Shared Variable

When declared as shared, a variable can be accessed by more than one sequential code and also by concurrent code, though only one sequential unit should modify its value.

Additionally, the value of a shared variable can be passed to a signal in an assignment made outside the sequential code. For the carry ripple adder shown below a and b are the input vectors to be added, cin is the carry-in bit, s is the sum vector, and cout is the carry-out bit.

Example: Counter with SHARED VARIABLE

Design a 00-to-99 counter employing shared variables for both digits.

Solution:

The code below implements a circuit that counts from 00 to 99 and then automatically restarts from 00.

Even though a single process would do, two were employed in order to illustrate the use of shared variables (temp1 and temp2, declared in line 8).

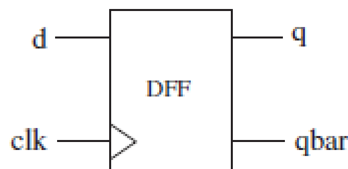
Note that each variable is modified by only one process (proc1 for temp1, proc2 for temp2) and that the passing of their values to signals can be done outside the processes (lines 35–36).

```
1 ---------------------------------------------------
2 ENTITY counter_with_sharedvar IS
3       PORT (clk: IN BIT;
4                 digit1, digit2: OUT INTEGER RANGE 0 TO 9);
5 END ENTITY;
6 ---------------------------------------------------
7 ARCHITECTURE counter OF counter_with_sharedvar IS
8               SHARED VARIABLE temp1, temp2: INTEGER RANGE 0 TO 9;
9 BEGIN
10 ---------------------------------
11 proc1: PROCESS (clk)
12      BEGIN
13              IF (clk'EVENT AND clk='1') THEN
```

```
14                  IF (temp1=9) THEN
15                        temp1 := 0;
16                  ELSE
17                        temp1 := temp1 + 1;
18                  END IF;
19            END IF;
20      END PROCESS proc1;
21 ---------------------------------
22 proc2: PROCESS (clk)
23 BEGIN
24            IF (clk'EVENT AND clk='1') THEN
25                  IF (temp1=9) THEN
26                        IF (temp2=9) THEN
27                              temp2 := 0;
28                        ELSE
29                              temp2 := temp2 + 1;
30                        END IF;
31                  END IF;
32            END IF;
33 END PROCESS proc2;
34 ---------------------------------
35      digit1 <= temp1;
36      digit2 <= temp2;
37 END ARCHITECTURE;
38 -----------------------------------------------------
```

Example: DFF with q and qbar #1

We want to implement the DFF. This DFF has reset and the qbar. The presence of qbar will help understand how an assignment to a SIGNAL is made (recall that a PORT is a SIGNAL by default).



```
1 ---- Solution 1: not OK ---------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
5 ENTITY dff IS
6       PORT ( d, clk: IN STD_LOGIC;
7                   q: BUFFER STD_LOGIC;
8                   qbar: OUT STD_LOGIC);
9 END dff;
10 ---------------------------------------
11 ARCHITECTURE not_ok OF dff IS
12 BEGIN
13      PROCESS (clk)
14              BEGIN
15                      IF (clk'EVENT AND clk='1') THEN
16                              q <= d;
17                              qbar <= NOT q;
18                      END IF;
19 END PROCESS;
20 END not_ok;
21 ---------------------------------------
```

```
1 ---- Solution 2: OK -------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
5 ENTITY dff IS
6       PORT ( d, clk: IN STD_LOGIC;
7                       q: BUFFER STD_LOGIC;
8                   qbar: OUT STD_LOGIC);
9 END dff;
10 ---------------------------------------
11 ARCHITECTURE ok OF dff IS
12 BEGIN
13      PROCESS (clk)
14              BEGIN
15                IF (clk'EVENT AND clk='1') THEN
16                  q <= d;
17                END IF;
18      END PROCESS;
19 qbar <= NOT q;
20 END ok;
21 ---------------------------------------
```

Comments:
In solution 1, the assignments q<=d (line 16) and qbar<=NOT q (line 17) are both synchronous, so their new values will only be available at the conclusion of the PROCESS. This is a problem for qbar, because the new value of q has not propagated yet.

Therefore, qbar will assume the reverse of the old value of q. In other words, the right value of qbar will be one clock cycle delayed, thus causing the circuit not to work correctly.

In solution 2, we have placed qbar<=NOT q (line 30) outside the PROCESS, thus operating as a true concurrent expression.

Example: Bad versus Good Multiplexer
In this example, we will implement the multiplexer. This is, indeed, a classical example regarding the choice of a SIGNAL versus a VARIABLE.

```
1 -- Solution 1: using a SIGNAL (not ok) --
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
5 ENTITY mux IS
6       PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7                   y: OUT STD_LOGIC);
8 END mux;
9 ---------------------------------------
10 ARCHITECTURE not_ok OF mux IS
11 SIGNAL sel : INTEGER RANGE 0 TO 3;
12 BEGIN
13      PROCESS (a, b, c, d, s0, s1)
14              BEGIN
15                      sel <= 0;
16                      IF (s0='1') THEN sel <= sel + 1;
17                      END IF;
18                      IF (s1='1') THEN sel <= sel + 2;
19                      END IF;
20              CASE sel IS
21                      WHEN 0 => y<=a;
22                      WHEN 1 => y<=b;
```

```
23                      WHEN 2 => y<=c;
24                      WHEN 3 => y<=d;
25      END CASE;
26      END PROCESS;
27 END not_ok;
28 ----------------------------------------
```

Comments:
A common mistake when using a SIGNAL is not to remember that it might require a certain amount of time to be updated.

Therefore, the assignment sel <= sel + 1 in the first solution (line 16) will result in one plus whatever value had been previously propagated to sel, for the assignment sel <= 0 (line 15) might not have had time to propagate yet.

The same is true for sel <= sel + 2 (line 18). This is not a problem when using a VARIABLE, for its assignment is always immediate.

A second aspect that might be a problem in solution 1 is that more than one assignment is being made to the same SIGNAL (sel, lines 15, 16, and 18), which might not be acceptable.

Generally, only one assignment to a SIGNAL is allowed within a PROCESS, so the software will either consider only the last one (sel <= sel + 2 in solution 1) or simply issue an error message and stop compilation. Again, this is never a problem when using a VARIABLE.

```
1 -- Solution 2: using a VARIABLE (ok) ----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
5 ENTITY mux IS
6       PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7                 y: OUT STD_LOGIC);
8 END mux;
9 ----------------------------------------
10 ARCHITECTURE ok OF mux IS
11      BEGIN
12      PROCESS (a, b, c, d, s0, s1)
```

```
13          VARIABLE sel : INTEGER RANGE 0 TO 3;
14      BEGIN
15          sel := 0;
16          IF (s0='1') THEN sel := sel + 1;
17          END IF;

18          IF (s1='1') THEN sel := sel + 2;
19          END IF;
20      CASE sel IS
21          WHEN 0 => y<=a;
22          WHEN 1 => y<=b;
23          WHEN 2 => y<=c;
24          WHEN 3 => y<=d;
25 END CASE;
```

Number of Registers

We will discuss the number of flip-flops inferred from the code by the compiler.

The purpose is not only to understand which approaches require less registers, but also to make sure that the code does implement the expected circuit.

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

Such assignment, being synchronous, can only happen inside a PROCESS, FUNCTION, or PROCEDURE (usually following a declaration of the type ‘‘IF signal'EVENT . . .’’ or ‘‘WAIT UNTIL . . .’’).

A VARIABLE, on the other hand, 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.

A VARIABLE also generates a register when it is used before a value has been assigned to it.

Rules for Variables and Signals

Rule 1: Local of Declaration

SIGNAL: Can be declared in the declarative part of ENTITY, ARCHITECTURE, PACKAGE, BLOCK, or GENERATE.

VARIABLE: Can only be declared in sequential units (PROCESS and subprograms). The only exception is for shared variables, which can be declared in the same places as SIGNAL, but should only be modified by one sequential unit.

Rule 2: Scope (Local of Use)

SIGNAL: Can be global (seen and modified in the whole code, including in sequential units).

VARIABLE: Always local (seen and modified only inside the sequential unit where it was created). To leave that unit, its value must be passed directly or indirectly to a signal. The only exception is for a shared variable, which can be global (seen by more than one sequential unit and also by concurrent statements, though it should be modified only by one sequential unit).

Rule 3: Update

SIGNAL: A new value is only available after the conclusion of the present run of the process or subprogram.

VARIABLE: Updated immediately, so its new value is ready to be used in the next line of code.

Rule 4: Assignment Operator

SIGNAL: Values are assigned using "<=" (example: sig <= 5;).

VARIABLE: Values are assigned using ":=" (example: var := 5;).

Rule 5: Multiple Assignments

SIGNAL: Only one effective assignment is allowed in the whole code.

VARIABLE: Because its update is immediate, multiple assignments are fine.

Rule 6: Inference of Registers

SIGNAL: Flip-flops are inferred when an assignment to a signal occurs at the transition of another signal.

VARIABLE: Flip-flops are inferred when an assignment to a variable occurs at the transition of another signal and this variable's value is eventually passed directly or indirectly to a signal.

Rule	SIGNAL	VARIABLE
1. Local of declaration	ENTITY, ARCHITECTURE, BLOCK, GENERATE, and PACKAGE (declaration in sequential code is forbidden)	Only in sequential units (PROCESS and subprograms), except shared variables (declared in ENTITY, ARCHITECTURE, BLOCK, GENERATE, PACKAGE)
2. Scope (local of use and of modification)	Can be global (used and modified anywhere in the code)	Local (used and modified only inside its own sequential unit), except shared variables (can be global, but modified by only one sequential unit)
3. Update	New value available only at the end of the current cycle	Updated immediately (new value ready to be used in the next line of code)
4. Assignment operator	Values are assigned using "<=" Example: sig<=5;	Values are assigned using ":=" Example: var:=5;
5. Multiple assignments	Only one assignment is allowed	Multiple assignments are fine (because update is immediate)
6. Inference of registers	Flip-flops are inferred when an assignment to a signal occurs at the transition of another signal	Flip-flops are inferred when an assignment to a variable occurs at the transition of a signal and this variable's value eventually affects a signal's value

Example: In the process shown below, output1 and output2 will both be stored (that is, infer flip-flops), because both are assigned at the transition of another signal (clk).

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

Example: In the next process, only output1 will be stored (output2 will make use of logic gates).

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

Example: In the process below, temp (a variable) will cause x (a signal) to be stored.

```
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;
```

Example: DFF with q and qbar #2

Let us consider the DFF once again. Both solutions presented below function properly. The difference between them, however, resides in the number of flip-flops needed in each case.

Solution 1 has two synchronous SIGNAL assignments (lines 16–17), so 2 flip-flops will be generated. This is not the case in solution 2, where one of the assignments (line 19) is no longer synchronous.

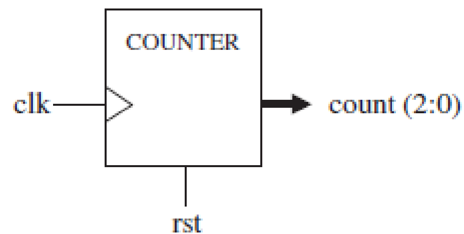
```
 ---- Solution 1: Two DFFs ---------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
```

```
5 ENTITY dff IS
6       PORT ( d, clk: IN STD_LOGIC;
7                 q: BUFFER STD_LOGIC;
8                 qbar: OUT STD_LOGIC);
9 END dff;
10 ---------------------------------------
11 ARCHITECTURE two_dff OF dff IS
12 BEGIN
13      PROCESS (clk)
14      BEGIN
15              IF (clk'EVENT AND clk='1') THEN
16                      q <= d; -- generates a register
17                      qbar <= NOT d; -- generates a register
18              END IF;
19 END PROCESS;
20 END two_dff;
21 ---------------------------------------
```

```
1 ---- Solution 2: One DFF ---------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
5 ENTITY dff IS
6       PORT ( d, clk: IN STD_LOGIC;
7                 q: BUFFER STD_LOGIC;
8                 qbar: OUT STD_LOGIC);
9 END dff;
10 ---------------------------------------
11 ARCHITECTURE one_dff OF dff IS
12 BEGIN
13      PROCESS (clk)
14              BEGIN
15                      IF (clk'EVENT AND clk='1') THEN
16                              q <= d; -- generates a register
17                      END IF;
18      END PROCESS;
19 qbar <= NOT q; -- uses logic gate (no register)
20 END one_dff;
21 ---------------------------------------
```

Example: Counter

Consider the 0-to-7 counter.




```
1 ------ Solution 1: With a VARIABLE --------
2 ENTITY counter IS
3       PORT ( clk, rst: IN BIT;
4       count: OUT INTEGER RANGE 0 TO 7);
5 END counter;
6 -------------------------------------------
7 ARCHITECTURE counter OF counter IS
8       BEGIN
9       PROCESS (clk, rst)
10              VARIABLE temp: INTEGER RANGE 0 TO 7;
11      BEGIN
12              IF (rst='1') THEN
13                      temp:=0;
14              ELSIF (clk'EVENT AND clk='1') THEN
15                      temp := temp+1;
16              END IF;
17 count <= temp;
18 END PROCESS;
19 END counter;
20 -------------------------------------------
```

```
1 ------ Solution 2: With SIGNALS only -------
2 ENTITY counter IS
3       PORT ( clk, rst: IN BIT;
4                       count: BUFFER INTEGER RANGE 0 TO 7);
5 END counter;
6 -------------------------------------------
7 ARCHITECTURE counter OF counter IS
8       BEGIN
```

```
9       PROCESS (clk, rst)
10              BEGIN
11                      IF (rst='1') THEN
12                              count <= 0;
13                      ELSIF (clk'EVENT AND clk='1') THEN
14                              count <= count + 1;
15                      END IF;
16      END PROCESS;
17 END counter;
18 ---------------------------------------------
```

In the first, a synchronous VARIABLE assignment is made (lines 14–15). In the second, a synchronous SIGNAL assignment occurs (lines 13–14).

From either solution, three flip-flops are inferred (to hold the 3-bit output signal count). Solution 1 is an example that a VARIABLE can indeed generate registers. The reason is that its assignment (line 15) is at the transition of another signal (clk, line 14) and its value does leave the PROCESS (line 17).

Solution 2, on the other hand, uses only SIGNALS. Notice that, since no auxiliary signal was used, count needed to be declared as of mode BUFFER (line 4), because it is assigned a value and is also read (used) internally (line 14). Still regarding line 14 of solution 2, notice that a SIGNAL, like a VARIABLE, can also be incremented when used in a sequential code.

Finally, notice that neither in solution 1 nor in solution 2 was the std_logic_1164 package declared, because we are not using std_logic data types in this example.

Making Multiple Signal Assignments

Ex: Parity detector

```
1 ----------------NOT OK-----------------------
2 ENTITY parity_det IS
3 GENERIC (N: POSITIVE := 8);
4       PORT (x: IN BIT_VECTOR(N-1 DOWNTO 0);
5                y: OUT BIT);
6 END ENTITY;
7 --------------------------------------
```

```
8 ARCHITECTURE not_ok OF parity_det IS
9               SIGNAL temp: BIT;
10 BEGIN
11              temp <= x(0);
12              gen: FOR i IN 1 TO N-1 GENERATE
13              temp <= temp XOR x(i);
14              END GENERATE;
15 y <= temp;
16 END ARCHITECTURE;
17 ---------------------------------------
```

Soln-2 (NOT-OK)

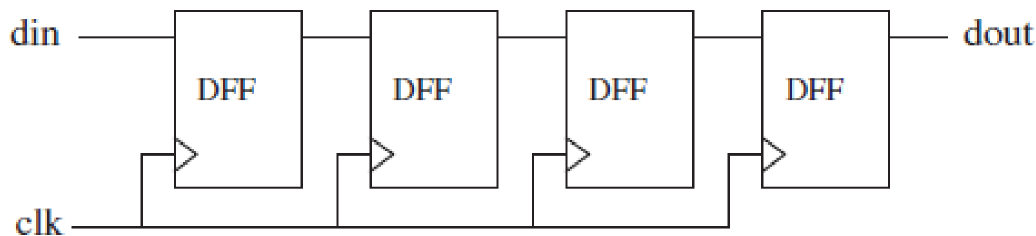
```
7 ----------------------------
8 ARCHITECTURE not_ok OF ...
9       SIGNAL temp: BIT;
10      BEGIN
11      PROCESS (x)
12      BEGIN
13              temp <= x(0);
14              FOR i IN 1 TO N-1 LOOP
15                      temp <= temp XOR x(i);
16              END LOOP;
17 y <= temp;
18 END PROCESS;
19 END ARCHITECTURE;
20 ----------------------------
```

Sol-3 (OK)

```
7 -----------------------------------------
8 ARCHITECTURE ok OF parity_det IS
9       SIGNAL temp: BIT_VECTOR(N-1 DOWNTO 0);
10 BEGIN
11      temp(0) <= x(0);
12      gen: FOR i IN 1 TO N-1 GENERATE
13              temp(i) <= temp(i-1) XOR x(i);
14      END GENERATE;
15 y <= temp(N-1);
16 END ARCHITECTURE;
17 -----------------------------------------
```

Example: Shift Register

We are now interested in examining what happens to the 4-stage shift register when different VARIABLE and SIGNAL assignments are made.




```
1 -------- Solution 1: -----------------
2 ENTITY shift IS
3       PORT ( din, clk: IN BIT;
4                   dout: OUT BIT);
5 END shift;
6 -------------------------------------
7 ARCHITECTURE shift OF shift IS
8 BEGIN
9       PROCESS (clk)
10              VARIABLE a, b, c: BIT;
11      BEGIN
12              IF (clk'EVENT AND clk='1') THEN
13              dout <= c;
14              c := b;
15              b := a;
16              a := din;
17              END IF;
18      END PROCESS;
19 END shift;
20 -------------------------------------
```

In solution 1, three VARIABLES are used (a, b, and c, line 10). However, the variables are used before values are assigned to them (that is, in reverse order, starting with dout, line 13, and ending with din, line 16). Consequently, flip-flops will be inferred, which store the values from the previous run of the PROCESS.

```
1 -------- Solution 2: -----------------
2 ENTITY shift IS
3       PORT ( din, clk: IN BIT;
4                 dout: OUT BIT);
5 END shift;
6 --------------------------------------
7 ARCHITECTURE shift OF shift IS
8       SIGNAL a, b, c: BIT;
9 BEGIN
10      PROCESS (clk)
11      BEGIN
12              IF (clk'EVENT AND clk='1') THEN
13                      a <= din;
14                      b <= a;
15                      c <= b;
16                      dout <= c;
17              END IF;
18      END PROCESS;
19 END shift;
20 --------------------------------------
```

In solution 2, the variables were replaced by SIGNALS (line 8), and the assignments are made in direct order (from din to dout, lines 13–16). Since signal assignments at the transition of another signal do generate registers, here too the right circuit will be inferred. However, the program is not OK in this way of writing.

```
1 -------- Solution 3: -----------------
2 ENTITY shift IS
3       PORT ( din, clk: IN BIT;
4                 dout: OUT BIT);
5 END shift;
6 -------------------------------------
7 ARCHITECTURE shift OF shift IS
8       BEGIN
9             PROCESS (clk)
10             VARIABLE a, b, c: BIT;
11            BEGIN
12                  IF (clk'EVENT AND clk='1') THEN
13                        a := din;
14                        b := a;
15                        c := b;
16                        dout <= c;
17                  END IF;
18            END PROCESS;
19 END shift;
20 -------------------------------------
```

In solution 3, the same variables of solution 1 were employed, but in direct order (from din to dout, lines 13–16). Recall, however, that an assignment to a variable is immediate, and since the variables are being used in direct order (that is, after values have been assigned to them), lines 13–15 collapse into one line, equivalent to c := din. The value of c does leave the process in the next line (line 16), however, where a signal assignment (dout <= c) occurs at the transition of clk.

Therefore, one register will be inferred from solution 3, thus not resulting the correct circuit.