PC Based Synthesis Tool

Paulo Flores

ESPRIT III Project # 6043 "QuickChips"
Task 1–5.2.4 - Deliverable 10

INESC
Instituto de Engenharia de Sistemas e Computadores

April 1995

Contents

1 Introduction

Synthesis is the automatic generation of hardware from an initial description expressed in an appropriate language, usually a hardware description language (HDL). The use of this technology has several fundamental advantages over a traditional design methodology. It eliminates the former gate-level design bottleneck and reduces circuit design time and errors introduced when hand-translating a specification to gates. The synthesis technology can also help the designer to explore the design space. Thus, an optimized hardware implementation that meets the circuit constraints, in terms of area and/or speed, can be "more easily" found.

The emergence of VHDL as a worldwide standard hardware description language make it appealing as a formalism to specify the input and output of synthesis applications. However, VHDL was developed as a language for digital system modeling. As a consequence, its use for synthesis applications is not straightforward. 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. Moreover, an unsuitable description style can generate incorrect or non-optimized circuit implementations, independently of the quality and capability of the synthesis tool used.

The evaluation of the synthesis technology in the QuickChips project has resulted in the choice of Synopsys synthesis tool, which has been integrated in the QC design environment. In previous reports we have described the restrictions necessary to impose to a VHDL description so, it can be synthesized and acceptable quality implementations can be obtained from this tool.

In order to make accessible the facilities offer by the QuickChips Consortium (specially the fabrication of integrated circuits) to SMEs, is necessary to offer an affordable set of tools that allows the project of integrated circuits in-house and can be integrated in the QC design environment. An entry point in the QC design environment for SMEs has already been established using OrCad software. Through a traditional design methodology, SMEs can accomplish to design and describe the circuit at logic level using the OrCad schematic editor. To extend the use of synthesis technology to SEMs is necessary that low cost and PC based synthesis tools become available in the QC design environment.

In this report we will evaluate the Viewlogic PC based synthesis tool, `vhdldes`, and illustrate how to integrate this tool in the QC CAD system. The synthesizable subset of constructs supported by the tool is analyzed in some detail and a set of synthesis example that show a suitable description style for this tools is presented. Using some VHDL descriptions from the IC-Blocks library [Flores 95] portability issues are discussed in the last section. Finally some conclusions are drawn about the use of Viewlogic synthesis tool in the QC CAD system.

2 Integration in the design environment

In the QuickChips project it was implemented a top-down design methodology based on synthesis technology. The project design flow enforces this methodology adopting a set of tasks that the designer must fulfill for a successful circuit design. Each one of the design tasks is executed by a high performance tool that runs on a workstation. All the tools, described in [Flores 94a], are integrated in a common design environment - the QC design environment - as presented in [Abreu 94].

The use of low-cost and PC based tools to execute some of the high level tasks of the design flow, concede to SMEs the opportunity to design the circuit by themselves. Many tools are available for the tasks concerning design entry and verification. One important aspect to consider in the selection of a tool is its capability to be integrated in the existent design environment, it must have internal or external parsers to read/write some of the formats in which the different tools communicate [Flores 94b].

The synthesis tool evaluated for this report – `vhdldes V2.05` – is integrated in the Viewlogic design environment - Workview 4.1.3. In this environment it is also provided an EDIF (Electronic Data Interchange Format) parser - `vl2edif2 V4.1.2` - which provide a way to transfer the synthesized circuit to the main QC design environment. In figure 1 it can be identified the QuickChips "standard" data flow and the supported extension for the Viewlogic synthesis tool. Having an EDIF representation of the circuit in the main design environment gives us the opportunity to re-synthesize the circuit for further optimization and/or to re-map it to a new technology.

3 Synthesis Tool

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 floating-point arithmetic and to use files for input/output, are very convenient and sophisticated for system modeling, but requires unrealistic capabilities from the synthesis tools [Levitan 89].

Since the VHDL semantic is adapted to simulation, most simulators support the full language. The ones that do not support all the VHDL constructs just exclude a small set which is 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 dependent on the actual synthesis tool.

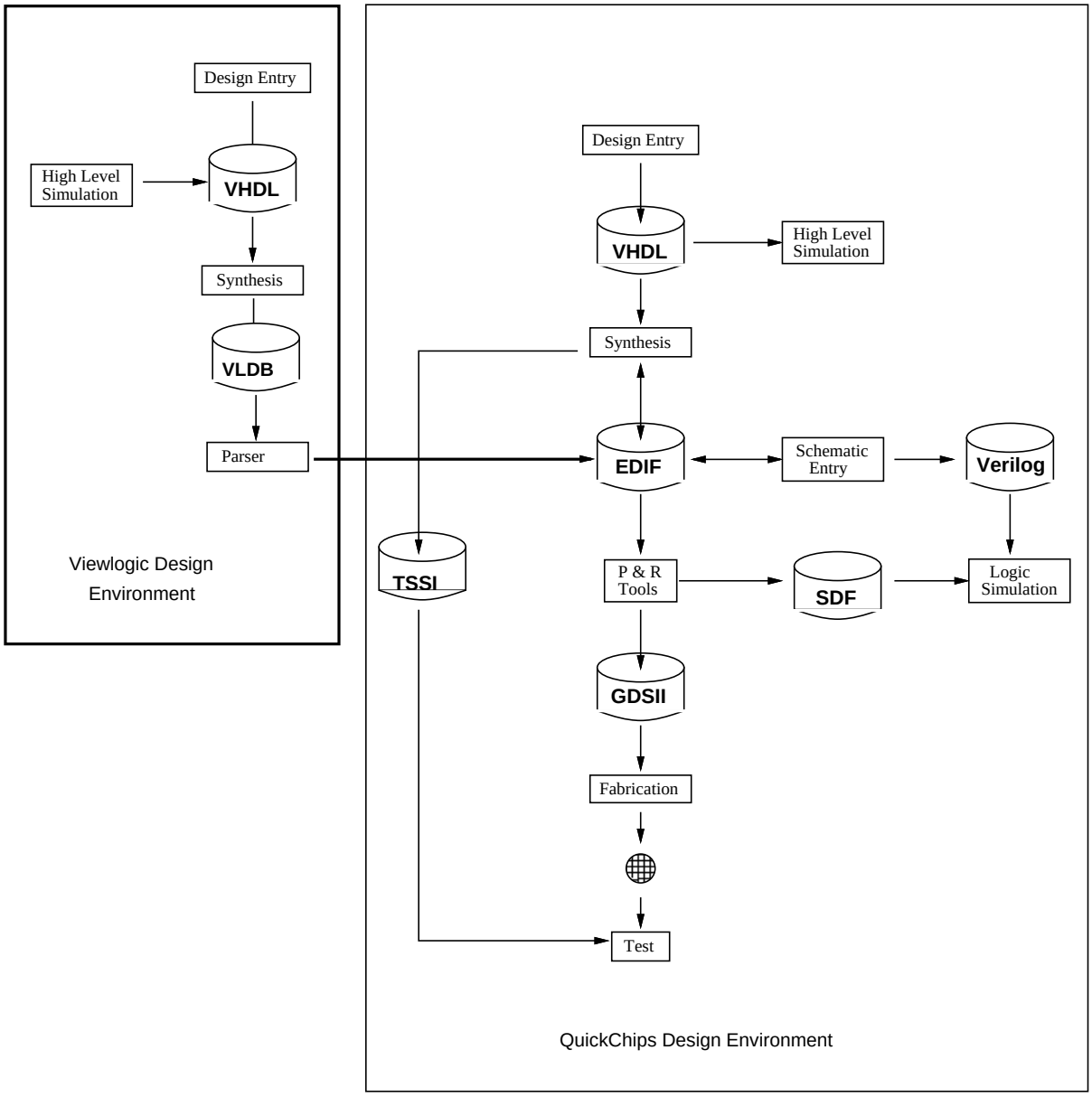

Figure 1: QuickChips "standard" data flow and extension for Viewlogic.

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 Viewlogic synthesis tool, `vhdldes` version `V2.05`, provides for each VHDL construct [Viewlogic 92b, Viewlogic 92a].

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 not 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 declaration defines its name and interface to the enclosing design. It represents the external view of a component, the type and number of inputs and outputs but does not include any functionality description. Table 1 presents the restrictions imposed on an entity for synthesis.

Architectures

An entity's architecture body defines its functionality. It describes how the outputs are obtained from the inputs.

Each architecture body is associated, by name, with one entity declaration. The behavior description of an architecture can range from an algorithm (a set of sequential statements within a process) to a structural netlist (a set of instantiated components). Although VHDL accepts several architecture associated to the same entity, in this synthesis tool each entity has to have only one associated architecture body. The table 2 shows the architecture support for the Viewlogic synthesis tool.

Configurations

The configuration unit specifies which pair entity/architecture is chosen for the components instantiated. This is due to the possibility of existence of several architectures

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

Item	Restrictions
generic_declaration	Supported, but can not interfere with the synthesis of the circuit
port_declaration	Only signal of type `vlbit` or one-dimensional arrays of type `vlbit`, `vlbit_vector` (see section 3.1.2)
entity_declarative_part	Unsupported
entity_statement_part	Unsupported

Table 1: Entity support

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

Item	Restrictions
architecture_declarative_part	Only types, constants, signals, subprograms, components declaration or attributes declarations and specifications.
statements	Support as presented in sections 3.1.5 and 3.1.4.

Table 2: Architecture support

for the same entity. So, it is necessary for each entity instantiated, to define which architecture to use. The Viewlogic synthesis tools does not support configurations

Packages

A package is a library unit (design unit) holding a collection of items that can be used in other design units. All the items specified in the package declaration are accessible to other design units either by selection or directly after an appropriate use clause.

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 3, these two views of a package are partial 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	Only declarations of subprograms, constants, types, components, attributes, and specifications attributes.
package_body_declarative_part	Only declarations of subprograms, constants, types and specifications of subprograms bodies.

Table 3: Package support

The IEEE standard package `std_logic_1164` that defines a nine value logic system for modeling is not supported. To describe a design using a multi-value logic system, only the Viewlogic predefined types, `vlbit` and `vlbit_vector`, explained in the next, section are allowed.

3.1.2 Objects, Data Types and Attributes

VHDL objects are containers of data within a model. Each object has a type and a class. The object class indicates what can be done with its data. There are three classes of objects: constants, variables and signals. Table 4 describes the how this three classes of objects are supported.

Object	Restrictions
Constants	Deferred constants are unsupported
Variables	Initial values are unsupported
Signals	May only be of type `vlbit` or one-dimensional array of type `vlbit` `vlbit_vector` (see below in this section). `register`, `bus` and resolution functions are unsupported. Initial values are unsupported.

Table 4: 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 5 with the restrictions imposed by this synthesis tool.

Types	Restrictions
Integer	Infinite-precision arithmetic is unsupported, a maximum of 32 bits is allowed
Floating point	Unsupported.
Physical	Ignored.
Enumerates	Fully supported.
Array	Only integers ranges are supported.
Records	Unsupported.
Access	Unsupported.
File	Unsupported.
Incomplete data types	Unsupported.

Table 5: Supported types

The Viewlogic synthesis tool has a predefined type, `vlbit`, for signal objects. The `vlbit` type is an extension to the IEEE type `bit`. It contains four values: *logic-zero* (`'0'`), *logic-one* (`'1'`), *high-impedance* (`'Z'`) and *don't-care* (`'X'`)[1] The `vlbit_vector` is defined as unconstraint array of `vlbit` type.

Attributes describe characteristics about associated entities. Although the language has a set of predefined attributes it also allows user defined attributes. Since VHDL was

[1] In simulation `'X'` means *unknown*.

developed for simulation, there are some attributes with no meaning for synthesis. The only supported attributes for synthesis are presented in table 6.

Attributes Class	Restrictions
Predefined attributes	Only the following attributes are supported: `left, right, high, low, range, reverse_range` and `length`.
User-defined attributes	Unsupported.

Table 6: 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, using functions, for any kind 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 table 7 along with the restrictions imposed by the synthesis tool.

Class	Operators	Restrictions
Logical	*and*, *or*, *nand*, *nor*, *xor*, *not*	Fully supported
Relational	$=$, $/=$, $<$, $<=$, $>$, $>=$	Fully supported
Adding	$+$, $-$, $\&$	Fully supported
Sign	$+$, $-$, *abs*	Fully supported
Multiplying	$*$, $/$, *mod*, *rem*	Supported when both operands are constants or the second operand is a constants power of 2.
Miscellaneous	$**$	Unsupported.

Table 7: Operators supported

Operands

As already mentioned operands determine the data 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 8.

Operands	Restrictions
Identifiers	Fully supported.
Selected names	Unsupported.
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	Unsupported
Subprograms calls	Function conversions on input ports are unsupported.
Qualified expressions	Fully supported.
Type conversion	Only using predefined functions of Viewlogic.
Allocators	Unsupported.
Static expressions	Fully supported.
Universal expressions	Floating-point expressions are unsupported.

Table 8: Operands supported

3.1.4 Sequential Statements

Sequential statements define the algorithms that express the behavior of design entities.

Sequential statements are encapsulated inside processes or subprograms. They are executed sequentially, in their order of appearance. Table 9 lists the VHDL sequential statements and the restrictions imposed.

3.1.5 Concurrent Statements

Concurrency is an important and natural concept in hardware description. Concurrent statements are used to describe behavior or structure of architectures. Unlike sequential statements, the order in which the concurrent statements are written does not modify the description functionality. The support of concurrent statements is presented in table 10.

Statements	Restrictions
wait	Supported only on the following forms: **wait until** signal = *logic_value* **;** **wait until** signal**'event and** signal = *logic_value* **;** **wait until** (signal**'event and** signal = *logic_value*) **or** sig = *l_value;* where *logic_value* is '0' or '1' for synchronization with falling or rising edge of signal, respectively. sig and *l_value* allow asynchronous signal to execute the process. Wait statements can not be used in subprograms and must be the first statement in a `process`.
Assertions	Ignored.
Signal assignment	Multiple waveform elements and `transport` delay in signal assignment statements are unsupported. The `after` clause is ignored.
Variable assignment	Fully supported.
Subprogram call	Type conversion on formal parameters is unsupported. Parameter association is made by position.
If	Fully supported.
Case	Case choices can not be a range. The choice `others` is required to ensure that no value is omitted.
Loops For loops While loops	Unsupported. The loop range must be known at analysis time. Unsupported.
Next	Unsupported
Exit	Fully supported.
Return	Only supported in a function body, and it must be the last statement.
Null	Fully supported.

Table 9: Sequential statements support

Statements	Restrictions
Block	Ports and generics in block statement and guarded blocks are unsupported.
Process	All signals read in a process without a `wait` statement must be in the sensitivity list.
Subprogram calls	Type conversion on formal parameters is unsupported. Parameter association is made by position.
Assertions	Ignored.
Signal assignment	The `guarded` and `transport` keywords are unsupported and `after` clause is ignored. Multiple waveforms are unsupported. In a selected signal assignment a range choice is not supported.
Component instantiation	Type conversion on the formal port of a component specification is unsupported. Generic map association is made by positional association.
Generate	Unsupported.

Table 10: Concurrent statements support

3.2 Description Styles

VHDL definition is mainly oriented towards simulation applications, and its use for synthesis raises many issues. These problems encompass all aspects of synthesis and module generation, including functional and timing aspects as well as technology-dependent issues.

Describing a circuit using the synthesizable subset of a tool, as enumerated in the previous sections, does not guarantee to a designer that a hardware implementation can be synthesized from it. Or, if it can, that the resulting circuit has the desire performance. Only knowing the description style accepted by the synthesis tool allows to write descriptions that can be synthesized and, at the same time, getting the most from the 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 circuit that implements a multiplexer. In this multiplexer the selection input is dependent on the `if` condition, as shown in figure 2.

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


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

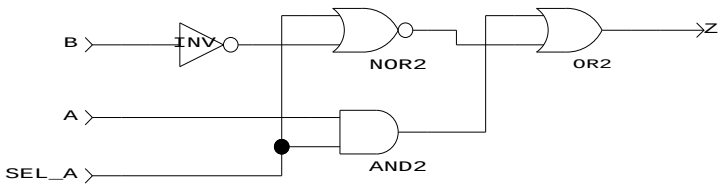

Figure 2: Synthesis of an `if-then-else` inside a process.

If the signal assignment is only done in one branch of the `if` instruction, then a memory element is synthesized. This memory element is necessary to hold the signal value for the cases in which the `if` condition is false. As shown in the figure 3, the synthesis of an `if` instruction results in a latch that it is enabled by the `if` condition.

The use of the `wait` instruction allows the description of circuits that are synchro-

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

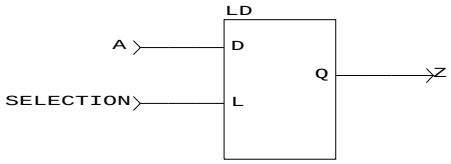

Figure 3: Synthesis of an `if` without the `else` branch.

nized with some signal, usually the clock of the circuit. Although most of the synthesis tools agree with this style of description for synchronous circuits, problems arise in the description style adopted for initialization of the circuit.

Figure 4 presents the synthesis of a flip-flop with synchronous reset through the use of an `wait` and `if` statement. In this description the instructions in any branch of the `if` statement are only executed on the positive transition of the `clock` signal.

For the cases that an asynchronous initialization is required, the description in figure 5 can be used. In this description the `wait` statement hold the execution of the process until either a clock impulse occur or the reset signal became active. Then, an `if` statement determines the asynchronous or synchronous behavior of the description.

Finite state machines are commonly used in the project of digital circuits. Its description in VHDL using a synthesizable style requires the use of two process. One is responsible for the description of the sequential behavior of the machine and its synchronous and/or asynchronous initialization. The other, describes the combinational part of the machine, the transitions between states and the generation of the outputs values.

Figure 6 describes a Moore finite state machine for the detection of `"111"` input sequence. Table 11, that defines the state transitions of the machine, is specified in the combinational process of the VHDL description (process with the label `comb`). In this process, using a `case` controlled by the present state of the machine, the next state and the outputs are assigned. If a Mealy machine is required, the assignment of the outputs variables should be done in the branches of the `if` statements. In that case the outputs will depend both on the state of the machine and the present values of the inputs.

The process with the label `sync` of figure 6 is responsible for asynchronous initialization of the machine and the update of its state on each clock cycle. Therefore, it is in this

```
entity circuit is
    port(data, clock, reset: in vlbit;
         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;
```

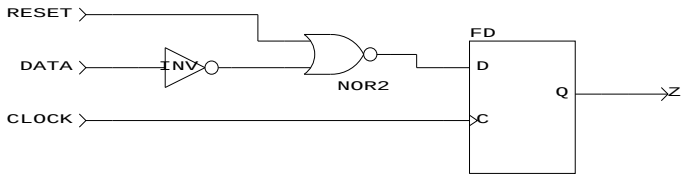

Figure 4: Synthesis of a flip-flop with synchronous reset.

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


architecture async_rst of circuita is
begin
    process
    begin
        wait until (clock'event and clock = '1') or reset = '1' ;
        if reset = '1' then
            z <= '0';                    -- asynchronous reset
        else
            z <= data;                   -- asynchronous operation
        end if;
    end process;
end async_rst;
```

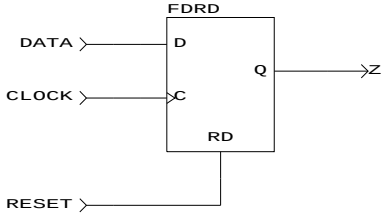

Figure 5: Synthesis of a flip-flop with asynchronous reset.

process that the sequential behavior of the circuit is specified. This process is needed

both for Moore and Mealy finite state machine descriptions.

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 11: Transition table of the finite state machine.

In some circuits the use of a two-value logic system (*logic-one*, `'1'`, and *logic-zero*, `'0'`) is not sufficient for its correct specification. In such circuits, which contain references to *high-impedance* or *don't-care* values, the use of multi-value logic is required. The version evaluated of the Viewlogic synthesis tool does not yet support the standardized package from IEEE for multi-logic values - `std_logic_1164`. So, the predefined types `vlbit` and/or `vlbit_vector`, that support in the Viewlogic tool a multi-logic value, should be used.

When the circuit outputs are not completely specified for all the inputs combinations, a *don't-care* value, `'X'`, should be assigned to them. In this way it is possible for the synthesis tool to generate a better circuit using 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 `vlbit_vector` type allows the assignment of the output to a vector of *don't-cares*, for input values from `"1010"` to `"1111"`, and force all outputs to *high-impedance* when the decoder is not enabled. For the Viewlogic tool the synthesis of three-state drivers is only possible if the assignment of the "high-impedance" value to the outputs is done in a conditional signal assignment, as presented in figure 7.

```
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              -- synchronous behavior
    begin
        wait until (clock'event and clock = '1') or reset = '1' ;
        if reset = '1' then
            present_state <= zero;
        else
            present_state <= next_state;
        end if;
    end process;

end Moore;
```

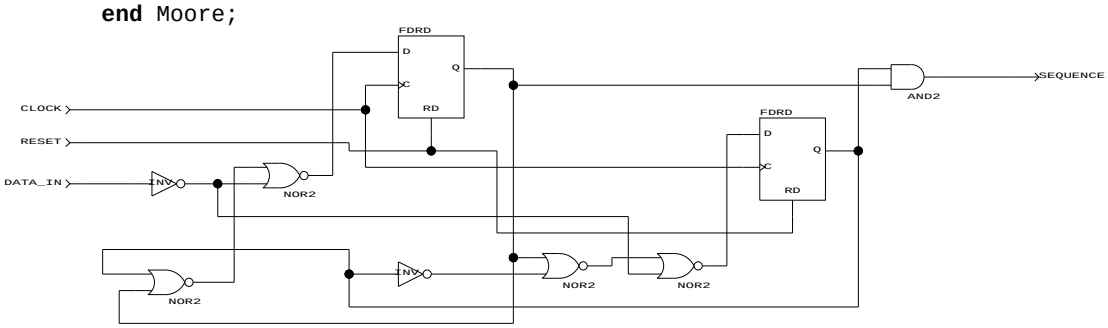

Figure 6: Moore state machine to detect the sequence "111".

```
entity bcd_7seg is
    port(data_in: in vlbit_vector(3 downto 0);
         enable: in vlbit;
         data_out: out vlbit_vector(6 downto 0));
end bcd_7seg;

architecture combinational of bcd_7seg is
    signal data_out_temp : vlbit_vector(6 downto 0);
begin

    data_out <= data_out_temp when enable = '1'
                else "ZZZZZZZ";                          -- three-state output

    process (data_in)
    begin
        case data_in is                                  --abcdefg
            when "0000" => data_out_temp <= "1111110";
            when "0001" => data_out_temp <= "1100000";
            when "0010" => data_out_temp <= "1011011";
            when "0011" => data_out_temp <= "1110011";
            when "0100" => data_out_temp <= "1100101";
            when "0101" => data_out_temp <= "0110111";
            when "0110" => data_out_temp <= "0111111";
            when "0111" => data_out_temp <= "1100010";
            when "1000" => data_out_temp <= "1111111";
            when "1001" => data_out_temp <= "1110111";
            when others => data_out_temp <= "XXXXXXX"; -- don't-care output
        end case;
    end process;
end combinational;
```

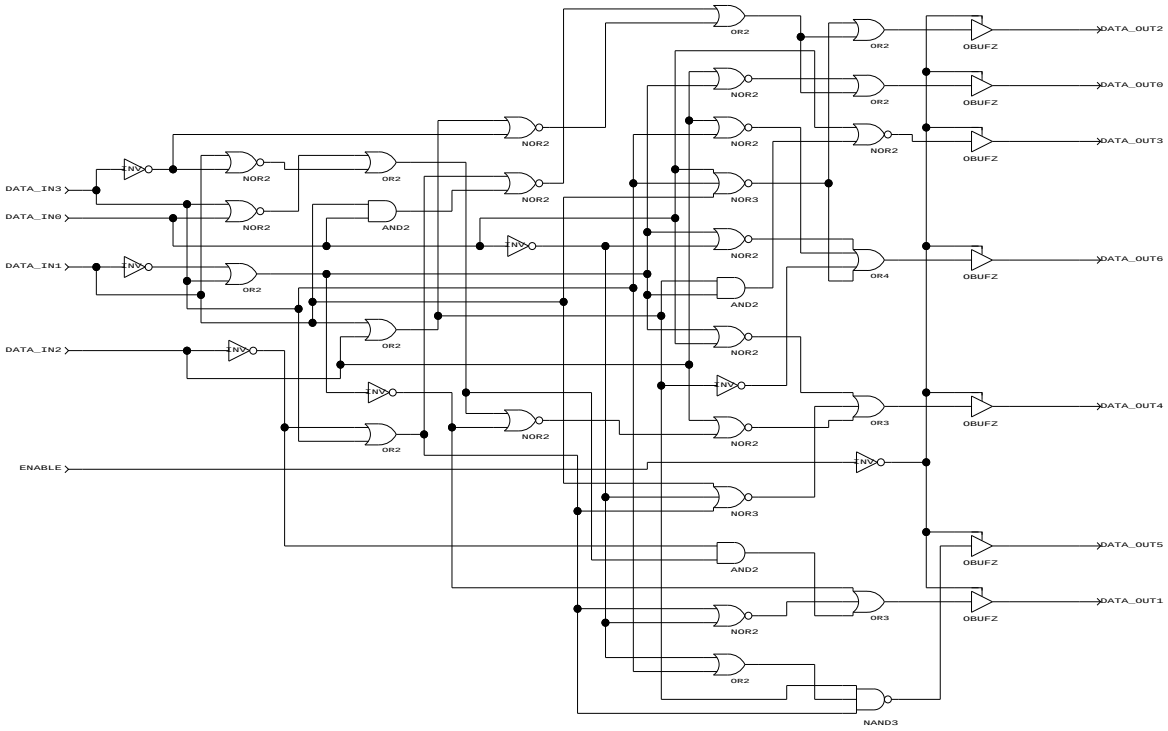

Figure 7: The use of multi-value logic values.

4 Portability Issues

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 maintain 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. The synthesis package under development by the IEEE 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.

As presented is section 3.1 and 3.2 the Viewlogic synthesis tool has more limitations, regarding the supported subset and description style, than the Synopsys tool, which was the one selected for the QuickChips design environment [Flores 94a]. Thus, to port a description into the Viewlogic synthesis tool is necessary to perform, at least, some basic changes to the code:

1. Change the standard types `std_logic` and `std_logic_vector` to the Viewlogic correspondent multi-value logic types, `vlbit` and `vlbit_vector`, respectively. Remove all references to the `IEEE` libraries.
2. Identify the all the sequential processes of the description and change them in accordance with the style described in table 9.
3. Transform all signal assignments to *high-impedance* logic values into conditional signal assignments.
4. Change (or eliminate) the generic values if in the synthesized circuit a value different form the default ones is wanted or they are assigned to a signal. In this case replace the generic by the desired value.

[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.

To exemplify how to port a VHDL description to the Viewlogic synthesis tool, we selected a some circuits from the IC-Blocks library [Flores 95]. As examples of combinational circuits we selected an adder and a decoder.

Figure 8 shows the original VHDL code of the adder from IC-Blocks library. Figure 9 shows this description changed according to the above basic rules and the synthesized result of four bit adder. Note that in this description only the rules 1 and 4 were applied because it is a pure combinational description without any assignments to *high-impedance* logic value.

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

entity ADD is
    generic(WIDTH: integer := 8);
    port(A: in std_logic_vector(WIDTH - 1 downto 0);
         B: in std_logic_vector(WIDTH - 1 downto 0);
         CIN: in std_logic;
         SUM: out std_logic_vector(WIDTH - 1 downto 0);
         COUT: out std_logic );
end ADD;

architecture BEHAVIORAL of ADD is
begin
    process (A,B,CIN)
        variable a_aux, b_aux : std_logic_vector(WIDTH downto 0);
        variable sum_aux : std_logic_vector(WIDTH downto 0);
    begin
        a_aux := '0' & A;
        b_aux := '0' & B;

        sum_aux := a_aux + b_aux;
        sum_aux := sum_aux + CIN;


        SUM <= sum_aux(WIDTH - 1 downto 0);
        COUT <= sum_aux(WIDTH);
    end process;
end BEHAVIORAL;
```

Figure 8: VHDL code of the ADD component from IC-Blocks library.

```
entity ADD is
    generic(WIDTH: integer := 4);
    port(A: in vlbit_vector(WIDTH - 1 downto 0);
         B: in vlbit_vector(WIDTH - 1 downto 0);
         CIN: in vlbit;
         SUM: out vlbit_vector(WIDTH - 1 downto 0);
         COUT: out vlbit );
end ADD;

architecture BEHAVIORAL of ADD is
begin
    process (A,B,CIN)
        variable a_aux, b_aux : vlbit_vector(WIDTH downto 0);
        variable sum_aux : vlbit_vector(WIDTH downto 0);
    begin
        a_aux := '0' & A;
        b_aux := '0' & B;

        sum_aux := a_aux + b_aux;
        sum_aux := sum_aux + CIN;


        SUM <= sum_aux(WIDTH - 1 downto 0);
        COUT <= sum_aux(WIDTH);
    end process;
end BEHAVIORAL;
```

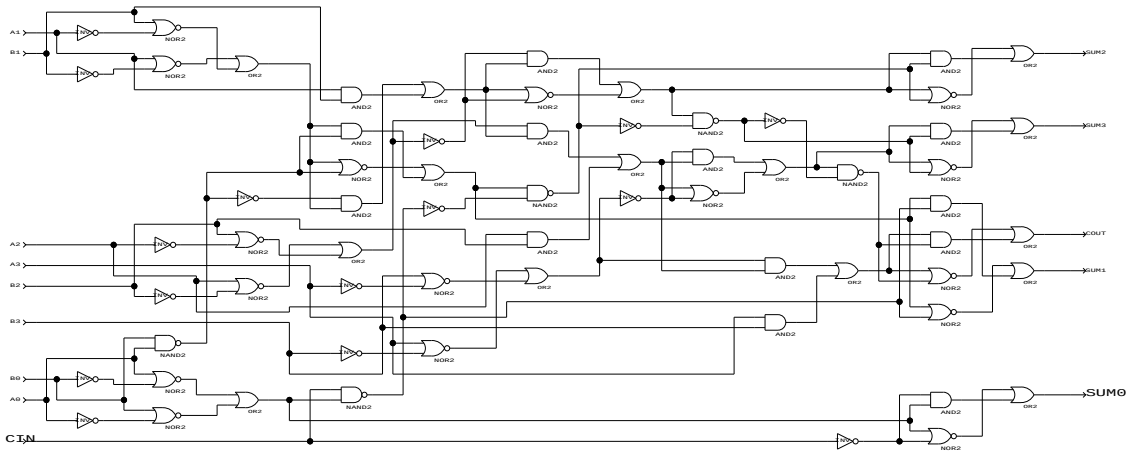

Figure 9: Synthesized adder description for Viewlogic tool and its implementation.

In some cases changing a description using only the basic rules is not enough to make it synthesizable by this synthesis tool. Figure 10 describe a decoder with parameterizable width and parameterizable values for active and inactive outputs. To synthesize this combinational description, all generic values had to be substituted by specific values and all the redundant code eliminated. Figure 11 shows the resulting VHDL description and the synthesized circuit, for the following case: three inputs, active output set to *logic-one* (`'1'`) and inactive outputs set to *logic-zero* (`'0'`).

```
library ieee;
use ieee.STD_LOGIC_1164.all;
use ieee.STD_LOGIC_unsigned.all;

entity DECOD is
    generic(WIDTH_DECOD: integer := 3;
            INACTIVE_OUT: integer range 0 to 2 := 0;
            ACTIVE_OUT: integer range 0 to 2 := 1);
    port(DATA_IN: in STD_LOGIC_VECTOR(WIDTH_DECOD-1 downto 0);
         DATA_OUT: out STD_LOGIC_VECTOR(2**WIDTH_DECOD - 1 downto 0));
end DECOD;

architecture behavioral of DECOD is
begin
    process (DATA_IN)
        variable data_aux: STD_LOGIC_VECTOR(2**WIDTH_DECOD -1 downto 0);
        variable index: integer range 0 to 2**WIDTH_DECOD-1;
    begin
        index := CONV_INTEGER(DATA_IN);
        lp: for i in DATA_OUT'range loop
            if i /= index then
                case INACTIVE_OUT is
                    when 0 =>
                        DATA_OUT(i) <=  '0';
                    when 1 =>
                        DATA_OUT(i) <=  '1';
                    when 2 =>
                        DATA_OUT(i) <=  'Z';
                end case;
            else
                case ACTIVE_OUT is
                    when 0 =>
                        DATA_OUT(i) <= '0';
                    when 1 =>
                        DATA_OUT(i) <= '1';
                    when 2 =>
                        DATA_OUT(i) <= 'Z';
                end case;
            end if;
        end loop lp;
    end process;
end behavioral;
```

Figure 10: VHDL code of the DECODER component from IC-Blocks library.

```
entity DECOD is
    port(DATA_IN: in VLBIT_VECTOR(2 downto 0);
         DATA_OUT: out VLBIT_VECTOR(7 downto 0));
end DECOD;

architecture behavioral of DECOD is
begin
    process (DATA_IN)
        variable data_aux: VLBIT_VECTOR(7 downto 0);
        variable index: integer range 0 to 7;
    begin
        index := DATA_IN;
        lp: for i in DATA_OUT'range loop
            if i /= index then
                DATA_OUT(i) <=  '0';
            else
                DATA_OUT(i) <=  '1';
            end if;
        end loop lp;
    end process;
end behavioral;
```

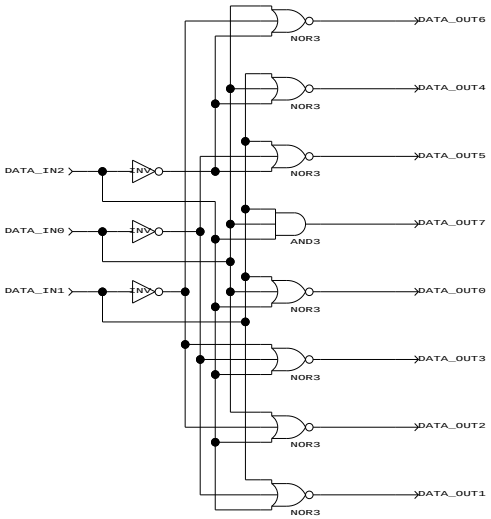

Figure 11: Synthesized decoder description for Viewlogic tool and its implementation.

As an example of a sequential circuit the component REGFF from the IC-Blocks library was selected. This component is a register bank, made of D type flop-flops, with parameterizable width, synchronous initialization value, asynchronous initialization value and outputs values when the register bank is not selected. Figure 12 presents the VHDL description of this component in IC-Blocks library.

To obtain the synthesizable description of figure 13 it was necessary to change it according to all the four rules presented above. All reference to `IEEE` standards were removed and the Viewlogic supported types were used. The synchronization and initialization of the process that describes this sequential circuit was changed according to the coding style, using a `wait` statement. The attribution of *high-impedance* values (`'Z'`) to outputs was accomplish using a conditional signal assignment statement. Some generic

declarations were eliminated and the generic references replaced by the actual values. The synthesized circuit, also presented in figure 13, is a 8 register data bank with asynchronous initialization to `"00000000"`, synchronous initialization to `"11111111"` and with three-state output drivers.

To port a VHDL description from the Viewlogic synthesis environment to the QC design environment minor changes are necessary. Actually, in most of the cases it should be enough to change the Viewlogic predefined types to the IEEE standard types. This results from fact the QC selected tool[3] for the synthesis task has a more flexible description style and supports a larger subset of the language.

5 Conclusion

The use of synthesis technology allows to obtain "automatically" a logic implementation of a circuit, mapped in cells of a given technology and satisfying a set of constraints (for example, area and/or delay), from a specification written in a hardware description language. The use of this technology in the project of integrated circuits can significantly reduce the overall design turnaround time.

In order to make accessible the synthesis technology to SMEs it was necessary to create an interface, in the QuickChips design environment, to a low-cost, commercially available, PC-based synthesis tool.

In this report we evaluated the PC version of the Viewlogic synthesis tool, `vhdldes V2.05`. It was explained how to integrate this tool in the QC design flow, through the use of one of the design environment selected languages, EDIF. For a fast design process is very important to know the supported synthesizable VHDL subset and the description style adopted by the synthesis tool. Only then, the advantages of using synthesis technology are significant to improve the design process. So, we describe in detail the VHDL constructs supported by Viewlogic tool and presented a suitable description style for synthesis using some basic examples.

Although being VHDL a standard language for modeling, its use for synthesis is not standard. Each synthesis tool support a subset of the language and has its own allowed description style. In this report some considerations were made about code portability and a set of rules to import a description form the main QC design environment was given. Some descriptions from the IC-Blocks library, have been ported to the Viewlogic tool, and synthesized, to illustrate the presented rules.

[3] `design_compiler` from Synopsys Inc.

```
library IEEE;

use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_arith.all;

entity REGFF is
    Generic (WIDTH : INTEGER := 8;
             AINITIAL : integer := 0;
             SINITIAL : integer := 0;
             NO_READ : integer range 0 to 3 := 2);
    Port (CLK : In STD_LOGIC;
          DATA_IN : In STD_LOGIC_VECTOR (width-1 downto 0);
          LOAD : In STD_LOGIC;
          READ : In STD_LOGIC;
          AINIT: In STD_LOGIC;
          SINIT: In STD_LOGIC;
          DATA_OUT : out STD_LOGIC_VECTOR (width-1 downto 0) );
end REGFF;

architecture BEHAVIORAL of REGFF is
    signal data_out_dummy: STD_LOGIC_VECTOR(width-1 downto 0);
begin
    process (READ, data_out_dummy)
    begin
        if READ = '1' then
            data_out <= data_out_dummy;
        else
            case NO_READ is
                when 0 =>
                    data_out <= (others => '0');
                when 1 =>
                    data_out <= (others => '1');
                when 2 =>
                    data_out <= (others => 'Z');
                when 3 =>
                    data_out <= (others => '-');
            end case;
        end if;
    end process;
    process(AINIT, clk)
    begin
        if AINIT = '1' then
            data_out_dummy <= CONV_STD_LOGIC_VECTOR
                (CONV_UNSIGNED(AINITIAL,WIDTH),WIDTH);
        elsif clk'event and clk = '1' then
            if SINIT = '1' then
                data_out_dummy <= CONV_STD_LOGIC_VECTOR
                    (CONV_UNSIGNED(SINITIAL,WIDTH),WIDTH);
            else
                if LOAD = '1' then
                    data_out_dummy <= data_in;
                else
                    data_out_dummy <= data_out_dummy;
                end if;
            end if;
        end if;
    end process;
end BEHAVIORAL;
```

Figure 12: VHDL code of the REGFF component from IC-Blocks library.

```
entity REGFF is
    Generic (WIDTH : INTEGER := 8);
    Port (CLK : In VLBIT;
          DATA_IN : In VLBIT_VECTOR (width-1 downto 0);
          LOAD, READ : In VLBIT;
          AINIT, SINIT: In VLBIT;
          DATA_OUT : out VLBIT_VECTOR (width-1 downto 0) );
end REGFF;

architecture BEHAVIORAL of REGFF is
    signal data_out_dummy: VLBIT_VECTOR(width-1 downto 0);
begin
    data_out <= data_out_dummy when READ = '1'
                else "ZZZZZZZZ";          -- three-state output
    process
    begin
        wait until (clk'event and clk = '1') or AINIT = '1';
        if AINIT = '1' then
            data_out_dummy <= "00000000";
        else
            if SINIT = '1' then
                data_out_dummy <= "11111111";
            else
                if LOAD = '1' then
                    data_out_dummy <= data_in;
                else
                    data_out_dummy <= data_out_dummy;
                end if;
            end if;
        end if;
    end process;
end BEHAVIORAL;
```

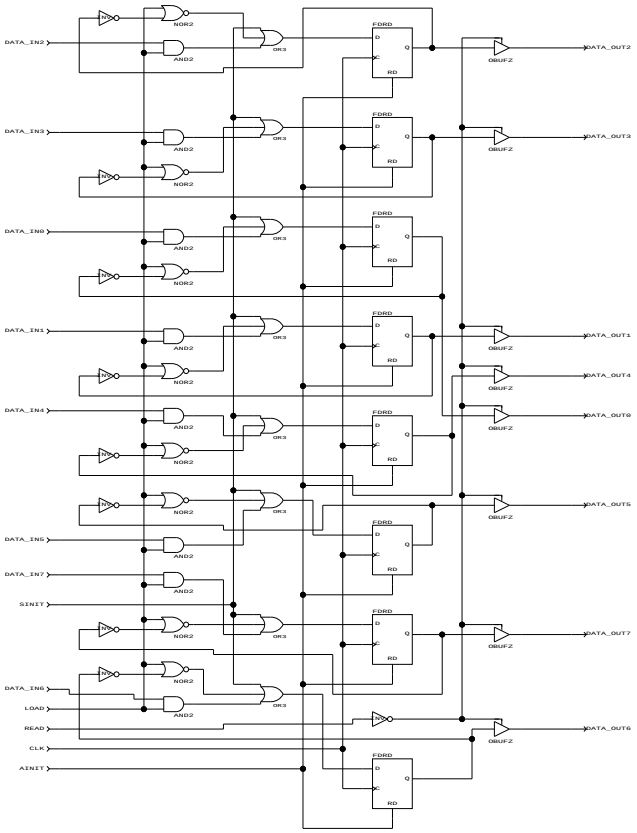

Figure 13: Synthesized register description for Viewlogic tool and its implementation.

References

[Abreu 94] José Pedro Abreu. Alpha Version of the CAD System. Technical report, Project ESPRIT 6043, April 1994.

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

[Flores 94a] Paulo Flores. Demonstration of Tools. Technical report, Project ESPRIT 6043, May 1994.

[Flores 94b] Paulo Flores. Selected Languages and Formats. Technical report, Project ESPRIT 6043, May 1994.

[Flores 95] Paulo Flores. IC Blocks Reference Manual. Technical report, Project ESPRIT 6043, April 1995.

[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.

[Viewlogic 92a] Viewlogic Systems, Inc. *VHDL Designer User's Guide and Tutorial*, January 1992. Version F.

[Viewlogic 92b] Viewlogic Systems, Inc. *VHDL Reference Manual for Synthesis*, January 1992. Version A.