Sequential Code
PROCESS, FUNCTIONS, PROCEDURES

We study the statements required to create a sequential code.

VHDL

VHSIC HARDWARE DESCRIPTION LANGUAGE

Sequential Code
Introduction

- **Sequential** code is not limited to sequential logic
- **A Sequential** block (code) is still executed concurrently with the rest of the code (outside the sequential part of the code)
- A sequential code is writing inside a **PROCESS**, **FUNCTION** or **PROCEDURE** (we see functions and procedures in next section) using the sequential statements **IF**, **WAIT**, **CASE** and **LOOP.**
- **VARIABLES** can be used only in sequential code.
- A **VARIABLE** can not be global. Therefore, its value can not be passed out directly.

Sequential Code

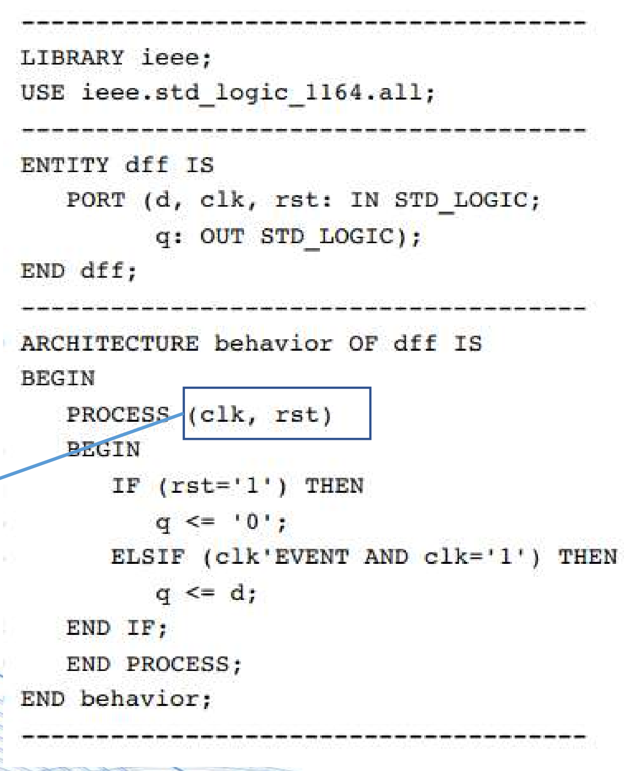
PROCESS

- Initializing **PROCESS** indicates that a piece of sequential code will be written.
- Sequential code is characterized by containing IF, WAIT, CASE and/or LOOP statements and by a sensitivity list.
- A **PROCESS** is executed every time a signal in the sensitivity list changes. (or the condition of WAIT is fulfilled)
- Syntax of **PROCESS** is:

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

- VARIABLES are optional and its initial value is not synthesizable.
- Use of label is also optional. It is useful for code readability.

PROCESS is executed when clk or rst changes



```
---------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY dff IS
   PORT (d, clk, rst: IN STD_LOGIC;
         q: OUT STD_LOGIC);
END dff;
---------------------------------------
ARCHITECTURE behavior OF dff IS
BEGIN
   PROCESS (clk, rst)
   BEGIN
      IF (rst='1') THEN
         q <= '0';
      ELSIF (clk'EVENT AND clk='1') THEN
         q <= d;
   END IF;
   END PROCESS;
END behavior;
---------------------------------------
```

Sequential Code
PROCESS-SIGNALS and VARIABLES

It is important to note the differences between SIGNAL and VARIABLE:

- To pass non-static values within and between circuits we can use SIGNALS or VARIABLES
- A **SIGNAL** can be declared in **PACKAGE**, **ENTITY** and **ARCHITECTURE.** It can be global.
- A **VARIABLE** can be declared only in a **PROCESS**. It is only local. (it can not go out of the PROCESS directly).
- To send out of process a VARIABLE, we need to assign its value to a SIGNAL.
- VARIABLES are immediately updated. Its new value can be used in next line of code. A signal within a PROCESS can only guarantee its updated value the next time the code enters to the PROCESS.
- Remember that assignment for SIGNAL is "<=" . Assignment for VARIABLE is ":=".

Sequential Code

PROCESS-IF

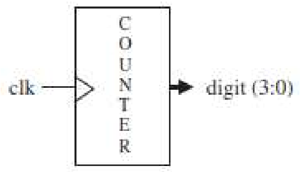
- Syntax of **IF/ELSE** is:

```
IF conditions THEN assignments;
        ELSIF conditions THEN assignments;
        ...
        ELSE assignments;
END IF;
```

- Example:

```
IF (x<y) THEN temp:="11111111";
ELSIF (x=y AND w='0') THEN temp:="11110000";
ELSE temp:=(OTHERS =>'0');
```

Make a 1-digit counter (0 to 9) with 4bit output and a clk signal as input. Counter resets automatically (from 9 to 0).



```
---------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY counter IS
   PORT (clk : IN STD_LOGIC;
         digit : OUT INTEGER RANGE 0 TO 9);
END counter;
---------------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
   count: PROCESS(clk)
      VARIABLE temp : INTEGER RANGE 0 TO 10;
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         temp := temp + 1;
         IF (temp=10) THEN temp := 0;
         END IF;
      END IF;
      digit <= temp;
   END PROCESS count;
END counter;
---------------------------------------------
```

Note that we are comparing against a constant. This is made by a simple and cheap comparator (very good for saving FPGA resources). If a programable parameter is used instead of a constant, a full comparator is required, which uses much more resources of FPGA. Only use that when needed!

Sequential Code
PROCESS-WAIT

- Operator **WAIT** is similar to IF.
- **PROCESS** waits until its condition is fulfilled.
- When **WAIT** is used, no sensitivity list is allowed in **PROCESS**.
- There are three forms of **WAIT**, its syntaxis are:

❑ **WAIT UNTIL** signal_condition;	// Only one signal can be used.
❑ **WAIT ON** signal1 [, signal2, ...];	// Accepts multiple SIGNALS. Process hold until any of the signal changes.
❑ **WAIT FOR** time;	//Only for simulations (e.g. WAIT FOR 5ns;)

- WAIT UNTIL is more appropriate for Synchronous code (governed by a clock) as only one SIGNAL is accepted. WAIT ON is more appropriated for Asynchronous code, which output depends on changes in the input.
- For both cases, PROCESS is hold until signal condition is met (WAIT UNTIL) or signals change (WAIT ON).

```
PROCESS          -- no sensitivity list
BEGIN
   WAIT UNTIL (clk'EVENT AND clk='1');
   IF (rst='1') THEN
      output <= "00000000";
   ELSIF (clk'EVENT AND clk='1') THEN
      output <= input;
   END IF;
END PROCESS;
```

```
PROCESS
BEGIN
   WAIT ON clk, rst;
   IF (rst='1') THEN
      output <= "00000000";
   ELSIF (clk'EVENT AND clk='1') THEN
      output <= input;
   END IF;
END PROCESS;
```

Sequential Code
PROCESS-CASE

- Operator **CASE** syntax is:

  CASE identifier IS

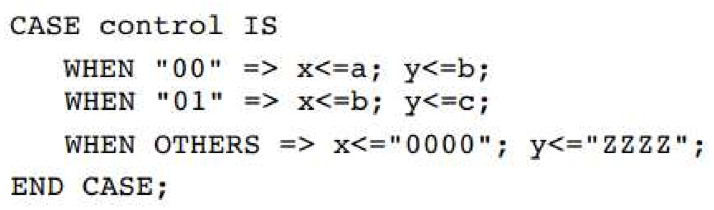
  WHEN value => assignments;

  WHEN value => assignments;

  ...

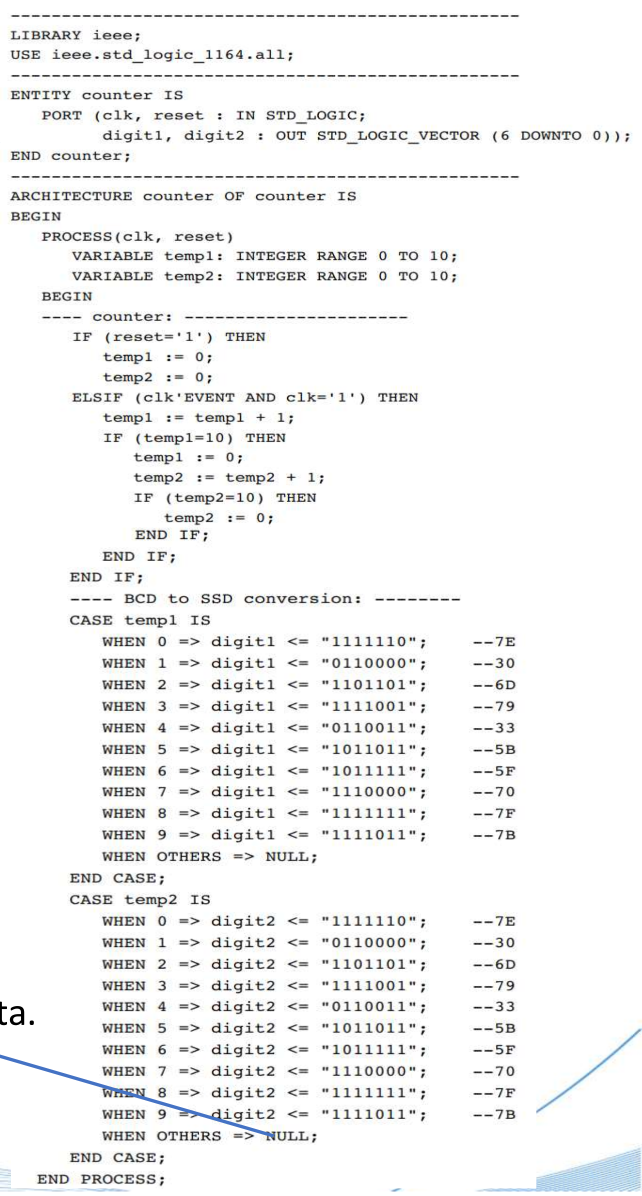
  END CASE;

- CASE (sequential) might look similar to WHEN (concurrent), also all permutations must be tested. However, CASE allow multiple assignments for each test condition, WHEN allows only one.



```
CASE control IS
   WHEN "00" => x<=a; y<=b;
   WHEN "01" => x<=b; y<=c;
   WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

2-digit counter in 7 segment format



```
---------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------------
ENTITY counter IS
   PORT (clk, reset : IN STD_LOGIC;
         digit1, digit2 : OUT STD_LOGIC_VECTOR (6 DOWNTO 0));
END counter;
---------------------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
   PROCESS(clk, reset)
      VARIABLE temp1: INTEGER RANGE 0 TO 10;
      VARIABLE temp2: INTEGER RANGE 0 TO 10;
   BEGIN
   ---- counter: ----------------------
      IF (reset='1') THEN
         temp1 := 0;
         temp2 := 0;
      ELSIF (clk'EVENT AND clk='1') THEN
         temp1 := temp1 + 1;
         IF (temp1=10) THEN
            temp1 := 0;
            temp2 := temp2 + 1;
            IF (temp2=10) THEN
               temp2 := 0;
            END IF;
         END IF;
      END IF;
      ---- BCD to SSD conversion: --------
      CASE temp1 IS
         WHEN 0 => digit1 <= "1111110";    --7E
         WHEN 1 => digit1 <= "0110000";    --30
         WHEN 2 => digit1 <= "1101101";    --6D
         WHEN 3 => digit1 <= "1111001";    --79
         WHEN 4 => digit1 <= "0110011";    --33
         WHEN 5 => digit1 <= "1011011";    --5B
         WHEN 6 => digit1 <= "1011111";    --5F
         WHEN 7 => digit1 <= "1110000";    --70
         WHEN 8 => digit1 <= "1111111";    --7F
         WHEN 9 => digit1 <= "1111011";    --7B
         WHEN OTHERS => NULL;
      END CASE;
      CASE temp2 IS
         WHEN 0 => digit2 <= "1111110";    --7E
         WHEN 1 => digit2 <= "0110000";    --30
         WHEN 2 => digit2 <= "1101101";    --6D
         WHEN 3 => digit2 <= "1111001";    --79
         WHEN 4 => digit2 <= "0110011";    --33
         WHEN 5 => digit2 <= "1011011";    --5B
         WHEN 6 => digit2 <= "1011111";    --5F
         WHEN 7 => digit2 <= "1110000";    --70
         WHEN 8 => digit2 <= "1111111";    --7F
         WHEN 9 => digit2 <= "1111011";    --7B
         WHEN OTHERS => NULL;
      END CASE;
   END PROCESS;
```

NULL is used in CASE for unaffected data.

Sequential Code
PROCESS-LOOP

- Operator **LOOP** is useful when a piece of code needs to be instantiated several times. Loop can be in different forms, as following syntaxis:

FOR/LOOP:

[label:] **FOR** identifier IN range **LOOP**

(sequential statements)

END LOOP [label];

WHILE/LOOP:

[label:] **WHILE** condition **LOOP**

(sequential statements)

END LOOP [label];

EXIT is used to end/scape of the LOOP.

[label:] EXIT [label] [WHEN condition];

NEXT is used for skipping the loop steps

[label:] NEXT [loop_label] [WHEN condition];

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

FOR range must be static

```
WHILE (i < 10) LOOP
   WAIT UNTIL clk'EVENT AND clk='1';
   (other statements)
END LOOP;
```

```
FOR i IN data'RANGE LOOP
   CASE data(i) IS
      WHEN '0' => count:=count+1;
      WHEN OTHERS => EXIT;
   END CASE;
END LOOP;
```

It exits the loop as son as a value different to 0 is found in data vector

```
FOR i IN 0 TO 15 LOOP
   NEXT WHEN i=skip;    -- jumps to next iteration
      (...)
END LOOP;
```

If I is equal to the value of “skip”, then this loop cycle is jumped and cycle continues with the next i.

Sequential Code

Case and IF

- CASE and IF after optimization can (and usually do) generate same circuits. Examples for same Physical multiplexer:

```
---- With IF: --------------
IF (sel="00") THEN x<=a;
ELSIF (sel="01") THEN x<=b;
ELSIF (sel="10") THEN x<=c;
ELSE x<=d;
```

```
---- With CASE: ------------
CASE sel IS
   WHEN "00" => x<=a;
   WHEN "01" => x<=b;
   WHEN "10" => x<=c;
   WHEN OTHERS => x<=d;
END CASE;
```

- CASE and WHEN might look similar, but they are intended for different type of codes:

	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

Equivalent codes, But CASE is whitin PROCESS

```
---- With WHEN: ----------------
WITH sel SELECT
   x <=    a WHEN "000",
           b WHEN "001",
           c WHEN "010",
           UNAFFECTED WHEN OTHERS;
```

```
---- With CASE: ----------------
CASE sel IS
   WHEN "000" => x<=a;
   WHEN "001" => x<=b;
   WHEN "010" => x<=c;
   WHEN OTHERS => NULL;
END CASE;
--------------------------------
```

Sequential Code

Clock Test Condition

- Assignment to the **same signal in both clock transitions** usually can not be synthesized.

```
PROCESS (clk)
BEGIN
   IF(clk'EVENT AND clk='1') THEN
      counter <= counter + 1;
   ELSIF(clk'EVENT AND clk='0') THEN
      counter <= counter + 1;
   END IF;
   ...
END PROCESS;
```

Probably compiler would complain. Specially in CPLDs that uses single-Edge flip-flops

- **CLOCK TEST CONDITION:** Note that EVENT attribute of clock must be related to a test condition, i.e. **(clk'EVENT and clk='1').** Avoiding the test condition make the compiler fails or assume ONE arbitrary test condition (either 0 or 1 – Not both).

- When a **signal used in the sensitivity** list of PROCESS does not appear in any of the assignments that compose the process, the compiler simply **ignore** it.

```
PROCESS (clk)
BEGIN
   counter := counter + 1;
   ...
END PROCESS;
```

```
----------------------
PROCESS (clk)
BEGIN
   IF(clk'EVENT AND clk='1') THEN
      x <= d;
   END IF;
END PROCESS;
----------------------
PROCESS (clk)
BEGIN
   IF(clk'EVENT AND clk='0') THEN
      y <= d;
   END IF;
END PROCESS;
```

Here we write a good code, avoiding all previous problems. Two PROCESS are needed.

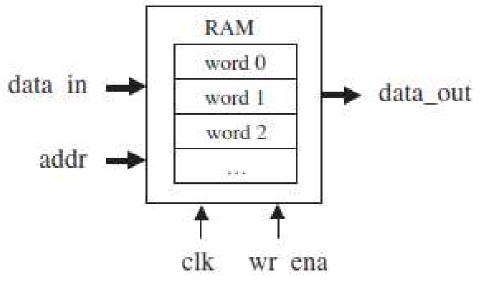
Sequential Code
Final Comments

- **Sequential code** can be used to implement **sequential** or **combinational** circuits.
- If the **sequential code** is intended for **combinational** circuit, the complete **truth-table** should be specified.
- **Registers** are necessary when making **sequential circuits** with **sequential code**.
  - Register: sequential(clocked) memory storing devices.
- Tips for writing proper sequential code for combinational circuits:
  1. Make sure **all input (read) signals used in the PROCESS appear in its sensitivity list**.
     - Otherwise, you get warning and the compiler include the signal to the PROCESS anyway. (not that serious, but good to know)
  2. Make **sure all combinations of input/output signals are defined in the code**.
     - Otherwise, compiler can create a latch for the undefined cases, which saves the last value of the signal. This create more hardware unnecessarily.

Note: Latches appear in combinatorial logic where a variable does not get assigned a value in every possible path. You find latches by checking every if, else, case, when etc. to see if at that point in the code a value has been assigned to every variable. This is not required for clocked circuits. There a variable holds its previous value if no assignment has been made.

Sequential Code
Exercise

- Create a RAM like the figure:
  - Input data is 8 bits
  - It can save up to 16 words
  - If wr_ena is enable, data_in must be stored at the specified address "addr" at the next rising edge clock.
  - data_out must always show the data stored in the value of addr.



```
---------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------------
ENTITY ram IS
GENERIC ( bits: INTEGER := 8;     -- # of bits per word
          words: INTEGER := 16);  -- # of words in the memory
   PORT ( wr_ena, clk: IN STD_LOGIC;
          addr: IN INTEGER RANGE 0 TO words-1;
          data_in: IN STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
          data_out: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
END ram;
---------------------------------------------------
ARCHITECTURE ram OF ram IS
   TYPE vector_array IS ARRAY (0 TO words-1) OF
      STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
   SIGNAL memory: vector_array;
BEGIN
   PROCESS (clk, wr_ena)
   BEGIN
      IF (wr_ena='1') THEN
         IF (clk'EVENT AND clk='1') THEN
            memory(addr) <= data_in;
         END IF;
      END IF;
   END PROCESS;
   data_out <= memory(addr);
END ram;
---------------------------------------------------
```

POWER & ENERGY CONVERSION LABORATORY

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

END-Sequential Code PROCESS, FUNCTIONS, PROCEDURES

We study the staments required to create a sequential code.

VHDL

VHSIC HARDWARE DESCRIPTION LANGUAGE