Counter Appendix

Overview of the basic elements of the synchronous sequential logic

In contrast to combinational logic (e.g. comparator and different adders discussed in the previous labs), the output of sequential logic depends not only on the current input values, but also on the previous input history. This means that sequential logic possesses an internal state that changes in accordance with the applied sequence of input values. The result is that the same set of input values can generate a different output since the internal state of the sequential logic element might have been different.

The sequential circuits can be of asynchronous or synchronous type. In asynchronous circuits the internal state starts changing immediately in response to the change of inputs. In synchronous circuits the internal state can change only during certain periods of times that are determined with the clock signal. Clock signal is a series of repeating pulses with fixed frequency. Synchronous circuits can be configured to change internal state at the rising edge (clock signal changes from LOW to HIGH) or falling edge (clock signal changes from HIGH to LOW). This manual focuses only on the synchronous type sequential circuits.

The basic element of synchronous sequential logic is a flip-flop. It has two states (LOW and HIGH) that can be changed by applying appropriate values to flip-flop's control inputs. A D-type flip-flop is shown in Figure 1 (as indicated with the "D" letter on its data input). The triangle-shaped input denotes clock signal input, meaning this sequential circuit is synchronous (edge-sensitive). Reset and enable are two commonly used control signals that govern the state change process. When reset input is active, the flip-flow goes into predefined reset state (usually LOW). Reset input usually has the highest priority, meaning that other control signals are ignored when it is active. Enable input, when set, allows the state change to take place. When enabled (reset is inactive and enable is active) the D-type flip-flop stores the value that is fed to the data input D on the active edge. The internal state of the flip-flop is then propagated to the data output Q.

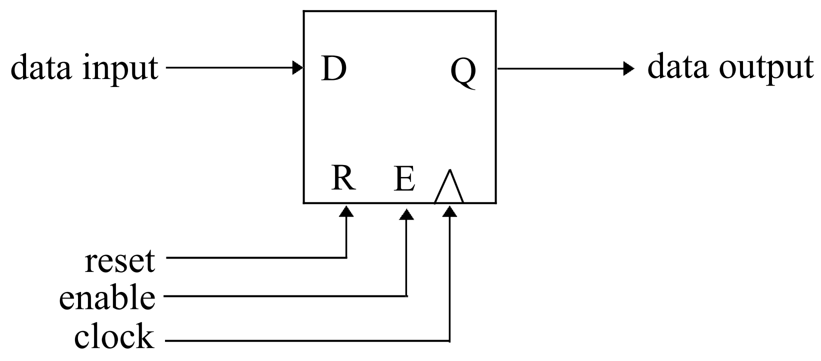

Figure 1: Block Diagram of D-type Flip-flop

VHDL description of the D-type flip-flop depicted above is shown in Listing 1. Signal “Q” infers a register during synthesis, as it is assigned a value inside an edge-sensitive **if** statement (with “clock'event **and** clock = '1' ” condition, alternatively “rising_edge(clock)” can be used as well). This makes the **else** path excessive, as the storage element should hold the data unchanged in any other case. The state change occurs on the rising edge of the “clock”, because the condition implies that the signal changes from ’0’ to ’1’. The “reset” input signal forces flip-flop into reset state (’0’) when active (’1’) and has priority over “enable” since it is evaluated first in the description. When “reset” is inactive the flip-flop stores the value from input “D” when “enable” is set (’1’). Otherwise, the flip-flop’s value remains unchanged.

Listing 1: VHDL Description of D-type Flip-flop

```
process (clock)
begin
  if clock'event and clock = '1' then
    if reset = '1' then
      Q <= '0';
    elsif enable = '1' then
      Q <= D;
    end if;
  end if;
end process;
```

One flip-flop can store only one bit of information. However, the data items in digital systems can be much bigger. In order to store multi-bit data flip-flops are grouper into arrays called registers (Figure 2). Each flip-flop has individual input/output that can be accessed in parallel, but all of them share clock and control signals (reset is also shared, but is not shown in Figure 2) that allow joint operation. When arranged in such a way, the data comprising of several bits can be simultaneously written to/read from all the flip-flops. The VHDL description of a register is basically the same as in Listing 1, but “Q” and “D” signals are of *std_logic_vector* type with the appropriate width (instead of *std_logic* as in the case of a flip-flop).

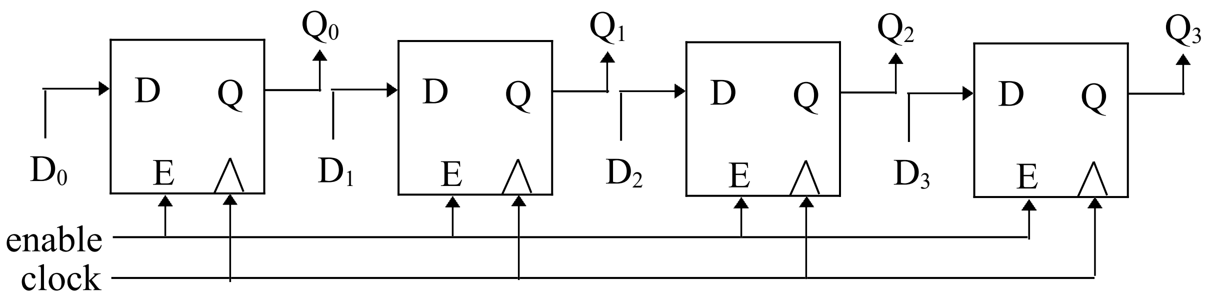

Figure 2: Block Diagram of 4-bit Register Built from Four D-type Flip-flops

Another possible arrangement of flip-flops in a register is to cascade them (data output of one flip-flop goes to the data input of the next one). This type of register is called shift register. An example of a shift register connected in serial-in, parallel-out (SIPO) fashion is shown in Figure 3. All flip-flops share clock and control signals for joint operation, and the content of all flip-flops can be read in parallel (hence “parallel-out”). However, saving of data is performed via 1-bit input D (i.e. serially, hence “serial-in”). Thus it requires four clock cycles to input (shift in) a 4-bit data into such register. Technically, in the same manner data can be read out serially as well (shifted out).

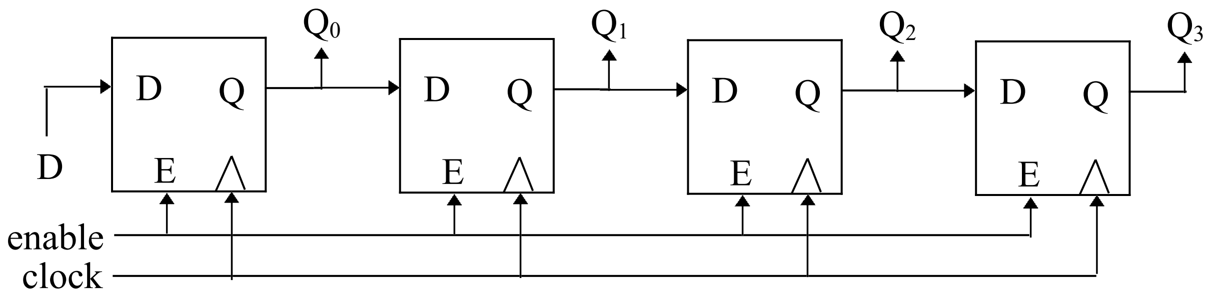

Figure 3: Block Diagram of 4-bit SIPO Shift Register Built from Four D-type Flip-flops

VHDL description of the SIPO shift register depicted above is shown in Listing 2. It is very similar to the description of an ordinary register. The only difference is that when the shift register is enabled, the states of flip-flops from 0 to 2 overwrite the states of neighboring flip-flops from 1 to 3 (the content is moved one position to the right).

Listing 2: VHDL Description of 4-bit SIPO Shift Register

```
process (clock)
begin
  if clock'event and clock = '1' then
    if reset = '1' then
      Q <= (others => '0');
    elsif enable = '1' then
      Q(0) <= D;
      Q(1 to 3) <= Q(0 to 2);
    end if;
  end if;
end process;
```

The leftmost flip-flop stores the value from input “D”. However, during shift the state of the rightmost flip-flop is lost. Similarly, the shifting may be arranged in a circular fashion when instead of storing values from external input “D” the register stores its rightmost value “Q(3)”. Also, it can be easily seen from the VHDL code that signal “Q” is of *std_logic_vector (0 to 3)* type, while signal “D” is *std_logic*.

Another widely used sequential circuit is a counter. Counter stores the number of clock cycles it has been active (enabled). The principle structure of a generic 4-bit counter is shown in Figure 4. It comprises of a 4-bit register whose inputs and outputs are connected to a combinational logic block that computes the next digit based on value of the one that is currently stored. The exact logic inside the “next value function” depends on the numerical system that is being employed. It is possible to count in, for example, binary, BCD or Gray encoded formats. Similarly, the direction of counting can be implemented as incrementing or decrementing.

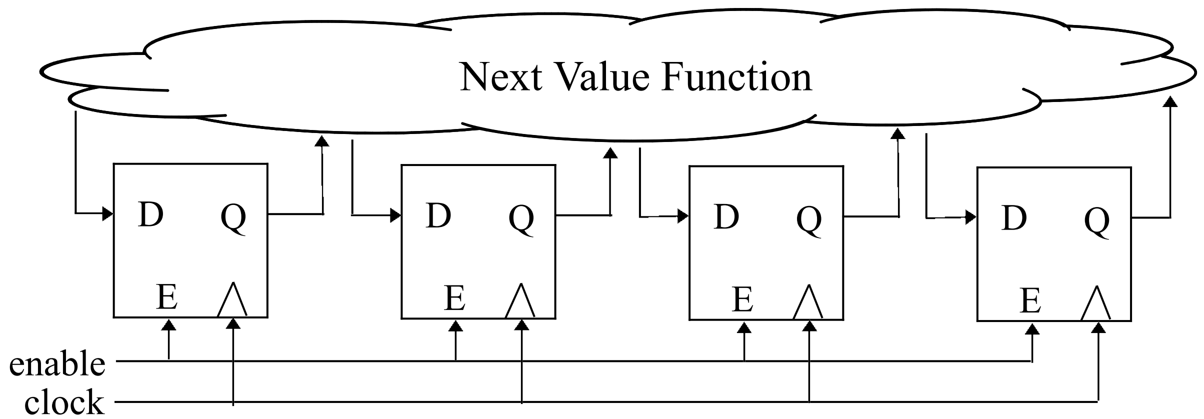

Figure 4: Block Diagram of a Generic 4-bit Counter

VHDL description of the 4-bit binary up (incrementing) counter is shown in Listing 3. Essentially it is the description of an ordinary register, but its next state is calculated as addition of one to the value that is currently stored. Note that the value of the *std_logic_vector* type signal is not a number, thus arithmetic operations are not actually defined for this type. In order to use addition with *std_logic_vector* type signal it must be typecasted to *unsigned* type first (the value is treated as a binary number without sign). The result of the addition is then typecasted back to *std_logic_vector* type since signal "Q" on the left side of the assignment is of that type. The *unsigned* type and addition operation for that type are defined in the *ieee.numeric_std* standard VHDL package that must be declared in order for the VHDL code in Listing 3 to work. Also, note that *ieee.numeric_std* defines addition of an *integer* type value to *unsigned* type value (there is no need to convert *integer* type to *unsigned* type or vice versa).

Listing 3: VHDL Description of 4-bit Binary Up Counter
```
process (clock)
begin
  if clock'event and clock = '1' then
    if reset = '1' then
      Q <= (others => '0');
    elsif enable = '1' then
      Q <= std_logic_vector(unsigned(Q) + 1);
    end if;
  end if;
end process;
```