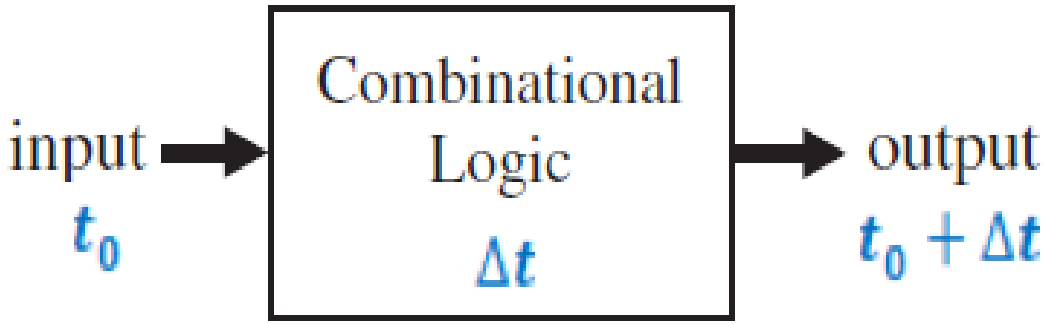
VHDL - Sequential Code (PROCESS elements)

Introduction

- VHDL code can be concurrent (combinational logic) or sequential (sequential logic).
- **Combinational logic - definition:**

  The output is a pure function of the present input only (implemented by Boolean circuits, using conventional logic gates only – no memory, no feedback).
- Intuitively, the circuit information flows in parallel.

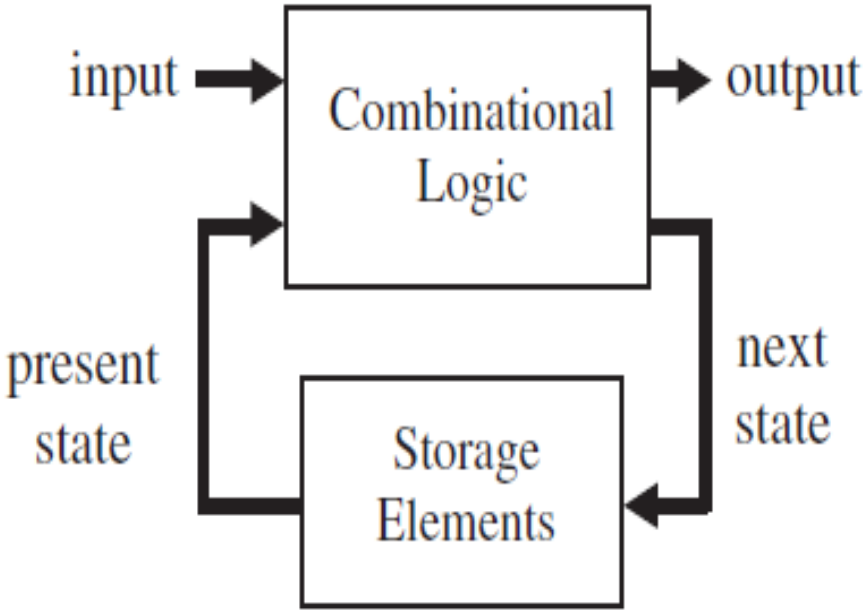


Introduction

- **Sequential logic - definition**:

  The output does depend on present inputs and previous inputs (implemented using storage, flip-flops elements, which are connected to the combinational logic block through a feedback loop).

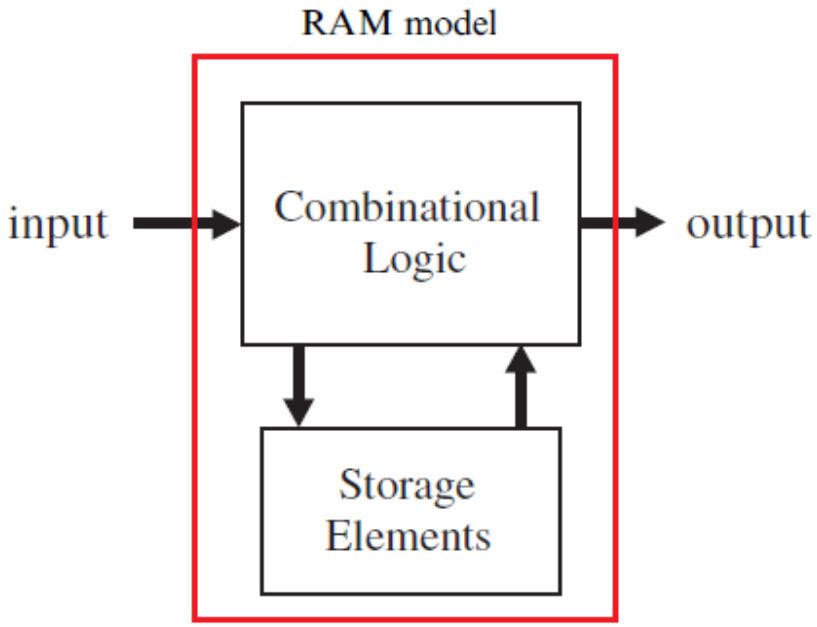
- Intuitively, the circuit information flows in serial triggered by clk signal.



Introduction

- **Note**: not any circuit that possesses storage elements is sequential.
- Example: RAM memory.

  The storage elements appear in a forward path rather than in a feedback loop. The memory-read operation depends only on the present address vector input (with nothing to do with previous memory accesses).



Introduction

- There are four types of **ARCHITECTURE** Modeling styles:
  - ✓ Dataflow modeling = Concurrent Code
  - ✓ Structural modeling = Concurrent Code
  - ✓ **Behavioral modeling = Sequential Code**
  - ✓ Mixed modeling = Concurrent Code

Sequential (behavioral) Code

- VHDL code is inherently concurrent but there are three sequential design units **PROCESSES**, **FUNCTIONS** and **PROCEDURES** are the only sections of code that are executed sequentially. However, as a whole, any of these blocks is still concurrent with any other statements placed outside it.
- One important aspect of sequential code is that it is not limited to sequential logic. Indeed, with it we can build sequential circuits as well as combinational circuits (but not mixed together, which called Mixed PROCESS).
- Sequential statements are allowed only inside **PROCESSES**, **FUNCTIONS**, or **PROCEDURES** as concurrent statements are allowed only outside of them.
- In this section we will concentrate on **PROCESSES** only. The design units **FUNCTIONS** and **PROCEDURES** are very similar, but are intended for system-level design (will be discussed later).

PROCESS

- A **PROCESS** is a sequential section of VHDL code. It is placed only in the ARCHITECTURE body (after BEGIN) and can contain the sequential statements **IF, WAIT, CASE, LOOP** (in addition to signals assignment) and use of **VARIABLES**.
- Syntax:

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

- The use of a label (improves code readability) and **VARIABLES** are optional.
- The **VARIABLES** initial value is not synthesizable (for simulation only).

PROCESS

- Syntax:

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

- A **PROCESS** is executed every time a signal in the sensitivity list changes (on an EVENT occurrence) or the condition related to WAIT is fulfilled.

```
proc1 : PROCESS (a,b)
    BEGIN
        sequential statements;
END PROCESS proc1 ;
```

```
proc2 : PROCESS
    BEGIN
        WAIT ON (a,b);
        sequential statements;
END PROCESS proc2 ;
```

- PROCESS as a code element is performed as a single concurrent statement inside ARCHITECTURE body, thus using of two PROCESS (or more) inside ARCHITECTURE constitute using of two (or more) concurrent statements.

Signals assignment inside **PROCESS**

- A **PROCESS** is a sequential section of VHDL code. It is placed only in the ARCHITECTURE body (after BEGIN) and can contain the sequential statements **IF, WAIT, CASE, LOOP** (in addition to signals assignment).
- Signals assignment:

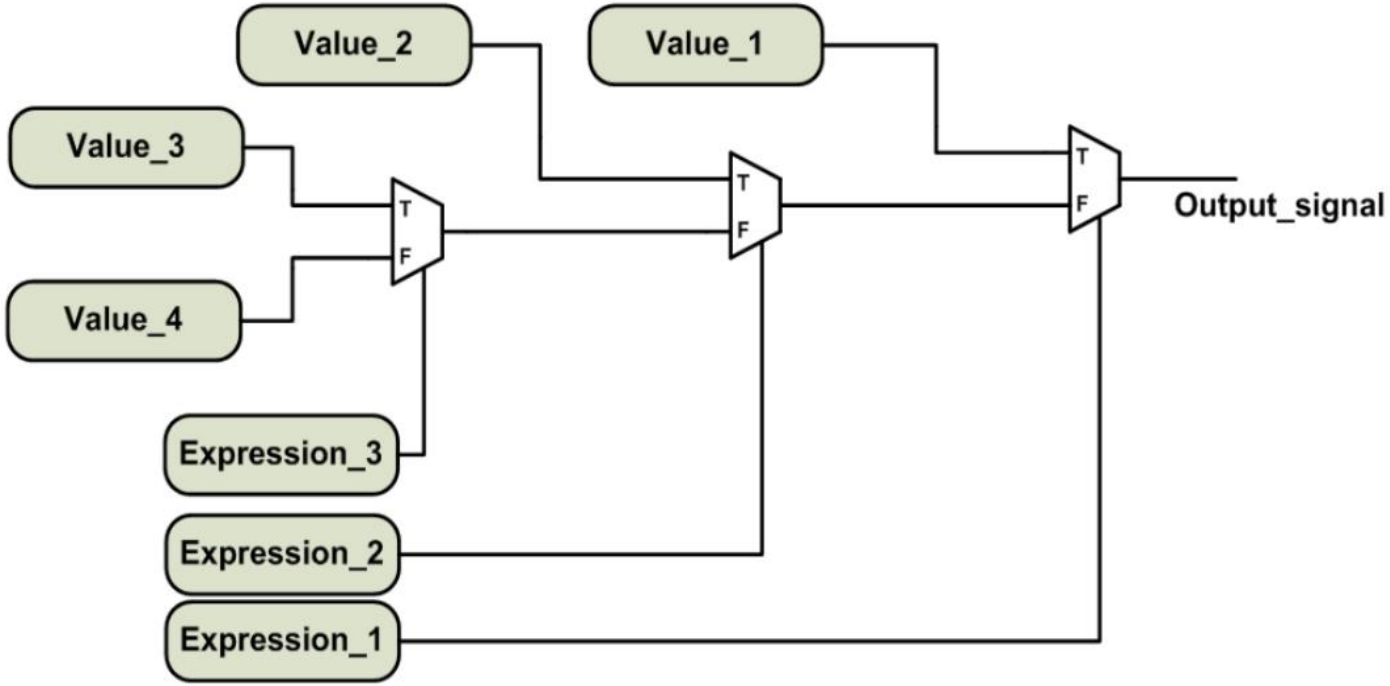
```
signal_name <= expression;
```

- When **SIGNAL** is used in a **PROCESS**, its new value is generally only guaranteed to be available after the conclusion (different between combinatorial or sequential PROCESS) of the present run of the **PROCESS.**
- In case of multiple assignments to the same SIGNAL, only the last assignment is taken into consideration by the compiler, the rest are ignored (differ from the case of concurrent code – multiple driven).

Sequential statement IF-THEN

Syntax:

```
process(all_inputs)
begin
    if Expression_1 then
        Output_signal <= Value1;
    elsif Expression_2 then
        Output_signal <= Value2;
    elsif Expression_3 then
        Output_signal <= Value3;
    else
        Output_signal <= Value4;
    end if;
end process;
```



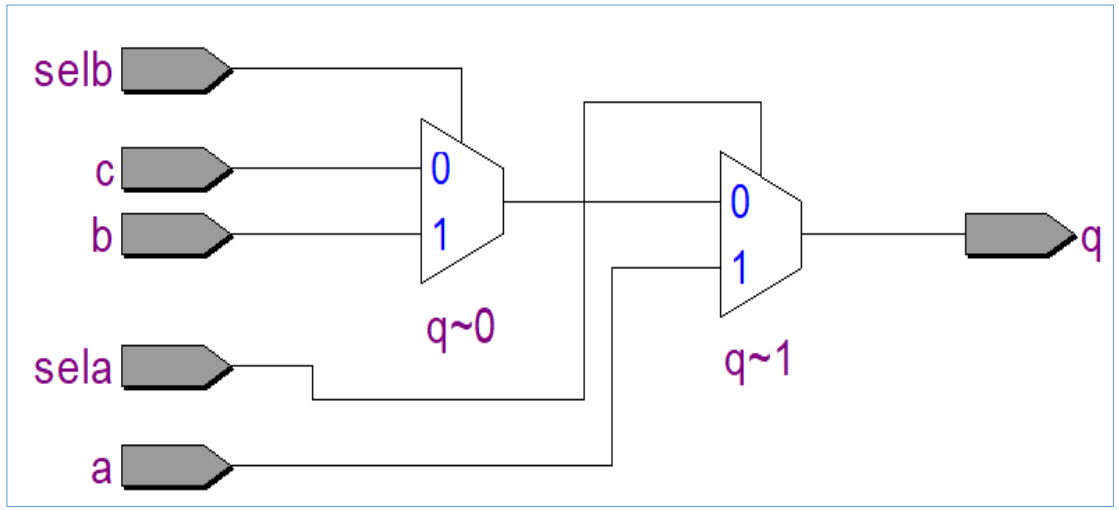
Notes:

- When the command contains conflicting terms, the first of them will be executed
- Multiple separate IF-THEN statement contains assignment to the same SIGNAL must be avoided (causes sick hardware).

Sequential statement IF-THEN

- Example:

```
PROCESS (sela, selb, a, b, c)
BEGIN
     IF  sela='1' THEN
         q <= a ;
     ELSIF  selb='1'  THEN
         q <= b ;
     ELSE
         q <= c ;
     END IF;
END PROCESS ;
```

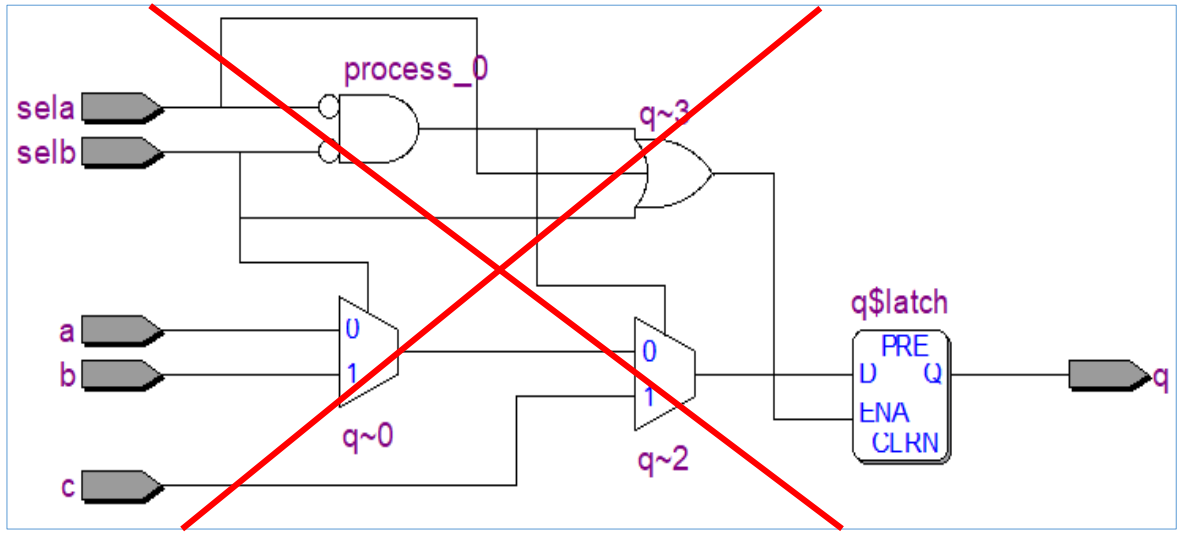


- Example – Sick Hardware:

```
PROCESS (sela, selb, a, b, c)
BEGIN
     IF  sela='1' THEN
         q <= a ;
     END IF;

     IF  selb='1'  THEN
         q <= b ;
     END IF;

     IF sela='0' and selb='0' THEN
         q <= c ;
     END IF;
END PROCESS ;
```

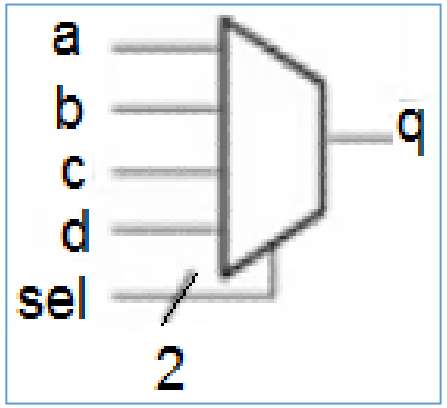


Sequential statement CASE

Syntax:

```
CASE  expression IS
     WHEN condition1 =>
          -- sequential statements
     WHEN condition2 =>
          -- sequential statements
     WHEN OTHERS =>
          -- sequential statements
END CASE;
```

Example:




```
PROCESS (sel, a, b, c, d)
BEGIN
      CASE  sel IS
          WHEN "00" =>
                q <= a;
          WHEN "01" =>
                q <= b;
          WHEN "10" =>
                q <= c;
          WHEN OTHERS =>
                q <= d;
      END CASE ;
END PROCESS ;
```

CASE (sequential) vs WHEN (concurrent)

- For both of them all permutations must be tested, so the keyword **OTHERS** is often helpful (differ from **IF-THEN** statement). This means that both will be synthesized mux based.
- Another important keyword is **NULL** for **CASE** (for **WHEN** is **UNAFFECTED**), which should be used when no action is to take place.

	WHEN	CASE
Statement type	Concurrent	Sequential
Usage	Only outside PROCESSES, FUNCTIONS, or PROCEDURES	Only inside PROCESSES, FUNCTIONS, or PROCEDURES
All permutations must be tested	Yes for WITH/SELECT/WHEN	Yes
Max. # of assignments per test	1	Any
No-action keyword	UNAFFECTED	NULL

Sequential statements LOOP, FOR ,WHILE

- As the name says, LOOP kinds is useful when a piece of code must be instantiated several times (the compiler unfold the inner body loop statements statically), can only be used inside a **PROCESS**, **FUNCTION**, or **PROCEDURE**.
- **FOR / LOOP**:
  - ✓ The loop is repeated a fixed number of times, the loop limits of the range must be static (0 **TO** 5 or 5 **DOWNTO** 0).
  - ✓ Using of label is optional.

```
label : FOR identifier IN range LOOP
    sequential statements;
END LOOP label;
```

```
FOR i IN 0 TO 5 LOOP
    x(i) <= enable AND w(i+2);
    y(0, i) <= w(i);
END LOOP;
```

Sequential statements LOOP, FOR ,WHILE

- **WHILE / LOOP**:
  - ✓ The loop is repeated until a condition no longer holds.
  - ✓ Using of label is optional.

```
label : WHILE condition LOOP
    sequential statements;
END LOOP label;
```

```
Shift: process (Input_X)
    variable i : NATURAL;
begin
    i := 0;
    while i <= 8 loop
        Output_X(i) <= Input_X(i+8);
        i := i + 1;
    end loop;
end process Shift;
```

Sequential statements LOOP, FOR ,WHILE

- **EXIT**: Used for ending the loop.
  - ✓ Syntax:

```
loop_label: EXIT loop_label WHEN condition;
```

  - ✓ Example:

```
    i := 0;
L1: while i >= 0 loop
    Output_X(i) <= Input_X(i+8);
    i := i + 1;
    exit L1 when i > 8;
    end loop L1;
```

- **NEXT**: Used for skipping loop steps.
  - ✓ Syntax:

```
loop_label: NEXT loop_label WHEN condition;
```

  - ✓ Example:

```
    i := 0;
L1: while i <= 8 loop
    Output_X(i) <= Input_X(i+8);
    i := i + 1;
    next L1 when i = 4; -- skip to the next iteration
    end loop L1;
```

Sequential statements LOOP, FOR ,WHILE

- **LOOP**:
  - ✓ The loop is repeated forever until a **EXIT** condition holds.
  - ✓ Using of label is optional.

```
loop_label: LOOP
    sequential statements;
    NEXT  loop_label  WHEN condition1; --optional
    EXIT  loop_label  WHEN condition2;
END LOOP loop_label;
```

```
    i := 0;
L2: loop
    Output_X(i) <= Input_X(i+8);
    i:= i+1;
    exit L2 when i > 8;
    end loop L2;
```

Sequential statement **WAIT** (for simulation generally)

- When **WAIT** is employed the **PROCESS** cannot have a sensitivity list.
- There are two synthesizable forms of **WAIT** (**WAIT UNTIL** and **WAIT ON**).
- **WAIT UNTIL:**
  - ✓ Syntax:

```
WAIT UNTIL signal_condition;
```

```
WAIT UNTIL (clk'EVENT AND clk='1');
```

  - ✓ The **WAIT UNTIL** statement accepts only one signal (thus being more appropriate for synchronous code).
  - ✓ Since the PROCESS has no sensitivity list in this case, **WAIT UNTIL** must be the first statement in the PROCESS instead.

Sequential statement **WAIT** (for simulation generally)

- **WAIT ON:**
  - ✓ Syntax:

```
WAIT ON signal_list;
```

```
WAIT ON clk,rst;
```

  - ✓ The **WAIT ON**, accepts multiple signals.
  - ✓ Since the PROCESS has no sensitivity list in this case, **WAIT ON** must be the first statement in the PROCESS instead.
  - ✓ The PROCESS is put on hold until any of the signals listed changes (on EVENT occurrence).

Sequential statement WAIT (for simulation generally)

- Examples:

```
proc1 : PROCESS
    BEGIN
        WAIT UNTIL (clk'EVENT AND clk='1');
        sequential statements;
END PROCESS proc1;
```

```
proc2 : PROCESS
    BEGIN
        WAIT ON (a,b);
        sequential statements;
END PROCESS proc2;
```

WAIT FOR <time_expression> - simulation only

- **WAIT FOR** is intended for simulation only (intended for waveform generation for testbenches).
- Examples:

✓ Stop the process until the 5ns elapsed.

```
WAIT FOR 5ns;
```

```
tb_b : process
begin
   b <= '0';
   wait for 50 ns;
   b <= not b;
   wait for 50 ns;
end process tb_b;
```

```
tb_b : process
begin
   b <= '0';
   for i in 0 to 7 loop
      wait for 50 ns;
      b <= not b;
   end loop;
   wait;
end process tb_b;
```

✓ Can be combined: Stop the process until the condition holds for 5µs elapsed.

```
WAIT UNTIL (a='1') FOR 5us;
```

Signals and Variables

- A **SIGNAL** can be declared in a **PACKAGE**, **ENTITY** or **ARCHITECTURE** (in its declarative part), while a **VARIABLE** can only be declared inside a piece of sequential code (in a **PROCESS**, **FUNCTION**, **PROCEDURE**). Therefore, while the value of **SIGNAL** can be global, the value of **VARIABLE** is always local.

- VHDL has two ways of passing non-static values, by means of a **SIGNAL** or a **VARIABLE**. The value of a **VARIABLE** can never be passed out of the PROCESS directly, if necessary, then it must be assigned to a **SIGNAL**.

- The update of a **VARIABLE** is immediate (we can promptly count on its new value in the next line of code). When **SIGNAL** is used in a **PROCESS**, its new value is generally only guaranteed to be available after the conclusion of the present run of the **PROCESS**.

VARIABLE

- Contrary to **CONSTANT** and **SIGNAL**, a **VARIABLE** represents only local information. It can only be used inside a **PROCESS**, **FUNCTION**, or **PROCEDURE** (its declaration can only be done in the declarative part), and its value can not be passed out directly.
- **VARIABLE** update is immediate, so the new value can be promptly used in the next line of code (When **SIGNAL** is used in a **PROCESS**, its new value is generally only guaranteed to be available after the conclusion of the present run of the **PROCESS**).
- **VARIABLE** is a local element, its main purpose is for Intermediate calculations (for simulation. Vanished in synthesis - uses as a “glue” between **SIGNALS**, that is wires) inside **PROCESS.**
- We can not track **VARIABLE** in simulation.

VARIABLE

- **VARIABLE** declaration (inside the PROCESS's declarative part):

```
VARIABLE  name : data_type := initial_value;
```

like in the case of a **SIGNAL**, the initial value of **VARIABLE** is not synthesizable, being only considered in simulation.

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

- **VARIABLE** use (inside the PROCESS's body):

```
temp := "10101010" ;
temp := x"AA" ;
temp(7) := '1' ;
temp(7 DOWNTO 4) := "1010" ;
```

Rule of using **VARIABLES** in PROCESS

- Do not use SIGNALS for intermediate calculations, use only VARIABLES for that purpose.
- The Rule - Separate the PROCESS into three sub parts:
  - ✓ Upper part:

    Set **VARIABLES** value (from **constants** or **SIGNALS**).
  - ✓ Middle part:

    Intermediate calculations using those **VARIABLES.**
  - ✓ Lower part:

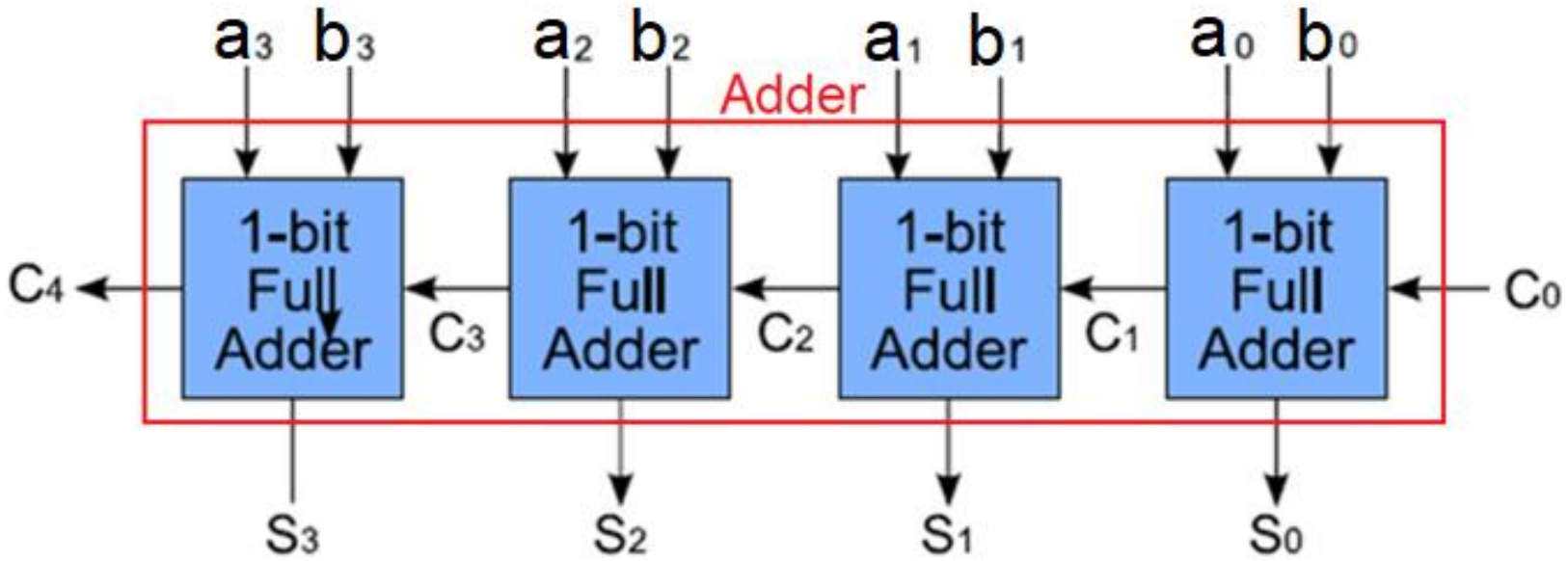
    SIGNALS assignments using those **VARIABLES.**

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

Carry Ripple Adder – Behavioral modeling

- Solution 1 – using **VARIABLES** of type STD_LOGIC_VECTOR
- Solution 2 – using **VARIABLES** of type INTEGER



$$s_j = a_j \; XOR \; b_j \; XOR \; c_j$$

$$c_{j+1} = (a_j \; AND \; b_j) \; OR \; (a_j \; AND \; c_j) \; OR \; (b_j \; AND \; c_j)$$

Carry Ripple Adder – solution 1

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  ---------------------------------------
4  ENTITY adder IS
5    GENERIC (length : INTEGER := 8);
6    PORT ( a, b: IN STD_LOGIC_VECTOR (length-1 DOWNTO 0);
7           cin: IN STD_LOGIC;
8             s: OUT STD_LOGIC_VECTOR (length-1 DOWNTO 0);
9          cout: OUT STD_LOGIC);
10 END adder;
11 ------------------------------------------------
12 ARCHITECTURE adder OF adder IS
13 BEGIN
14   PROCESS (a, b, cin)
15     VARIABLE carry : STD_LOGIC_VECTOR (length DOWNTO 0);
16   BEGIN
17     carry(0) := cin;
18     FOR i IN 0 TO length-1 LOOP
19       s(i) <= a(i) XOR b(i) XOR carry(i);
20       carry(i+1) := (a(i) AND b(i)) OR (a(i) AND
21       carry(i)) OR (b(i) AND carry(i));
22     END LOOP;
23     cout <= carry(length);
24   END PROCESS;
25 END adder;
```

Set **VARIABLES** value

Intermediate calculations using those **VARIABLES**

SIGNALS assignments using those **VARIABLES**

Carry Ripple Adder – solution 2

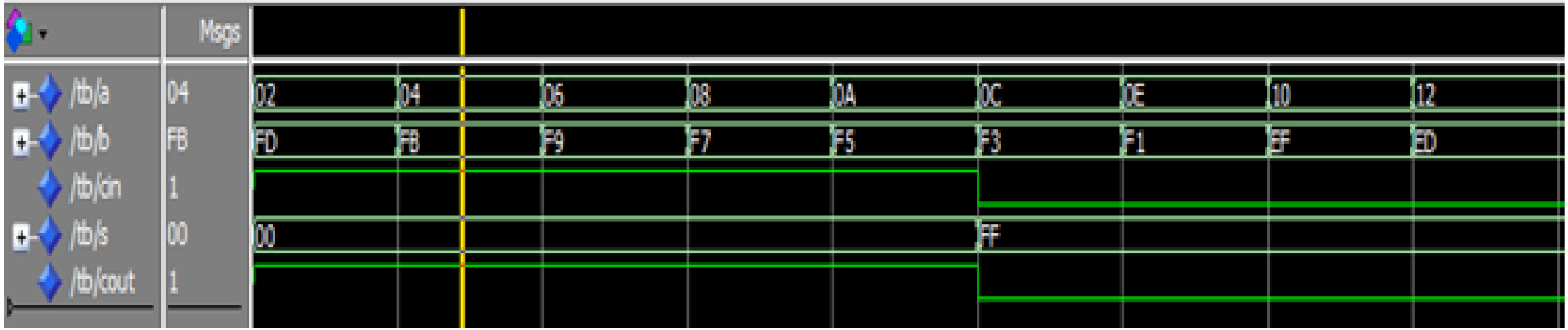
```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  ---------------------------------------
4  ENTITY CarryRippleAdderSol2 IS
5    GENERIC (length : INTEGER := 8);
6    PORT ( a, b: IN INTEGER RANGE 0 TO 2**length-1; -- same as RANGE 0 TO 255
7           cin: IN STD_LOGIC;
8             s: OUT INTEGER RANGE 0 TO 2**length-1; -- same as RANGE 0 TO 255
9          cout: OUT STD_LOGIC);
10 END CarryRippleAdderSol2;
11 ------------------------------------------------
12 ARCHITECTURE adder OF CarryRippleAdderSol2 IS
13 BEGIN
14   PROCESS (a, b, cin)
15     VARIABLE temp : INTEGER RANGE 0 TO 2**(length+1)-1; -- same as RANGE 0 TO 511
16   BEGIN
17     IF (cin='1') THEN temp:=1;
18     ELSE temp:=0;
19     END IF;
20     temp := a + b + temp;
21     IF (temp > 2**length-1) THEN
22        cout <= '1';
23        temp := temp-2**length; -- temp := temp-256, to get 8-bit only
24     ELSE cout <= '0';
25     END IF;
26     s <= temp;
27   END PROCESS;
28 END adder;
```

Set **VARIABLES** value

Intermediate calculations using those **VARIABLES**

SIGNALS assignments using those **VARIABLES**

Carry Ripple Adder – Simulation Results



- Given: the sum of **a** and **b** solely, is equal to 0xFF
- For cin=1, the 9-bit output [s=0x00, cout=1] equal to 256
- For cin=0, the 9-bit output [s=0xFF, cout=0] equal to 255