EASTERN MEDITERRANEAN UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT
CMPE 224 DIGITAL LOGIC SYSTEMS

***EXPERIMENT III*: VHDL IMPLEMENTATION OF SEQUENTIAL CIRCUITS: FLIP-FLOPS AND SIMPLE EXAMPLES**

Objectives:
VHDL implementation of four flip-flop units and sequential circuits containing different types of flip-flops will be studied.

Preliminary Work:
1. ***Read the explanations in "Additional VHDL Topics Used in This Experiment".***
2. ***Write down and save the source codes of experimental steps in VHDL. You will demonstrate the remaining steps in labaoratory.***

Additional VHDL Topics Used in This Experiment:

A. **Implementation of Sequential Circuits in VHDL**: In VHDL, you can describe the behavior of a sequential logic element such as a latch or flip-flop, as well as specifying the behavior of more complex sequential machines. Remember that the behavior of a sequential logic element (latch or flip-flop) is to save a value of a signal over time.

As in combinational circuits, there are often several ways to describe a particular sequential behavior. However, the two commonly-used methods used to describe sequential behavior are conditional (**if-then**) specifications and **wait** statements.

Conditional Specification

Describing sequential logic with a conditional specification relies on the inherent behavior of a VHDL if statement. The convention used for conditional statements that describe positive-edge sensitive clocking logic is:

```
process(clk)
begin
        if clk'event and clk='1' then
                y <= a;
        end if;
end process;
```

If you are using the IEEE 1164 std_logic (or std_ulogic) data types, you can simplify the description of clock edges (and improve the accuracy of simulations) by using the ***rising_edge()*** function:

```
process(clk)
begin
        if rising_edge(clk) then
                y <= a;
        end if;
end process;
```

Wait Statement

The second method uses a **wait** statement within the process:

```
process
wait until expression;
.

.
end process;
```

This statement suspends evaluation (over time) of a process or a procedure until an event occurs, and the expression evaluates to **true**. When a wait statement is used in a process, no process sensitivity list is required (or allowed). An example on the use of wait statement may be described as:

```
process
        wait until clk'event and clk='1'
        y <= a;
end process;
```

A constraint of the VHDL synthesizer is that wait statements must be located at either the beginning or end of a process, and there may not be more than one wait statement in a process. In addition to this, Wait statements are not recommended for use in synthesizable designs. If-then conditional statements are a more universally accepted method of describing sequential logic.

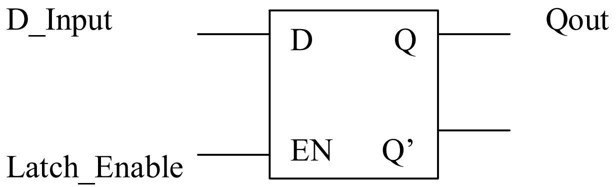
B. Latches and Flip-Flops

Remember, the most basic memory element is the latch – a device that holds a value until it is told to change that value by a (conditional) change in its inputs. VHDL implements latches as if they are incomplete multiplexers. For example a D-latch with a Latch_Enable input is implemented as:

```
Process (Latch_Enable, D_Input)
Begin
        If (Latch_Enable='1') then
```

```
                Qout <=D_Input;
        End if;
End Process;
```

Notice that**, if Latch_Enable is '0'**, **Qout does not change** – its old value is retained. This code synthesizes to the following D-Latch:



One of the key points to get from this example is that **Qout is not given value under all conditions, but only when Latch_Enable is '1'**. This creates the need for VHDL to put down a latch to mimic this VHDL's functionality exactly in hardware. This is called latch inferencing in VHDL. Related with this, it is possible to fool ourselves into thinking that a "resetable" latch can be implemented in VHDL like:

```
Process (Latch_Enable, D_Input)
Begin
        If (Latch_Enable='1') then
                Qout <=D_Input;
Else
                Qout='0';
        End if;
End Process;
```

which is synthesized as a simple 2-input AND gate. All discussions and claims of this subsection **is valid for VHDL implementation of all asynchronous latches.**

In synchronous (clocked) operation, values of signals or variables is determined as a function of a clock signal. For example, VHDL implementation of a rising-edge sensitive D flipflop is

```
Process (Clock, D_Input)
Begin
    If (Clock'event and Clock='1') then
            Qout <= D_Input;
    End if;
End process;
```

Note that, D_Input is not absolutely necessary in the process's sensitivity list for synchronous operation because the value of D_Input is not important until the Clock's edge changes. Hence, it is also perfectly OK to rewrite the above VHDL code as

```
Process (Clock)
Begin
    If (Clock'event and Clock='1') then
            Qout <= D_Input;
    End if;
End process;
```

Design of flip-flops with asynchronous set or reset inputs, the following style using if-elseif construct can be used:

```
Process (Reset, Clock)
Begin
    If (Reset='1') then   -- asynchronous reset
            Qout<='0';
    elseif (Clock'event and Clock='1') then
            Qout <= D_Input;
    End if;
End process;
```

When using this style, it is IMPORTANT to put all the asynchronous signals or variables before the final "elseif (Clock'event and Clock='1') then" statement. Hence, ***the following code is not buildable because it is impossible to build hardware that runs purely on the no-rising edge of clock***.

```
Process (Reset, Clock)
Begin
    if (Clock'event and Clock='1') then  --synchronous
            Qout <= D_Input;
    elseif (Reset='1') then           -- asynchronous reset
            Qout<='0';
    End if;
End process;
```

C. Example VHDL Codes:

I. **The following is a VHDL code of a positive edge triggered D flip-flop. This is your first building block to learn when constructing sequential machines.**

```
    library ieee;
    use ieee.std_logic_1164.all;
    entity dflipflop is port (
      d,clk: in std_logic;   -- defines the inputs
      q:  out std_logic );   -- defines the output
    end dflipflop;
```

```
architecture example of dflipflop is
    begin
   process (clk) begin
   if (clk'event and clk = '1') then
   q <=d;    -- q is assigned the valu of d
   end if;
    end process;
end example;
```

II. **The following is a VHDL code of a negative edge triggered d-type flip-flop.**

```
library ieee;
use ieee.std_logic_1164.all;
   entity dflipflopn is port (
               d,clk: in std_logic;
               q:  out std_logic );
   end dflipflopn;

architecture example of dflipflopn is
   begin
      process(clk) begin
               if (clk'event and clk = '0') then
                              q <=d;
               end if;
     end process;
end example;
```

III. **VHDL implementation of SR-Latch**

```
library ieee;
use ieee.std_logic_1164.all;
   entity SR_Latch is
        port (S, R: in std_logic;
               Q,Qbar: out std_logic);
   end SR_Latch;

architecture behaviour of SR_Latch is
   begin
      p0: process(R,S)
               begin
                    case std_logic_vector'(R,S) is
                           when "01" =>
                                      Q <='1';
                                      Qbar<='0';
                           when "10" =>
```

```
                                        Q<=’0’;
                                        Qbar<=’1’;
                              when “11” =>
                                        Q<=’1’;
                                        Qbar<=’1’^;
                              when others =>
                                        null;
                              end case;

                    end process p0;
          end architecture behaviour;
```

IV. **VHDL implementation of JK FF**

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

---------------------------------------------

entity JK_FF is
port (    clock:              in std_logic;
          J, K:               in std_logic;
          reset:              in std_logic;
          Q, Qbar: out std_logic
);
end JK_FF;

----------------------------------------------

architecture behv of JK_FF is

   -- define the useful signals here

   signal state: std_logic;
   signal input: std_logic_vector(1 downto 0);

begin

   -- combine inputs into vector
   input <= J & K;

   p: process(clock, reset) is
   begin

          if (reset='1') then
```

```
            state <= '0';
         elsif (rising_edge(clock)) then

      -- compare to the truth table
            case (input) is
                    when "11" =>
                       state <= not state;
                    when "10" =>
                       state <= '1';
                    when "01" =>
                       state <= '0';
                    when others =>
                       null;
                    end case;
         end if;

   end process;

   -- concurrent statements
   Q <= state;
   Qbar <= not state;

end behv;
```

V. **VHDL implementation of T FF**

```
      library ieee;
      use ieee.std_logic_1164.all;
        entity T_FF is
            port (T, clock, Reset: in std_logic;
                   Q,Qbar: out std_logic);
        end T_FF;

      architecture behaviour of T_FF is
        signal state: std_logic;
        begin
           p0: process(clock, Reset)
                   begin
                       if (Reset='0') then
                             state <='0';
                        elseif rising_edge(clock) then
                             if T = '1' then
                                        state <= not state;
                             end if;
                      end if;
```

```
                        end process p0;
                        Q<=state;
                        Qbar<= not state;
          end architecture behaviour;
```

Experimental Work

Simulate examples from I to V using Quartus II. Show the FF excitations under different input and present-state conditions.

Good Luck.

Dr. Adnan Acan