BLOCK

There are two kinds of BLOCK statements: Simple and Guarded.

Simple BLOCK

The BLOCK statement, in its simple form, represents only a way of locally partitioning the code.

It allows a set of concurrent statements to be clustered into a BLOCK, with the purpose of turning the overall code more readable and more Manageable.

Its syntax is shown below.

```
label: BLOCK
  [declarative part]
BEGIN
  (concurrent statements)
END BLOCK label;
```

Therefore, the overall aspect of a "blocked" code is the following:

```
------------------------
ARCHITECTURE example ...
BEGIN
...
block1: BLOCK
        BEGIN
                ...
        END BLOCK block1
        ...
block2: BLOCK
        BEGIN
                ...
        END BLOCK block2;
        ...
END example;
------------------------
```

Example:

```
b1: BLOCK
        SIGNAL a: STD_LOGIC;

        BEGIN
                a <= input_sig WHEN ena='1' ELSE 'Z';
        END BLOCK b1;
```

A BLOCK (simple or guarded) can be nested inside another BLOCK. The corresponding syntax is shown below.

```
label1: BLOCK

        [declarative part of top block]

        BEGIN

        [concurrent statements of top block]

                label2: BLOCK
                        [declarative part nested block]
                BEGIN
                        (concurrent statements of nested block)
                END BLOCK label2;

        [more concurrent statements of top block]

END BLOCK label1;
```

Guarded BLOCK

A guarded BLOCK is a special kind of BLOCK, which includes an additional expression,
called guard expression. A guarded statement in a guarded BLOCK is executed only when the guard expression is TRUE.

Guarded BLOCK:

```
label: BLOCK (guard expression)
        [declarative part]
BEGIN
        (concurrent guarded and unguarded statements)
END BLOCK label;
```

Example: Latch Implemented with a Guarded BLOCK

The example presented below implements a transparent latch.

In it, clk='1' is the guard expression, while q<=GUARDED d is a guarded statement. Therefore, q<=d will only occur if clk='1'.

```
-------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------
ENTITY latch IS
        PORT (d, clk: IN STD_LOGIC;
                q: OUT STD_LOGIC);
END latch;
-------------------------------
ARCHITECTURE latch OF latch IS

BEGIN
        b1: BLOCK (clk='1')
        BEGIN
                q <= GUARDED d;
        END BLOCK b1;
END latch;
-------------------------------
```

Example: DFF Implemented with a Guarded BLOCK

In the code below

clk'EVENT AND clk='1' (line 12)   is the guard expression,

while q <= GUARDED '0' WHEN rst='1'   is a guarded statement.

Therefore, q<='0' will occur when the guard expression is true and rst is '1'.

```
1 -------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -------------------------------
5 ENTITY dff IS
6       PORT ( d, clk, rst: IN STD_LOGIC;
7               q: OUT STD_LOGIC);
8 END dff;
9 -------------------------------
10 ARCHITECTURE dff OF dff IS
11      BEGIN
12        b1: BLOCK (clk'EVENT AND clk='1')
13              BEGIN
14                      q <= GUARDED '0' WHEN rst='1' ELSE d;
15              END BLOCK b1;
16      END dff;
17 -------------------------------
```

Sequential Code

Concurrent code is intended only for the design of combinational circuits, while sequential code can be used indistinctly to design both sequential and combinational circuits.

The statements intended only for completely concurrent code, referred to as concurrent statements, are WHEN, SELECT, and GENERATE, while those for sequential code, referred to as sequential statements, are IF, WAIT, LOOP, and CASE.

In VHDL, there are three kinds of sequential code: PROCESS, FUNCTION and PROCEDURE (the last two are called subprograms). PROCESS is intended for the architecture body (main code, for example)

Main properties of SIGNAL:

A signal can only be declared outside sequential code (though it can be used there).

A signal is not updated immediately (when a value is assigned to a signal inside sequential code, the new value will only be ready after the conclusion of that run).

A signal assignment, when made at the transition of another signal, will cause the inference of registers (given that the signal affects the design entity).

Only a single assignment is allowed to a signal in the whole code (even though the compiler might accept multiple assignments to the same signal in PROCESS or subprograms, only the last one will be effective, so again it is just one assignment).

Main properties of VARIABLE:

A variable can only be declared and used inside a PROCESS or subprogram

(if it is a shared variable, then the declaration is made elsewhere, but it still should only be modified inside a sequential unit).

A variable is updated immediately (hence the new value can be used/tested in the next line of code).

A variable assignment, when made at the transition of another signal, will cause the inference (cikarim, sonuc) of registers (assuming that the variable’s value affects a signal, which in turn affects the design entity).

Multiple assignments are fine.

PROCESS:

A PROCESS is a sequential section of VHDL code. It is characterized by the presence of IF, WAIT, CASE, or LOOP, and by a sensitivity list. A PROCESS must be installed in the main code, and is executed every time a signal in the sensitivity list changes (or the condition related to WAIT is fulfilled).

Its syntax is shown below.

```
[label:] PROCESS (sensitivity list)

[VARIABLE name type [range] [:= initial_value;]]
        BEGIN
                (sequential code)
        END PROCESS [label];
```

VARIABLES are optional. If used, they must be declared in the declarative part of the PROCESS (before the word BEGIN, as indicated in the syntax above). The initial value is not synthesizable, being only taken into consideration in simulations. The use of a label is also optional. Its purpose is to improve code readability. The label can be any word, except VHDL reserved words.

Example

The (partial) process below is executed whenever clk or rst changes. It contains three variable declarations (a, b, c), the first two specified as INTEGER, the last one as BIT_VECTOR. Only for c a default value (optional) was entered.

```
PROCESS (clk, rst)

        VARIABLE a, b: INTEGER RANGE 0 TO 255;
        VARIABLE c: BIT_VECTOR(7 DOWNTO 0) := "00001111";

BEGIN
...
END PROCESS;
```

The IF Statement

IF, WAIT, LOOP, and CASE are the statements intended for sequential code (they can only be used inside a PROCESS or subprogram)

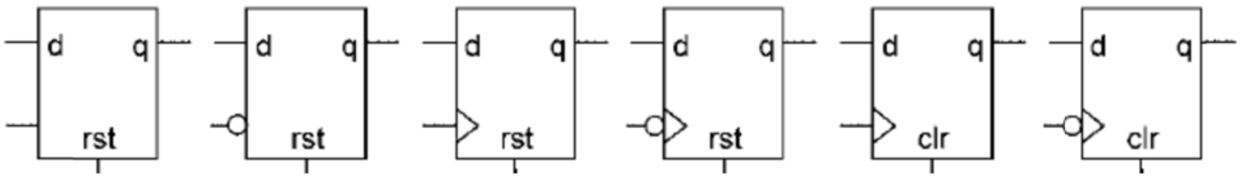
A simplified syntax for IF is shown below

```
[label:] IF conditions THEN
        assignments;
ELSIF conditions THEN
        assignments;
        ...
ELSE
        assignments;
END IF [label];
```

Example

```
IF (x<y) THEN
        temp:= "00001111";
ELSIF (x=y AND w='0') THEN
        temp:= "11110000";
ELSE
        temp:=(OTHERS => '0');
END IF;
```

DFFs with Reset and Clear



Whenever rst = '1' occurs, the output is immediately zeroed, regardless of the clock

Whenever clr is asserted we still need to wait until the proper clock transition occurs for the output to be zeroed

Example:
DFFs with Reset and Clear. Employing the IF statement, write a code that implements the DFFs

```
---------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY flipflops IS
    PORT (d1, d2, clk, rst, clr: IN STD_LOGIC;
    q1, q2: OUT STD_LOGIC);
END ENTITY;
---------------------------------------------
ARCHITECTURE flipflops OF flipflops IS
BEGIN
    ---DFF with reset---
    with_reset: PROCESS (clk, rst)
    BEGIN
        IF (rst='1') THEN
            q1 <= '0';
        ELSIF (clk'EVENT AND clk='1') THEN
            q1 <= d1;
        END IF;
    END PROCESS with_reset;

---DFF with clear:---
    with_clear: PROCESS (clk)
    BEGIN
        IF (clk'EVENT AND clk='1') THEN
            IF (clr='1') THEN
                q2 <= '0';
            ELSE
                q2 <= d2;
            END IF;
        END IF;
    PROCESS with_clear;
END ARCHITECTURE;
---------------------------------------------
```