CPE 462
VHDL: Simulation and Synthesis

Topic #06 - e) Signals and Variables

SIGNAL

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

Examples:

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

- When SIGNAL, when used inside a section of sequential code (PROCESS, for example), its update is not immediate.
- Be careful when multiple assignments are made to the same SIGNAL. The compiler might complain and quit synthesis, or might infer the wrong circuit.

Count Ones (not OK)

- Since the value of a signal is not updated immediately, line 18 conflicts with line 15, for the value assigned in line 15 might not be ready until the conclusion of the PROCESS, in which case a wrong value would be computed in line 18.
- Use a variable in these situations!

```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY count_ones IS
6     PORT ( din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            ones: OUT INTEGER RANGE 0 TO 8);
8  END count_ones;
9  ----------------------------------------
10 ARCHITECTURE not_ok OF count_ones IS
11    SIGNAL temp: INTEGER RANGE 0 TO 8;
12 BEGIN
13    PROCESS (din)
14    BEGIN
15       temp <= 0;
16       FOR i IN 0 TO 7 LOOP
17          IF (din(i)='1') THEN
18             temp <= temp + 1;
19          END IF;
20       END LOOP;
21       ones <= temp;
22    END PROCESS;
23 END not_ok;
24 ----------------------------------------
```

Variable

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

Examples:

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

- Since a VARIABLE can only be used in sequential code, its declaration can only be done in the declarative part of a PROCESS, FUNCTION, or PROCEDURE.
- Recall that the assignment operator for a VARIABLE is ":=". Also, like in the case of a SIGNAL, the initial value in the syntax above is not synthesizable, being only considered in simulations.

Count Ones (OK)

- The only difference in the solution is that an internal VARIABLE is employed instead of a SIGNAL.
- Since the update of a variable is immediate, the initial value is established correctly and no complains regarding multiple assignments will be issued by the compiler.

```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY count_ones IS
6     PORT ( din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            ones: OUT INTEGER RANGE 0 TO 8):
8  END count_ones;
9  ----------------------------------------
10 ARCHITECTURE ok OF count_ones IS
11 BEGIN
12    PROCESS (din)
13       VARIABLE temp: INTEGER RANGE 0 TO 8;
14    BEGIN
15       temp := 0;
16       FOR i IN 0 TO 7 LOOP
17          IF (din(i)='1') THEN
18             temp := temp + 1;
19          END IF;
20       END LOOP;
21       ones <= temp;
22    END PROCESS;
23 END ok;
24 ----------------------------------------
```

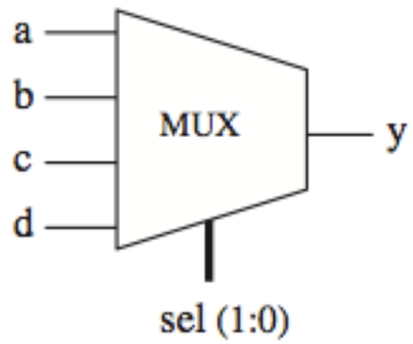
Signals vs Variables

Comparison between SIGNAL and VARIABLE.

	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

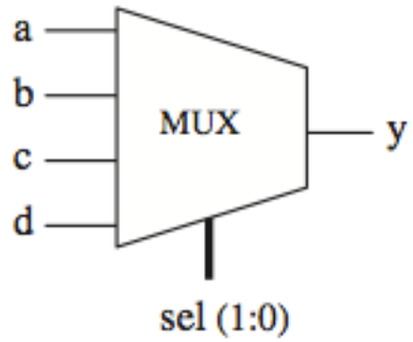
Bad Multiplexer

```
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 ---------------------------------------
```

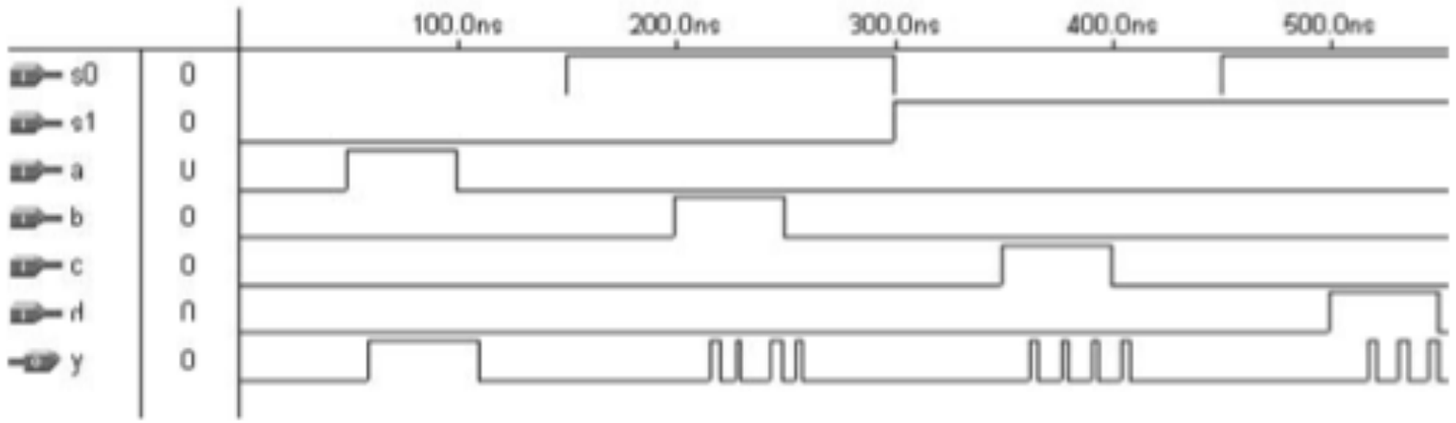


- A common mistake when using a SIGNAL is not to remember that it might require a certain amount of time to be updated.
- Therefore, the assignment sel <=sel + 1 (line 16) will result in one plus whatever value had been previously propagated to sel, for the assignment sel <=0 (line 15) might not have had time to propagate yet. The same is true for sel <= sel +2 (line 18).

Bad Multiplexer

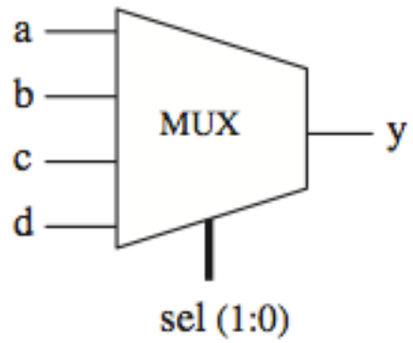



```
1  -- Solution 1: using a SIGNAL (not ok) --
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7            y: OUT STD_LOGIC);
8  END mux;
9  ---------------------------------------
10 ARCHITECTURE 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 ---------------------------------------
```

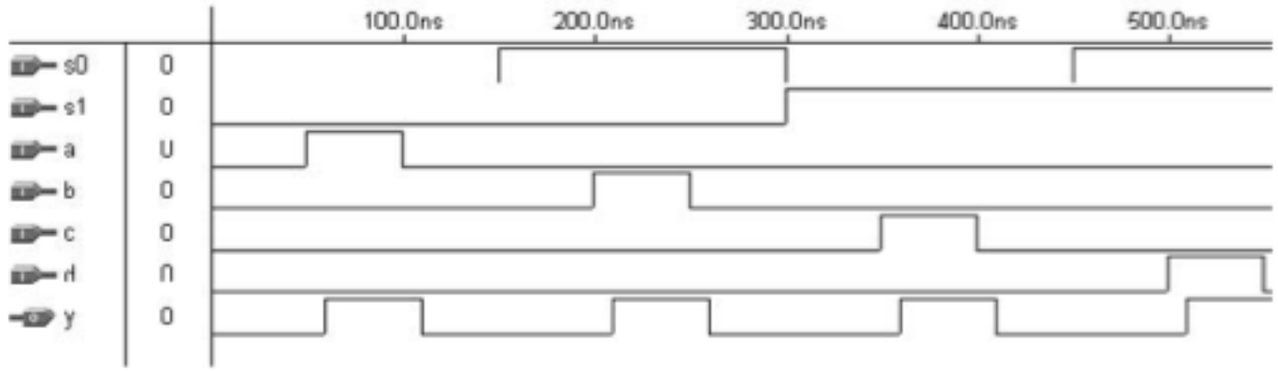


Good Multiplexer

```
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;
26    END PROCESS;
27 END ok;
28 ----------------------------------------
```



- There are no problems with a VARIABLE, for all assignments are always immediate.



Number of Registers

- A SIGNAL generates a flip-flop whenever an assignment is made at the transition of another signal.
- Such assignments can only happen inside a PROCESS, FUNCTION, or PROCEDURE.
- 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.

Example: Two inferred Flip-Flops

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: Single inferred Flip-Flop

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;
```

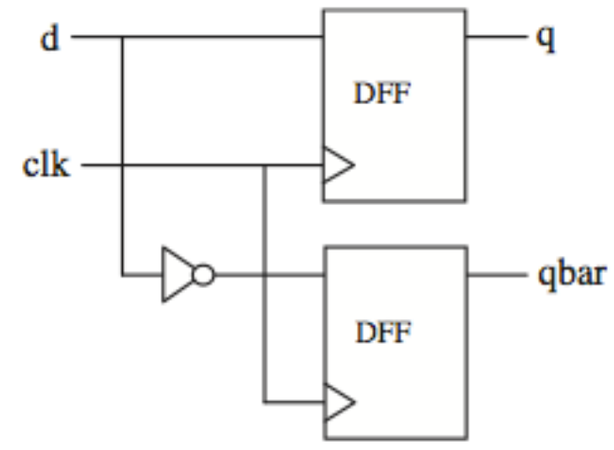
A variable causing a signal to be stored.

- In the process below, temp (a variable) will cause x (a signal) to be stored.
- Variable temp will leave the process... so it will infer a Flip-Flop!

```
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;
```

DFF with q and qbar

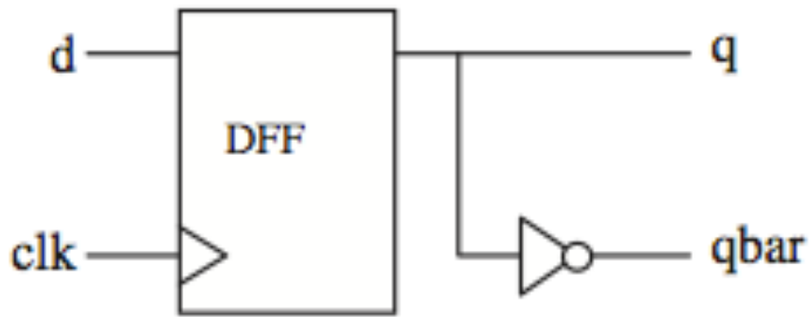
```
1  ---- Solution 1: Two DFFs ---------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY dff IS
6     PORT ( d, clk: IN STD_LOGIC;
7            q: BUFFER STD_LOGIC;
8            qbar: OUT STD_LOGIC);
9  END dff;
10 ---------------------------------------
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 ---------------------------------------
```



- Two FF are generated in hardware...
- Who cares? We do! We want to minimize our design area!

DFF with q and qbar

```
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 ---------------------------------------
```



Single FF is inferred!