Signals and Variables
When and where to use them

We study the importance and differences between signals and variables

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE

Signals and Variables

Introduction

- VHDL provides two objects to deal with non-static data values: **SIGNAL** and **VARIABLE**.
- VHDL provides **CONSTANT** and **GENERIC** for establishing default static values.
- **CONSTANT** and **SIGNAL** can be global (seen by the whole code) and used in sequential or concurrent code.
- **VARIABLE** is local and only used within the **sequential code (PROCESS, FUNCTION or PROCEDURE)**.

Signals and Variables

CONSTANT

- CONSTANT is used to stablish default values. Syntax:

  CONSTANT name : type := value;

```
CONSTANT set_bit : BIT := '1';
CONSTANT datamemory : memory := (('0','0','0','0'),
                                 ('0','0','0','1'),
                                 ('0','0','1','1'));
```

A **CONSTANT** can be declared in a **PACKAGE** (truly global), **ENTITY** (global for the entity and all its architectures) or **ARCHITECTURE** (global for this architecture).

CONSTANT are commonly used in **PACKAGE** or **ARCHITECTURE**, less common in **ENTITY**

Signals and Variables

SIGNAL

- **SIGNAL** is the object to pass values in and out of the circuit also among internal units.
- **SIGNAL** can be seen as the wires that interconnect circuits (all PORTS of an ENTITY are signals by default).

Syntax:

SIGNAL name : type [range] [:= initial_value];

```
SIGNAL control: BIT := '0';
SIGNAL count: INTEGER RANGE 0 TO 100;
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

A **SIGNAL** can be declared in a **PACKAGE** (truly global), **ENTITY** (global for the entity and all its architectures) or **ARCHITECTURE** (global for this architecture).

A **SIGNAL** used in **sequential code** (PROCESS, PROCEDURE, FUNCTION) is **not updated immediately** (need conclusion of the sequential code first).

Signals and Variables

SIGNAL

```
---------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY count_ones IS
   PORT ( din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
          ones: OUT INTEGER RANGE 0 TO 8);
END count_ones;
---------------------------------------
ARCHITECTURE not_ok OF count_ones IS
   SIGNAL temp: INTEGER RANGE 0 TO 8;
BEGIN
   PROCESS (din)
   BEGIN
      temp <= 0;
      FOR i IN 0 TO 7 LOOP
         IF (din(i)='1') THEN
            temp <= temp + 1;
         END IF;
      END LOOP;
      ones <= temp;
   END PROCESS;
END not_ok;
---------------------------------------
```

- This circuit counts the number of 1s in the input.
- The code is not properly written as the signal temp is not updated until the PROCESS is finished. Therefore, we should not assign a “new” value to it.

- For this example, it is better to use **VARIABLE**.

- Notice that defining “ones” as **BUFFER** instead of **OUT** is also possible. In this case, “ones” can be used internally and be assigned by values. Nevertheless, use the temporal variable “temp” and “ones” as output is a better practice as “ones” is a real output of the circuit.

Signals and Variables

VARIABLE

- Unlike CONSTANT or SIGNAL, **VARIABLE** is local information. Only used in **PROCESS FUNCTION** or **PROCEDURE**.
- A VARIABLE is updated immediately. That is very good feature to work in sequential code.

VARIABLE name : type [range] [:= init_value];

```
VARIABLE control: BIT := '0';
VARIABLE count: INTEGER RANGE 0 TO 100;
VARIABLE y: STD_LOGIC_VECTOR (7 DOWNTO 0) := "10001000";
```

- Obviously, a VARIABLE can be declared only on the declaration section of PROCESS, FUNCTION and PROCEDURE.

```
-------------------------------------
ARCHITECTURE ok OF count_ones IS
BEGIN
   PROCESS (din)
      VARIABLE temp: INTEGER RANGE 0 TO 8;
   BEGIN
      temp := 0;
      FOR i IN 0 TO 7 LOOP
         IF (din(i)='1') THEN
            temp := temp + 1;
         END IF;
      END LOOP;
      ones <= temp;
   END PROCESS;
END ok;
```

Now the code is well written

Sequential Code

SIGNAL vs VARIABLE Summary

```
-- Solution 1: using a SIGNAL (not ok) --
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
          y: OUT STD_LOGIC);
END mux;
---------------------------------------
ARCHITECTURE not_ok OF mux IS
   SIGNAL sel : INTEGER RANGE 0 TO 3;
BEGIN
   PROCESS (a, b, c, d, s0, s1)
   BEGIN
      sel <= 0;
      IF (s0='1') THEN sel <= sel + 1;
      END IF;
      IF (s1='1') THEN sel <= sel + 2;
      END IF;
      CASE sel IS
         WHEN 0 => y<=a;
         WHEN 1 => y<=b;
         WHEN 2 => y<=c;
         WHEN 3 => y<=d;
      END CASE;
   END PROCESS;
END not_ok;
---------------------------------------
```

Simulate and compare results!

This is not immediately assigned.

Also, several assignments to signal are present.

Generally, only one assignment is allowed within one PROCESS. So, probably only the last assignment will be taken (sel<=sel+1) or error.

```
-- Solution 2: using a VARIABLE (ok) ----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
          y: OUT STD_LOGIC);
END mux;
---------------------------------------
ARCHITECTURE ok OF mux IS
BEGIN
   PROCESS (a, b, c, d, s0, s1)
      VARIABLE sel : INTEGER RANGE 0 TO 3;
   BEGIN
      sel := 0;
      IF (s0='1') THEN sel := sel + 1;
      END IF;
      IF (s1='1') THEN sel := sel + 2;
      END IF;
      CASE sel IS
         WHEN 0 => y<=a;
         WHEN 1 => y<=b;
         WHEN 2 => y<=c;
         WHEN 3 => y<=d;
      END CASE;
      END PROCESS;
END ok;
---------------------------------------
```

Signals and Variables

SIGNAL vs VARIABLE Summary

	SIGNAL	VARIABLE
Assignment	<=	:=
Utility	Represents circuit interconnects (wires)	Represents local information
Scope	Can be global (seen by entire code)	Local (visible only inside the corresponding PROCESS, FUNCTION, or PROCEDURE)
Behavior	Update is not immediate in sequential code (new value generally only available at the conclusion of the PROCESS, FUNCTION, or PROCEDURE)	Updated immediately (new value can be used in the next line of code)
Usage	In a PACKAGE, ENTITY, or ARCHITECTURE. In an ENTITY, all PORTS are SIGNALS by default	Only in sequential code, that is, in a PROCESS, FUNCTION, or PROCEDURE

Signals and Variables

Efficient use of FPGA resoruces

- A **SIGNAL generates a flip-flop whenever an assignment is made at the transition of another signal** (synchronous assignment. This only happens inside a PROCESS, FUNCTION, or PROCEDURE (e.g. ‘‘IF signal’EVENT . . .’’ or ‘‘WAIT UNTIL . . .’’).

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

Assigment of output1 and output2 requires flipflops as they are assigned during transition of another signal (clk)

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

Assigment of output1 require fliflop but output2 does not!. Output2 Will make use of **only logic gates.**

Signals and Variables

Efficient use of FPGA resoruces

- **A VARIABLE not necessarily generate a flip-flop** (register) if its value never leaves the sequential code.
- A **VARIABLE generate a flip-flop** when a value is assigned to it, based on the transition of another signal, and that value **is passed to a signal that leaves the sequential code.**
- A **VARIABLE** also generate a register when it is used before a value has been assigned to it.

```
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;
```

Temp leaves the sequential code and its vulue was assigned based on transition of another signal

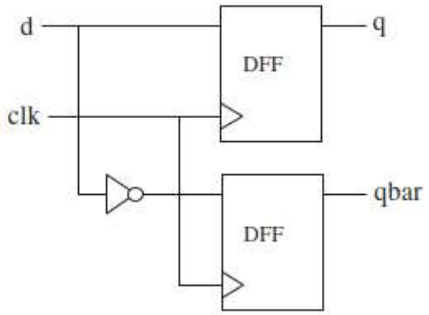
:=

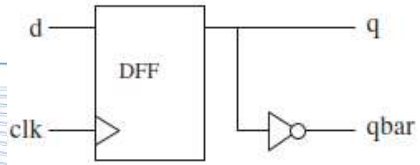
Signals and Variables

Efficient use of FPGA resoruces

Two different solutions for a D-Type Flip-Flop (DFF).
It copies the input to the output at rising or falling Edge of the clock.

```
---- Solution 1: Two DFFs ---------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY dff IS
   PORT ( d, clk: IN STD_LOGIC;
          q: BUFFER STD_LOGIC;
          qbar: OUT STD_LOGIC);
END dff;
-----------------------------------------
ARCHITECTURE two_dff OF dff 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;
-----------------------------------------
```





This SIGNAL assignment is not Synchronous, sparing one register.

```
---- Solution 2: One DFF ----------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY dff IS
   PORT ( d, clk: IN STD_LOGIC;
          q: BUFFER STD_LOGIC;
          qbar: OUT STD_LOGIC);
END dff;
-----------------------------------------
ARCHITECTURE one_dff OF dff 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;
-----------------------------------------
```

END-Signals and Variables
When and where to use them

We study the importance and differences between signals and variables

VHDL
VHSIC HARDWARE
DESCRIPTION LANGUAGE