Set of Synthesis Tools

Paulo Flores

ESPRIT III Project # 6043 "QuickChips"
Task 1–5.2.4 - Deliverable 9

INESC
Instituto de Engenharia de Sistemas e Computadores

November 1993

Contents

1 Introduction

The project of an integrated circuit usually begins with a high-level specification of its behavior and a set of restrictions to be met (for example, maximum area of the circuit, minimum or maximum delays of some signals). The synthesis process of an digital circuit pretends to get from its high-level representation the optimal circuit, mapped in a specific technology library, that assures the imposed restriction.

The representation of the circuit in a hardware description language is the first step of a project in a synthesis environment. The use of a powerful language, such as the VHDL (IEEE standard), requires that the designer has a deep knowledge of the language to take advantage of all its features. Moreover, the designer must take into account the restrictions imposed by the synthesis tools, in order to be assured that a description can be synthesized and that the resulting circuit has the desired performance. The description of circuits in an unsuitable style for synthesis, usually generates incorrect or non-optimized circuits, independently of the quality and capability of the synthesis tool used.

Earlier in this project an evaluation of synthesis technology has result in the choice of Synopsys synthesis tools. This report describes the design flow used in the synthesis environment. It emphasizes the steps needed to obtain a gate-level representation of the circuit and the iterative/interactive nature of the process. The fundamental issue of identifying a synthesizable VHDL subset is analyzed in some detail, focusing on the language constructs supported by the Synopsys synthesis tool, Design Compiler [Synthesis 92], and giving some attention to the questions of description portability between tools from different vendors. Examples that shows a suitable description style for this tool are also presented. Additional portability issues are discussed in the last section before conclusions.

2 Synthesis design process

The synthesis process to produce a gate-level netlist from a high-level description is not done in a single step. The design flow typically used in a synthesis environment involves the following four steps (see figure 1):

1. The VHDL description of the circuit. This description should be done in an abstraction level such that the designer is liberated from the implementation details, but at the same time it can be used with current synthesis tools.

2. The validation of the VHDL circuit description through the use of a VHDL simulator. This design step assures that the desired functionality is present in the description.

3. The logic synthesis of the circuit to produce a gate level representation mapped in a specific technology that fulfills a set of constraints imposed by the designer.

4. The verification of the synthesized circuit performance, using a gate-level simulator.

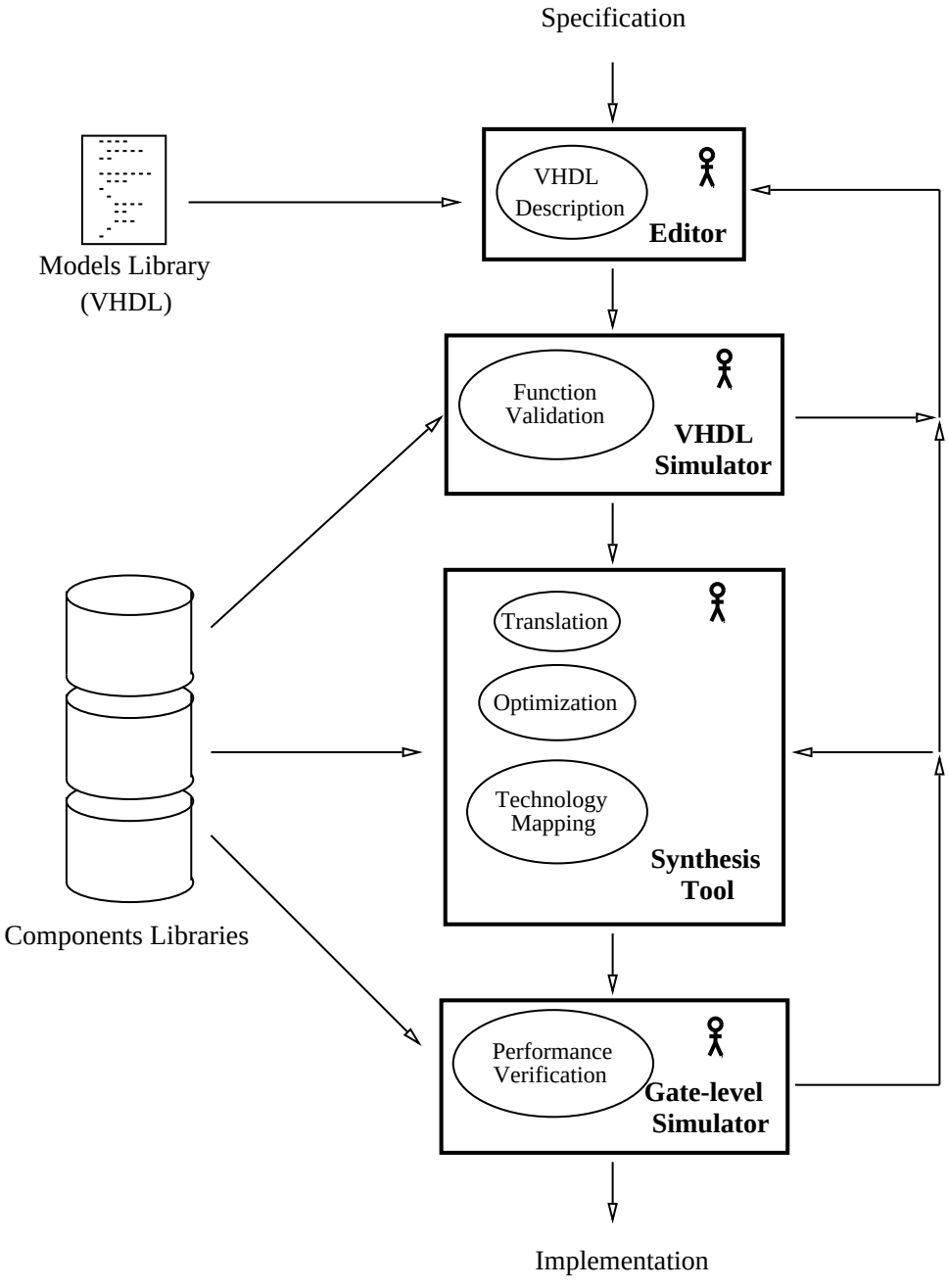

Figure 1: Synthesis design flow.

Every time a new description of the circuit is obtained, it is necessary to verify its functionality through a simulation. When the simulation results do not verifies all of the designer constraints, it is necessary to go back to a previous step, and after some changes repeat the remaining steps. This iterative process can be done in three levels:

- At language level: while the description of the circuit does not met the desired functionality or it can not be synthesized, it will be necessary to rewrite the VHDL code.
- At the synthesis tool level: when the changes to the circuit result from modifications of parameters on the synthesis tools, so that the constraints imposed by the designer are satisfied.
- At global level: if the gate-level simulation does not demonstrate the functionality or the performance desired for the circuit. In this case it could be necessary to rewrite the VHDL code and repeat all design steps again.

The global level iteration should not be necessary if the synthesis tools could generate circuits with correct timing and functionality. However, there are two reasons that justify the gate-level simulation after the synthesis. Firstly, because differents semantic meanings could be given by the VHDL simulator and the synthesis tool to some constructs. Secondly, because only after technology mapping it is possible to simulate the circuit with realist timings.

The design flow shown is iterative as well as interactive. This means that during the synthesis step the designer has to "guide" the tools so that it can obtain circuits of an acceptable quality. This process consists in the characterization of the circuit and imposition of restrictions. The former supplies to the synthesis tool additional information about the environment in which the circuit will be used (timing and electrical information), the latter establishes a set of objectives to be accomplished by the synthesis tool.

The libraries that allow the representation of the circuit at differents levels have distinct features according to their target. The model library is written in VHDL, in a technology independent way, and has the purpose to help the designer to write and test their circuits. An example of a synthesizable model library can be found in [Sim oes 93]. The component libraries have a representation of each cell belonging to a technology through a description of its functionality and a set of parameters which differs from tool to tool. For the synthesis tools both the area and timing of each cell are important, for the simulation tools only the timing is meaningful. The use of a non-coherent set of component libraries can increase the number of iterations or even make the synthesis process impossible.

3 Synthesis Tools

The VHDL language was developed for the description and simulation of digital circuits. For this reason, many statements that can be used to model a circuit are related to

a simulation environment and are not generally synthesized. For instance, the VHDL capability to support a floating-point arithmetic and to use files for input/output, are very convenient and sophisticated for system modeling, but requires unrealistic capabilities to the synthesis tools [Levitan 89].

Since the VHDL semantic is adapted to simulation, most of simulators support the full language. The ones that do not support all VHDL constructs just exclude a small set which are certainly not supported by synthesis tools. Therefore, the VHDL constructs that can be used in a synthesizable description are mainly limited by the synthesis tools.

3.1 Supported VHDL

The subset of statements supported for synthesis is not yet standardized, despite the effort that has been developed by the synthesis working group [Harper 92]. Hence, the synthesizable VHDL subset is actually dependent on the synthesis tool.

From the synthesis viewpoint, the VHDL constructs can be divided in three categories:

- *Ignored* - means that the construct is allowed in the VHDL source, but is ignored in synthesis.
- *Unsupported* - means that the construct is not allowed in the VHDL source code. If unsupported constructs are present in a description, the analyzer will flag them as errors and the description can not be synthesized.
- *Supported* - means that the construct can be used in a synthesizable VHDL description. Some restrictions may be imposed if the construct is not fully supported.

The reminder of this section is dedicated to present the support that Synopsys synthesis tool do for each VHDL constructs [CRM 92].

3.1.1 Design Units

The VHDL language defines design units as the minimal set of instruction that can be analyzed separately. Each design unit is analyzed to a project library. By default the project library is denominated `WORK`. The use of libraries and separate analysis of design units are fully supported.

The design units specified in the VHDL language are presented in the following subsections.

Entities

The VHDL entity is a hardware abstraction that can represent a whole system, a board, an integrated circuit or just a cell. Each VHDL entity specifies the interface between the block it represents and the environment in which it will be used. The declaration of an entity defines the type and the number of inputs and outputs, but does not include

entity*entity_name* **is**
[**generic**(*generic_declaration*)**;**]
[**port**(*port_declaration*)**;**]
[*entity_declarative_part*]
[**begin**
entity_statement_part]
end [*entity_name*];

Item	Restrictions
generic_declaration	Only of type integer
port_declaration	Default values for ports are ignored
entity_declarative_part	Alias declaration and disconnection specification are unsupported
entity_statement_part	Ignored

Table 1: Entity support

any functionality description. Table 1 presents the restrictions imposed on an entity for synthesis.

Architectures

An architecture defines the functionality of an entity. It describes how the outputs are obtained from the inputs.

Each entity can have associated an undetermined number of architectures. The "same" functionality in each one of them can range from an algorithm (a set of sequential statements within a process) to a structural netlist (a set of components instantiated). The table 2 shows the architecture support for the Synopsys synthesis tool.

architecture *architecture_name* **of** *entity_name* **is**
[*architecture_declarative_part*]
begin
[*statements*]
end [*architecture_name*]**;**

Item	Restrictions
architecture_declarative_part	Only subprograms (declaration or body), types, subtypes, constants, signals or component declarations.
statements	Support as presented in sections 3.1.5 and 3.1.4.

Table 2: Architecture support

Configurations

Due to the possibility of existence of several architectures for the same entity, it is necessary for each entity instantiated, to define which architecture to use.

The configuration unit allows the definition of the entities-architectures pairs that will be used for each component instantiated inside a given architecture. In table 3 are described the restrictions to the use of configurations units for synthesis.

configuration *configuration_name* **of** *entity_name* **is**
    block_configuration
end [*configuration_name*]**;**

Item	Restrictions
block_configuration	Only to specify the top-level architecture for a top-level entity. Component configuration and nested block configuration are unsupported.

Table 3: Configuration support

Packages

The use of packages permits a better organization of circuit description, through the use of code that can be shared, such as: constants or type definitions, component declarations or subprograms.

A package can have two parts, the *declaration*, where the public view of the package is defined, and the *body*, where the private information and the subprogram implementations of the package are included. As shown in table 4, these two views of a package are fully supported.

package *package_name* **is** – – package declaration
    [*package_declarative_part*]
end [*package_name*] **;**

package_body *package_name* **is** – – package body
    [*package_body_declarative_part*]
end [*package_name*]**;**

Item	Restrictions
package_declarative_part	Fully supported
package_body_declarative_part	Fully supported

Table 4: Package support

The IEEE standard package `std_logic_1164` defines a nine value logic system for modeling. This package also defines the logic operators and conversion functions necessary

to use a nine value logic. The package is fully supported excluding the following functions: `rising_edge`, `falling_edge` and `is_x`.

3.1.2 Objects, Data Types and Attributes

VHDL objects are elements that can hold values of a given type. There are three classes of objects: constants, variables and signals.

Object	Restrictions
Constants	Deferred constants are unsupported
Variables	Initial values are unsupported
Signals	`register` and `bus` declarations are unsupported. Resolution functions are supported for wired and three-state functions only. Initial values are unsupported.

Table 5: Supported objects

The type of an object determines which are the values that an object can hold. The four basic scalar types are: integers, floating point, physical and enumerates. Composite types, like arrays or structures, can be defined using the basic types. Access types and file types provide a way to access objects of a given type. Subtypes can be defined through the imposition of restrictions on another type. The types defined in VHDL language are listed in table 6 with the restrictions imposed by synthesis.

Types	Restrictions
Integer	Infinite-precision arithmetic is unsupported. Integer types are converted to vectors whose width is as small as possible to accommodate all possible values of the type range (binary for non-negative ranges and 2's-complement for range that include negative numbers).
Floating point	Unsupported.
Physical	Ignored.
Enumerates	Fully supported.
Array	Only integers range are supported. Multidimensional arrays are unsupported, but arrays of arrays are supported.
Records	Fully supported.
Access	Unsupported.
File	Unsupported.
Incomplete data types	Unsupported.

Table 6: Supported types

Attributes are characteristics (values, functions, etc) that can be associated with certain elements in VHDL. Although the language has a set of predefined attributes it also

allows user defined attributes.

Many synthesis tools define attributes with special meaning for the tool. Although the characterization and the imposition of constraints could be done through the use of these, this method should be avoided for a portable VHDL description[1]. Because VHDL was developed for simulation, there are some attributes with no meaning for synthesis. The only supported attributes for synthesis are presented in table 7.

Attributes Class	Restrictions
Predefined attributes	Only the following attributes are supported: `base, left, high, low, range, reverse_range` and `length`.
User-defined attributes	Unsupported.
Synthesis-attributes	Are supported to characterize components, input and output ports, or to provide design and output constraints.

Table 7: Supported attributes

3.1.3 Expressions

Expressions perform arithmetic or logical computations by applying an operator to one or more operands. Operators specify the computation to be performed, while operands are the data for the computation.

Operators

A VHDL operator is characterized by: name, function, number and type of operands, and result type. It is possible to define new operators, with functions, for any type of operand and result type, as well as assigning new functions to the predefined VHDL operators (operator overloading). The VHDL predefined operators are presented in the table 8 along with the restrictions imposed by the synthesis tool.

Operands

As already mentioned operands determine the date used by the operator to compute its value. The operands in an expression can include identifiers, names, literals, aggregates, subprograms calls, qualified expressions, type conversions, allocators and another expressions surrounded by parentheses. The operands supported are presented in table 9.

3.1.4 Sequential Statements

The VHDL statements describe internal organization and/or operation of a circuit. Sequential statements are used when it is pretended to describe a circuit in a behavioral way.

[1] Characterization and imposition of constraints on the design can, and should be done, in the synthesis tool (interactively or using a script).

Class	Operators	Restrictions
Logical	*and*, *or*, *nand*, *nor*, *xor*	Fully supported
Relational	=, / =, <, <=, >, >=	Fully supported
Adding	+, −, &	Fully supported
Sign	+, −	Fully supported
Multiplying	∗, /, *mod*, *rem*	The ∗ operator is fully supported. The /, *mod* and *rem* operators are supported only when both operands are constant or when the right operand is a *computable*[1] power of 2.
Miscellaneous	∗∗, *abs*, *not*	The ∗∗ operator is supported only when both operands are constant or when the left operand is a *computable*[1] power of 2. The *abs* and *not* operators are fully supported.

[1]Their value can be statically determined during the analysis of the code.

Table 8: Operators supported

Operands	Restrictions
Identifiers	Fully supported.
Selected names	Fully supported.
Indexed names	Fully supported.
Slice names	Null slices are unsupported.
Attribute names	Fully supported for the attributes defined in section 3.1.2.
Literals	Null literals are unsupported. Physical literals are ignored.
Aggregates	The use of types as aggregates choices is unsupported. Record aggregates are unsupported.
Subprograms calls	Function conversions on input ports are unsupported.
Qualified expressions	Fully supported.
Type conversion	Fully supported.
Allocators	Unsupported.
Static expressions	Fully supported.
Universal expressions	Floating-point expressions are unsupported. Precision is limited to 32 bits; all intermediate results are converted to integer.

Table 9: Operands supported

The algorithm is set up in a step by step fashion as in high level programming languages. All sequential statements are encapsulated inside processes or subprograms. Table 10 lists the VHDL sequential statements and the restrictions imposed by the synthesis tool.

Statements	Restrictions
wait	Supported only on the following forms: **wait until** signal = *logic_value* **;** **wait until** signal**'event and** signal = *logic_value* **;** **wait until not** signal**'stable and** signal = *logic_value* **;** where *logic_value* is '0' or '1' for synchronization with falling or rising edge of signal, respectively. Wait statements cannot be used in subprograms or for loops.
Assertions	Ignored.
Signal assignment	Multiple waveform elements in signal assignment statements are unsupported. `Transport` keyword and `after` clause are ignored.
Variable assignment	Fully supported.
Subprogram call	Type conversion on formal parameters is unsupported.
If	Fully supported.
Case	Fully supported.
Loops For loops While loops	The loop body must contain at least one wait statements. The loop range must be a computable, and the loop body must not contain a wait statement. The loop body must contain at least one wait statement.
Next	Fully supported.
Exit	Fully supported.
Return	Fully supported.
Null	Fully supported.

Table 10: Sequential statements support

3.1.5 Concurrent Statements

Since electronic circuits work in parallel, the VHDL language has included the concurrent capability to model this behavior. Unlike sequential statements, which are executed one after another, concurrent statements are executed continuously and in parallel without any predefined execution order. The support of concurrent statements is presented in table 11.

3.2 Description Styles

As important as knowning the VHDL subset that can be used in a synthesized description, is knowning the description style that synthesis tools accept. Only in this way it is possible to write descriptions that can be synthesized and at the same time get the most from the synthesis tool.

It is also of great importance to understand how descriptions are mapped to gates. For instance, the synthesis of the sequential statement `if-then-else` inside a process generates a multiplexer, in which the selection input is dependent on the `if` condition, as shown in figure 2.

```
entity examp is
    port(a, b, selection: in bit;
         z: out bit);
end examp;


architecture ifthenelse of examp is
begin
    process(selection, a, b)
    begin
        if selection = '1' then          -- if instruction
            z <= a;
        else                             -- else branch
            z <= b;
        end if;
    end process;
end ifthenelse;
```

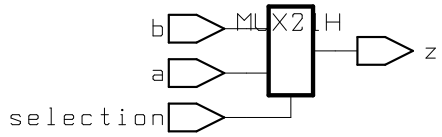

Figure 2: Synthesis of an `if-then-else` inside a process.

However, if the signal assignment is only done in one branch of the `if` instruction, the previous solution is not so simple. This situation is represented in figure 3, through the use of an `if` without the `else` branch. In this case, the synthesized circuit is a latch enabled by the `if` condition. The use of this memory element is necessary because while the `if` condition is false the output signal must keep the previous value.

Statements	Restrictions
Block	Ports and generics in block statement and guarded blocks are unsupported.
Process	Sensitivity list is ignored.
Subprogram calls	Type conversion on formal parameters is unsupported.
Assertions	Ignored.
Signal assignment	The `guarded` and `transport` keywords and `after` clause are ignored. Multiple waveforms are unsupported.
Component instantiation	Type conversion on the formal port of a component specification is unsupported.
Generate	Fully supported.

Table 11: Concurrent statements support

```
entity examp is
    port(a, selection: in bit;
         z: out bit);
end examp;


architecture ifthen of examp is
begin
    process(selection, a)
    begin
        if selection = '1' then          -- if instruction
            z <= a;                      -- without else branch
        end if;
    end process;
end ifthen;
```

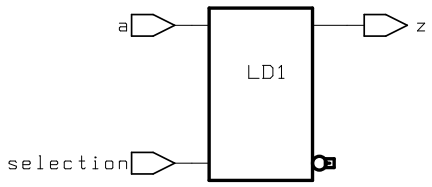

Figure 3: Synthesis of an `if` without the `else` branch.

The description of synchronous circuits, in which memory elements (flip-flops) are used, can be done using either the `wait` or the `if` instruction. The figure 4 presents the synthesis of a flip-flop with synchronous reset through the use of an `wait` and `if` statement.

If a flip-flop with asynchronous reset is required, the description in figure 5 can be used, which consists on two chained `if` instructions. The former represents the asynchronous reset and the later the normal synchronous operation of the circuit.

The description of finite state machines in VHDL can be achieved in different ways, since a specific statement for their representation does not exist. The most common descriptions of state machines use two process: one is responsible for the definition of the synchronous behavior of the circuit; the other, describes the combinational part of the circuit, the transitions between states and the generation of the outputs values.

The example of figure 6 describes a Moore finite state machine that detects the input sequence `"111"`. The state table of this machine, represented in table 12, is described in the combinational process through the use of a `case` statement controlled by the `present_state`. Each branch of the `case` assigns a value to the output and defines the next state of the machine using an `if`, whose condition depends on the input value. This process is identified in the code of the figure 6 with the label `comb`. The process identified with the `sync` label is responsible for the synchronous operation of the state machine.

The example presented in figure 6 illustrates some points that should be considered when describing a finite state machine, such as:

- Use of an enumerate type to represent the states of the machine. The use of this type allows the description of the state machine at a symbolic level. In this way it

```
entity circuit is
    port(data, clock, reset: in bit;
         z: out bit);
end circuit;


architecture async_rst of circuit is
begin
    process
    begin
        wait until clock'event and clock = '1' ;
        if reset = '1' then
            z <= '0';                    -- synchronous reset
        else
            z <= data;                   -- synchronous operation
        end if;
    end process;
end sync_rst;
```

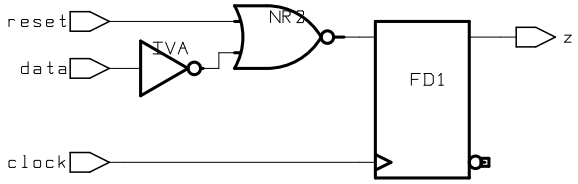

Figure 4: Synthesis of a flip-flop with synchronous reset.

```
entity circuit is
    port(data, clock, reset: in bit;
         z: out bit);
end circuit;


architecture async_rst of circuit is
begin
    process (reset, clock)
    begin
        if reset = '1' then              -- asynchronous reset
            z <= '0';
        elsif clock'event and clock = '1' then
            z <= data;                   -- synchronous operation
        end if;
    end process;
end async_rst;
```

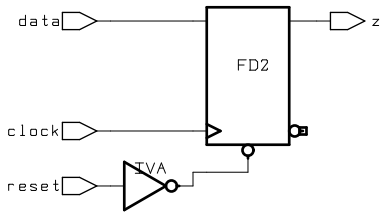

Figure 5: Synthesis of a flip-flop with asynchronous reset.

is left to the synthesis tool to choose the best state codification. However, if the optimal codification is known it should be given to the synthesis tool.

```
entity fsm is
    port(data_in, clock, reset: in bit;
         sequence: out bit);
end fsm;


architecture Moore of fsm is
    type states is (zero, one, two, three); -- enumerat type for
    signal present_state,                   -- the state variables
        next_state: states;
begin

    comb: process(data_in, present_state)   -- combinational logic
    begin
        case present_state is
            when zero =>
                sequence <= '0';
                if data_in = '1' then
                    next_state <= one;
                else
                    next_state <= zero;
                end if;
            when one =>
                sequence <= '0';
                if data_in = '1' then
                    next_state <= two;
                else
                    next_state <= zero;
                end if;
            when two =>
                sequence <= '0';
                if data_in = '1' then
                    next_state <= three;
                else
                    next_state <= zero;
                end if;
            when three =>
                sequence <= '1';
                if data_in = '1' then
                    next_state <= three;
                else
                    next_state <= zero;
                end if;
        end case;
    end process;

    sync: process(reset, clock)             -- synchronous behavior
    begin
        if reset = '1' then
            present_state <= zero;
        elsif clock'event and clock = '1' then
            present_state <= next_state;
        end if;
    end process;

end Moore;
```

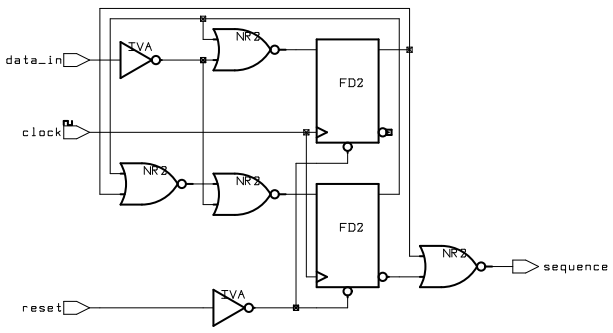

Figure 6: Moore state machine to detect the sequence `"111"`.

Present state	Next State		Output sequence
	`data_in` = 0	`data_in` = 1	
zero	zero	one	0
one	zero	two	0
two	zero	three	0
three	zero	three	1

Table 12: Transition table of the finite state machine.

- Initialization of the state machine using a dedicated signal (reset). Other types of initialization are possible in VHDL: implicitly, in which the state variable is initialized to a default value, or explicitly when the variable is declared. Both types of initialization are appropriate for simulation, but when synthesized the circuit can evaluate to an unpredictably state after the power on.

- Use of `if-then-else` statements in the combinational process to completely specify in all states which are the values for each output and next state. This method avoids the use of latchs that will force the circuit to an unpredictably state after the power on.

In descriptions where the use of a multi-value logic is required for the correct specification of the circuit, the IEEE standard logic system defineed in package `std_logic_1164`, should be used.

For circuits in which the outputs are not completely specified for all inputs combination the don't-care value, `'-'`, should be used. In this way it is possible for the synthesis tool to generate a better circuit if it uses this information during the logic optimization phase. For circuits that need three-state lines the high-impedance value `'Z'` can be used, so that the synthesis tool provides three-state drivers.

The example of figure 7 describes a BCD to 7-segments decoder with three-state output. The use of the `std_logic_vector` type allows to force all outputs to three-state, when the decoder is not enabled, or to assign to the output (for input values from `"1010"` to `"1111"`) a vector of don't-cares.

4 Code Portability

To guarantee the portability of a description between synthesis tools, it is necessary to use a common subset of VHDL constructs and a compatible description style. Even if these restrictions may be difficult to sustain between some tools, there are some constructs that should not be used in a portable description, such as:

- Specific attributes that are just recognized by one synthesis tool. As mentioned previously these include the attributes for characterization and imposition of restrictions to the circuit. The synthesis package under development by the IEEE

```
library ieee;
use ieee.std_logic_1164.all;

entity bcd_7seg is
    port(data_in: in std_logic_vector(3 downto 0);
         enable: in std_logic;
         data_out: out std_logic_vector(6 downto 0));
end bcd_7seg;

architecture combinational of bcd_7seg is
begin
    process (data_in, enable)
    begin
        if enable = '0' then
            data_out <= (others => 'Z');              -- three-state output
        else
            case data_in is                 --abcdefg
                when "0000" => data_out <= "1111110";
                when "0001" => data_out <= "1100000";
                when "0010" => data_out <= "1011011";
                when "0011" => data_out <= "1110011";
                when "0100" => data_out <= "1100101";
                when "0101" => data_out <= "0110111";
                when "0110" => data_out <= "0111111";
                when "0111" => data_out <= "1100010";
                when "1000" => data_out <= "1111111";
                when "1001" => data_out <= "1110111";
                when others => data_out <= "-------"; -- don't-care output
            end case;
        end if;
    end process;
end combinational;
```

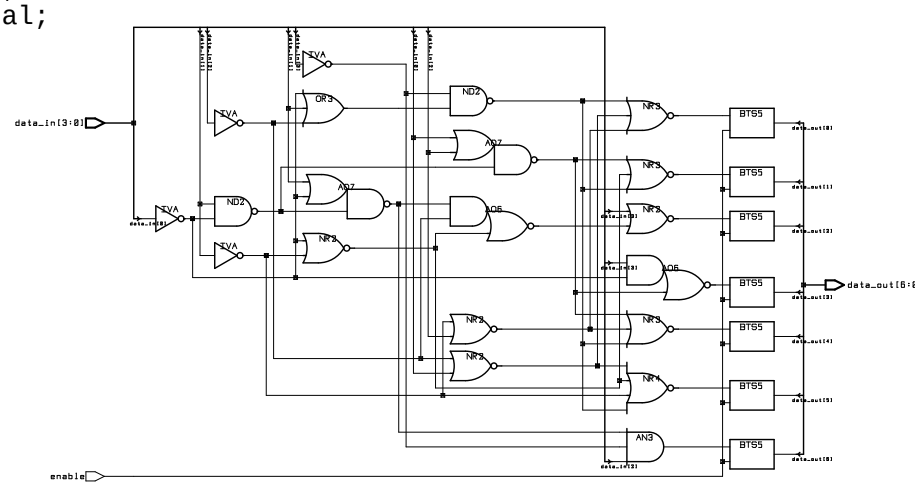

Figure 7: Use the IEEE multi-value logic values.

synthesis working group addresses this problem, defining a set of attributes for synthesis that will become an IEEE standard.

- Synthetic comments[2] that allow incorporation in the VHDL description of commands for the synthesis tool. An example of a synthetic comment that most tools support, but differently, is the one which allows to cancel or re-initialize the analysis of specific portions of code.

- Functions or procedures described in packages which are integrated in the tool. For these only the package declaration exists, but the body has no VHDL representation because it is embedded in the tool.

It should be noticed that to get the most from the synthesis tool it maybe necessary to use some of this constructs at the cost of getting a non-portable description.

For a multi-level logic system it is encouraged to use the standard package `std_logic_1164`. This package defines a nine-value logic system and a set of functions to manipulate this logic system, and is supported by most synthesis tools.

5 Conclusion

In this report we presented the design flow usually used in a synthesis environment. The information on the synthesizable VHDL subset and the use of a proper description style are vital for a faster design process, reducing the time spent in each design step or decreasing the number of iteration cycles needed. The study of the supported VHDL constructs for the synthesis tool of choice was presented and some examples on a suitable description style were given.

Some considerations have been made about code portability. The use of the standard package `sdt_logic_1164` is a first step for a portable description in order to avoid the use of tool specific constructs. Special attention is maintained on the follow-up of the work being developed by the IEEE synthesis working group which is addressing these problems.

References

[CRM 92] Synopsys, Inc. *VHDL Compiler Reference Manual*, November 1992. Version 3.0.

[Flores 93] Paulo Flores. Especificação funcional de Sistemas Electrónicos Digitais em Ambiente de Síntese. Master's thesis, Instituto Superior Técnico, June 1993.

[2] Comments in which the first word has a especial meaning to force the tool to interpret the rest of the line as a command.

[Harper 92] P. Harper and K. Scott. Towards A Standard VHDL Synthesis Package. In *Proceedings of European Design Automation Conference / Proceedings of Euro-VHDL*, September 1992.

[Levitan 89] S. Levitan, A. Martello, R. Owens, and M. Irwin. Using VHDL as a Language for Synthesis of CMOS VLSI Circuits. In *Proceedings of the Ninth IFIP Symposium on Computer Hardware Description Languages and their Applications*, June 1989.

[Sim oes 93] Leonel Sim oes. IC Blocks Reference Manual. Technical report, Project ESPRIT 6043, November 1993.

[Synthesis 92] Synopsys, Inc. *Design Compiler Reference Manual*, December 1992. Version 3.0.