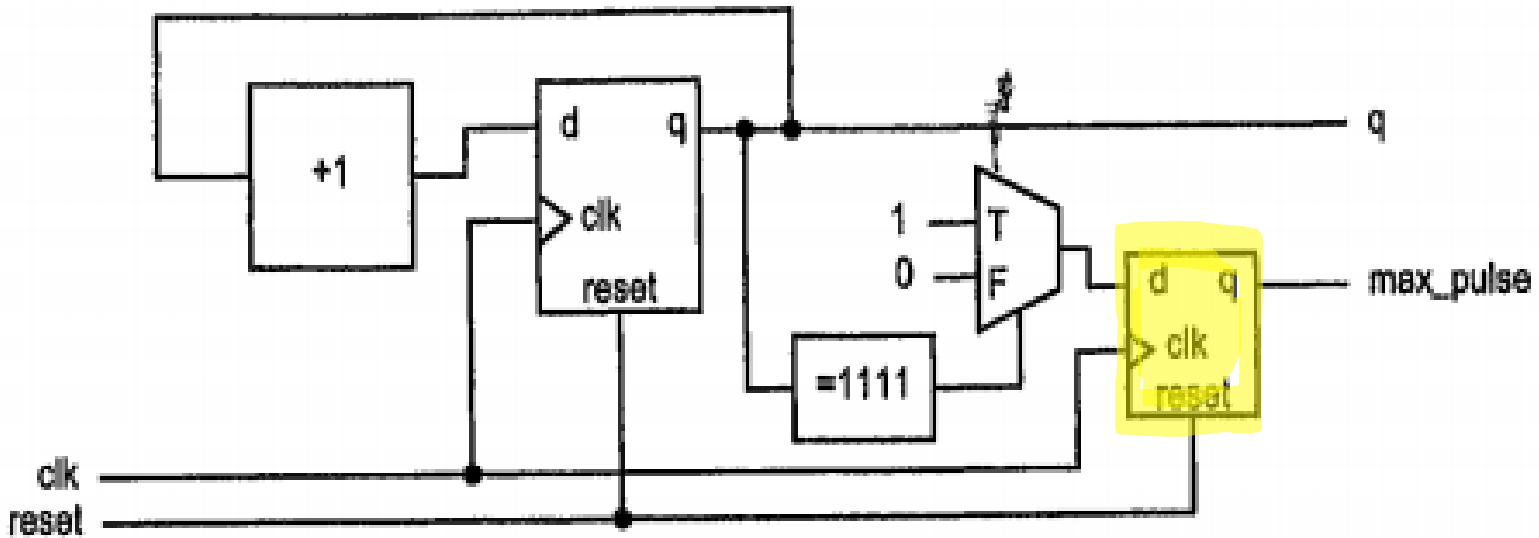


Figure 8.18 Free-running binary counter with an unintended output buffer.

8.7.2 Summary

Reasons not to use the one-segment style.

When we combine the memory elements and next-state logic in the same process, it is much harder to "visualize" the circuit and to map the VHDL statements into hardware components. This style may make code more compact for a few simple circuits, as in the first three examples. However, when a slightly more involved feature is needed, as the `max_pulse` output or the incrementor sharing of the last two examples, the one-segment style makes the code difficult to understand and error-prone. Although we can correct the problems, the resulting code contains extra statements and is far worse than the codes in Section 8.5. Furthermore, since the combinational logic and memory elements are mixed in the same process, it is more difficult to perform optimization and to fine-tune the combinational circuit. In summary, although the two-segment code may occasionally appear cumbersome, its benefits far outweigh the inconvenience, and we generally use this style in this book.

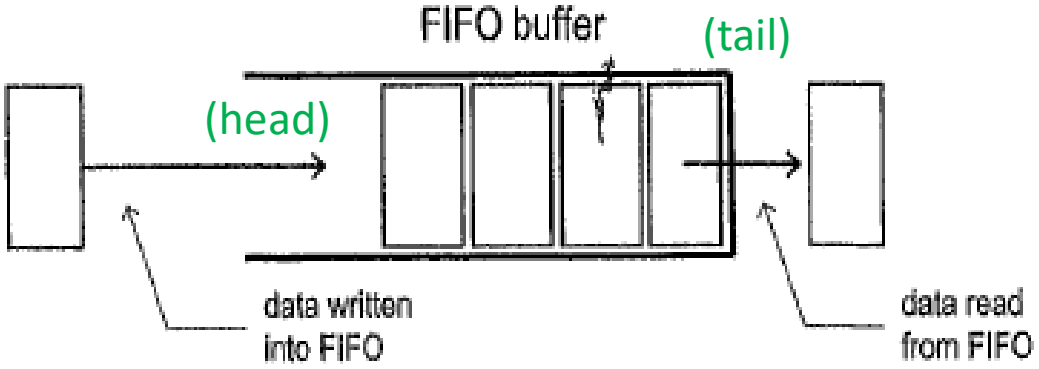


Figure 4.10 Conceptual diagram of a FIFO buffer.

(pronounced with a long I, as in "file")

typo
tail

e.g., two subsystems on different clock frequencies

4.5.3 FIFO buffer

A FIFO (first-in-first-out) buffer is an "elastic" storage between two subsystems, as shown in the conceptual diagram of Figure 4.10. It has two control signals, `wr` and `rd`, for write and read operations. When `wr` is asserted, the input data is written into the buffer. The read operation is somewhat misleading. The head of the FIFO buffer is normally always available and thus can be read at any time. The `rd` signal actually acts like a "remove" signal. When it is asserted, the first item (i.e., head) of the FIFO buffer is removed and the next item becomes available.

FIFO buffer is a critical component in many applications and the optimized implementation can be quite complex. In this subsection, we introduce a simple, genuine circular-queue-based design. More efficient, device-specific implementation can be found in the Xilinx literature.

and in the Altera library of parameterized modules

Circular-queue-based implementation One way to implement a FIFO buffer is to add a control circuit to a register file. The registers in the register file are arranged as a circular queue with two pointers. The *write pointer* points to the head of the queue, and the *read pointer* points to the tail of the queue. The pointer advances one position for each write or read operation. The operation of an eight-word circular queue is shown in Figure 4.11.

A FIFO buffer usually contains two status signals, `full` and `empty`, to indicate that the FIFO is full (i.e., cannot be written) and empty (i.e., cannot be read), respectively. One of the two conditions occurs when the read pointer is equal to the write pointer, as shown in Figure 4.11(a), (f), and (i). The most difficult design task of the controller is to derive a mechanism to distinguish the two conditions. One scheme is to use two FFs to keep track of the empty and full statuses. The FFs are set to '1' and '0' during system initialization and then modified in each clock cycle according to the values of the `wr` and `rd` signals. The code is shown in Listing 4.20.

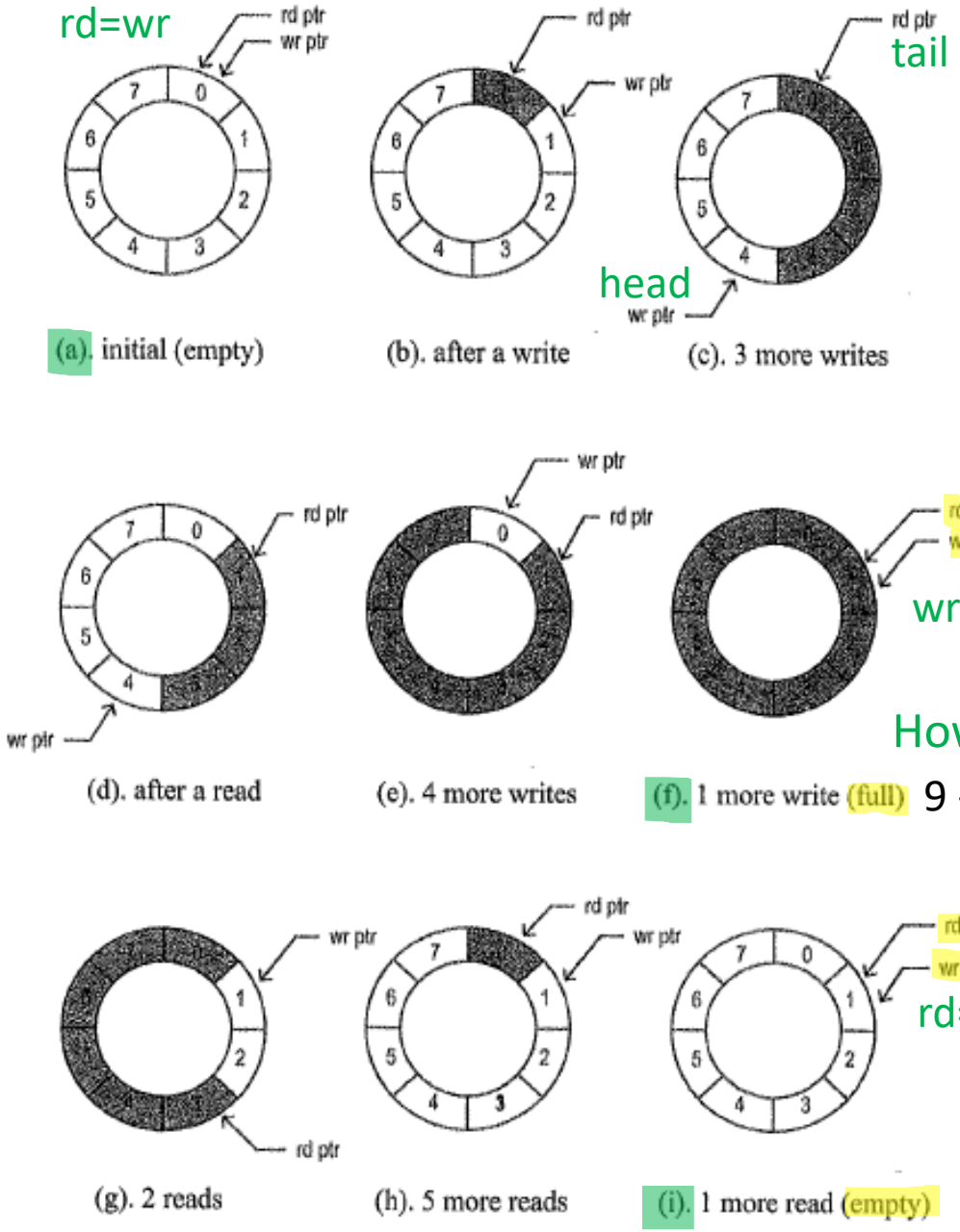


Figure 4.11 FIFO buffer based on a circular queue.

FIFO controller has 2-bit control SIGNAL wr_op (concatenated from RD and WR pins)

"00" No Op

"01" READ

"10" WRITE

"11" READ and WRITE

The code is divided into a register file and a FIFO controller. The controller consists of two pointers and two status FFs. Its next-state logic examines the wr and rd signals and takes actions accordingly. For example, let us consider the "10" case, which implies that only a write operation occurs. The status FF is checked first to ensure that the buffer is not full. If this condition is met, we advance the write pointer by one position and clear the empty status FF. Storing one extra word to the buffer may make it full. This happens if the new write pointer "catches" the read pointer, which is expressed by the `w_ptr_succ=r_ptr_reg` expression.

Listing 4.20 FIFO buffer

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo is
    generic(
        B: natural:=8; -- number of bits
        W: natural:=4 -- number of address bits
    );
    port(
        clk, reset: in std_logic;
        rd, wr: in std_logic;
        w_data: in std_logic_vector (B-1 downto 0);
        empty, full: out std_logic;
        r_data: out std_logic_vector (B-1 downto 0)
    );
end fifo;
```

Natural is same as Integer but without the zero

16-byte FIFO

No address pins are needed because the FIFO read and write addresses are manipulated internally using pointers (registers)

User type: 2-dimensional array

```
architecture arch of fifo is
    type reg_file_type is array (2**W-1 downto 0) of
         std_logic_vector(B-1 downto 0);
    signal array_reg: reg_file_type;
    signal w_ptr_reg, w_ptr_next, w_ptr_succ:
       std_logic_vector(W-1 downto 0);
    signal r_ptr_reg, r_ptr_next, r_ptr_succ:
       std_logic_vector(W-1 downto 0);
    signal full_reg, empty_reg, full_next, empty_next:
           std_logic;
    signal wr_op: std_logic_vector(1 downto 0);
    signal wr_en: std_logic;
begin
```

15

8

Internal address registers, i.e., pointers

successive

-- flags

```
-- register file
process(clk,reset)
begin
   if (reset='1') then
      array_reg <= (others=>(others=>'0'));
   elsif (clk'event and clk='1') then
      if wr_en='1' then
-- write port  array_reg(to_integer(unsigned(w_ptr_reg)))
-- if enabled          <= w_data;
      end if;
   end if;
end process;
-- read port (always)
r_data <= array_reg(to_integer(unsigned(r_ptr_reg)));
-- write enabled only when FIFO is not full
wr_en <= wr and (not full_reg);
```

(segment 1)

(segment 2)

Pointers keep track of where you are in the FIFO.

Memory (segment 1 cont'd)

```
-- fifo control logic
-- register for read and write pointers
process(clk,reset)
begin
    if (reset='1') then
        w_ptr_reg <= (others=>'0');
        r_ptr_reg <= (others=>'0');
        full_reg  <= '0';
        empty_reg <= '1';
    elsif (clk'event and clk='1') then
        w_ptr_reg <= w_ptr_next;
        r_ptr_reg <= r_ptr_next;
        full_reg  <= full_next;
        empty_reg <= empty_next;
    end if;
end process;
```

-- D FF flags

Variable number of zeros to match generic definition

Combinatorial logic (segment 2 cont'd)

```
-- successive pointer values "middle men"
w_ptr_succ <= std_logic_vector(unsigned(w_ptr_reg)+1);
r_ptr_succ <= std_logic_vector(unsigned(r_ptr_reg)+1);

-- next-state logic for read and write pointers
wr_op <= wr & rd;
process(w_ptr_reg,w_ptr_succ,r_ptr_reg,r_ptr_succ,wr_op,
        empty_reg,full_reg)
begin
   w_ptr_next <= w_ptr_reg;  -- default NS values (keep old values)
   r_ptr_next <= r_ptr_reg;  -- they can be overridden below
   full_next  <= full_reg;   --
   empty_next <= empty_reg;  --
   case wr_op is
      when "00" => -- no op   -- use default values
      when "01" => -- read
         if (empty_reg /= '1') then -- not empty
            r_ptr_next <= r_ptr_succ;
            full_next <= '0';  --( can't be full if just did a read)
            if (r_ptr_succ=w_ptr_reg) then
               empty_next <='1';
            end if;
         end if;
```

Sensitivity includes ALL inputs to combinatorial circuits, usually on right side of <= below

If RD will equal WR due to this read operation, then FIFO will be empty after this read.

```
            when "10" => -- write
                if (full_reg /= '1') then -- not full
                    w_ptr_next <= w_ptr_succ;
                    empty_next <= '0';
                    if (w_ptr_succ=r_ptr_reg) then
                        full_next <='1';
                    end if;
                end if;
            when others => -- write/read;
                w_ptr_next <= w_ptr_succ;
                r_ptr_next <= r_ptr_succ;
        end case;
    end process;
    -- output
    full <= full_reg;
    empty <= empty_reg;
end arch;
```

(can't be empty if just did a write)

If WR will equal RD due to this write operation, then FIFO will be full after this write.

11

Cancels out, so no change to full or empty

-- send flags out to pins

Verification circuit The verification circuit examines the operation of 4 -by-3 FIFO buffer. We use three switches to generate the input data and use two buttons for the `wr` and `rd` signals. The 3-bit readout and the `full` and `empty` status signals are displayed in five discrete LEDs. Because of bounces of the mechanical contact, a debouncing circuit is needed to generate a clean, one-clock-cycle tick. The debouncing module, named `debounce`, is discussed in Section 5.9 but for now can be treated as a predesigned module. The original button inputs are `btn(0)` and `btn(1)`, and the debounced signals are `db_btn(0)` and `db_btn(1)`. The code is shown in Listing 4.21.

Listing 4.21 Testing circuit for a FIFO buffer

```
library ieee;
use ieee.std_logic_1164.all;
entity fifo_test is
    port(
        clk, reset: in std_logic;
        btn: std_logic_vector(1 downto 0);
        sw: std_logic_vector(2 downto 0);
        led: out std_logic_vector(7 downto 0)
    );
end fifo_test;
```

will not be using 3 of these LEDs

Using top-level hierarchy to implement the generic FIFO as 4 word x 3 bit FIFO and wire up inputs to debounced buttons and outputs to LEDs

```
architecture arch of fifo_test is
    signal db_btn: std_logic_vector(1 downto 0);
begin
    -- debouncing circuit for btn(0)
    btn_db_unit0: entity work.debounce(fsmd_arch)
        port map(clk=>clk, reset=>reset, sw=>btn(0),
                 db_level=>open, db_tick=>db_btn(0));
    -- debouncing circuit for btn(1)
    btn_db_unit1: entity work.debounce(fsmd_arch)
        port map(clk=>clk, reset=>reset, sw=>btn(1),
                 db_level=>open, db_tick=>db_btn(1));
    -- instantiate a 2^2-by-3 fifo
    fifo_unit: entity work.fifo(arch)
        generic map(B=>3, W=>2)
        port map(clk=>clk, reset=>reset,
                 rd=>db_btn(0), wr=>db_btn(1),
                 w_data=>sw, r_data=>led(2 downto 0),
                 full=>led(7), empty=>led(6));
    -- disable unused leds
    led(5 downto 3)<=(others=>'0');
end arch;
```

9.1 POOR DESIGN PRACTICES AND THEIR REMEDIES

Synchronous design is the most important design methodology for developing a large, complex, reliable digital system. In the past, some poor, non-synchronous design practices were used. Those techniques failed to follow the synchronous principle and should be avoided in RT-level design. Before continuing with more examples, we examine those practices and their remedies. The most common problems are:

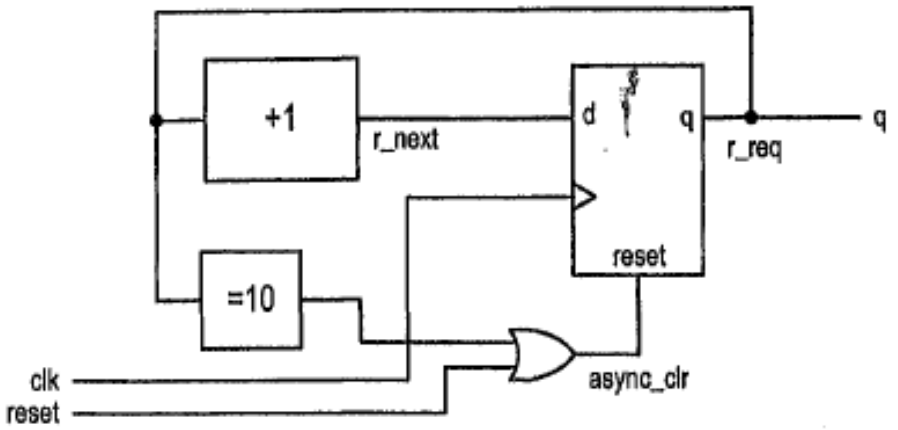
- Misuse of the asynchronous reset.
- Misuse of the gated clock.
- Misuse of the derived clock.

Some of those practices were used when a system was realized by SSI and MSI devices and the silicon real estate and printed circuit board were a premium. Designers tended to cut corners to save a few chips. These legacy practices are no longer applicable in today's design environment and should be avoided. The following subsections show how to remedy these poor non-synchronous design practices.

SSI – small scale integration
MSI – medium scale integration

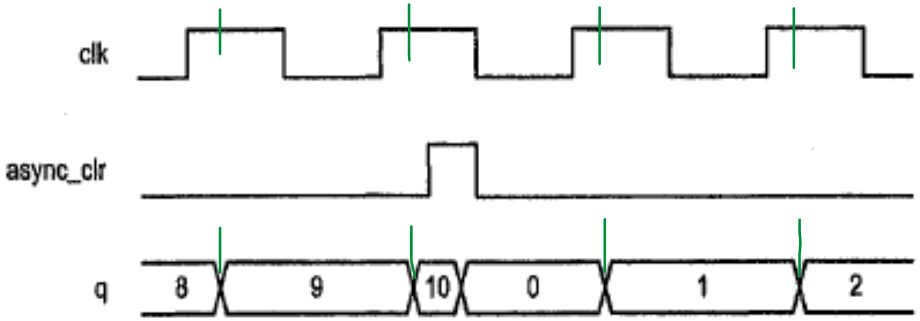
9.1.1 Misuse of asynchronous signals

In a synchronous design, we utilize only asynchronous reset or preset signals of FFs for system initialization. These signals should not be used in regular operation. A decade (mod-10) counter based on asynchronous reset is shown in Figure 9.1(a). The idea behind the design is to clear the counter to "0000" immediately after the counter reaches "1010". The timing diagram is shown in Figure 9.1(b). If we want, we can write VHDL code for this design, as in Listing 9.1.



(a) Block diagram

After propagation delay



(b) Timing diagram

Figure 9.1 Decade counter using asynchronous reset.

There are several problems with this design. First, the transition from state "1001" (9) to "0000" (0) is noisy, as shown in Figure 9.1(b). In that clock period, the counter first changes from "1001" (9) to "1010" (10) and then clears to "0000" (0) after the propagation delay of the comparator and reset. Second, this design is not very reliable. A combinational circuit is needed to generate the clear signal, and glitches may exist. Since the signal is connected to the asynchronous reset of the register, the register will be cleared to "0000" whenever a glitch occurs. Finally, because the asynchronous reset is used in normal operation, we cannot apply the timing analysis technique of Section 8.6. It is very difficult to determine the maximal operation clock rate for this design.

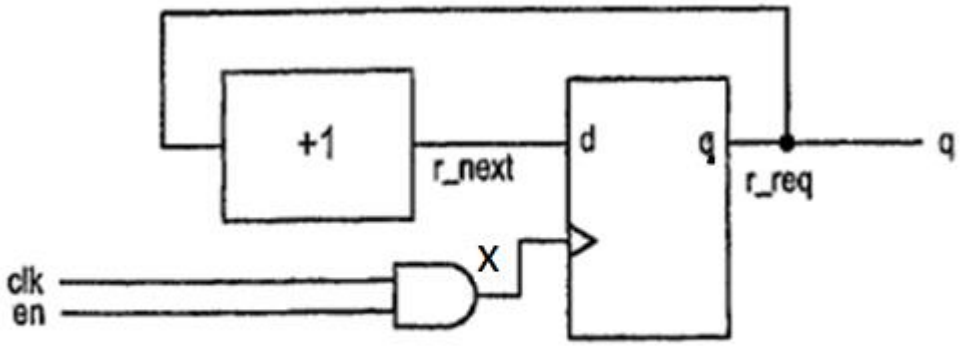
The remedy for this design is to load "0000" in a synchronous fashion. We can use a multiplexer to route "0000" or the incremented result to the input of the register. The code was discussed in Section 8.5.5 and is listed in Listing 9.2 for comparison. In terms of the circuit complexity, the synchronous design requires an additional 4-bit 2-to-1 multiplexer.

Chapter 6 Wiatrowski – Combinatorial circuits can have HAZARDS that generate glitches.

9.1.2 Misuse of gated clocks

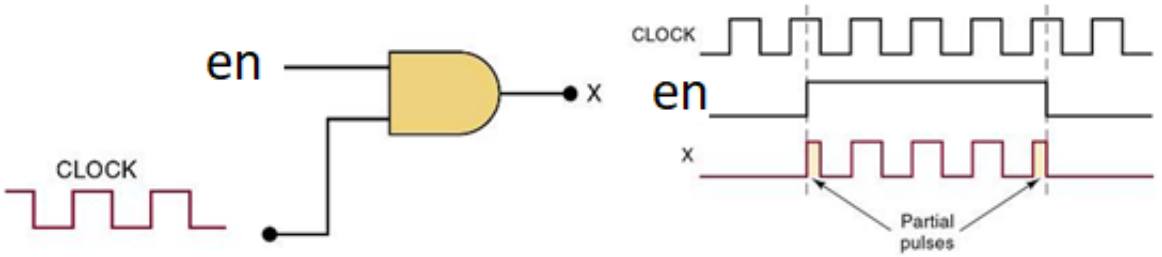
Correct operation of a synchronous circuit relies on an accurate clock signal. Since the clock signal needs to drive hundreds or even thousands of FFs, it uses a special distribution network and its treatment is very different from that of a regular signal. We should not manipulate the clock signal in RT-level design.

One bad RT-level design practice is to use a gated clock to suspend system operation, as shown in Figure 9.2. The intention of the design is to pause the counter operation by disabling the clock signal. The design suffers from several problems. First, since the enable signal, en, changes independent of the clock signal, the output pulse can be very narrow and cause the counter to malfunction.



Disabling FF with a gated clock.

The asynchronous signal en can produce partial pulses at *X*.



9.1.2 Misuse of gated clocks Cont'd

Second, if the en signal is not glitch-free, the glitches will be passed through the and cell and be treated as clock edges by the counter. Finally, since the and cell is included in the clock path, it may interfere with the construction and analysis of the clock distribution network.

Also, the and gate's (cell's) propagation delay creates clock skew.

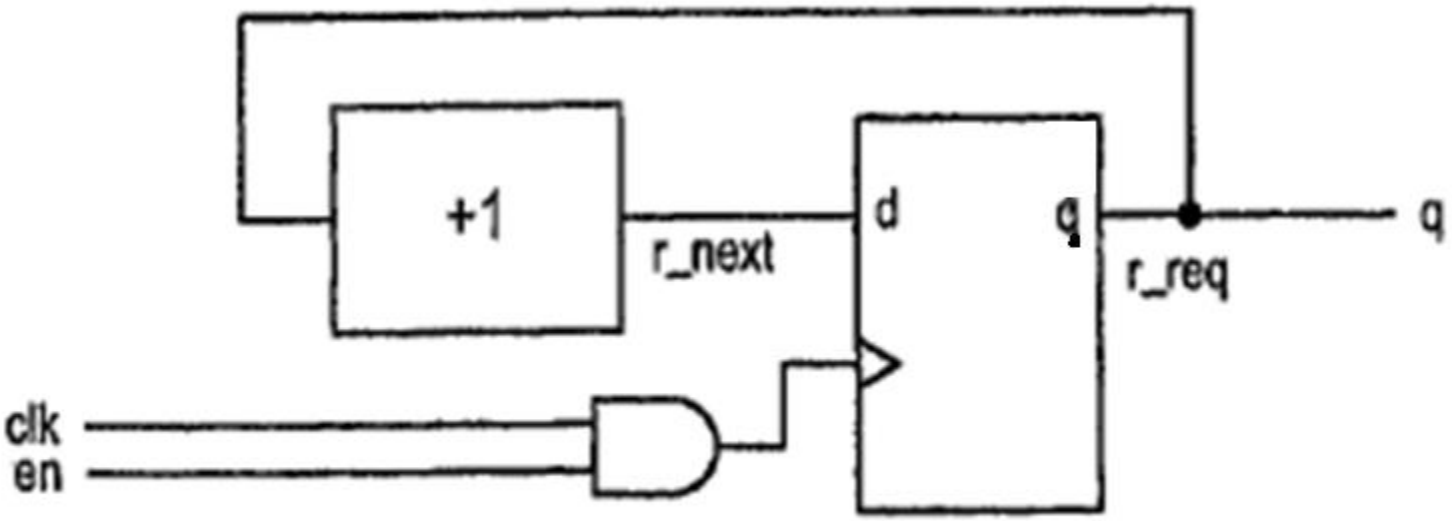


Figure 9.2 Disabling FF with a gated clock.

The remedy for this design is to use a synchronous enable signal for the register, as discussed in Section 8.5.1. We essentially route the register output as a possible input. If the en signal is low, the same value is sampled and stored back to the register and the counter appears to be "paused." The VHDL codes for the original and revised designs are shown in Listings 9.3 and 9.4. In terms of the circuit complexity, the synchronous design requires an additional 2-to-1 multiplexer.

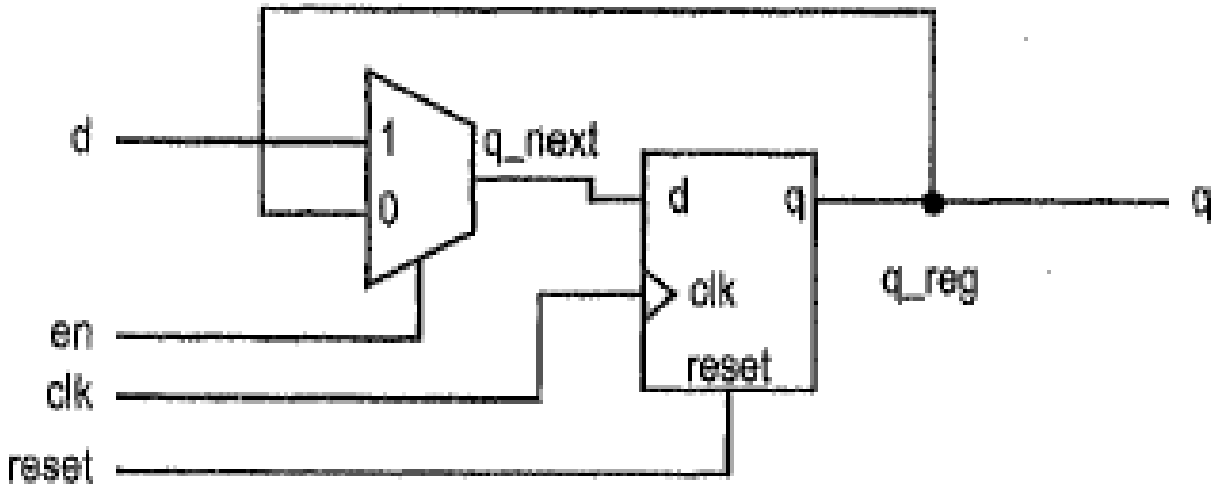


Figure 4.3 D FF with synchronous enable.

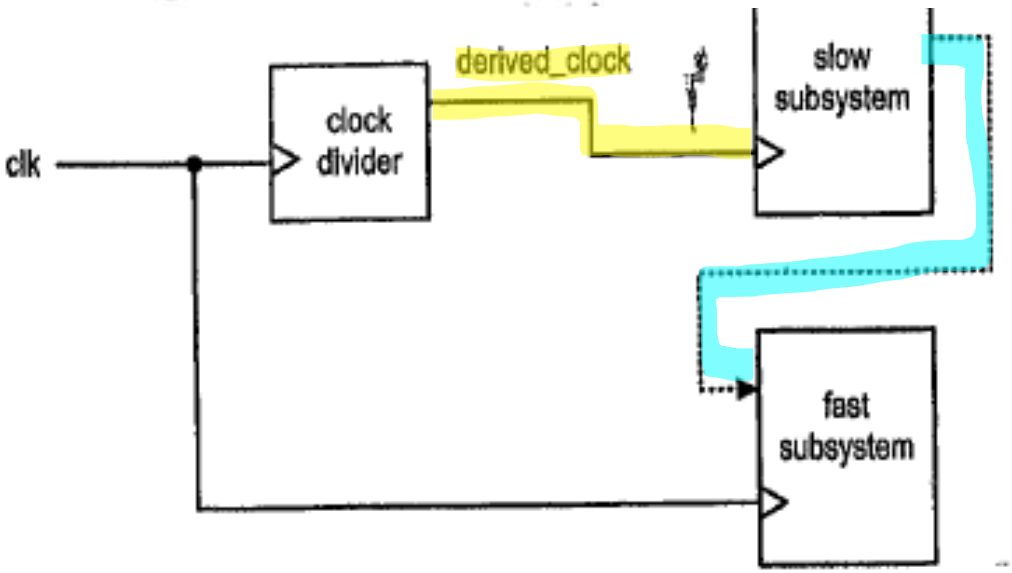
Power consumption is one important design criterion in today's digital system. A commonly used technique is to gate the clock to reduce the unnecessary transistor switching activities. However, this practice should not be done in RT-level code. The system should be developed and coded as a normal sequential circuit. After synthesis and verification, we can apply special power optimization software to replace the enable logic with a gated clock systematically.

9.1.3 Misuse of derived clocks

Clock Dividers

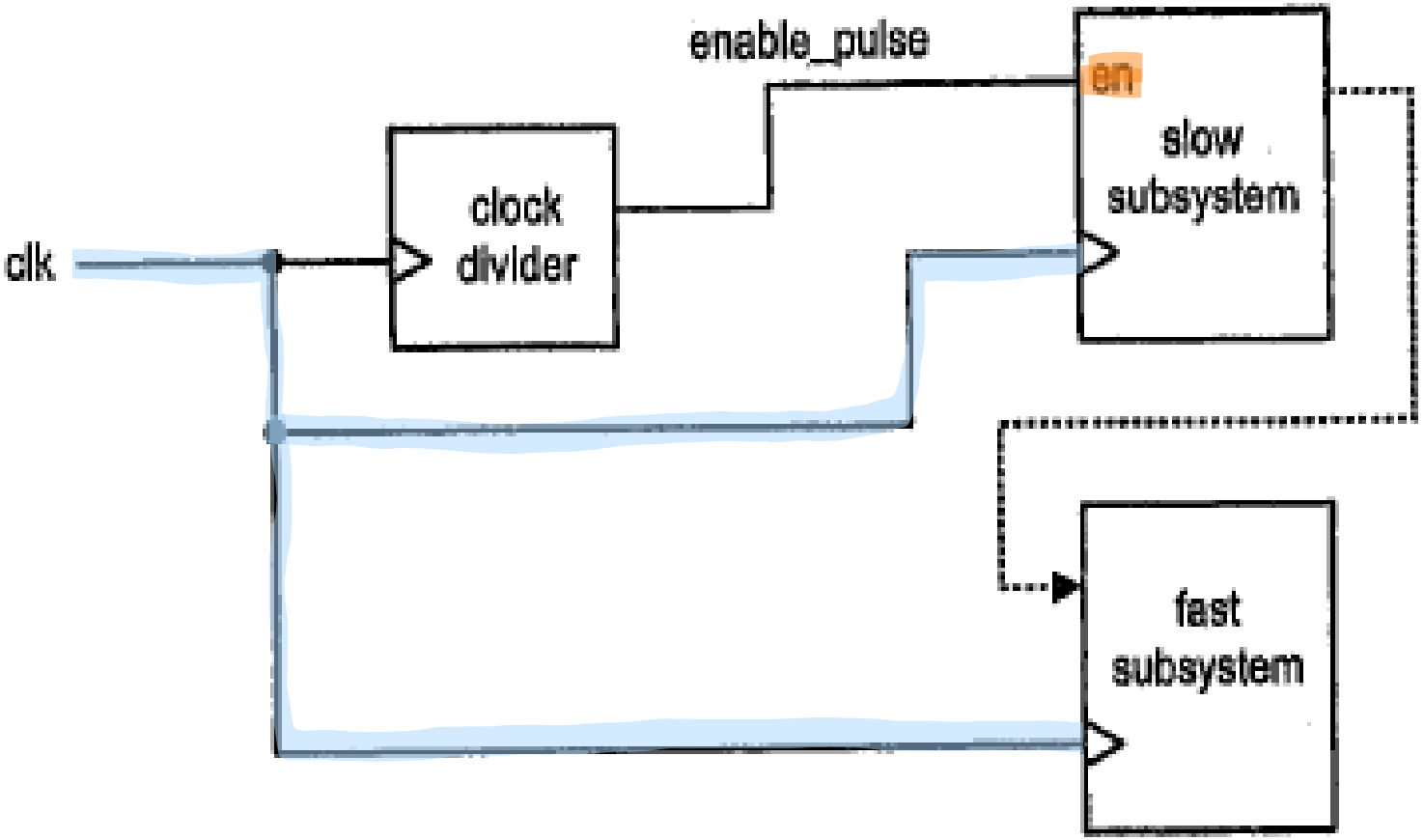
A large digital system may consist of subsystems that operate in different paces. For example, a system may contain a fast processor and a relatively slow I/O subsystem. One way to accommodate the slow operation is to use a clock divider (i.e., a counter) to derive a slow clock for the subsystem. The block diagram of this approach is shown in Figure 9.3(a). There are several problems with this approach. The most serious one is that the system is no longer synchronous. If the two subsystems interact, as shown by the dotted line in Figure 9.3(a), the timing analysis becomes very involved. The simple timing model of Section 8.6 can no longer be applied and we must consider two clocks that have different frequencies and phases. Another problem is the placement and routing of the multiple clock signals. Since a clock signal needs a special driver and distribution network, adding derivative clock signals makes this process more difficult.

Global low-skew clock lines



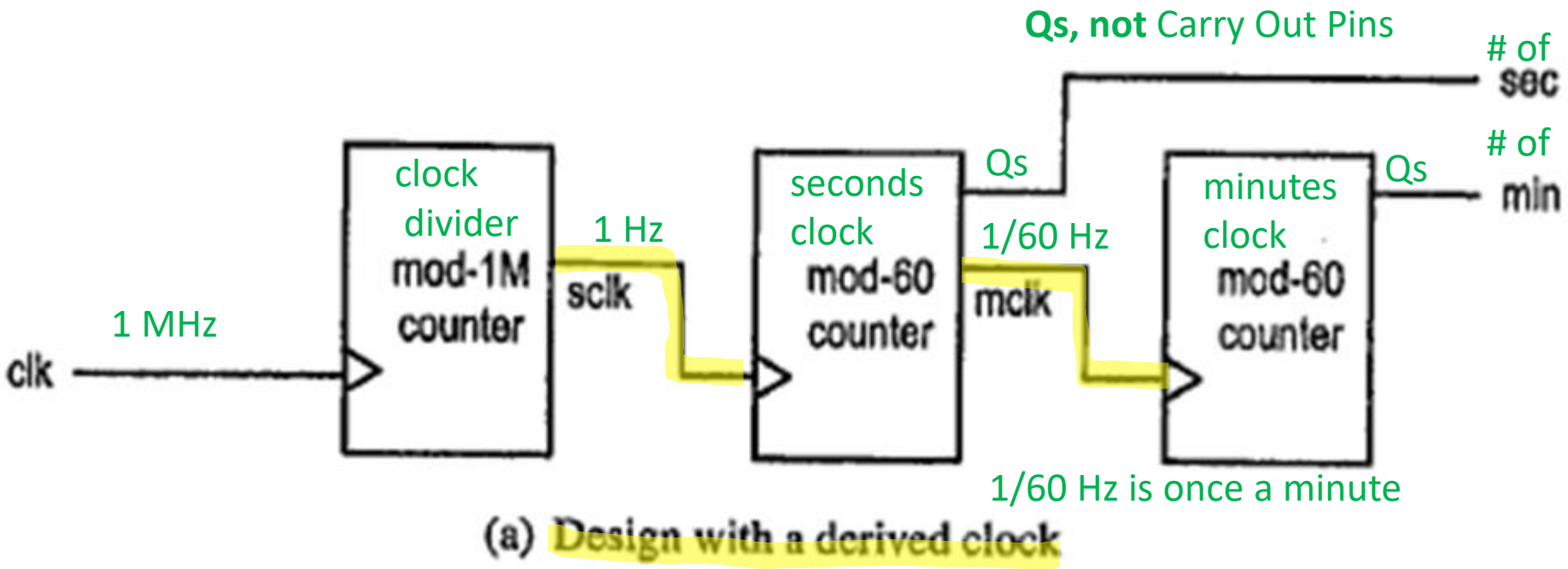
(a) System with a derived clock

A better alternative is to add a synchronous enable signal to the slow subsystem and drive the subsystem with the same clock signal.
Instead of generating a derivative clock signal, the clock divider generates a low-rate single-clock enable pulse. This scheme is shown in Figure 9.3(b).



(b) System with a single synchronous clock

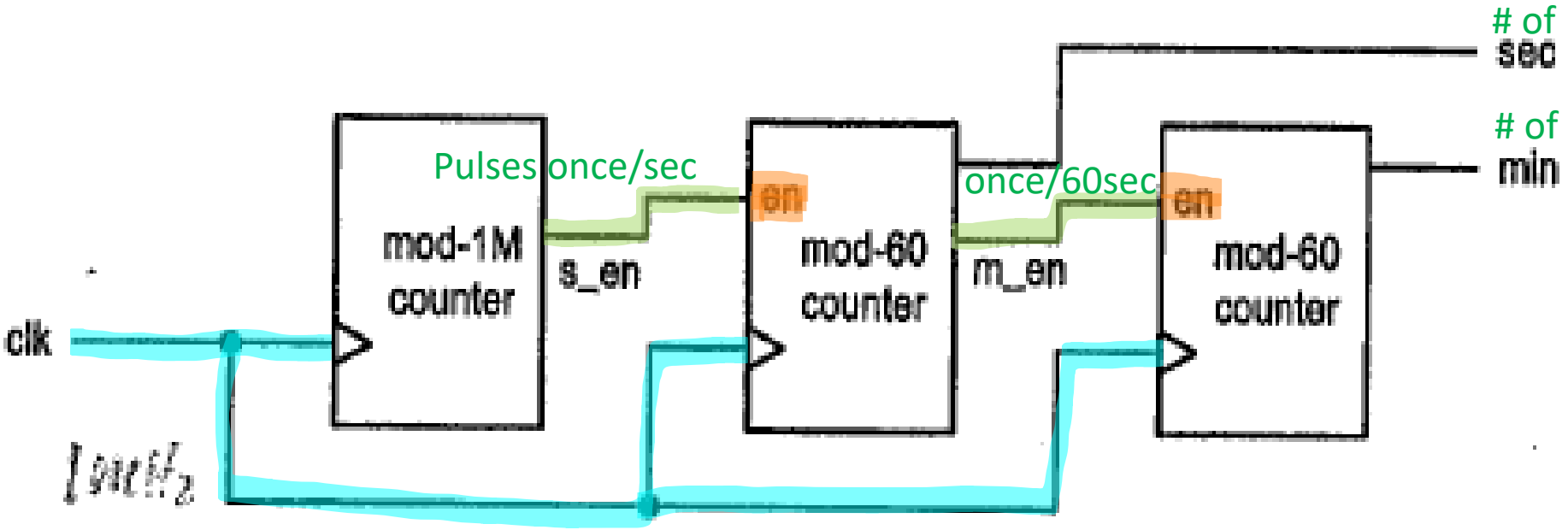
Let us consider a simple example. Assume that the system clock is 1 MHz and we want a timer that counts in minutes and seconds. The first design is shown in Figure 9.4(a). It first utilizes a mod-1000000 counter to generate a 1-Hz squared wave, which is used as a 1-Hz clock to drive the second counter. The second counter is a mod-60 counter, which in turn generates a $\frac{1}{60}$-Hz signal to drive the clock of the minute counter.



(a) Design with a derived clock

To convert the design to a synchronous circuit, we need to make two revisions. First, we add a synchronous enable signal for the mod-60 counter. The enable signal functions as the en signal discussed in examples in Section 8.5.1. When it is deasserted, the counter will pause and remain in the same state. Second, we have to replace the 50% duty cycle clock pulse with a one-clock-period enable pulse, which can be obtained by decoding a specific value of the counter. The revised diagram is shown in Figure 9.4(b), and the VHDL code is shown in Listing 9.6.

Using an AND gate with inverters on certain inputs.



(b) Design with a single synchronous clock