VHDL - Sequential PROCESS Logic Synthesis

Introduction

- The next step after HDL code Simulation is HDL code Synthesis.
- Synthesis step contains the next:
  - ✓ conversion of the high-level VHDL (or Verilog) language, which describes the circuit at the Register Transfer Level (RTL), into a netlist at the gate level.
  - ✓ Optimization of the gate-level netlist for speed (minimize critical path) and for area (minimize Logic function).
  - ✓ Implementation of the optimized gate-level netlist based on **MUXs+LUTs, Latches, FFs** (in case of FPGA as a target Hardware).
- The last step is a place and route (fitter), software will generate the physical layout for a FPGA chip or will generate the masks for an ASIC chip.

Synthesis coding approach

- Synthesis tools and Simulation tools translate PROCESS based HDL code in a different way (concurrent code translated in the same way).
- Synthesis tools search for adjustment of VHDL code to one of the next three template kinds (*ieee-1076.6 standard*):

  Combinational Logic, Synchronous Logic, Latch based Logic.
- **Our goal:**
  - ✓ writing of HDL code which will be translated in the same exact way by all Synthesis and Simulation tools.
  - ✓ Avoid of HDL code which synthesized with hardware errors in the required design (avoid from Sick Hardware).
  - ✓ **Important rule:** when you write HDL code, think Hardware!

Note: Unusual and Unsupported design approach

With a guarded BLOCK or with WHEN statements (using concurrent code) even very simple sequential circuits can be constructed. This, however, is Unusual and Unsupported design approach.

In conclusion: Synchronous design will be described using PROCESS only!

DFF implementation example

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY dff IS
  PORT ( d, clk, rst: IN STD_LOGIC;
         q: OUT STD_LOGIC);
END dff;
-----------------------------------------
ARCHITECTURE dff OF dff IS
BEGIN
  b1: BLOCK (clk'EVENT AND clk='1')
  BEGIN
     q <= GUARDED '0' WHEN rst='1' ELSE d;
  END BLOCK b1;
END dff;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------
ENTITY dff IS
  PORT ( d, clk : IN STD_LOGIC;
                q : BUFFER STD_LOGIC);
END dff;
---------------------------------
ARCHITECTURE dff OF dff IS
BEGIN
    q <= d WHEN clk'event and clk='1' ELSE q;
END dff;
```

PROCESS Logic Synthesis

The way we write a PROCESS affects its synthesis and is associated with one of the following two synthesis types:

- **Combinatorial PROCESS (Combinational Logic Circuit):**
  - ✓ PROCESS that its sensitivity list contains all its internal input SIGNALS and the PROCESS doesn't contain IF-THEN statement which its condition on SIGNAL event.
  - ✓ This kind of PROCESS describes a combinational logic circuit.
  - ✓ If we write a partial sensitivity list, the compiler completes it, differ from simulation environment.
- **Sequential PROCESS (Synchronous / Asynchronous Logic Circuit):**
  - ✓ PROCESS that its sensitivity list contains a input SIGNAL and the PROCESS contains IF-THEN statement which its condition on SIGNAL event.
  - ✓ This kind of PROCESS describes FFs based sequential logic circuit triggered by a SIGNAL event.

Sequential PROCESS (A/Synchronous Circuit)

In order the compiler will synthesize the PROCESS as a Synchronous Logic Circuit we must obey the next two rules (Synchronous Logic template):

Rule 1: Make sure that only the trigger input SIGNAL (mostly named clk) and its Asynchronous SIGNAL (mostly named rst) in the required Synchronous circuit, appear in the PROCESS sensitivity list.

Rule 2: Use a main **IF-THEN** statement (from the only next two patterns) which its condition on SIGNAL event is a one of two forms (at the beginning, without using of any ELSIF **or** at the end, at the last ELSIF), positive edge trigger or negative edge trigger.

```
PROCESS (clk)
BEGIN                          Synchronous part
    IF (clk'EVENT AND clk='1') THEN
        sequential_statements;
    END IF;                    Combinational Logic
END PROCESS;
```

```
PROCESS (clk,rst)
BEGIN                          Asynchronous part
    IF (rst='1')
        signals_assignment;
    ELSIF (clk'EVENT AND clk='1') THEN
        sequential_statements;
    END IF;                    Combinational Logic
END PROCESS;                   Synchronous part
```

Sequential PROCESS (Synchronous Circuit)

- Positive edge trigger condition:

```
IF (clk'EVENT AND clk='1') THEN
    sequential_statements;
END IF;
```

```
IF (rising_edge(clk)) THEN
    sequential_statements;
END IF;
```

- Negative edge trigger condition:

```
IF (clk'EVENT AND clk='0') THEN
    sequential_statements;
END IF;
```

```
IF (falling_edge(clk)) THEN
    sequential_statements;
END IF;
```

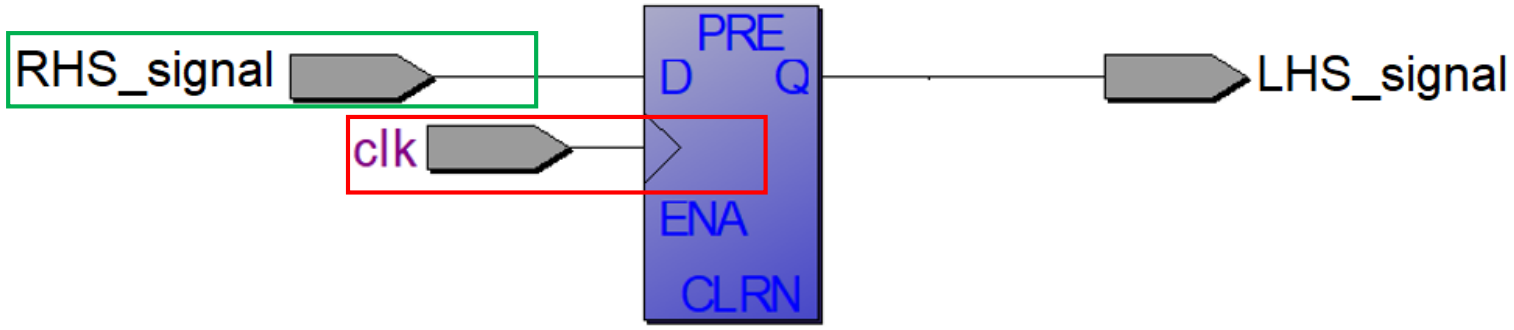
Three FFs (register) templates

Template No.1 (pure synchronous) – FF inferred version 1:

```
PROCESS (clk)
BEGIN
    IF (clk'EVENT AND clk='1') THEN
        LHS_signal <= RHS_signal;
    END IF;
END PROCESS;
```

Synchronous part

Synchronous Combinational Logic



FF inferred version 1: A SIGNAL generates a flip-flop whenever an assignment is made at the transition of another signal, that is, when a synchronous assignment occurs.

Three FFs (register) templates

Template No.1 (including asynchronous logic) – FF inferred version 1:

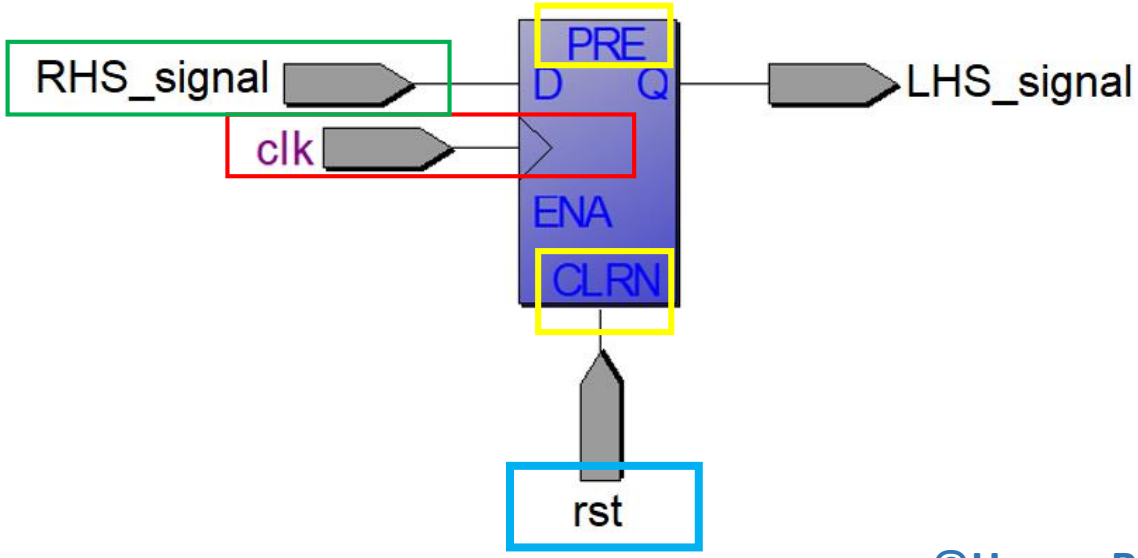
```
PROCESS (clk,rst)
BEGIN
     IF (rst='1') THEN
          LHS_signal <= '0';
     ELSIF (clk'EVENT AND clk='1') THEN
          LHS_signal <= RHS_signal;
     END IF;
END PROCESS;
```

Asynchronous Combinational Logic

Asynchronous part (setting of all circuit outputs which are described by the PROCESS)

Synchronous Combinational Logic

Synchronous part



Note: if the asynchronous part was as the next code, PRE input would been used.

```
IF (rst='1')
     LHS_signal <= '1';
ELSIF (clk'EVENT AND clk='1') THEN
     LHS_signal <= RHS_signal;
END IF;
```

Three FFs (register) templates

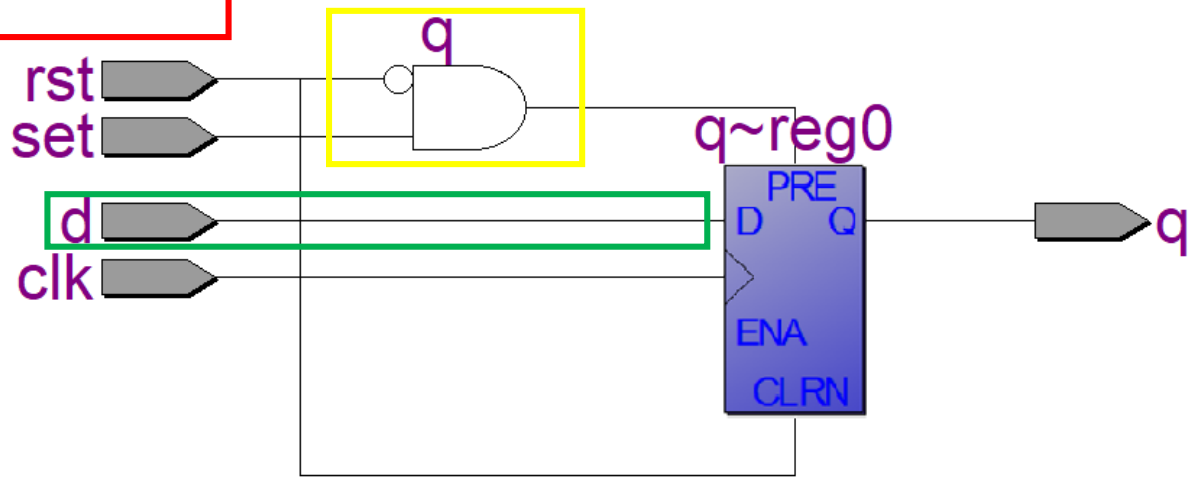
Template No.1 (including asynchronous logic) – FF inferred version 1:

```
PROCESS (clk,rst,set)
BEGIN
     IF (rst='1') THEN
          q <= '0';
     ELSIF (set='1') THEN
          q <= '1';
     ELSIF (clk'EVENT AND clk='1') THEN
          q <= d;
     END IF;
END PROCESS;
```

Asynchronous Combinational Logic

Synchronous part

Synchronous Combinational Logic



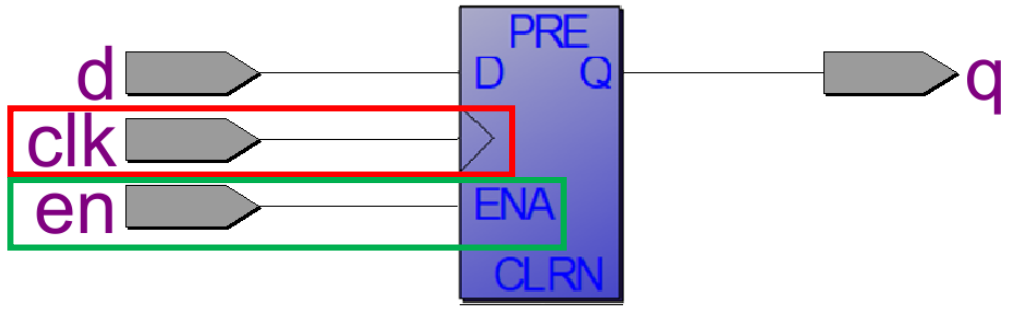
Three FFs (register) templates

Template No.1 (including ENA logic) – FF inferred version 1:

```
PROCESS (clk)
BEGIN
    IF (clk'EVENT AND clk='1') THEN
        IF (en='1') THEN
            q <= d;
        END IF;
    END IF;
END PROCESS;
```

Synchronous part

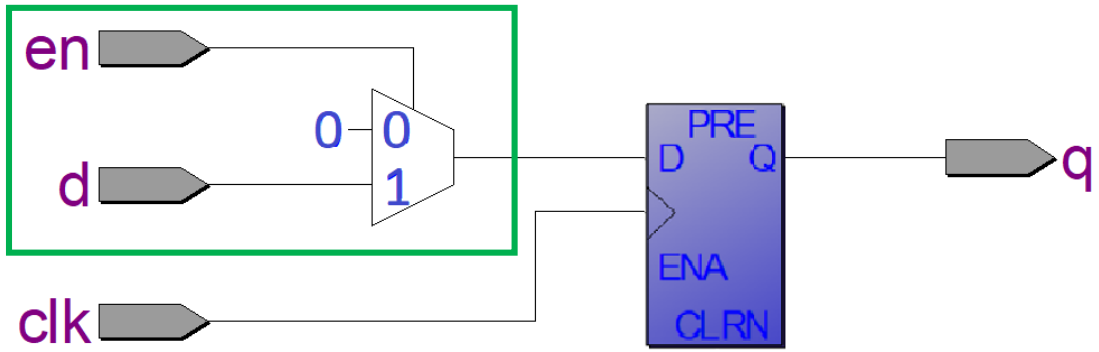
ENA logic, in case of IF-THEN nested without ELSE/ELSIF



Three FFs (register) templates

Template No.1 (D input logic) – FF inferred version 1:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY en_dff IS
    PORT ( d,clk,en: IN STD_LOGIC;
                q: OUT STD_LOGIC);
END en_dff;
---------------------------------------
ARCHITECTURE rtl OF en_dff IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'EVENT AND clk='1') THEN
            IF (en='1') THEN
                q <= d;
            ELSE
                q <= '0';
            END IF;
        END IF;
    END PROCESS;
END rtl;
```



FF inferred version 1- examples

```
PROCESS (clk)
BEGIN
    IF (clk'EVENT AND clk='1') THEN
        output1 <= temp; -- output1 stored
        output2 <= a;    -- output2 stored
    END IF;
END PROCESS;
```

Two FFs inferred

```
PROCESS (clk)
BEGIN
    IF (clk'EVENT AND clk='1') THEN
        output1 <= temp; -- output1 stored
    END IF;
    output2 <= a; -- output2 not stored
END PROCESS;
```

Only one FFs inferred

FF inferred version 2

A VARIABLE, will not necessarily generate flip-flops if its value never leaves the PROCESS (or FUNCTION, or PROCEDURE). However, if a value is assigned to a variable at the transition of another signal, and such value is eventually passed to a signal (which leaves the process), then flip-flops will be inferred.

```
PROCESS (clk)
    VARIABLE temp: BIT;
BEGIN
    IF (clk'EVENT AND clk='1') THEN
        temp := a;
    END IF;
    x <= temp; -- temp causes x to be stored
END PROCESS;
```

FF inferred version 3

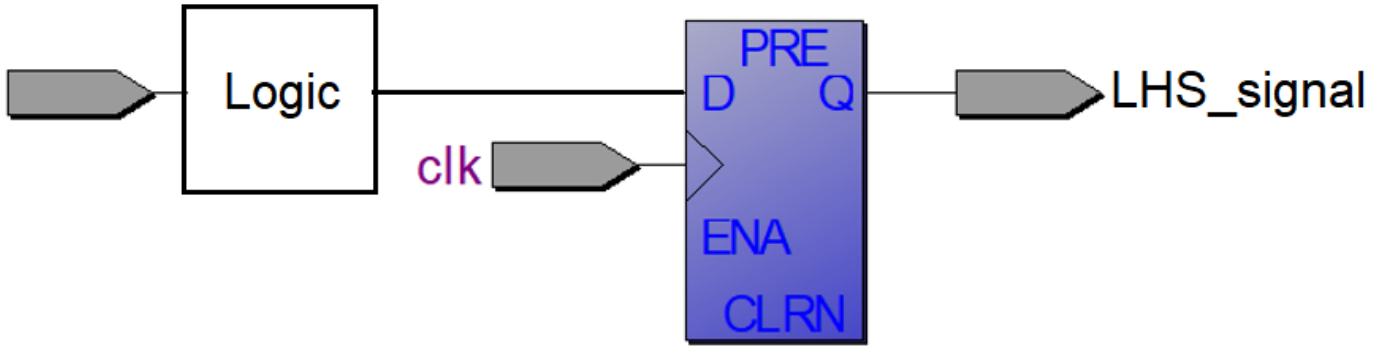
A VARIABLE also generates a register when it is used before a value has been assigned to it. In this case a VARIABLE is used wrongly, instead of use VARIABLE for intermediate calculations it is used as a memory element.

Remember: don't read from VARIABLE before a value has been assigned to it.

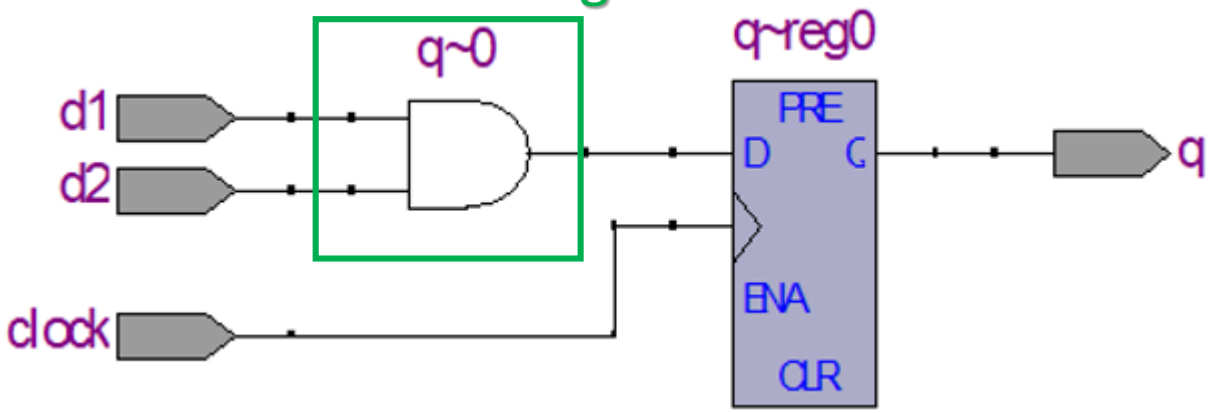
```
PROCESS (clk)
    VARIABLE a : BIT;
BEGIN
    IF (clk'EVENT AND clk='1') THEN
        output <= a;  -- output stored
        a := input;
    END IF;
END PROCESS;
```

Three FFs (register) templates

Template No.2:



Combinational Logic

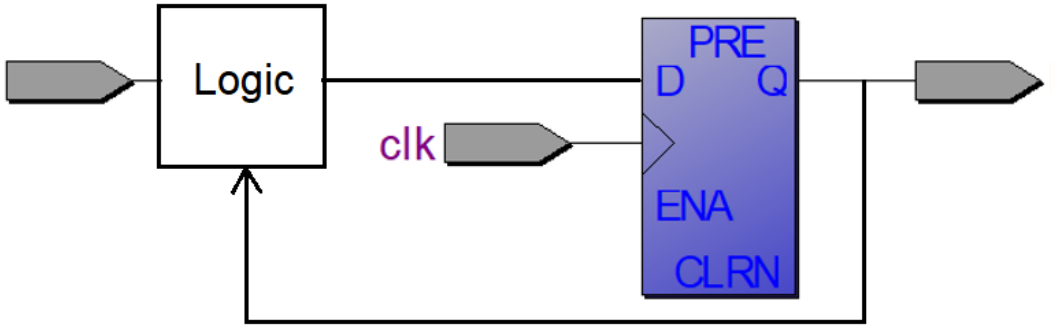


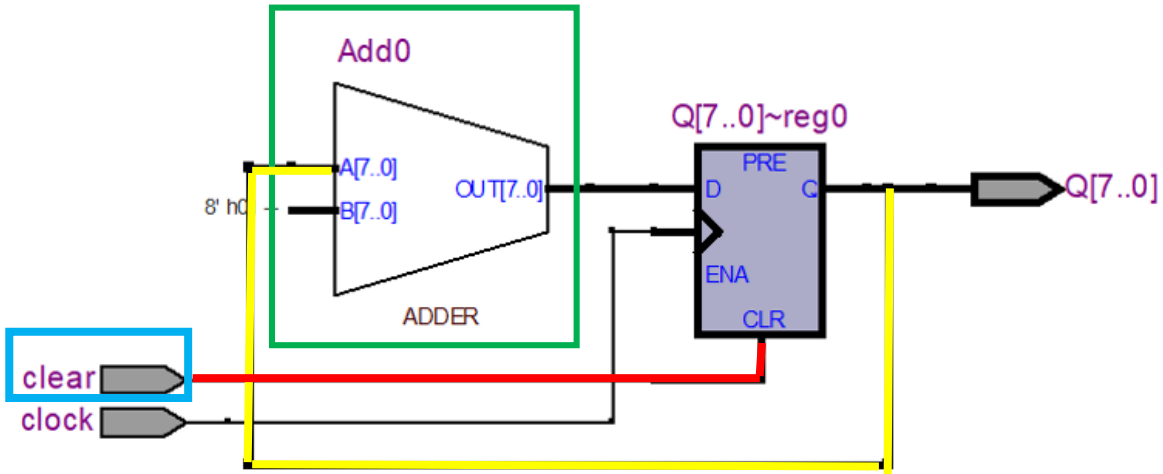

```
entity sync_logic is
port(
     d1:            in std_logic;
     d2:            in std_logic;
     clock:         in std_logic;
     q:             out std_logic
);
end sync_logic;

architecture rtl of sync_logic is
begin
     process(clock)
     begin
          if rising_edge(clock) then
               q <= d1 and d2;
          end if;
     end process;
end rtl;
```

Combinational Logic

Three FFs (register) templates

Template No.3:



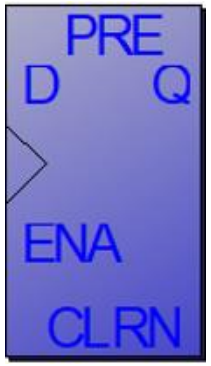



```
entity counter is
port(
    clock   : in std_logic;
    clear   : in std_logic; -- Async reset
    Q       : buffer std_logic_vector(7 downto 0));
end counter;

architecture rtl of counter is
begin
    process(clock,clear)
    begin
        if clear = '1' then
            Q <= "00000000";
        elsif rising_edge(clock) then
            Q <= Q + "00000001";
        end if;
    end process;
end rtl;
```

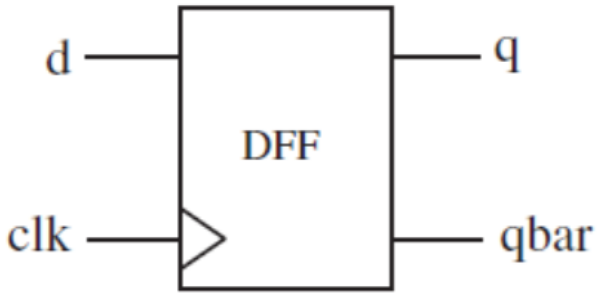
DFF with q and qbar - example

Q: How do we implement the next DFF

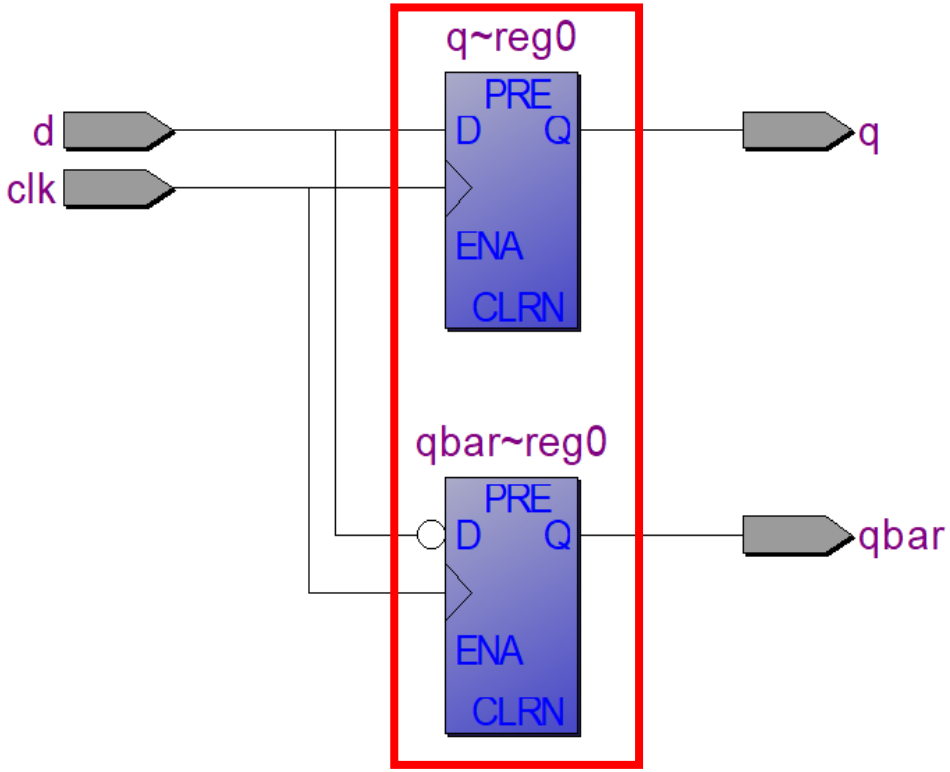
A1: remember that the base FF is based on the form:




```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------
ENTITY dff1 IS
     PORT ( d, clk: IN STD_LOGIC;
                 q: BUFFER STD_LOGIC;
              qbar: OUT STD_LOGIC);
END dff1;
```



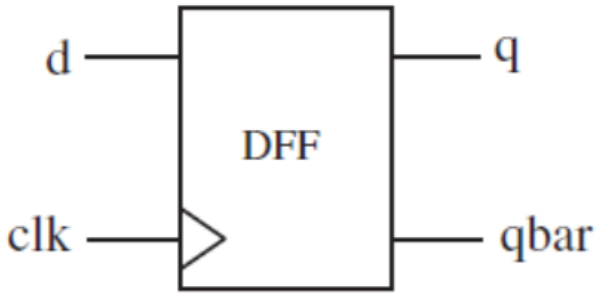

```
ARCHITECTURE two_dff OF dff1 IS
BEGIN
     PROCESS (clk)
     BEGIN
          IF (clk'EVENT AND clk='1') THEN
               q <= d;          -- generates a register
               qbar <= NOT d;   -- generates a register
          END IF;
     END PROCESS;
END two_dff;
```

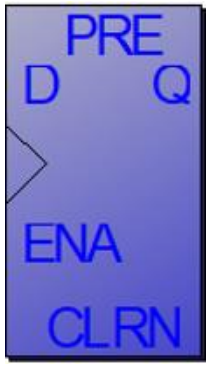


DFF with q and qbar - example

Q: How do we implement the next DFF

A2: remember that the base FF is based on the form:






```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------
ENTITY dff1 IS
     PORT ( d, clk: IN STD_LOGIC;
                 q: BUFFER STD_LOGIC;
              qbar: OUT STD_LOGIC);
END dff1;
```

```
ARCHITECTURE one_dff OF dff2 IS
BEGIN
     PROCESS (clk)
     BEGIN
          IF (clk'EVENT AND clk='1') THEN
               q <= d; -- generates a register
          END IF;
     END PROCESS;
     qbar <= NOT q; -- uses logic gate (no register)
END one_dff;
```