Module 6 - Looping Construct

loop

- Introduction to Looping Constructs
- For Loop
- While Loop
- Loop Control - Next & Exit
- For-Generate Loop
- When & Which

Introduction to Looping Constructs

Introduction to Looping Constructs

Looping constructs are VHDL instructions that allow a program to repeat the same block of code, a process known as **iteration** (similar to C).

Loops in VHDL are divided into two main categories based on how they function and what they create.

---

1. Sequential Loops (Inside a `process`)

These loops are similar to those in traditional programming languages. They describe an **algorithm** or a sequence of operations that execute step-by-step.

- They can **only** be used inside a `process`, `function`, or `procedure`.
- They are synthesized into hardware that performs an operation over multiple clock cycles or as a large block of combinational logic.

There are two main types of sequential loops:

- `for` **loop** -> Repeats for a specific number of times. Use this when you know the exact number of iterations.
- `while` **loop** -> Repeats as long as a certain condition is `true`. Use this when the number of iterations is unknown.

---

2. Concurrent Loops (Outside a `process`)

This type of loop is unique to hardware description languages. It doesn't describe an algorithm but instead describes the **replication of hardware structures**. It acts like copy and pasting ICs on a breadboard.

image

(well, that's the gist of it)

- It creates multiple instances of concurrent statements (like component instantiations).
- It is **only** used **outside** of a `process`, in the main architectural body.
- The main type is the `for-generate` **loop**.

For Loop

For Loop

The `for` **loop** is the most common type of sequential loop in VHDL. It is used to repeat a block of code a specific, pre-determined number of times. This makes it ideal for tasks where you know exactly how many iterations you need, such as processing the bits of a vector.

Syntax

The basic structure of a `for` loop is:

```
loop_label: for <index_variable> in <range> loop
    -- code code code...
end loop loop_label;
```

Example:

```
testloop: for i in 0 to 2 loop
    -- code cedo deco ... .. . .
end loop
```

- **`loop_label`** is the name of the loop (**recommended!**). It's not mandatory, but helps a lot in coding.
- **`<index_variable>`** is a temporary var that holds the value of the current iteration.
- **`<range>`**, the sequence of values the index will take. This is defined with `to` for an ascending range (`0 to 7`) or `downto` for a descending range (`7 downto 0`).

Notes

When using a `for` loop, there are a few important rules to remember:

- The index variable (`i`) is created automatically by the loop and does **not** need to be declared in the process.
- You can read the value of `i` within the loop, but you **cannot** manually assign a new value to it.
- The loop index will always increment or decrement by one in each iteration. You cannot specify a different step value.

Example: Init Memory

The code below shows the `for` loop iterating through every address of the memory to write a '0' value to it, like setting a RAM's content to 0.

```
type t_memory is array(0 to 63) of std_logic_vector(7 downto 0);


-- Inside your architecture...
p_Memory_Reset: process (i_Clock, i_Reset)
    variable v_ram : t_memory;
begin
    if (i_Reset = '1') then
        -- Initialize all 64 locations of the RAM to zero.
        -- We use a label "RAM_INIT_LOOP" for clarity.
        RAM_INIT_LOOP: for i in v_ram'range loop
        -- RAM_INIT_LOOP: for i in 0 to 63 loop <- would work too. variable`range is more
flexible
            v_ram(i) := (others => '0');
        end loop RAM_INIT_LOOP;

    elsif rising_edge(i_Clock) then
        -- ... normal memory operation code ...
    end if;
end process p_Memory_Reset;
```

This loop is much cleaner than writing 64 individual lines of code. It uses `v_ram'range` (variable'range) to automatically loop through the entire size of the array.

While Loop

While Loop

The `while` **loop** is used where the number of repetitions is not known from the start. A `while` loop continues to execute as long as a specified condition is `true`.

Syntax

The basic structure of a `while` loop is:

```
loop_label: while <condition> loop
    -- Going through...
    -- IMPORTANT: Must include logic to eventually make the condition false!
end loop loop_label;
```

- `<condition>` is a **boolean expression** that is checked before each iteration. If it's `true`, the loop body executes. If it's `false`, the loop terminates.

Warning: Infinite Loops

A `while` loop can create an **infinite loop**. This occurs if the loop's condition never becomes `false`. Unlike in C, Java, Python etc. where infinite loops are harmless, VHDL **shouldn't have these**.

- In a simulation, an infinite loop will cause the simulator to hang and never finish.
- In synthesis, it can be interpreted as a feedback path that creates a latch, or the synthesizer might fail with an error.

To avoid this, ensure that the logic inside the loop will eventually cause the condition to become `false`.

Example: Finding the First '1'

In this example, we'll search a vector from left to right (from the most significant bit) to find the position of the first bit that is a '1'.

Notice that unlike a `for` loop, we must **manually declare and update** our index variable (`i`).

```
-- Inside a process...
p_Find_First_One: process(a_vector)
```

```
    -- We must declare our own index variable for a while loop
    variable i : integer := a_vector'left; -- Start at the leftmost bit
    variable i_position : integer := -1; -- -1 if no 1 is found
begin
    -- Reset variables for each run of the process
    i_position := -1;
    i := a_vector'left;

    FIRST_ONE: while (i >= a_vector'right) loop

        if a_vector(i) = '1' then
            i_position := i;
            exit FIRST_ONE; -- Quit if found
        end if;

        -- Manually decrement the index to check the next bit
        i := i - 1;

    end loop SEARCH_LOOP;

    -- Assign the result to a signal
    found_position_signal <= i_position;

end process p_Find_First_One;
```

In this code, the loop continues as long as `i` is within the vector's bounds. If a '1' is found, the `exit` statement terminates the loop early. If no '1' is found, the loop completes naturally when `i` goes out of bounds.

Loop Control - Next & Exit

Loop Control - Next & Exit

The two control statements, `next` and `exit`, allow you to skip an iteration or terminate the loop entirely, giving you more precise control over your sequential code.

These statements can be used in both `for` and `while` loops.

One, The `next` Statement

The `next` statement immediately stops the **current** loop iteration and jumps to the beginning of the **next** one. Any code that comes after the `next` statement within the loop body is skipped for that specific iteration (same thing as in C!).

The syntax can be written in two ways:

```
-- Form 1: Using an if-statement
if <condition> then
    next;
end if;

-- Form 2: Using the 'when' keyword
next when <condition>;
```

Example: Summing Only Odd Numbers

To skip even numbers, the `next` statement is perfect!

```
-- Inside a process...
variable sum : integer := 0;
...
SUM_ODD_LOOP: for i in 1 to 10 loop
    -- If the number is even, skip to the next iteration
    next when (i mod 2 = 0);

    -- This line is only reached for odd numbers
    sum := sum + i;
```

```
end loop SUM_ODD_LOOP;
-- At the end, sum will be 1+3+5+7+9 = 25
```

Two, The `exit` Statement

The `exit` statement terminates the loop **entirely**. As soon as `exit` is executed, the program moves to the line after `end loop`.

The syntax is similar to `next`:

```
-- Form 1: Using an if-statement
if <condition> then
    exit;
end if;

-- Form 2: Using the 'when' keyword
exit when <condition>;
```

Example: Finding a Value in Memory

Say, we want to search through a memory to find an address that holds a specific value. Once we find it, there is no reason to continue searching.

```
-- Inside a process...
constant SEARCH_VALUE : std_logic_vector(7 downto 0) := x"A5";
variable found_addr : integer := -1; -- Default value
...
SEARCH_MEM_LOOP: for i in ram'range loop
    -- If the value at the current address matches,
    -- store the address and exit the loop immediately.
    if ram(i) = SEARCH_VALUE then
        found_addr := i;
        exit SEARCH_MEM_LOOP;
    end if;
end loop SEARCH_MEM_LOOP;

-- The code goes on...
```

Using `exit` makes the search efficient. It prevents the loop from doing unnecessary work after the item has been found.

For-Generate Loop

The Concurrent 'for-generate' Loop

We now switch from sequential loops to a **concurrent** one.

The `for-generate` statement is **not** a loop that executes over time inside a process. Instead, it's a command that tells the synthesizer to create multiple copies of hardware structures.

It's good for repetitive structures like registers and chains of components. A key rule is that `for-generate` is used **outside** of a `process`, in the main architectural body.

Main Differentiators

- This loop is **Concurrent**, not **Sequential**. All the hardware instances created by the loop exist simultaneously and operate in parallel. The synthesizer "unrolls" the loop, creating all the copies before the design is even simulated or synthesized.
- This is most often used to create multiple instances of a `component`, but **can also be used for concurrent signal assignments or even entire `process` blocks**.

Syntax

The basic structure of a `for-generate` statement is:

```
generate_label: for <identifier> in <range> generate
    -- Concurrent statements to be replicated go here...
    -- (e.g., component instantiations)
end generate generate_label;
```

- Label, identifier, and range, all follow the same rules as the other loops (`for` and `while`).

Example: 4-Bit Adder

We start with a **1-bit full adder** and use `for-generate` to create and connect four of them in a chain to make a **4-bit adder**.

1. Replication Target First, we need the definition of the 1-bit full adder that we want to copy.

```
component full_adder is
    port (
```

```
        A, B, Cin  : in  std_logic;
        S, Cout    : out std_logic
    );
end component;
```

2. 4-Bit Adder Architecture Next, we use `for-generate` to create four instances of the `full_adder` and wire them together.

```
entity four_bit_adder is
    port (
        A, B    : in  std_logic_vector(3 downto 0);
        Cin     : in  std_logic;
        S       : out std_logic_vector(3 downto 0);
        Cout    : out std_logic
    );
end entity;

architecture structural of four_bit_adder is
    -- Internal signal to wire the carry chain between the adders.
    -- It needs 5 bits to include the first Cin and final Cout.
    signal C : std_logic_vector(4 downto 0);
begin

    -- The first carry wire is connected to the adder's carry-in pin
    C(0) <= Cin;

    -- Generate the chain of 4 full adders
    ADDER_CHAIN: for i in 0 to 3 generate
        -- Create one instance of the full_adder in each "iteration"
        FA_INSTANCE: full_adder
            port map (
                A    => A(i),       -- Connect to the i-th bit of input A
                B    => B(i),       -- Connect to the i-th bit of input B
                Cin  => C(i),       -- The carry-in for this bit
                S    => S(i),       -- The sum output for this bit
                Cout => C(i+1)      -- The carry-out for this bit
            );
    end generate ADDER_CHAIN;

    -- The final carry-out of the chain is the adder's carry-out pin
```

```
    Cout <= C(4);


end architecture structural;
```

Physically (RTL wise), this is the exact same as copy-pasting the 4 adders manually. So, this approach is much cleaner and more organized.

When & Which

When & Which?

Comparison

Feature	`for` Loop	`while` Loop	`for-generate` Statement
Execution	**Sequential**	**Sequential**	**Concurrent**
Usage Location	Inside a `process`	Inside a `process`	**Outside** a `process`
Iteration	Fixed number of times	Repeats while a condition is `true`	Creates N physical copies
Purpose	Algorithmic tasks	Searching or Polling	Hardware Replication

---

Quick Guide

- Use a `for` **loop** when...
  - You need to repeat an action a **specific number of times**.
  - *Iterating through all the bits of a vector, initializing every address in a memory.*
- Use a `while` **loop** when...
  - You need to repeat an action **until a condition changes**, and you don't know how long that will take.
  - *Searching for the first occurrence of a value, waiting for a status flag to be set in a testbench.*
- Use a `for-generate` **statement** when...
  - You need to create **multiple, regular instances of hardware components** or concurrent statements.
  - *Building an N-bit register from N flip-flops, creating a chain of adders, connecting several identical modules to a bus.*

Remember, all loops have the same syntax rules, so you just need to remember the structure. Good luck! :+1: