4.7 VHDL

"VHDL" stands for "VHSIC Hardware Description Language." VHSIC, in turn, stands for "Very High Speed Integrated Circuit, which was a joint program between the US Department of Defense and IEEE in the mid-1980s to research on high-performance IC technology.

VHDL was standardized by the IEEE in 1987 (VHDL-87) and extended in 1993 (VHDL-93).

Features:

- Designs may be decomposed hierarchically.
- Each design element has both
  1. a well-defined interface (for connecting it to other elements) and
  2. a precise behavioral specification (for simulating it).
- Behavioral specifications can use either an algorithm or an actual hardware structure to define an element's operation.
- Concurrency, timing, and clocking can all be modeled.
- VHDL handles asynchronous as well as synchronous sequential-circuit structures.
- The logical operation and timing behavior of a design can be simulated.

VHDL synthesis tools are programs that can create logic-circuit structures directly from VHDL behavioral descriptions.

Using VHDL, you can design, simulate, and synthesize anything from a simple combinational circuit to a complete microprocessor system on a chip.

4.7.1. Design Flow

Steps in the design flow:

1. **Hierarchical / block diagram**. Figuring out the basic approach and building blocks at the block-diagram level. Large logic designs are usually hierarchical, and VHDL gives you a good framework for defining modules and their interfaces and filling in the details later.
2. **Coding**. Actual writing of VHDL code for modules, their interfaces, and their internal details.
3. **Compilation**. Analyses your code for syntax errors and checks it for compatibility with other modules on which it relies. Compilation also creates the internal information that is needed for simulation.
4. **Simulation**. A VHDL simulator allows you to define and apply inputs to your design, and to observe its outputs. Simulation is part of a larger step called *verification*. A **functional verification** is performed to verify that the circuit's logical operation works as desired independent of timing considerations and gate delays.
5. **Synthesis**. Converting the VHDL description into a set of primitives or components that can be assembled in the target technology. For example, with PLDs or CPLDs, the synthesis tool may generate two-level sum-of-products equations. With ASICs, it may generate a *netlist* that specifies how the gates should be interconnected.
6. **Fitting / Placement & Routing**. Maps the synthesized components onto physical devices.
7. **Timing verification**. At this stage, the actual circuit delays due to wire lengths, electrical loading, and other factors are known, so precise timing simulation can be performed. Study the circuit's operation including estimated delays, and we verify that the setup, hold, and other timing requirements for sequential devices like flip-flops are met.

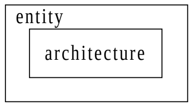
4.7.2 Program Structure

A key idea in VHDL is to define the interface of a hardware module while hiding its internal details.

A VHDL **entity** is simply a declaration of a module's inputs and outputs, i.e. its external interface signals or *ports*.

A VHDL **architecture** is a detailed description of the module's internal structure or behavior.

You can think of the entity as a "wrapper" for the architecture, hiding the details of what's inside while providing the "hooks" for other modules to use it.



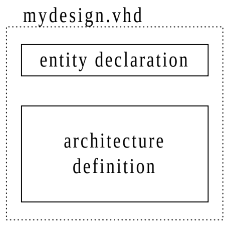
VHDL actually allows you to define multiple architectures for a single entity, and it provides a configuration management facility that allows you to specify which one to use during a particular synthesis run.

In the text file of a VHDL program, the entity declaration and architecture definition are separated.

Example – VHDL program for an "inhibit" gate:

```
entity Inhibit is      -- also known as 'BUT-NOT'
  port (X, Y: in BIT;  -- as in 'X but not Y'
        Z: out BIT);
end Inhibit;

architecture Inhibit_arch of Inhibit is
begin
  Z <= '1' when X='1' and Y='0' else '0';
end Inhibit_arch;
```



Keywords: *entity, port, is, in, out, end, architecture, begin, when, else,* and *not.*
Comments: begin with two hyphens (--) and end at the end of a line.
Identifiers: begin with a letter and contain letters, digits, and underscores. (An underscore may not follow another underscore or be the last character in an identifier.)
Keywords and identifiers are not case sensitive.

A basic **entity** declaration has the syntax as shown below:

```
entity entity-name is
    port (signal-names : mode signal-type;
          signal-names : mode signal-type;
          …
          signal-names : mode signal-type);
end entity-names;
```

In addition to the keywords, an entity declaration has the following elements:
- *entity-name*: A user-selected identifier to name the entity.
- *signal-names*: A comma-separated list of one or more user-selected identifiers to name external-interface signals.
- *mode*: One of four reserved words, specifying the signal direction:
  - **in** – The signal is an input to the entity.
  - **out** – The signal is an output of the entity. Note that the value of such a signal cannot be "read" inside the entity's architecture, only by other entities that use it.
  - **buffer** – The signal is an output of the entity, and its value can also be read inside the entity's architecture.
  - **inout** – The signal can be used as an input or an output of the entity. This mode is typically used for three-state input/output pins on PLDs.
- *signal-type*: A built-in or user-defined signal type. See below.

A basic **architecture** definition has the syntax as shown below:

```
architecture architecture-name of entity-name is
    type declarations
    signal declarations
    constant declarations
    function definitions
    procedure definitions
    component declarations
begin
    concurrent-statement
    …
    concurrent-statement
end architecture-name;
```

The *architecture-name* is a user-selected identifier, usually related to the entity name.

An architecture’s external interface signals (ports) are inherited from the port-declaration part of its corresponding entity declaration.
An architecture may also include signals and other declarations that are local to that architecture.
Declarations common to multiple entities can be made in a separate “package” used by all entities. See 4.7.5.
The declarations can appear in any order.

signal declaration.

> **signal** *signal-names* : *signal-type*;

4.7.3 Types and constants

All signals, variables, and constants must have an associated “type.” The **type** specifies the set or range of values that the object can take on.
Some predefined types are:
bit, **bit_vector**, **boolen**, **character**, **integer**, **real**, and **string**.

A *user-defined* ***enumerated*** *types* have the following syntax:

> **type** *type-name* **is (***value-list***);**
> **--** *value-list* is a comma-separated list of all possible values of the type.
>
> -- subtypes of a type. Values must be a contiguous range of values of the base type, from *start* to *end*.
> **subtype** sub*type-name* **is** *type-name* **range** *start* **to** *end***;** -- ascending order
> **subtype** sub*type-name* **is** *type-name* **range** *start* **downto** *end***;** -- descending order
>
> **constant** *constant-name***:** *type-name* **:=** *value***;**

Example (**enumerated** type):

```
type traffic_light_state is (reset, stop, wait, go);
type std_logic is ( ‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘-’);
subtype fourval_logic is std_logic range ‘X’ to ‘Z’;
subtype bitnum is integer range 31 downto 0;
constant bus_size: integer := 32;   -- width of component
constant MSB: integer := bus_size-1;     -- bit number of MSB
constant Z: character := ‘Z’;-- synonym for Hi-Z value
```

array types have the following syntax:

> **type** *type-name* **is array (***start* **to** *end***) of** *element-type***;**
> **type** *type-name* **is array (***start* **downto** *end***) of** *element-type***;**
> **type** *type-name* **is array (***range-type***) of** *element-type***;**
> **type** *type-name* **is array (***range-type* **range** *start* **to** *end***) of** *element-type***;**
> **type** *type-name* **is array (***range-type* **range** *start* **downto** *end***) of** *element-type***;**

Example (**array** type):

```
type monthly_count is array (1 to 12) of integer;
type byte is array (7 downto 0) of std_logic;
constant word_len: integer := 32;
type word is array (word_len – 1 downto 0) of std_logic; -- note: a range
                                    value can be a simple expression.
constant num_regs: integer := 8;
type reg_file is array (1 to num_regs) of word; -- a two dimensional array
type statecount is array (traffic_light_state) of integer; -- an enumerated
                                    type can be the range.
```

Array elements are considered to be ordered from left to right, in the same direction as index range. Thus, the leftmost elements of arrays of types:
monthly_count is 1, byte is 7, word is 31, reg_file is 1, and statecount is reset.

Array elements are accessed using parentheses. If M, B, *W*, R, and S are signals or variables of the five array types above, then

M(11), B(5), *W*(word_len – 5), R(0, 0), S(reset).

Array literals can be specified using different formats:

B := (‘1’, ‘1’, ‘1’, ‘1’, ‘1’, ‘1’, ‘1’, ‘1’);
B := “11111111”;
W := (0=>‘0’, 8=>‘0’, 16=>‘0’, 24=>‘0’, **others** => ‘1’);
W := “11111110111111101111111011111110”;

It is also possible to refer to a contiguous subset or *slice* of an array. eg.

M(6 to 9), B(3 downto 0), W(15 downto 8), S(stop to go).

You can combine arrays or array elements using the *concatenation operator* **&** which joins arrays and elements in the order written, from left to right. eg.

‘0’ & ‘1’ & “1Z” = “011Z”
B(6 downto 0) & B(7) yields a 1-bit left circular shift of the 8-bit array B.

The most important array type is:

type STD_LOGIC_VECTOR **is array (**natural **range <>) of** STD_LOGIC;

-- example of an *unconstrained array type* – the range of the array is unspecified

4.7.4 Functions and procedures

A *function* accepts a number of arguments and returns a result.

```
function function-name (
    signal-names : signal-type;
    signal-names : signal-type;
    …
    signal-names : signal-type
) return return-type is
    type declarations
    constant declarations
    variable declarations
    function definitions
    procedure definitions
begin
    sequential-statement
    …
    sequential -statement
end function-name;
```

A *procedure* does not return a result. Their arguments can be specified with type **out** or **inout** to “return” results.

A function can be used in the place of an expression.
A procedure can be used in the place of a statement.

Example – an “inhibit” function:

```
architecture Inhibit_archf of Inhibit is

function ButNot (A, B: bit) return bit is
begin
  if B = ‘0’ then return A;
  else return ‘0’;
  end if;
end ButNot;

begin
  Z <= ButNot(X, Y);
end Inhibit_archf;
```

Can have *overloaded* operators. the compiler picks the definition that matches the operand types in each use of the operator.

4.7.5 Libraries and Packages

A *library* is a place where the VHDL compiler stores information about a particular design project, including intermediate files that are used in the analysis, simulation, and synthesis of the design.

For a given VHDL design, the compiler automatically creates and uses a library named “work” under the current design directory.

Use the *library clause* at the beginning of the design file to use a standard library.

library ieee;

The clause “library work;” is included implicitly at the beginning of every VHDL design file.

A library name in a design gives it access to any previously analyzed entities and architectures stored in the library, but it does not give access to type definitions.

A *package* is a file containing definitions of objects that can be used in other programs. The kind of objects that can be put into a package include signal, type, constant, function, procedure, and component declarations.

Signals that are defined in a package are “global” signals, available to any entity that uses the package.

A design can use a package with the statement

use ieee.std_logic_1164.all;

Here, “ieee” is the name of the library. Use the file named “std_logic_1164” within this library. The “all” tells the compiler to use all of the definitions in this file.

The **package** syntax:

```
package package-name is
    -- public section: visible in any design file that uses the package
    type declarations
    signal declarations
    constant declarations
    component declarations
    function declarations -- lists only the function name, arguments, and type.
    procedure declarations
end package-name;
package body package-name is
    -- private section: local to the package
    type declarations
    constant declarations
    function definitions -- the complete function definition
    procedure definitions
end package-name;
```

4.7.6 Structural design elements

The body of an architecture is a series of concurrent statements.

Each concurrent statement executes simultaneously with the other concurrent statements in the same architecture body. e.g. if the last statement updates a signal that is used by the first statement, then the simulator will go back to that first statement and update its results according to the signal that just changed.

The simulator will keep propagating changes and updating results until the simulated circuit stabilizes.

Component statement (a concurrent statement).

```
label: component-name port map (signal1, signal2, …, signal_n);
label: component-name port map(port1=>signal1, port2=>signal2, …, port_n=>signal_n);
```

Component-name is the name of a previously defined entity that is to be used, or *instantiated*, within the current architecture body.

Each instance must be named by a unique *label*.

The *port map* introduces a list that associates ports of the named entity with signals in the current architecture.

Before being instantiated in an architecture's definition, a component must be declared in a *component declaration* in an architecture's definition. It is essentially the same as the port-declaration part of the corresponding entity declaration.

The components used in an architecture may be ones that were previously defined as part of a design, or they may be part of a library.

A **component declaration**.

```
component component-name
    port (signal-names : mode signal-type;
          signal-names : mode signal-type;
          …
          signal-names : mode signal-type);
end component;
```

A VHDL architecture that uses components is often called a *structural description* or *structural design*, because it defines the precise interconnection structure of signals and entities that realize the entity. A pure structural description is equivalent to a schematic or a net list for the circuit.

In some applications, it is necessary to create multiple copies of a particular structure within an architecture. e.g. an *n*-bit "ripple adder" can be created by cascading *n* "full adders."

A **generate** statement allows you to create such repetitive structures using a kind of "for loop."

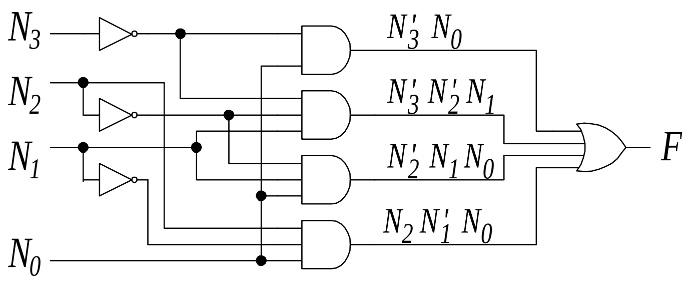
```
label: for identifier in range generate
    concurrent-statement
    end generate;
```

The value of a constant must be known at the time that a VHDL program is compiled. In many applications it is useful to design and compile an entity and its architecture while leaving some of its parameters, such as bus width, unspecified.

A **generic** statement lets you do this.

```
entity entity-name is
    generic (constant-names : constant-type;
                constant-names : constant-type;
                …
                constant-names : constant-type);
    port (signal-names : mode signal-type;
          …
          signal-names : mode signal-type);
end component;
```

Example – Given the schematic diagram for a prime-number detector, we can implement a **structural** VHDL program for the prime-number detector.



```
library IEEE;
use IEEE.std_logic_1164.all;

entity prime is
  port (  N: in STD_LOGIC_VECTOR (3 downto 0);
          F: out STD_LOGIC );
end prime;

architecture prime1_arch of prime is
  signal N3_L, N2_L, N1_L: STD_LOGIC;
  signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
  component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
  component AND2 port (I0,I1: in STD_LOGIC; O: out STD_LOGIC);
     end component;
  component AND3 port (I0,I1,I2: in STD_LOGIC; O: out STD_LOGIC);
     end component;
  component OR4 port (I0,I1,I2,I3: in STD_LOGIC; O: out STD_LOGIC);
     end component;

begin
  U1: INV port map (N(3), N3_L);
  U2: INV port map (N(2), N2_L);
  U3: INV port map (N(1), N1_L);
  U4: AND2 port map (N3_L, N(0), N3L_N0);
  U5: AND3 port map (N3_L, N2_L, N(1), N3L_N2L_N1);
  U6: AND3 port map (N2_L, N(1), N(0), N2L_N1_N0);
  U7: AND3 port map (N(2), N1_L, N(0), N2_N1L_N0);
  U8: OR4 port map (N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0, F);
end prim1_arch;
```

Example (**generate**) – an 8-bit inverter generated from one component.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity inv8 is
  port (  X: in STD_LOGIC_VECTOR (1 to 8);
          Y: out STD_LOGIC_VECTOR (1 to 8) );
end inv8;

architecture inv8_arch of inv8 is
component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;

begin
  g1: for b in 1 to 8 generate
    U1: INV port map (X(b), Y(b));
  end generate;
end inv8_arch;
```

Example (**generic**, **generate**) – an arbitrary-width bus inverter.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity businv is
  generic ( WIDTH: positive);
  port (  X: in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
          Y: out STD_LOGIC_VECTOR (WIDTH-1 downto 0) );
end businv;

architecture businv_arch of businv is
component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;

begin
  g1: for b in WIDTH-1 downto 0 generate
    U1: INV port map (X(b), Y(b));
  end generate;
end businv_arch;
```

4.7.7 Dataflow design elements

Several additional concurrent statements allow VHDL to describe a circuit in terms of the flow of data and operations on it within the circuit. This style is called a *dataflow description* or *dataflow design*.

concurrent signal-assignment statement.

signal-name **<=** *expression***;**

The type for expression must be identical or a sub-type of signal-name.
In the case of arrays, both the element type and the length must match; however, the index range and direction need not match.

Example (**concurrent signal-assignment**) – Dataflow VHDL architecture for the prime-number detector.

```
architecture prime2_arch of prime is
  signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;

begin
  N3L_N0     <= not N(3) and N(0);
  N3L_N2L_N1 <= not N(3) and not N(2) and N(1);
  N2L_N1_N0  <= not N(2) and N(1) and N(0);
  N2_N1L_N0  <= N(2) and not N(1) and N(0);
  F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
end prime2_arch;
```

conditional signal-assignment statement.

```
signal-name <= expression when boolean-expression else
               expression when boolean-expression else
               …
               expression when boolean-expression else
               expression;
```

Boolean operators: **and**, **or**, **not**.
Relational operators: =, /= (not equal), >, >=, <, <=.
The combined set of conditions in a single statement should cover all possible input combinations.

Example (**conditional signal-assignment**) – Dataflow VHDL architecture for the prime-number detector.

```
architecture prime3_arch of prime is
  signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;

begin
  N3L_N0     <= '1' when N(3)='0' and N(0)='1' else '0';
  N3L_N2L_N1 <= '1' when N(3)='0' and N(2)='0' and N(1)='1' else '0';
  N2L_N1_N0  <= '1' when N(2)='0' and N(1)='1' and N(0)='1' else '0';
  N2_N1L_N0  <= '1' when N(2)='1' and N(1)='0' and N(0)='1' else '0';
  F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
end prime3_arch;
```

selected signal-assignment statement.

```
with expression select
      signal-name <= signal-value when choices,
                     signal-value when choices,
                     …
                     signal-value when choices,
```

The *choices* may be a single value of *expression* or a list of values separated by vertical bars (|).
The *choices* for the entire statement must be mutually exclusive and all inclusive.

Example (**concurrent signal-assignment**) – Dataflow VHDL architecture for the prime-number detector.

```
architecture prime4_arch of prime is

begin
  with N select
    F <= '1' when "0001",
         '1' when "0010",
         '1' when "0011" | "0101" | "0111",
         '1' when "1011" | "1101",
         '0' when others;
end prime4_arch;
```

Example – A more behavioral description of the prime-number detector.

```
architecture prime5_arch of prime is

begin
  with CONV_INTEGER(N) select -- convert STD_LOGIC_VECTOR to INTEGER
    F <= '1' when 1 | 2 | 3 | 5 | 7 | 11 | 13,
         '0' when others;
end prime5_arch;
```

4.7.8 Behavioral design elements

VHDL's key behavioral element is the "process." A *process* is a collection of **sequential** statements that executes in parallel with other concurrent statements and other processes.

process statement.

process (*signal-name, signal-name, …, signal-name*)
  type declarations
  constant declarations
  variable declarations
  function definitions
  procedure definitions
begin
  sequential-statement
  …
  sequential -statement
end process;

A process statement can be used anywhere that a concurrent statement can be used.
A process is either *running* or *suspended*.
The list of signals in the process definition, called the *sensitivity list*, determines when the process runs.
A process initially is suspended. When any signal in its sensitivity list changes value, the process resumes execution, starting with its first sequential statement and continuing until the end.
If any signal in the sensitivity list change value as a result of running the process, it runs again. This continues until none of the signals change value.
In simulation, all of this happens in zero simulated time.

The sensitivity list is optional; a process without a sensitivity list starts running at time zero in simulation. One application of such a process is to generate input waveforms in a test bench.

A process may not declare signals. It can only declare variables.
A VHDL *variable* is similar to signals, except that they usually don't have physical significance in a circuit. They are used only in functions, procedures, and processes to keep track of the state within a process and is not visible outside of the process.

variable declaration.

variable *variable-names* : *variable-type*;

Example – A process-based dataflow VHDL architecture for the prime-number detector.

```
architecture prime6_arch of prime is

begin
  process(N)
    variable N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
  begin
    N3L_N0     := not N(3) and N(0);
    N3L_N2L_N1 := not N(3) and not N(2) and N(1);
    N2L_N1_N0  := not N(2) and N(1) and N(0);
    N2_N1L_N0  := N(2) and not N(1) and N(0);
    F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
  end process;

end prime6_arch;
```

VHDL has several kinds of **sequential** statements.

sequential signal-assignment statement – same as the concurrent signal-assignment statement. It is sequential because it occurs in the body of a process rather than an architecture.
Since we cannot have signal definitions in a process, the *signal-name* must be one from the sensitivity list.

```
signal-name <= expression;
```

variable-assignment statement -

```
variable-name := expression;          -- notice the different assignment operator for variables.
```

if statement.

```
if boolean-expression then sequential-statement
end if;

if boolean-expression then sequential-statement
else sequential-statement
end if;

if boolean-expression then sequential-statement
elsif boolean-expression then sequential-statement        -- note the spelling for the keyword elsif
…
elsif boolean-expression then sequential-statement
else sequential-statement
end if;
```

case statement.

```
case expression is
    when choices => sequential-statements       -- one or more sequential statements can be used
    …
    when choices => sequential-statements
    when others => sequential-statements        -- optional; to denote all values that have not yet been covered.
end case;
```

A *case* statement is usually more readable and may yield a better synthesized circuit.
The *choices* must be mutually exclusive and include all possible values of *expression*'s type.

loop statements.

```
for identifier in range loop          -- for loop
    sequential-statement
    …
    sequential-statement
end loop;

loop                                  -- infinite loop
    sequential-statement
    …
    sequential-statement
end loop;

exit;     -- transfers control to the statement immediately following the loop end

next;     -- causes any remaining statements in the loop to be bypassed and begins the next iteration of the loop.
```

The *identifier* is declared implicitly by its appearance in the *for loop* and has the same type as *range*. This variable may be used within the loop's sequential statements, and it steps through all of the values in *range*, from left to right, one per iteration.

Example – Prime-number detector using an **if** statement.

```
architecture prime7_arch of prime is

begin
  process(N)
    variable NI: integer;
  begin
    NI := CONV_INTEGER(N);
    if NI=1 or NI=2 then F <= '1';
    elsif NI=3 or NI=5 or NI=7 or NI=11 or NI=13 then F <= '1';
    else F <= '0';
    end if;
  end process;

end prime7_arch;
```

Example – Prime-number detector using an **case** statement.

```
architecture prime8_arch of prime is

begin
  process(N)
  begin
    case CONV_INTEGER(N) is
      when 1 => F <= '1';
      when 2 => F <= '1';
      when 3 | 5 | 7 | 11 | 13 => F <= '1';
      when others => F <= '0';
    end case;
  end process;

end prime8_arch;
```

Example – A truly behavioral description of a prime-number detector.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity prime9 is
  port (  N: in STD_LOGIC_VECTOR (15 downto 0);
          F: out STD_LOGIC );
end prime9;

architecture prime9_arch of prime9 is

begin
  process(N)
    variable NI: integer;
    variable prime: boolean;
  begin
    NI := CONV_INTEGER(N);
    prime := true;
    if NI=1 or NI=2 then null;       -- take case of boundary cases
    else
      for i in 2 to 253 loop
        if NI mod i = 0 then
          prime := false;
          exit;
        end if;
      end loop;
    end if;
    if prime then F <= '1';
    else F <= '0';
    end if;
  end process;
end prime9_arch;
```