PLS

VHDL

Table of Contents

VHDL ______ **1**
Introduction ______ 1
1. How to declare a circuit in VHDL ______ 2
1.1. General declaration ______ 2
1.1.1. Entity and architecture declaration ______ 2
1.1.1.1. Entity declaration ______ 2
1.1.1.2. Architecture declaration ______ 2
1.1.1.3. Generic parameters declaration ______ 2
1.1.2. Accepted VHDL types ______ 3
1.1.2.1. Accepted types in VHDL ______ 3
1.1.2.2. Additional accepted types in VHDL ______ 4
1.2 Hierarchical description ______ 5
1.2.1 How to assemble several blocks ______ 5
1.2.2. Particular case: repetitive or bit slice structures ______ 7
2. Data flow VHDL descriptions ______ 8
2.1. How to describe Boolean equations ______ 8
2.1.1. Constants ______ 8
2.1.2. Truth Table ______ 8
2.1.3. Don't care ______ 9
2.1.4. How the logic is synthesized ______ 10
2.2. How to describe multilevel logic ______ 11
2.2.1. Gate netlist ______ 11
2.2.2. Netlist using arithmetic operators ______ 11
2.2.3. Optimizations ______ 13
2.2.3.1. Resource folding and minimization of the number of multiplexors ______ 13
2.2.3.2. Recognition of common sub-expressions ______ 14
2.2.3.3. Synthesis of well-balanced trees ______ 15
2.2.3.4. Expression simplification ______ 16
2. 3. How to include memory elements using PLS prestored package ______ 16
3. Behavioral VHDL descriptions ______ 19
3.1. Combinational circuits descriptions using processes, functions and procedures ______ 19
3.1.1. Combinational process ______ 19
3.1.2. Truth tables ______ 21
3.1.3. Netlist declaration ______ 23
3.1.4. Repetitive or bit slice structure ______ 25
3.2. Sequential circuits descriptions using processes ______ 25
3.2.1. Description styles ______ 25
3.2.1.1. Sequential process with a sensitivity list ______ 25
3.2.1.2. Sequential process without a sensitivity list ______ 26
3.2.2. Examples : register and counter descriptions ______ 26
3.2.3. RAM inference ______ 28
3.2.4. Multiple *wait* statements descriptions ______ 31
3.3. Hierarchy handling through functions and procedures ______ 32
4. General examples using all the VHDL styles ______ 34
4.1. Example 1: timer/counter (prepbenchmark 2) ______ 34
4.2. Example 2: memory map (prepbenchmark 9) ______ 41
5. Finite State Machine Synthesis ______ 44
5.1. Explicit VHDL FSM template with attribute specifications ______ 44
5.1.1. VHDL template ______ 44
5.1.1.1. State register and next state equations ______ 44
5.1.1.2. Latched and non latched outputs ______ 45
5.1.1.3. Latched inputs ______ 45

5.1.2. State assignments 48
5.1.2.1. State assignment optimizations 48
5.1.2.2. User controlled state assignment 49
5.1.3. Choice of the flip-flops 49
5.2. Implicit VHDL FSM template 50
5.2.1. VHDL template 50
5.2.1.1. State register and next state equations 50
5.2.1.2. Latched and non latched outputs 50
5.2.1.3. Latched inputs 50
5.2.2. State assignments 55
5.2.2.1. State assignment optimizations 55
5.2.2.2. User controlled state assignment 55
5.3. Symbolic FSM identification 56
5.4. Templates using multiple wait statements 57
5.5. Handling FSMs within your design 58
5.5.1. Pre-processing or separate FSM handling 58
5.5.2. Embedded FSMs 58
6. Communicating Finite State Machines Synthesis 60
6.1. Introduction 60
6.2. Communicating FSMs 60
6.2.1. Concurrent communicating FSMs 60
6.2.2. Hierarchical or master-slave communicating FSMs 62
6.3. Processes based description 64
6.3.1. Modeling 64
6.3.2. Synthesis 65
6.4. Structural composition of FSMs 66
6.4.1. Modeling 66
6.4.2. Synthesis 69
7. State Chart Synthesis 71
7.1. Explicit VHDL state chart template 71
7.1.1. Modeling 71
7.1.2. Synthesis process with FSM and data path extraction with maximal resource sharing 75
7.1.2.1. Data path generation 75
7.1.2.2. Controller specification extraction and synthesis 76
7.1.2.3. Final circuit 77
7.1.3. Operator or function call 78
7.1.4. Data path optimization 79
7.1.5. Clocking scheme 80
7.2. Implicit VHDL state chart template 80
7.2.1. Modeling 80
7.2.2. Synthesis 82
7.3. General solution space exploration 83
8. Communicating State Charts Synthesis 84
8.1. Structural composition of state charts 84
8.1.1. Modeling 84
8.1.2. Synthesis 91
8.1.3. Synthesis of the example of §1.1 with FSMs and data paths with maximal sharing extraction 92
8.2. Processes based description 95
8.2.1. Modeling 95
8.2.2. Synthesis 99
9. Compatibility Between Synopsys and PLS/VHDL 100
9.1. Configuration unit 100
9.2. Recursive functions and procedures 101
9.3. Expressions 101
9.3.1. Expressions used in "port map" of component instantiations: 101
9.3.2. Expressions used with the "/", "mod" and "rem" operators: 102

9.4. Predefined packages and conversion functions 103
9.4.1. Using the vhdl option vhdl style synopsys 103
9.4.2. Using package "ARITHMETIC" of library "Synopsys": 103
9.4.3. Using package "BV_ARITHMETIC" of library "Synopsys": 104
9.4.4. Using package "ARITHMETIC" of library "MVL_7": 104
9.4.5. Using package "STD_LOGIC_ARITH" of "Synopsys" library: 105
9.5. Sensitivity list in a process 106
9.6. Indexed and sliced signals in a sensitivity list of a process 106
9.7. Synopsys predefined attributes 107
9.8. Compilation directives 107

Appendix 1: VHDL Subset 108

Appendix 2: Synthesis Directive File Syntax 113

VHDL

Introduction

The VHDL in the Programable Logic Synthesizer offers a very broad set of constructs for describing even the most complicated logic in a compact fashion. Dedicated macrogenerators, known as MACRO+, make design with arithmetic modules, counters, shifters a breeze and provide an advantage in performance and compactness of the design.

For finite state machines, several process type descriptions are accepted. To satisfy the CPLD/FPGA designers a special effort has been done on finite state machine synthesis (six state assignments) with an extension to communicating FSMs which may be fully concurrent or hierarchical. In this last case, a specific state assignment for each FSM can be called to obtain an optimized solution.

This manual assumes that the reader is familiar with the basic notions of VHDL. This manual is aimed at indicating how to use VHDL for synthesis purpose.

The supported subset of VHDL is described in appendix 1.

1. How to declare a circuit in VHDL

1.1. General declaration

1.1.1. Entity and architecture declaration

A circuit description is made up of two parts : the interface defining the I/O ports and the body. In VHDL, the entity corresponds to the interface and the architecture describes the behavior.

1.1.1.1. Entity declaration

In the entity, the I/O ports of the circuits are declared. Each port has a name, a mode (in, out, inout or buffer) and a type (See ports A,B,C,D,E in the figure 1).

1.1.1.2. Architecture declaration

In the architecture, internal signals may be declared. Each internal signal has a name and a type (See signal T in the figure 1). In figure 1, all bold words are key words of the VHDL language and are mandatory in the description.

```
entity EXAMPLE is
        port (    A,B,C : in BIT;
                  D,E :     out BIT );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
        signal T : BIT;
begin
        ...
end ARCHI;
```

Figure 1: *Entity and architecture example*

1.1.1.3. Generic parameters declaration

Generic parameters may also be declared in the entity declaration part. These parameters may be for example the width of the design.

1.1.2. Accepted VHDL types

1.1.2.1. Accepted types in VHDL

The accepted types are:

- the enumerated types :

  -BIT ('0','1'),

  -BOOLEAN (false, true),

  -STD_LOGIC ('U','X','0','1','Z','W','L','H','-'),
  where 'U' means unitialized,
  'X' means unknown,
  '0' means low,
  '1' means high,
  'Z' means high impedance,
  'W' means weak unknown,
  'L' means weak low,
  'H' means weak high,
  '-' means don't care.
  For PLS synthesis, the '0' and 'L' values are treated identically, idem for the '1' and 'H' values. The 'U', 'X', 'W' and '-' values are treated as don't care. The 'Z' value is treated as high impedance.

  -any enumerated type of identifiers defined by the user as for example:

  type COLOUR **is** (RED,GREEN,YELLOW);

- the bit vector types :

  -BIT_VECTOR

  -STD_LOGIC_VECTOR

  The unconstrained types which are types whose length is not defined are not accepted.

- the integer types:

  -INTEGER

The types BIT, BOOLEAN, BIT_VECTOR and INTEGER are VHDL predefined types.

The types STD_LOGIC and STD_LOGIC_VECTOR are declared in the package STD_LOGIC_1164 defined by the IEEE. This package is compiled in the library IEEE. So, when using one of these types, the following 2 lines has to be written in the VHDL specification:

library IEEE;
use IEEE.STD_LOGIC_1164.**all**;

Figures 11 and 12 show a complete example using these types.

1.1.2.2. Additional accepted types in VHDL

These additional types are:

- the enumerated types :

  -STD_ULOGIC which contains the same nine values as the STD_LOGIC type, but doesn't contain predefined resolution functions.

  -X01 : subtype of STD_ULOGIC containing the 'X', '0' and '1' values,

  -X01Z : subtype of STD_ULOGIC containing the 'X', '0', '1' and 'Z' values,

  -UX01 : subtype of STD_ULOGIC containing the 'U', 'X', '0' and '1' values,

  -UX01Z : subtype of STD_ULOGIC containing the 'U', 'X', '0' ,'1' and 'Z' values.

- the bit vector types :

  -STD_ULOGIC_VECTOR

  -UNSIGNED

  -SIGNED

  The unconstrained types which are types whose length is not defined are not accepted.

- bidimensional array types defined by the user:

  Note that the two arrays must be constrained. An example is shown below:

  subtype WORD8 **is** STD_LOGIC_VECTOR (7 **downto** 0);

  type TAB12 **is array** (11 **downto** 0) **of** WORD8;

- the integer types:

  -NATURAL

  -POSITIVE

  -any integer type within a given range defined by the user. As an example, "**type** MSB **is range** 8 **to** 15;" means any integer greater than 7 or less than 16.

The types NATURAL and POSITIVE are VHDL predefined types.

The types STD_ULOGIC (and its subtypes X01, X01Z, UX01, UX01Z), STD_LOGIC, STD_ULOGIC_VECTOR and STD_LOGIC_VECTOR are declared in the package STD_LOGIC_1164 defined by the IEEE. This package is compiled in the library IEEE. So, when using one of these types, the following 2 lines has to be written in the VHDL specification:

library IEEE;
use IEEE.STD_LOGIC_1164.**all**;

The types UNSIGNED and SIGNED (defined as an array of STD_LOGIC) is declared in the package SL_ARITH. This package is compiled in the library ASYL. So, when using these types, the following 2 lines has to be written in the VHDL specification:

library ASYL;
use ASYL.SL_ARITH.**all**;

Figures 11 and 12 show a complete example using these types.

The bidimensional array signals or variables can be used as explained below:

As an example, lets consider the following declarations:

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);

type TAB12 is array (11 downto 0) of WORD8;
signal WORD_A : WORD8;
signal TAB_A, TAB_B : TAB12;
```

A bidimensional array signal or variable can be completely used:

```
TAB_A <= TAB_B;
```

Or, just an index of the first array can be specified:

```
TAB_A (8) <= WORD_A;
```

Or, just indexes of the first and second arrays can be specified:

```
TAB_A (8) (0) <= '1';
```

Or, just a slice of the first array can be specified:

```
TAB_A (11 downto 8) <= TAB_B (3 downto 0);
```

Or, just an index of the first array and a slice of the second array can be specified:

```
TAB_A (6) (3 downto 0) <= TAB_B (7) (4 downto 1);
```

Note also that the indexes can be variable.

1.2 Hierarchical description

1.2.1 How to assemble several blocks

Structural descriptions assemble several blocks and allow to introduce hierarchy in a design. The basic concepts of hardware structure are the component, the port and the signal. The component is the building or basic block. A port is a component I/O connector. A signal corresponds to a wire between components.

In VHDL, a component is represented by a *design entity*. This is actually a composite consisting of an *entity declaration* and an *architecture body*. The entity declaration provides the "external" view of the component; it describes what can be seen from the outside, including the component ports. The architecture body provides an "internal" view; it describes the behavior or the structure of the component.

The connections between components are specified within *component instantiation statements*. Such a statement specifies an instance of a component occurring inside an architecture of an other component or the circuit. Each component instantiation statement is labeled with an identifier. Beside naming a component declared in a local component declaration, a component instantiation statement contains an association list (the parenthesized list following the reserved word *port map*) that specifies which actual signals or ports are associated with which local ports of the component declaration.

The example in figure 2 gives the structural description of a half adder composed of 4 nand2 gates. The synthesized top level netlist is shown in figure 3.

```
entity NAND2 is
        port (   A,B :     in BIT;
                 Y :       out BIT );
end NAND2;
architecture ARCHI of NAND2 is
begin
        Y <= A nand B;
end ARCHI;
entity HALFADDER is
        port (   X,Y :     in BIT;
                 C,S :     out BIT );
end HALFADDER;
architecture ARCHI of HALFADDER is
component NAND2
        port (   A,B :     in BIT;
                 Y :       out BIT );
end component;
for all : NAND2 use entity work.NAND2(ARCHI);
signal S1, S2, S3 : BIT;
begin
        NANDA : NAND2 port map (X,Y,S3);
        NANDB : NAND2 port map (X,S3,S1);
        NANDC : NAND2 port map (S3,Y,S2);
        NANDD : NAND2 port map (S1,S2,S);
        C <= S3;
end ARCHI;
```

Figure 2: *Structural description of a half adder*

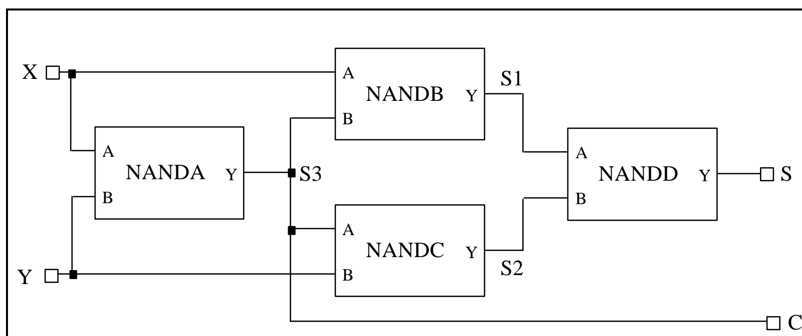

Figure 3: *Synthesized top level netlist*

1.2.2. Particular case: repetitive or bit slice structures

The repetitive structures are declared with the "generate" VHDL statement. For this purpose "for I in 1 to N generate" means that the bit slice description will be repeated N times. As an example, figure 4 gives the description of an 8 bit adder by declaring the bit slice structure.

```
entity EXAMPLE is
        port (   A,B :     in BIT_VECTOR (0 to 7);
                 CIN :     in BIT;
                 SUM :     out BIT_VECTOR (0 to 7);
                 COUT :          out BIT
                 );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
        signal C : BIT_VECTOR (0 to 8);
begin
        C(0) <= CIN;
        COUT <= C(8);
        LOOP_ADD : for I in 0 to 7 generate
                 SUM(I) <= A(I) xor B(I) xor C(I);
                 C(I+1) <= (A(I) and B(I)) or (A(I) and C(I))
                                or (B(I) and C(I));
        end generate;
end ARCHI;
```

Figure 4: *8 bit adder described with a "for...generate" statement*

The *"if <condition> generate"* statement is also supported for static (non dynamic) condition. Figure 5 shows such an example. It is a generic adder on N bits width but the width must be comprise between 4 and 32.

```
entity EXAMPLE is
        generic ( N : INTEGER := 8);
        port (   A,B :     in BIT_VECTOR (N downto 0);
                 CIN :     in BIT;
                 SUM :     out BIT_VECTOR (N downto 0);
                 COUT :    out BIT
                 );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
        signal C : BIT_VECTOR (N+1 downto 0);
begin
L1: if (N>=4 and N<=32) generate
        C(0) <= CIN;
        COUT <= C(N+1);
        LOOP_ADD : for I in 0 to N generate
                 SUM(I) <= A(I) xor B(I) xor C(I);
                 C(I+1) <= (A(I) and B(I)) or (A(I) and C(I))
                                or (B(I) and C(I));
        end generate;
end generate;
end ARCHI;
```

Figure 5: *N bit adder described with a "if...generate" and a "for ...generate" statement*

2. Data flow VHDL descriptions

2.1. How to describe Boolean equations

2.1.1. Constants

Constant values may be assigned to signals. In the example of figure 6, the output ports ZERO, ONE and TWO have 2 bits width and are assigned respectively to the constant binary values : "00", "01" and "10". (ZERO(0)=0, ZERO(1)=0, ONE(0)=1, ONE(1)=0, TWO(0)=0, TWO(1)=1).

```
entity EXAMPLE is
        port (  ZERO :          out BIT_VECTOR (1 downto 0);
                ONE :   out BIT_VECTOR (1 downto 0);
                TWO :   out BIT_VECTOR (1 downto 0) );

end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
        ZERO    <=      "00";
        ONE     <=      "01";
        TWO     <=      "10";
end ARCHI;
```

Figure 6: *Constants example*

2.1.2. Truth Table

This section describes how to declare Boolean equations in various formats. The standard way to declare a Boolean function is to declare its truth table. Consider for instance the example of figure 7. Instead of declaring globally a truth table, the output value may be given in a compact way declaring the 0 or 1 values or by successive decoding of the input variables.

A	B	S
0	0	1
0	1	1
1	0	0
1	1	1

Figure 7: Truth table

The figures 8 and 9 give two equivalent descriptions of the truth table of figure 7.

```
entity EXAMPLE is
        port (   A,B :     in BIT;
                 S :       out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
        S<=      '0' when (A = '1' and B = '0') else
                 '1';
end ARCHI;
```

Figure 8: *Truth table example*

```
entity EXAMPLE is
        port (   A,B :     in BIT;
                 S :       out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
        with (A and not B) select
        S<=      '0' when '1',
                 '1' when others;
end ARCHI;
```

Figure 9: *Truth table example*

2.1.3. Don't care

The declaration of don't care values is allowed. This means that these values have no importance for the circuit. They will be assigned later on for logic minimization purpose.

For the synthesis, the result of an equation : signal equal don’t care if a=’-‘ then ....will always return true, this may disagree with the simulation result

Figures 11 and 12 describe the truth table of figure 10. In figure 11, the output value is given by successive decoding of the input variables. In figure 12, the operator "&" is the concatenation operator and the operator "|" is used for values enumeration. For example, in figure 12, S=0 if (A,B,C)=1,0,0 or 1,0,1.

A	B	C	S
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	-
1	1	1	-

Figure 10: *Truth table with don't care*

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity EXAMPLE is
	port (	A,B,C :	in BIT;
		S :	out STD_LOGIC		);
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
	S<=	'1' when (A = '0') else
		'0' when (B = '0') else
		'-';
end ARCHI;
```

Figure 11: *Truth table example with don't care*

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity EXAMPLE is
	port (	A,B,C :	in BIT;
		S :	out STD_LOGIC		);
end EXAMPLE;

architecture ARCHI of EXAMPLE is
signal S1 : BIT_VECTOR (2 downto 0);
begin
	S1 <= A & B & C;
	with S1 select
	S<=	'1' when "000" | "001" | "010" | "011",
		'0' when "100" | "101",
		'-' when others;
end ARCHI;
```

Figure 12: *Truth table example with don't care*

2.1.4. How the logic is synthesized

The logic is synthesized by using the basic capabilities of PLS. These expressions are transformed according to the targets (binary decision diagrams for Actel, special ordered tree for Xilinx, various factorized forms for standard cells, etc...). The optimization criteria directly refer to PLS mappers and synthesis techniques. They are chosen according to the user requirements specified in the VHDL command (see later on).

2.2. How to describe multilevel logic

2.2.1. Gate netlist

Connection of gates are declared by using VHDL logical operators which are: and, nand, or, nor, not, xor, xnor. This is illustrated in the example of figure 13.

```
entity EXAMPLE is
       port (    A,B,C : in BIT;
                 S :      out BIT );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
       S<=       (A and B) or (not C);
end ARCHI;
```

Figure 13: *Gate netlist description*

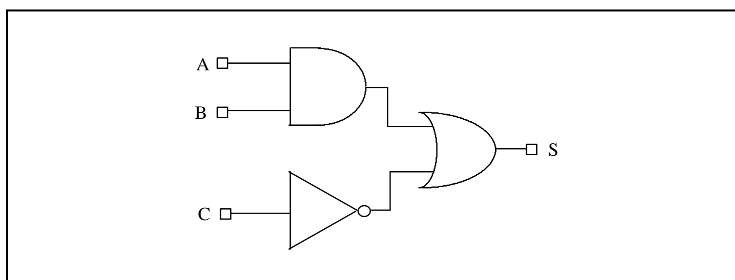

Figure 14: *Possible synthesized netlist*

2.2.2. Netlist using arithmetic operators

Arithmetic operators can be used instead of gates. These operators are the VHDL arithmetic operators: + (addition), - (subtraction), * (multiplication), / (division), mod (modulus), rem (remainder), abs (absolute value) and ** (exponentiation), as well as the VHDL relational operators : = (equality), /= (inequality), <, <=, > and >= (ordering). Some restrictions in the present version are made on the usage of some of these operators : the operators /, mod and rem are supported only if one of the two operands is a power of two, the abs operator is supported only for a constant operand, and the ** operator is supported only if the left operand is a power of two.

The VHDL operators are overloaded for BIT_VECTOR, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, UNSIGNED and SIGNED (Note that signed multipliers have not been implemented yet). The overloaded functions for types BIT_VECTOR, STD_LOGIC_VECTOR and STD_ULOGIC_VECTOR are written in the package ARITH which has been compiled in the ASYL library. So when using VHDL operators with BIT_VECTOR, STD_LOGIC_VECTOR or STD_ULOGIC_VECTOR operands, the following 2 lines have to be written in the VHDL specification:

```
library ASYL;
use ASYL.ARITH.all;
```

The overloaded functions for the types UNSIGNED and SIGNED are written in the package SL_ARITH which has been compiled in the ASYL library. So when using VHDL operators with UNSIGNED or SIGNED operands, the following 2 lines have to be written in the VHDL specification:

```
library ASYL;
use ASYL.SL_ARITH.all;
```

The number of bits of the operators is given by the width of the operands. In the example of figure 15, the two adders and the subtractor operators have 8 bits. The synthesized netlist from the previous description is given in figure 16.

Each operator can be instantiated in different manners according to the optimization criteria (speed, area, trade-off) specified in the synthesis command. This means that Boolean equations are stored for:

-4 types of adders (Ripple Carry Adder, Carry Skip Adder, Carry Look Ahead Adder, Conditional Sum Adder),

-4 types of subtractors based on the previous types of adders,

-comparators (=, /=, >, >=, <, <=) based on the previous types of subtractors,

-1 multiplier (multiplier of Braun).

These equations are carefully synthesized. For more details see the "Macro Block Handling: Macro+" part. If one of the operands is a constant, the iterative structure is left for a basic gate optimization.

```
library ASYL;
use ASYL.ARITH.all;
entity EXAMPLE is
        port (  A,B,C,D :       in BIT_VECTOR (7 downto 0);
                S :             out BIT_VECTOR (7 downto 0)
);
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
        S<=     (A + B) - (C + D);
end ARCHI;
```

Figure 15: *Example using arithmetic operators*

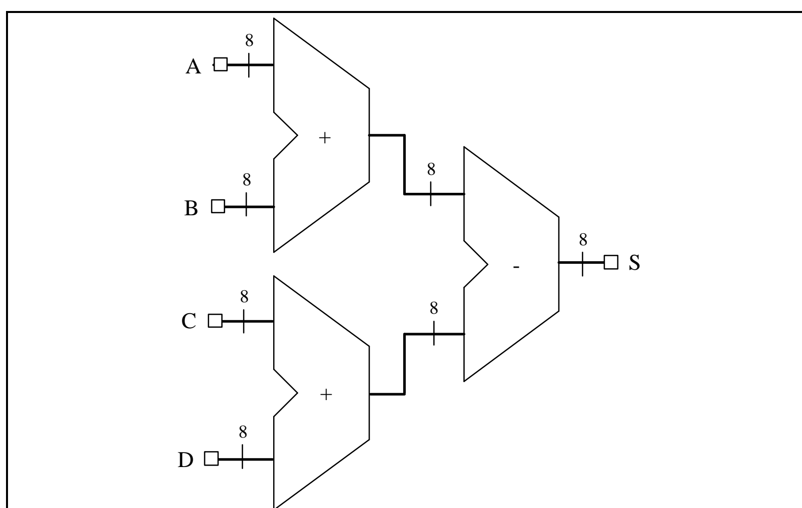

Figure 16: *Synthesized netlist*

2.2.3. Optimizations

2.2.3.1. Resource folding and minimization of the number of multiplexors

The optimizer first shares the operators and then reduces the number of required multiplexors by permuting the operands of the commutative operators. The example given in figure 17 illustrates the resource folding and the minimization of the number of multiplexors. The resulting netlist corresponding to the example described in figure 17 is shown in the figure 18. It contains 1 adder and 1 multiplexor instead of 2 adders or 1 adder and 2 multiplexors which may have been instantiated by direct reading.

```
library ASYL;
use ASYL.ARITH.all;

entity EXAMPLE is
        port (   A,B,C : in BIT_VECTOR (7 downto 0);
                 E :     in BIT;
                 S :     out BIT_VECTOR (7 downto 0)    );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
        with E select
        S<=      A + B when '1',
                 C + A when others;
end ARCHI;
```

Figure 17: *Example of resource folding*

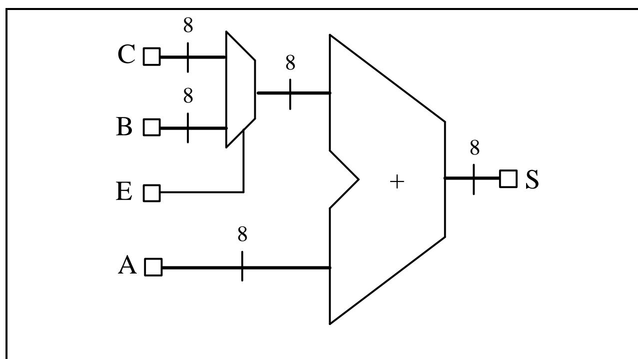

Figure 18: *Synthesized netlist*

2.2.3.2. Recognition of common sub-expressions

The optimizer recognizes common sub-expressions. In the example of figure 19, the optimizer recognizes the sub-expression "E * F". This sub-expression is only instantiated once and the design is synthesized using only 1 multiplier as shown in figure 20. For the sub-expressions "A * C" and "C * D", the optimizer shares the multiplier and instantiates one multiplexor. The adder is also shared . The final netlist is given in figure 20 and contains 1 adder, 2 multipliers and 1 multiplexor instead of 2 adders and 4 multipliers for an unoptimized synthesis.

```
library ASYL;
use ASYL.ARITH.all;

entity EXAMPLE is
        port (   A,B,C,D,E,F :   in BIT_VECTOR (7 downto 0);
                 S :             out  BIT_VECTOR  (15  downto
0) );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
        with B select
        S <=     E * F + A * C when "00000000",
                 C * D + E * F when others;
end ARCHI;
```

Figure 19: *Example of standard sub-expressions*

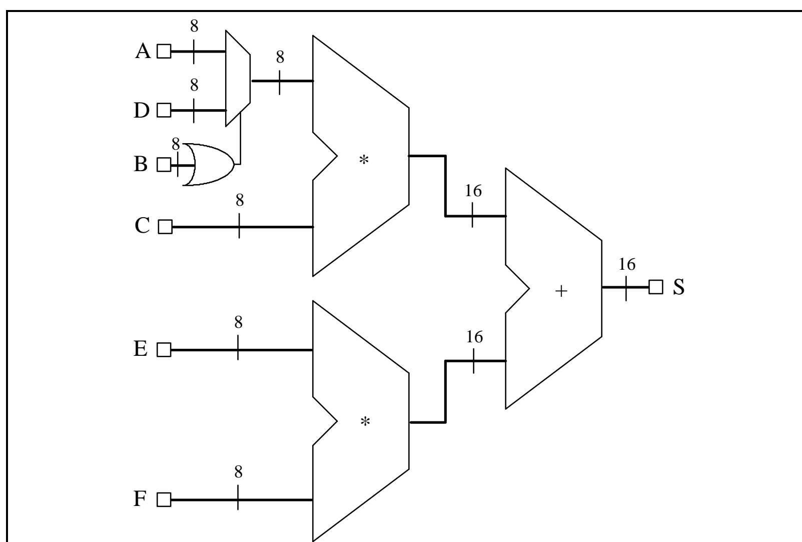

Figure 20: *Synthesized netlist*

2.2.3.3. Synthesis of well-balanced trees

The optimizer synthesizes well-balanced trees if an operator has a large number of inputs. In the example of figure 21 when recognizing the sub-expression "A + B + D + E", a multiplexor is instantiated allowing to add C or F to the sub-expression according to the value of G. The final netlist given in figure 22 contains 4 adders and 1 multiplexor. For the sub-expression "A + B + D + E" the optimizer creates a well-balanced minimal depth tree of adders which is a tree of [$\log_2$ levels] as 2 input adders only exist. In the synthesized netlist, data go through at most 3 adders instead of 4 between the inputs and the output.

```
library ASYL;
use ASYL.ARITH.all;

entity EXAMPLE is
        port (   A,B,C,D,E,F :   in BIT_VECTOR (7 downto 0);
                 G :             in BIT;
                 S :             out BIT_VECTOR (7 downto 0)
                 );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
        with G select
        S<=      A + B + C + D + E when '0',
                 E + B + D + A + F when others;
end ARCHI;
```

Figure 21: *Example of standard sub-expressions and well-balanced trees*

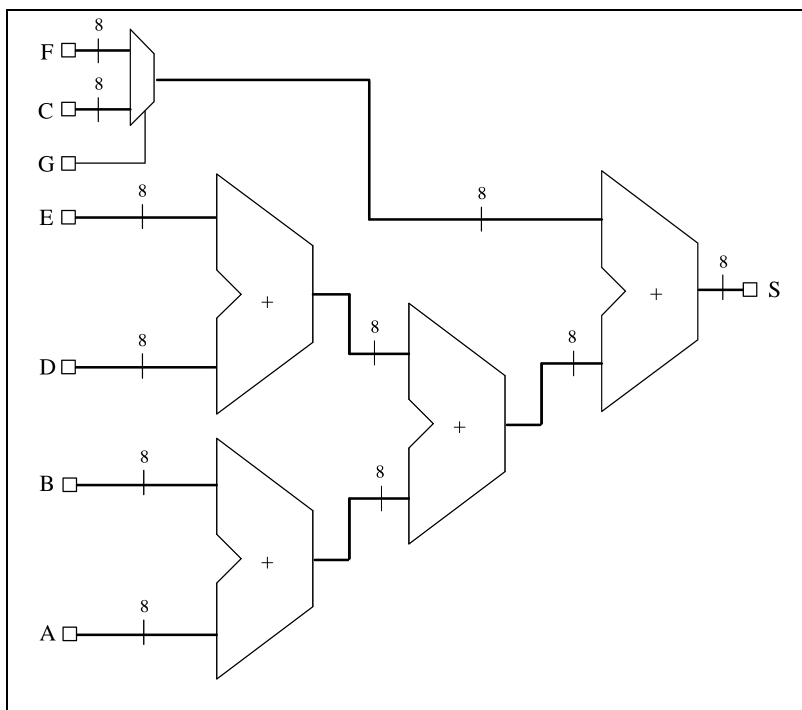

Figure 22: *Synthesized netlist*

2.2.3.4. Expression simplification

The optimizer is able to simplify expressions. In the example given in figure 23, it replaces the expressions [A - B"0000_0010" * A + A] and [A- A] by 0, so it replaces the output S by 0. In the synthesized netlist, all the bits of the output signal S are connected to the ground and no adder, subtractor or multiplier is instantiated.

```
library ASYL;
use ASYL.ARITH.all;

entity EXAMPLE is
        port (  A :     in BIT_VECTOR (7 downto 0);
                B :     in BIT;
                S :     out BIT_VECTOR (7 downto 0)
                );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
        with B select
        S<=     A - B"0000_0010" * A + A when '0',
                A - A when others;
end ARCHI;
```

Figure 23: *Example of expressions which are simplified*

2. 3. How to include memory elements using PLS prestored package

The instantiation of memory elements is made by prestored procedure calls. The general notion of user defined procedure will be defined later on (section 3.3.). PLS provides a package in which several procedures corresponding to single bit or vectored flip-flops and latches with or without asynchronous clear and preset inputs are described. The flip-flops are positive edge triggered and the latches are high level sensitive (transparent when the enable input is 1). These procedures are located in the package "ASYL_1164" and they are not user-defined procedures. This package has been compiled in the ASYL library. So when using these procedures, the 2 following lines have to be written in the VHDL specification:

```
library ASYL;
use ASYL.ASYL_1164.all;
```

These procedures are :

-DFF (DATA,CLOCK,Q) :
single bit D Flip-Flop,

-DFFC (DATA,CLEAR,CLOCK,Q) :
single bit D Flip-Flop with an active high asynchronous Clear,

-DFFP (DATA,PRESET,CLOCK,Q) :
single bit D Flip-Flop with an active high asynchronous Preset,

-DFFPC (DATA,PRESET,CLEAR,CLOCK,Q) :
single bit D Flip-Flop with active high asynchronous Clear and Preset,

-DFF_V (DATA,CLOCK,Q) :
vectored D Flip-Flop,

-DFFC_V (DATA,CLEAR,CLOCK,Q) :
vectored D Flip-Flop with an active high asynchronous Clear,

-DFFP_V (DATA,PRESET,CLOCK,Q) :
vectored D Flip-Flop with an active high asynchronous Preset,

-DFFPC_V (DATA,PRESET,CLEAR,CLOCK,Q) :
vectored D Flip-Flop with active high asynchronous Clear and Preset,

-DLATCH (DATA,ENABLE,Q) :
single bit D Latch,

-DLATCHC (DATA,CLEAR, ENABLE,Q) :
single bit D Latch with an active high asynchronous Clear,

-DLATCHP (DATA,PRESET, ENABLE,Q) :
single bit D Latch with an active high asynchronous Preset,

-DLATCHPC (DATA,PRESET, CLEAR, ENABLE,Q) :
single bit D Latch with active high asynchronous Clear and Preset,

-DLATCH_V (DATA,ENABLE,Q) :
vectored D Latch,

-DLATCHC_V (DATA,CLEAR, ENABLE,Q) :
vectored D Latch with an active high asynchronous Clear,

-DLATCHP_V (DATA,PRESET, ENABLE,Q) :
vectored D Latch with an active high asynchronous Preset,

-DLATCHPC_V (DATA,PRESET, ENABLE,Q) :
vectored D Latch with active high asynchronous Clear and Preset,

The procedures DFF, DFFC, DFFP, DFFPC, DLATCH, DLATCHC, DLATCHP and DLATCHPC are defined for the types BIT and STD_ULOGIC; the procedures DFF_V, DFFC_V, DFFP_V, DFFPC_V, DLATCH_V, DLATCHC_V, DLATCHP_V and DLATCHPC_V are defined for types BIT_VECTOR, STD_ULOGIC_VECTOR and STD_LOGIC_VECTOR. The full definition of these procedures is available for simulation purposes in the file "asyl_1164.vhdl" located in $ASYLDIR/vhdl/packages.

When PLS detects a call to such a procedure, it instantiates the corresponding memory element, assigning to the input/output the signals which are the parameters on the procedure call. For vectored flip-flops and latches, the number of bits of the input DATA and the output Q must be the same. This number fixes the width of the memory element. An example is given in figure 24. The synthesized netlist is shown in figure 25.

```
library ASYL;
use ASYL.ASYL_1164.all;

entity EXAMPLE is
	port (	DI :	in BIT_VECTOR (7 downto 0);
		CLK :	in BIT;
		DO :	out BIT_VECTOR (7 downto 0)
		);
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
	DFF_V ( DI, CLK, DO);
end ARCHI;
```

Figure 24: *Example of sequential netlist*

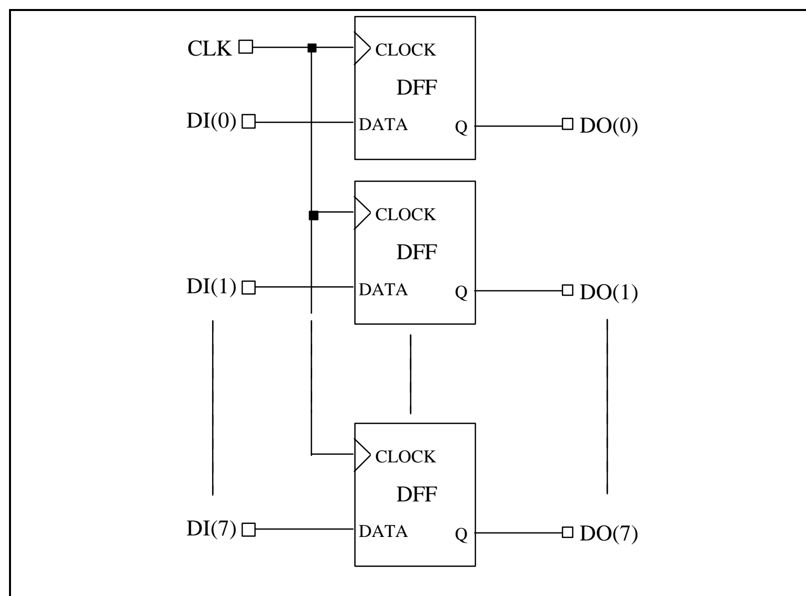

Figure 25: *Synthesized netlist*

3. Behavioral VHDL descriptions

The behavioral VHDL supported in this version includes combinational functions and procedures, combinational and sequential processes.

3.1. Combinational circuits descriptions using processes, functions and procedures

3.1.1. Combinational process

A combinational process assigns values to output Boolean functions called signals in a more sophisticated way than in the data flow style. The value assignments are made in a sequential mode. The latest assignments may cancel previous ones. An example is given in figure 26. First the signal S is assigned to 0, but later on for (A and B) = 1 the value for S is changed in 1.

```
entity EXAMPLE is
        port (   A, B :   in BIT;
                 S :      out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
process ( A, B )
begin
        S <= '0' ;
        if ((A and B) = '1') then
                 S <= '1' ;
        end if;
end process;
end ARCHI;
```

Figure 26: *Combinational process*

The example of figure 26 corresponds to the truth table of figure 27.

A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

Figure 27: *Truth table*

At the end of the process, the truth table of the outputs signals has to be completed. Therefore an output value has to be defined for each input value. This sequential mode declaration may simplify some descriptions. In the previous example '0' is treated as a default value before specifying the value '1' for A and B = '1'.

A combinational process has a sensitivity list appearing within parenthesis after the word "process". A process is activated if an event (value change) appears on one of the sensitivity list signals. This sensitivity list contains all condition signals and any signal appearing in the left part of an assignment. In the example of figure 28, the sensitivity list contains three signals which are the A, B and ADD_SUB signals.

A process may contain local variables. They are handled in a similar way than signals but of course are not outputs. In the example of figure 28 a variable named AUX is declared in the declarative part of the process and is assigned to a value with ":=" in the statement part of the process.

Figure 28 and figure 29 give two examples of combinational processes.

```
library ASYL;
use ASYL.ARITH.all;

entity ADDSUB is
     port (   A,B :          in BIT_VECTOR (3 downto 0) ;
              ADD_SUB : in BIT;
              S :             out BIT_VECTOR (3 downto 0));
end ADDSUB;

architecture ARCHI of ADDSUB is
begin
process ( A, B, ADD_SUB )
       variable AUX : BIT_VECTOR (3 downto 0);
begin
       if ADD_SUB = '1' then
              AUX := A + B ;
       else
              AUX := A - B ;
       end if;
       S <= AUX;
end process;
end ARCHI;
```

Figure 28: *Combinational process*

```
entity EXAMPLE is
       port (   A, B :   in BIT;
                S :        out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
process ( A,B )
       variable X, Y : BIT;
begin
       X := A and B;
       Y := B and A;
       if X = Y then
              S <= '1' ;
       end if;
end process;
end ARCHI;
```

Figure 29: *Combinational process*

Some examples described above by "Data flow and structural VHDL descriptions", will be described again by processes.

3.1.2. Truth tables

The truth table of figure 30 is recalled using the data flow style in figure 31 and for comparison using the behavioral style in figure 32. The conditional assignment in figure 31 is replaced by an if statement within a process in figure 32.

A	B	S
0	0	1
0	1	1
1	0	0
1	1	1

Figure 30 : *Truth table*

```
entity EXAMPLE is
        port (    A,B :     in BIT;
                  S :       out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
        S<=       '0' when (A = '1' and B = '0') else
                  '1';
end ARCHI;
```

Figure 31 : *Data flow description of a truth table*

```
entity EXAMPLE is
        port (    A,B :     in BIT;
                  S :       out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
process ( A, B )
begin
        if (A = '1' and B = '0') then
                  S<= '0';
        else
                  S<= '1';
        end if;
end process;
end ARCHI;
```

Figure 32: *Behavioral description of a truth table*

Similarly don't care values are supported in behavioral style. The truth table of figure 33 is described using the data flow style in figure 34 and using the behavioral style in figure 35. The selected assignment in figure 34 is replaced by a case statement within a process in figure 35.

A	B	C	S
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	-
1	1	1	-

Figure 33: *Truth table with don't care*

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity EXAMPLE is
        port (    A,B,C : in STD_LOGIC;
                  S :       out STD_LOGIC          );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
signal S1: STD_LOGIC_VECTOR (2 downto 0);
begin
        S1 <= A & B & C;
        with S1 select
        S<=       '1' when "000" | "001" | "010" | "011",
                  '0' when "100" | "101",
                  '-' when others;
end ARCHI;
```

Figure 34: *Data flow description of a truth table with don't care*

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity EXAMPLE is
        port (    A,B,C :  in STD_LOGIC;
                  S :      out STD_LOGIC          );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
signal S1: STD_LOGIC_VECTOR (2 downto 0);
begin
process ( A, B, C )
begin
        S1 <= A & B & C;
        case S1 is
                when "000" | "001" | "010" | "011" =>
                        S<= '1';
                when "100" | "101" =>
                        S<= '0';
                when others =>
                        S<= '-';
        end case;
end process; end ARCHI;
```

Figure 35: *Behavioral description of a truth table with don't care*

3.1.3. Netlist declaration

For netlist declaration, the process style does not alter at all the data flow description. The process and its sensitivity list are just put ahead. Figure 36 recalls the description of a gate netlist using the data flow style and figure 37 gives the same description using the behavioral style. The synthesized netlist stays the same.

```
entity EXAMPLE is
        port (    A,B,C :  in BIT;
                  S :      out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
        S<=       (A and B) or (not C);
end ARCHI;
```

Figure 36: *Gate netlist description using data flow style*

```
entity EXAMPLE is
        port (    A,B,C :  in BIT;
                  S :      out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
process ( A, B, C )
begin
        S<=       (A and B) or (not C);
end process;
end ARCHI;
```

Figure 37: *Gate netlist description using behavioral style*

The VHDL operators described in section 2 : "Data flow VHDL descriptions", are also supported in processes with the same restrictions and the same optimizations. Please refer to section 2.2 and 2.3 for more details. Figure 38 recalls an example using VHDL arithmetic operator. The synthesized netlist is identical. Similarly, for the adder, the 4 types of adders may be instantiated according to the user requirement.

```
library ASYL;
use ASYL.ARITH.all;

entity EXAMPLE is
        port (   A,B,C,D :          in BIT_VECTOR (7 downto 0);
                 S :                out BIT_VECTOR (7 downto 0)
                 );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
process ( A, B, C, D )
begin
        S<=      (A + B) - (C + D);
end process;
end ARCHI;
```

Figure 38: *Example using arithmetic operators*

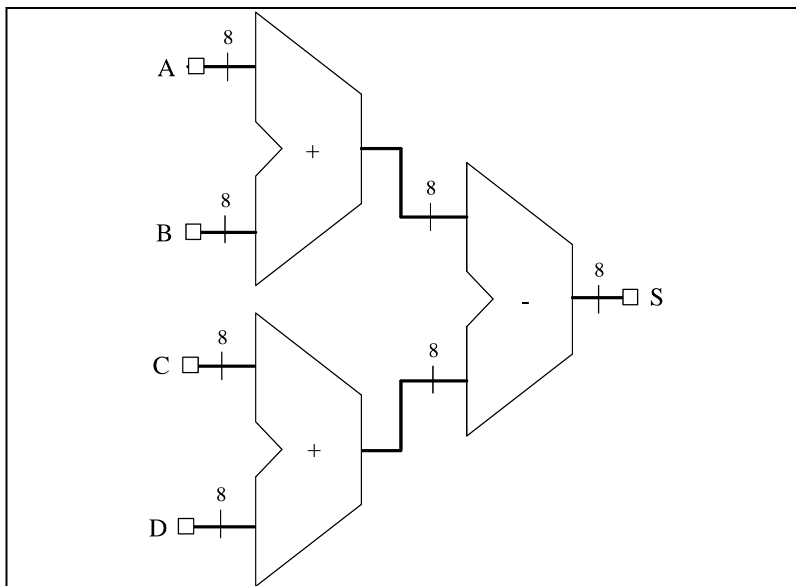

Figure 39: *Synthesized netlist*

3.1.4. Repetitive or bit slice structure

When using processes, repetitive or bit slice structure can also be described using the *"for...loop"* statement. Figure 40 gives an example of a 8 bits adder described with such a statement. *"For...loop"* statement is supported for constant bounds only.

```
entity EXAMPLE is
        port (   A,B :    in BIT_VECTOR (0 to 7);
                 CIN :    in BIT;
                 SUM :    out BIT_VECTOR (0 to 7);
                 COUT :           out BIT );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
        signal C : BIT_VECTOR (0 to 8);
begin
process (A,B,CIN,C)
begin
        C(0) <= CIN;
        for I in 0 to 7 loop
                 SUM(I) <= A(I) xor B(I) xor C(I);
                 C(I+1) <= (A(I) and B(I)) or (A(I) and C(I))
                                  or (B(I) and C(I));
        end loop;
        COUT <= C(8);
end process;
end ARCHI;
```

Figure 40: *8 bit adder described with a "for...loop" statement*

3.2. Sequential circuits descriptions using processes

We shall consider successively two types of descriptions : sequential processes with a sensitivity list and sequential processes without a sensitivity list.

3.2.1. Description styles

3.2.1.1. Sequential process with a sensitivity list

The sensitivity list contains only the clock signal and possibly a reset signal. One and only one "if" statement is accepted in such a process. An asynchronous part may appear before the synchronous part in the first branch of the "if" statement. Signals assigned in the asynchronous part must be assigned to constant values which must be ‘0’, ‘1’ or ‘Z’ or any vector composed of these values. These signals must also be assigned in the synchronous part which corresponds to the second branch of the "if" statement. The clock signal condition is the condition of the last branch of the "if" statement. Figure 41 shows the skeleton of such a process. Complete examples are shown in part 2.2.

```
process ( CLK, RST )
    ...
begin
    if RST = <'0' | '1'> then
    -- an asynchronous part may appear here
       ... -- signals must be assigned to constant values ('0','1' or
           -- 'Z' or any vector composed of these values)
    elsif <CLK'EVENT | not CLK'STABLE>
                and CLK = <'0' | '1'> then
    -- synchronous part
       ... -- signals assigned in the asynchronous part must also
be
           -- assigned here
    end if;
end process;
```

Figure 41: *Sequential process with an asynchronous reset*

3.2.1.2. Sequential process without a sensitivity list

Sequential processes without a sensitivity list contain a "wait" statement. The "wait" statement must be the first statement of the process and is the only "wait" statement in the process. The condition in the "wait" statement must be a condition on the clock signal. Several "wait" statements in the same process are not accepted. Asynchronous part can not be specified within processes without a sensitivity list. Figure 42 shows the skeleton of such a process. The clock condition can be a falling or a rising edge as shown in figure 42.

```
process  ...
begin
     wait until <CLK'EVENT | not CLK'STABLE>
                    and CLK = <'0'|'1'>;
     ...       -- a synchronous reset may be specified here.
end process;
```

Figure 42: *Sequential process without a sensitivity list*

3.2.2. Examples : register and counter descriptions

The example of figure 43 gives the description an 8 bit register using a process with a sensitivity list and in figure 44 the same example is described using a process without a sensitivity list containing a "wait" statement.

```
entity EXAMPLE is
     port (   DI :     in BIT_VECTOR (7 downto 0);
              CLK :    in BIT;
              DO :     out BIT_VECTOR (7 downto 0)    );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
process ( CLK )
begin
     if CLK'EVENT and CLK = '1' then
              DO <= DI ;
     end if;
end process;
end ARCHI;
```

Figure 43: *8 bit register description using a process with a sensitivity list*

```
entity EXAMPLE is
        port (   DI :      in BIT_VECTOR (7 downto 0);
                 CLK :     in BIT;
                 DO :      out BIT_VECTOR (7 downto 0)     );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
process begin
        wait until CLK'EVENT and CLK = '1';
        DO <= DI ;
end process;
end ARCHI;
```

Figure 44: *8 bit register description using a process without a sensitivity list*

The example of figure 45 gives the description of an 8 bit register with a clock signal and an asynchronous reset signal.

```
entity EXAMPLE is
        port (   DI :      in BIT_VECTOR (7 downto 0);
                 CLK :     in BIT;
                 RST :     in BIT;
                 DO :      out BIT_VECTOR (7 downto 0) );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
process ( CLK, RST )
begin
        if RST = '1' then
                 DO <= "00000000";
        elsif CLK'EVENT and CLK = '1' then
                 DO <= DI ;
        end if;  end process;
end ARCHI;
```

Figure 45: *8 bit register description using a process with a sensitivity list*

Figure 46 describes an 8 bit counter.

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity EXAMPLE is
       port (    CLK :    in BIT;
                 RST :    in BIT;
                 DO :     out BIT_VECTOR (7 downto 0)
       );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
process ( CLK, RST )
       variable COUNT : BIT_VECTOR (7 downto 0);
begin
       if RST = '1' then
                 COUNT := "00000000";
       elsif CLK'EVENT and CLK = '1' then
                 COUNT := COUNT + "00000001";
       end if;
       DO <= COUNT;
end process;
end ARCHI;
```

Figure 46: *8 bit counter description using a process with a sensitivity list*

3.2.3. RAM inference

Often, designers need to include RAMs in their designs. PLS offers the possibility to infer RAMs. In fact, when a RAM is recognized and inferred, a black box is generated in the final netlist. The ports of the black box correspond to the ports of the RAM. The user has to replace the black box by the corresponding RAM and the ports have to be connected correctly.

To infer a RAM, some rules have to be respected:

- a bidimentional signal or variable has to be used to represent the RAM.

- to write in the RAM, an indexed signal or an indexed variable has to be assigned by a simple signal or a simple variable. Concatenation and sliced signals are not allowed. The first index only of the array must be specified. This corresponds to a line of the RAM.

- to read the RAM, an indexed signal or an indexed variable has to be used in the right part of assignment. The first index only of the array must be specified.

- the index used as the write or the read address must be a signal or a variable of a range integer type.

- a command signal has to be used in the write mode.

The generated RAM has the following interface:

- a read address port,
- a write address port,
- a data in port,
- a data out port,
- a read/write command port,
- a clock optional port for synchronous RAM.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity REG is
port ( DIN : in       STD_LOGIC_VECTOR(7 downto 0);
       DOUT : out     STD_LOGIC_VECTOR (7 downto 0);
       RADR : in      STD_LOGIC_VECTOR(3 downto 0);
       WADR : in      STD_LOGIC_VECTOR(3 downto 0);
       CSB : in       STD_LOGIC;
       WRB : in       STD_LOGIC );
end REG;

architecture ARCH of REG is
    subtype ramword is STD_LOGIC_VECTOR(7 downto
0);
    type rammemory is array (15 downto 0) of ramword;
    signal ram : rammemory;
begin
read_process : process(ram, RADR)
variable raddr : integer range 0 to 15;

begin
    raddr := to_integer(RADR);
    DOUT <= ram(raddr);
end process read_process;

write_process : process(DIN, WRB, CBS, WADR)
variable waddr : integer range 0 to 15;

begin
    waddr := to_integer(WADR);
    if (CSB = '1' and WRB = '1') then
            ram(waddr) <= DIN;
    end if;
end process read_process;
end ARCH;
```

Figure 47: *RAM specification*

Figure 47 illustrates a RAM specification. The synthesized netlist is show in figure 48.

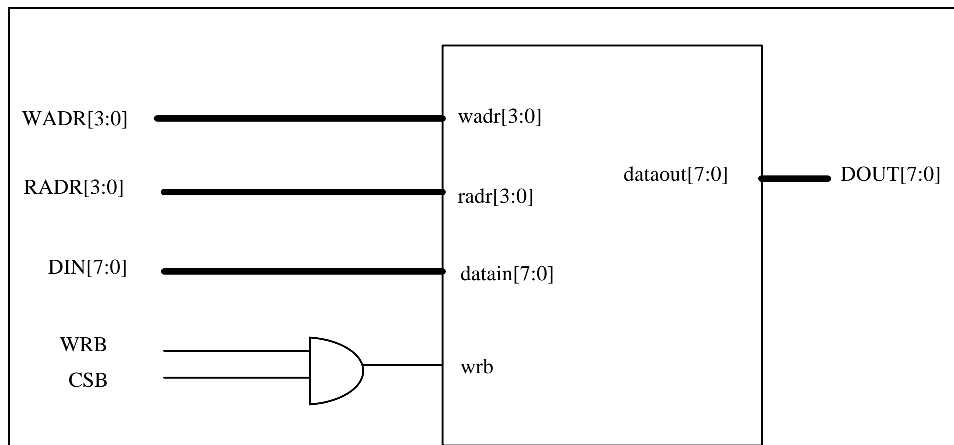

Figure 48 : Synthesized netlist

RAM can also be specified by a structural instantiation. An example using XBLOX is given in figure 49.

```
library asyl;
use asyl.xb_bv.all;

entity xbs_SRAM is
port ( Datain  : in     Bit_VECTOR(4 downto 0);
       WR      : in     Bit;
       add     : in     Bit_VECTOR(2 downto 0);
       Dataout: out     Bit_VECTOR(3 downto 0);
       AddErr           : in      Bit);
end xbs_SRAM;

architecture ARCH of xbs_SRAM is
begin

i_SRAM : SRAM
generic map (    N              => 5,
                 M              => 4,
                 DEPTH          => 16,
                 ENCODING => UBIN,
                 BOUND          => "",
                 TNM            => "")
port map(        D_in           => Datain,
                 WR_EN          => WR,
                 ADDR           => add,
                 D_OUT          => Dataout,
                 ADDR_ERR               => AddErr);
end ARCH;
```

Figure 49: Structural instantiation of *RAM with XBLOX*

For more details refer to "XBLOX macro block call" in the "Macro Block Handling: Macro+" part.

3.2.4. Multiple *wait* statements descriptions

Sequential circuits can be described with multiple *wait* statements in a process. In PLS, some rules must be respected to use multiple *wait* statements. These rules are:

- the process must only contain one *loop* statement,
- the first statement in the *loop* must be a *wait* statement,
- after each *wait* statement, a *next* or *exit* statement must be defined,
- the condition in the wait statements must be the same for each *wait* statement,
- this condition must use only one signal: the clock signal,
- this condition must have the following form:
  "**wait** [**on** <clock_signal>] **until** [(<clock_signal>**'EVENT** |
  not <clock_signal>**'STABLE**) **and**] <clock_signal> = <'0' | '1'>;"

An example using multiple *wait* statements is given in figure 50. This example describes a sequential circuit performing four different operations in sequence. The design cycle is delimited by two following rising edges of the clock signal. A synchronous reset is defined allowing to restart the sequence of operations at the beginning. The sequence of operations consists in putting each of the four inputs: DATA1, DATA2, DATA3 and DATA4 on the output: RESULT.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity EXAMPLE is
port ( DATA1, DATA2, DATA3, DATA4 : in
                          STD_LOGIC_VECTOR (3 downto 0);
      RESULT : out STD_LOGIC_VECTOR (3 downto 0);
      CLK : in STD_LOGIC;
      RST : in STD_LOGIC := '0' );
end EXAMPLE;

architecture ARCH of EXAMPLE is
begin
process begin
      SEQ_LOOP : loop
      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA1;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA2;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA3;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA4;

      end loop;
end process;
end ARCH;
```

Figure 50: *Sequential circuit using multiple wait statements*

3.3. Hierarchy handling through functions and procedures

The declaration of a function or a procedure aims at handling blocks used several times in a design. They can be declared in the declarative part of an entity, an architecture or in packages. The heading part contains the parameters: input parameters for functions and input and output parameters for procedures. These parameters can be unconstrained; it means that they are not constrained to a given bound. The content is similar to the combinational process content. Resolution functions are not supported except the one defined in the IEEE std_logic_1164 package. Recursive function and procedure calls are not supported.

Example of figure 51 shows a function declared within a package. The "ADD" function declared here is an one bit adder. This function is called 4 times with the right parameters in the architecture to create a 4 bit adder. The same example described with a procedure is shown in figure 52.

```
package PKG is
function ADD
        (A,B, CIN : BIT )
return BIT_VECTOR;
end PKG;

package body PKG is
function ADD
        (A,B, CIN : BIT )
return BIT_VECTOR is
variable S, COUT : BIT;
variable RESULT : BIT_VECTOR (1 downto 0);
begin
        S := A xor B xor CIN;
        COUT := (A and B) or (A and CIN) or (B and CIN);
        RESULT := COUT & S;
        return RESULT;          end ADD;
end PKG;

use work.PKG.all;

entity EXAMPLE is
        port (  A,B :   in BIT_VECTOR (3 downto 0);
                CIN :   in BIT;
                S :     out BIT_VECTOR (3 downto 0);
                COUT:   out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
        signal S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
begin
        S0 <= ADD ( A(0), B(0), CIN );
        S1 <= ADD ( A(1), B(1), S0(1) );
        S2 <= ADD ( A(2), B(2), S1(1) );
        S3 <= ADD ( A(3), B(3), S2(1) );
        S <= S3(0) & S2(0) & S1(0) & S0(0);
        COUT <= S3(1);
end ARCHI;
```

Figure 51: *Function declaration and function call*

```
package PKG is
procedure ADD
        (A,B, CIN :         in BIT;
         C :                out BIT_VECTOR (1 downto 0) );
end PKG;

package body PKG is
procedure ADD
        (A,B, CIN :         in BIT;
         C :                out BIT_VECTOR (1 downto 0) ) is
variable S, COUT : BIT;
begin
        S := A xor B xor CIN;
        COUT := (A and B) or (A and CIN) or (B and CIN);
        C := COUT & S;          end ADD;
end PKG;

use work.PKG.all;

entity EXAMPLE is
        port (  A,B :       in BIT_VECTOR (3 downto 0);
                CIN :       in BIT;
                S :         out BIT_VECTOR (3 downto 0);
                COUT : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
        process (A,B,CIN)
        variable S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
        begin
                ADD ( A(0), B(0), CIN, S0 );
                ADD ( A(1), B(1), S0(1), S1 );
                ADD ( A(2), B(2), S1(1), S2 );
                ADD ( A(3), B(3), S2(1), S3 );
                S <= S3(0) & S2(0) & S1(0) & S0(0);
                COUT <= S3(1);
        end process;    end ARCHI;
```

Figure 52: *Procedure declaration and procedure call*

4. General examples using all the VHDL styles

The following examples have been described to illustrate the structural, data flow and behavioral VHDL styles explained in this section.

4.1. Example 1: timer/counter (prepbenchmark 2)

Example 1 is an 8-bit timer/counter. It includes a loadable comparator and a multiplexor which allows the binary up-counter to be preloaded from either a latched value or a value available from a data bus. The block diagram is shown in figure 53.

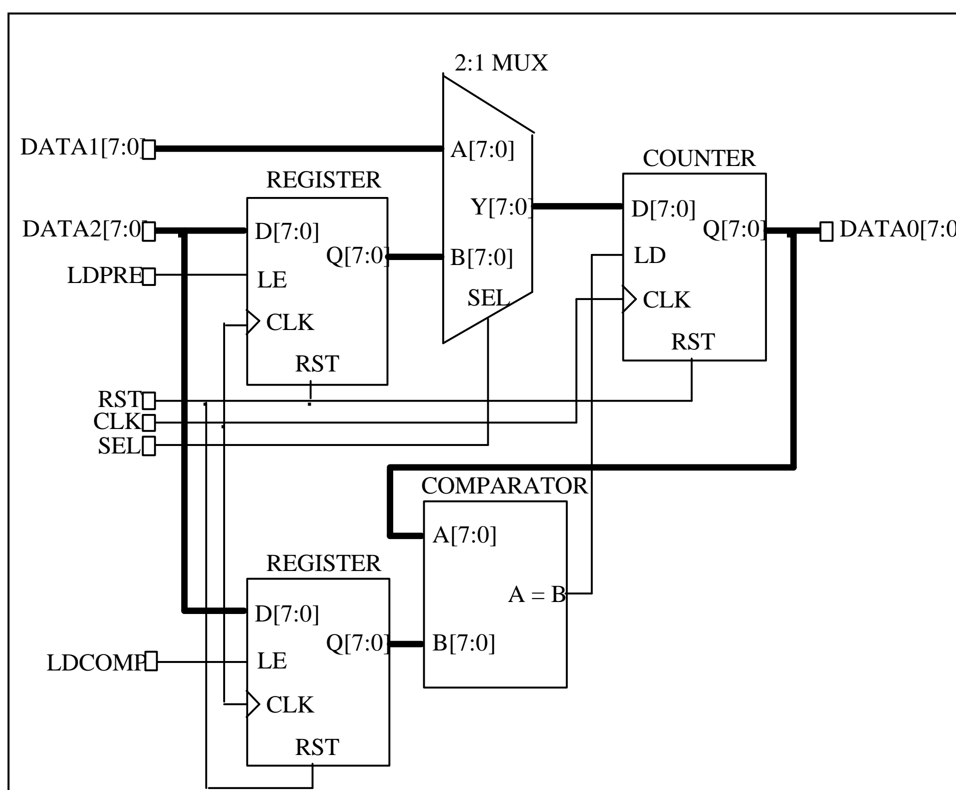

Figure 53: *Example 1: an 8-bit timer/counter*

Figure 60 gives the VHDL description of this example. It is a mixture of structural and data flow styles.

In the top level entity named “PREP2” the register, counter and comparator blocks are declared as components and their descriptions are given at the beginning of the vhdl file. The multiplexor is described in a data flow style using the *“with ... select”* statement. Figure 54 gives the top level VHDL description of the 8-bit timer/counter.

```
entity PREP2 is
   port (CLK,RST,SEL : in BIT;
         LDCOMP,LDPRE : in BIT;
         DATA1,DATA2: in BIT_VECTOR (7 downto 0);
         DATA0 : out BIT_VECTOR (7 downto 0));
end PREP2;

architecture TOP_LEVEL of PREP2 is
   component PREP2_REG
      port (  CLK, RST, LE: in BIT;
              D: in BIT_VECTOR (7 downto 0);
              Q: out BIT_VECTOR (7 downto 0) );
   end component;
   for all: PREP2_REG use entity
                        work.PREP2_REG(DATAFL);
   component PREP2_COUNT
      port (  CLK, RST, LD: in BIT;
              D: in BIT_VECTOR (7 downto 0);
              Q: out BIT_VECTOR (7 downto 0) );
   end component;
   for all: PREP2_COUNT use entity
                        work.PREP2_COUNT(DATAFL);
   component PREP2_COMP
      port (  A, B: in BIT_VECTOR (7 downto 0);
              EQ: out BIT);
   end component;
   for all: PREP2_COMP use entity
                        work.PREP2_COMP(DATAFL);
   signal QPRE, QCOMP, QX, YX:
                        BIT_VECTOR (7 downto 0);
   signal LD: BIT;
begin
   ONE:   PREP2_REG
         port map (CLK, RST, LDPRE, DATA2, QPRE);
   TWO:  PREP2_REG
         port map (CLK, RST, LDCOMP, DATA2, QCOMP);
   THREE:  PREP2_COUNT
         port map (CLK, RST, LD, YX, QX);
   FOUR: PREP2_COMP
         port map (QX, QCOMP, LD);
   with SEL select
      YX <=    DATA1 when '0',
               QPRE when others;
   DATA0 <= QX;
end TOP_LEVEL;
```

Figure 54: *The VHDL top level description of the 8-bit timer/counter*

The entity “PREP2_REG” describes the register used in this example. The block diagram is shown in figure 55.

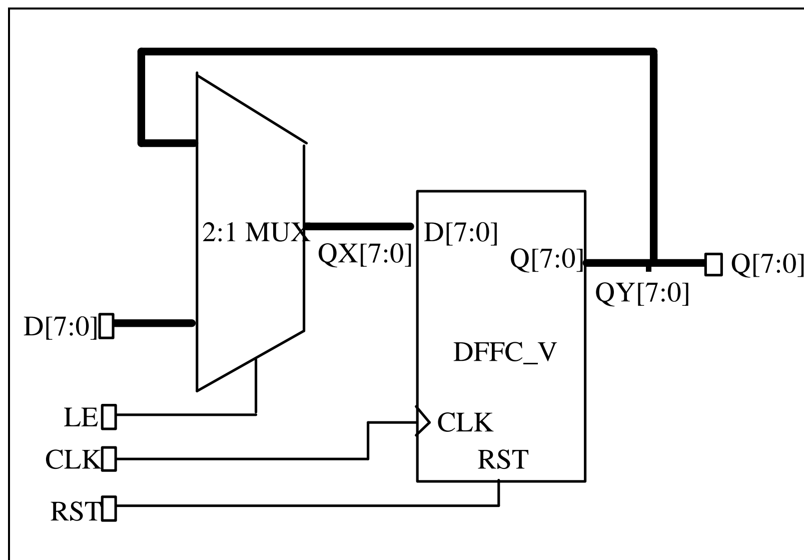

Figure 55: *The register block diagram*

This register has a clock input : CLK, an asynchronous reset input : RST and an enable input : LE which allows the transfer of the input data D[7:0] to the output data Q[7:0] when a clock rising edge occurs. The architecture “DATAFL” associated with the entity “PREP2_REG” uses the data flow style. The register is described as a simple register without enable command. Its input data is connected to the output data of a multiplexor. The command port of the multiplexor is connected to the enable port LE. The input data of the multiplexor is connected to the input data D[7:0] and output data Q[7:0] of the register. In the VHDL architecture, two internal signals : QX and QY are declared. QX is used to connect the output data of the multiplexor to the input data of the simple register without enable. QY is used to connect the output data of the register to an input of the multiplexor. This cannot be done directly using the output port Q[7:0] because an output port cannot be read in VHDL, so in internal signal has to be used. The multiplexor is described using the *“with ... select”* statement and the simple register without enable is described using the procedure call corresponding to such a register : DFF_V written in the package “ASYL_1164” which have been compiled in the ASYL library. Figure 56 gives the VHDL description of the register.

```
library ASYL;
use ASYL.ASYL_1164.all;

entity PREP2_REG is
    port (   CLK, RST, LE: in BIT;
             D: in BIT_VECTOR (7 downto 0);
             Q: out BIT_VECTOR (7 downto 0) );
end PREP2_REG;

architecture DATAFL of PREP2_REG is
    signal QX, QY: BIT_VECTOR (7 downto 0);
begin
    with LE select
        QX <=    QY when '0',
                 D when others;
    DFFC_V (QX, RST, CLK, QY);
    Q <= QY;
end DATAFL;
```

Figure 56: *The VHDL register description*

The entity “PREP2_COUNT” describes the counter. Figure 57 gives the block diagram.

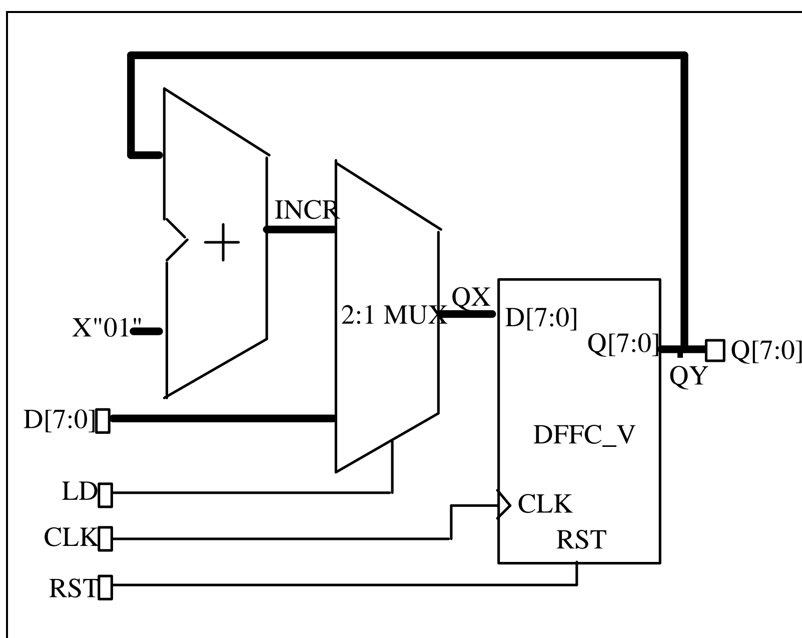

Figure 57: *The counter block diagram*

This counter has a clock input : CLK, an asynchronous reset input : RST and an enable input : LD which allows the transfer of the input data D[7:0] to the output data Q[7:0] when a clock rising edge occurs. If LD is low Q[7:0] is loaded with (Q[7:0] + 1). The architecture “DATAFL” associated with the entity “PREP2_COUNT” uses the data flow style similar to the register description. The incrementation is described using the VHDL operator “+” overloaded for the bit-vectors written in the package “ARITH” which have been compiled in the ASYL library. X”01” is the notation used to express 1 in hexadecimal mode on 8 bits. Figure 58 gives the VHDL representation.

```
library ASYL;
use ASYL.ASYL_1164.all;
use ASYL.ARITH.all;

entity PREP2_COUNT is
    port (  CLK, RST, LD: in BIT;
            D: in BIT_VECTOR (7 downto 0);
            Q: out BIT_VECTOR (7 downto 0) );
end PREP2_COUNT;

architecture DATAFL of PREP2_COUNT is
    signal QX, QY, INCR: BIT_VECTOR (7 downto 0);
begin
    with LD select
      QX <=   INCR when '1',
              D when others;
    INCR <= QY + X"01";
    DFFC_V (QX, RST, CLK, QY);
    Q <= QY;
end DATAFL;
```

Figure 58: *The VHDL counter description*

The entity "PREP2_COMP" describes the comparator used in this example. This comparator has two inputs : A[7:0] and B[7:0] and one output : EQ which is equal to one if the inputs are equal. It is described in data flow style using the *"when ... else"* statement using Figure 59 gives the VHDL description.

```
entity PREP2_COMP is
    port (  A, B: in BIT_VECTOR (7 downto 0);
            EQ: out BIT);
end PREP2_COMP;

architecture DATAFL of PREP2_COMP is
begin
    EQ <= '1' when (A = B) else
          '0';
end DATAFL;
```

Figure 59: *The VHDL comparator description*

The complete description is given below in figure 60.

```
library ASYL;
use ASYL.ASYL_1164.all;
entity PREP2_REG is
    port (   CLK, RST, LE: in BIT;
             D: in BIT_VECTOR (7 downto 0);
             Q: out BIT_VECTOR (7 downto 0) );
end PREP2_REG;

architecture DATAFL of PREP2_REG is
    signal QX, QY: BIT_VECTOR (7 downto 0);
begin
    with LE select
       QX <=    QY when '0',
                D when others;
    DFFC_V (QX, RST, CLK, QY);
    Q <= QY;
end DATAFL;

library ASYL;
use ASYL.ASYL_1164.all;
use ASYL.ARITH.all;
entity PREP2_COUNT is
    port (   CLK, RST, LD: in BIT;
             D: in BIT_VECTOR (7 downto 0);
             Q: out BIT_VECTOR (7 downto 0) );
end PREP2_COUNT;

architecture DATAFL of PREP2_COUNT is
    signal QX, QY, INCR: BIT_VECTOR (7 downto 0);
begin
    with LD select
       QX <=    INCR when '1',
                D when others;
    INCR <= QY + X"01";
    DFFC_V (QX, RST, CLK, QY);
    Q <= QY;
end DATAFL;
entity PREP2_COMP is
    port (   A, B: in BIT_VECTOR (7 downto 0);
             EQ: out BIT);
end PREP2_COMP;
architecture DATAFL of PREP2_COMP is
begin
    EQ <=  '1' when (A = B) else '0';
end DATAFL;

library ASYL;
use ASYL.ASYL_1164.all;
entity PREP2 is
    port (CLK,RST,SEL: in BIT;
          LDCOMP,LDPRE : in BIT;
          DATA1,DATA2: in BIT_VECTOR (7 downto 0);
          DATA0 : out BIT_VECTOR (7 downto 0));
end PREP2;
```

```
architecture TOP_LEVEL of PREP2 is
    component PREP2_REG
        port (  CLK, RST, LE: in BIT;
                D: in BIT_VECTOR (7 downto 0);
                Q: out BIT_VECTOR (7 downto 0) );
    end component;
    for all: PREP2_REG use entity work.PREP2_REG(DATAFL);
    component PREP2_COUNT
        port (  CLK, RST, LD: in BIT;
                D: in BIT_VECTOR (7 downto 0);
                Q: out BIT_VECTOR (7 downto 0) );
    end component;
    for all: PREP2_COUNT use entity
                        work.PREP2_COUNT(DATAFL);
    component PREP2_COMP
        port (  A, B: in BIT_VECTOR (7 downto 0);
                EQ: out BIT);
    end component;
    for all: PREP2_COMP use entity
                        work.PREP2_COMP(DATAFL);
    signal QPRE, QCOMP, QX, YX: BIT_VECTOR (7 downto 0);
    signal LD: BIT;
begin
    ONE:   PREP2_REG
          port map (CLK, RST, LDPRE, DATA2, QPRE);
    TWO:   PREP2_REG
          port map (CLK, RST, LDCOMP, DATA2, QCOMP);
    THREE:  PREP2_COUNT
          port map (CLK, RST, LD, YX, QX);
    FOUR: PREP2_COMP
          port map (QX, QCOMP, LD);
    with SEL select
        YX <=   DATA1 when '0',
                QPRE when others;
    DATA0 <= QX;
end TOP_LEVEL;
```

Figure 60: *Example 1: VHDL representation of the 8-bit timer/counter*

4.2. Example 2: memory map (prepbenchmark 9)

Example 2 implements a memory mapped I/O scheme of different sized memory spaces common to microprocessor systems.

Addresses are decoded when the address strobe (AS) is active according to an address space and each space has an output indicating that it is active. Addresses that fall outside the boundary of the decoder active a bus error (BE) signal.

Figure 61 represents the block diagram of this example. Figure 62 gives the outputs value according to the inputs value and figure 63 gives the VHDL description.

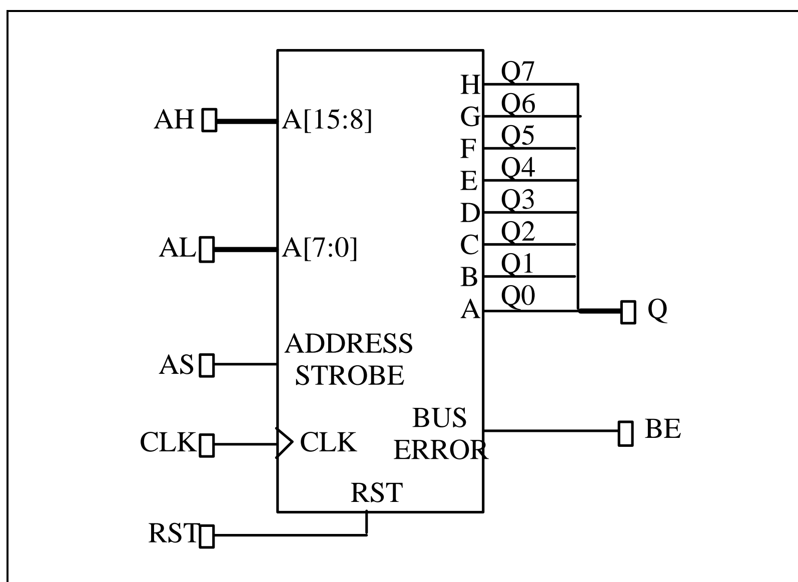

Figure 61: *Example 2: a memory map*

RST	AS	CLK	AH & AL	A	B	C	D	E	F	G	H	BE
1	X	X	X	0	0	0	0	0	0	0	0	0
0	0	¤	X	0	0	0	0	0	0	0	0	0
0	X	*	X	A	B	C	D	E	F	G	H	BE
0	1	¤	FFFF to F000	1	0	0	0	0	0	0	0	0
0	1	¤	EFFF to E800	0	1	0	0	0	0	0	0	0
0	1	¤	E7FF to E400	0	0	1	0	0	0	0	0	0
0	1	¤	E3FF to E300	0	0	0	1	0	0	0	0	0
0	1	¤	E2FF to E2C0	0	0	0	0	1	0	0	0	0
0	1	¤	E2BF to E2B0	0	0	0	0	0	1	0	0	0
0	1	¤	E2AF to E2AC	0	0	0	0	0	0	1	0	0
0	1	¤	E2AB	0	0	0	0	0	0	0	1	0
0	1	¤	E2AA to 0000	0	0	0	0	0	0	0	0	1

(*) Changes take place only on the active edge of the clock

Figure 62: *Example 2: table*

This example is described using the behavioral style. It uses a single process whose sensitivity list contains two signals which are the clock signals : CLK and the asynchronous reset signal : RST. In the asynchronous part the outputs Q[7:0] and BE are assigned to zero. In the synchronous part several *"if ... then"* statement are used to assigned the output according to the values of the AS, AH and AL inputs.

```
entity PREP9 is
    port (CLK, RST, AS : in BIT;
    AL,AH : in BIT_VECTOR (7 downto 0);
    BE : out BIT;
    Q : out BIT_VECTOR (7 downto 0));
end PREP9;

architecture TOP_LEVEL of PREP9 is
begin
    process(CLK, RST)
    begin
        if RST='1' then
            Q <= X"00";
            BE <= '0';
        elsif CLK='1' and CLK'EVENT then
            if AS='1' then
                BE <= '0';
                if AH >= X"F0" and AH <= X"FF" then
                    Q <= X"80";
                elsif AH <= X"EF" and AH >= X"E8" then
                    Q <= X"40";
                elsif AH <= X"E7" and AH >= X"E4" then
                    Q <= X"20";
                elsif AH <= X"E3" and AH >= X"E3" then
                    Q <= X"10";
                elsif AH = X"E2" then
                    if AL <= X"FF" and AL >= X"C0" then
                        Q <= X"08";
                    elsif AL <= X"BF" and AL >= X"B0" then
                        Q <= X"04";
                    elsif AL <= X"AF" and AL >= X"AC" then
                        Q <= X"02";
                    elsif AL = X"AB" then
                        Q <= X"01";
                    else
                        Q <= X"00";
                        BE <= '1';
                    end if;
                else
                    Q <= X"00";
                    BE <= '1';
                end if;
            else
                Q <= X"00";
                BE <= '0';
            end if;
        end if;
    end process;
end TOP_LEVEL;
```

Figure 63: *Example 2: VHDL description of a memory map*

5. Finite State Machine Synthesis

This module synthesizes synchronous Moore finite state machines i.e. finite state machines for which outputs are associated with states (and not with transitions).

There are many ways to describe a finite state machine or a state chart in VHDL. The important point is that the synthesis tool should optimize the corresponding logic in an efficient way both for speed and area. This is done by applying optimized automatic state assignments. As a large set of efficient state assignments is available in PLS, it is important to be able to try them all quickly to reach the best results.

In PLS, a first well defined VHDL template called an explicit FSM template is offered. It consists of identifying the state variables and the outputs by an attribute and to use a single process based on a *"case"* statement. This template is simple and allows the synthesis tool to call efficiently all types of state assignments and all types of flip-flops.

As a second option, PLS accepts process based descriptions (either a single process or two processes) without attributes identifying internal variables and outputs. This is called an implicit FSM template. The synthesis tool recognizes and extracts the FSM structure and is capable again to call state assignments procedure.

Finally PLS offers a third option. If a VHDL description may be identified as "equivalent" to a finite state machine, even not following the previous templates, the tool will be able to recognize and extract such a structure. The tool, in fact, looks if an application: "(State Variable) x (Inputs) -> (Next State Variable)" can be recognized without looking for a specific template.

5.1. Explicit VHDL FSM template with attribute specifications

In this case, the state register and the outputs of the FSM are explicitly declared by special attribute specifications to allow the use of a larger set of flip-flops (T, D, JK).

5.1.1. VHDL template

This template is based on a mono-cycle process. Such a finite state machine description must contain only one sequential process plus possibly data flow statements. The sensitivity list of the process must contain at most two signals which are the clock and the reset signals, and at least one signal which is the clock. The process contains only one assignment which is an *"if"* to specify the synchronous part and the asynchronous one if it exists. The reset may be asynchronous or not. The reset specified in the example of figure 65 is an asynchronous reset. For the clock, a rising or a falling edge can be declared.

5.1.1.1. State register and next state equations

• In the architecture, the attribute specification *"**attribute WIR_TYP of STATE : signal is STATE_REG;**"* indicates explicitly for the synthesis tool that the internal signal STATE is the output of the state register (cf. figure 65).

• To describe the transitions between states, a *"case"* statement must be used to identify which state is considered. In the *"case"*, all state register values have to be enumerated in *"when"* branches. In each branch, the state register may be assigned conditionally in an *"if"* statement, or not. The conditions are Boolean expressions of the input ports.

• If a reset part is declared, the state register must be assigned to the reset state in this part.

5.1.1.2. Latched and non latched outputs

• The output ports of the FSM are declared in the entity. The attribute specification *"**attribute WIR_TYP of Z0, Z1, Z2, Z3: signal is OUT_CONTROL;**"* indicates explicitly for the synthesis tool that the ports Z0, Z1, Z2 and Z3 are output ports of the controller (cf. figure 65).

• For outputs which are not latched, a combinational or a data flow part is described. These non latched outputs must be assigned using data flow conditional statements as showed at the beginning of the VHDL model skeleton of figure 66. Be careful: an output port assigned outside the process cannot be assigned within the process. In the example of figure 65, the output port Z3 is not latched.

• Latched outputs must be assigned in the *"when"* branches of the *"case"* statement in the sequential process. These outputs are stored in flip-flops. In each branch, output ports may be assigned conditionally in an *"if"* statement, or not. In the example of figure 65, the outputs : Z0, Z1 and Z2 are latched outputs.

• If a reset part is declared, the latched outputs may be assigned only to constant value in this part.

5.1.1.3. Latched inputs

• In this version and with this template, the latching of the inputs has to be declared separately in a structural mode.

Figure 64 gives an example of FSM where Z0, Z1 and Z2 are latched outputs and Z3 is a non latched output.

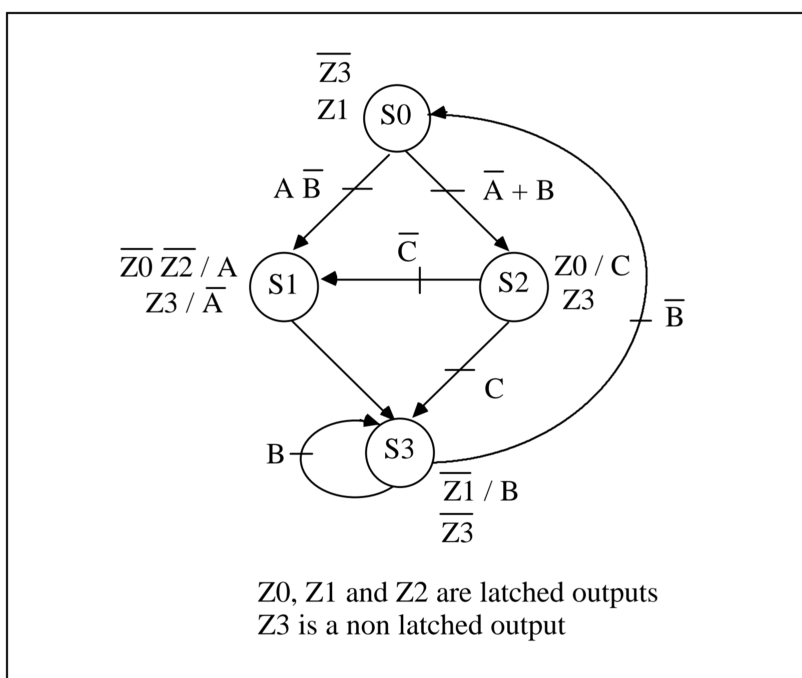

Figure 64: *Graphical representation of a Moore FSM*

Figure 65 gives the VHDL description of the FSM showed in figure 64. An asynchronous reset is described and the state register and the latched outputs are sampled on the rising edge of the clock.

```
library ASYL;
use ASYL.ASYL_RTL.all;

entity FSM is
port (          CLOCK, RESET :          in BIT;
                A, B, C :               in BIT;
                Z0, Z1, Z2, Z3 :        out BIT );
attribute WIR_TYP of Z0, Z1, Z2, Z3: signal is
OUT_CONTROL;
end FSM;

architecture SYNTHESIS of FSM is
signal STATE : INTEGER;
attribute WIR_TYP of STATE : signal is STATE_REG;
begin
Z3 <= '1' when ((STATE = 1 and A = '0') or STATE = 2) else
        '0';
process (CLOCK, RESET)
begin
        if RESET = '1' then STATE <= 0;
        elsif CLOCK'EVENT and CLOCK = '1' then
                case STATE is
                        when 0 =>
                                Z1 <= '1';
                                if (A and (not B)) = '1' then
                                        STATE <= 1;
                                else    STATE <= 2;
                                end if;
                        when 1 =>
                                if A = '1' then Z0 <= '0';
                                                Z2 <= '0';
                                end if;
                                STATE <= 3;
                        when 2 =>
                                if C = '1' then Z0 <= '1';
                                                STATE <= 3;
                                else            STATE <= 1;
                                end if;
                        when 3 =>
                                if B = '1' then Z1 <= '0';
                                end if;
                                if B = '1' then STATE <= 3;
                                else            STATE <= 0;
                                end if;
                        when others =>
                                null;
                end case;
        end if;
end process;
end SYNTHESIS;
```

Figure 65: *VHDL representation of a Moore FSM*

The skeleton for an PLS VHDL description is given below, in figure 66.

```
library ASYL;
use ASYL.ASYL_RTL.all;
-- package ASYL_RTL contains attributes used for synthesis
entity <entity_name> is
<port declaration>
<attribute specification>
end <entity_name>;
architecture <architecture_name> of <entity_name> is
<signal declaration>
<attribute specification>
begin
-- data flow part
<port_name> <= <expression>;
<port_name> <= <expression_1> when <condition_1> else
                    ....
                        <expression_N-1>  when  <condition_N-1>
else
                    <expression_N>;
-- <port_name> is the name of an output signal of the
controller.
-- <expression> can be a constant value or a Boolean
expression -- on  input signals (input ports of the controller).
-- <condition> can be a Boolean expression on input signals
-- and/or state register signal.
process (<clock_name>, <reset_name>)   -- sequential process
begin
      if <reset_name> = <‘0’ or ‘1’> then ...
      -- asynchronous reset, expressions are supported
      elsif <clock_name>'EVENT and
                        <clock_name> = <‘0’ or '1'> then
              <port_name> <= <expression>; -- default value
              case <state_name> is
              when <value_1> =>
                        <port_name> <= <expression>;
                        -- unconditional output assignment
                        ...
                        <state_name> <= <value_2>;
                        -- unconditional state assignment
              when <value_2> =>
                        -- conditional output assignment, Boolean
                        --    expressions    on    predicates    are
supported
                        if <predicate_name> = <‘0’ or '1'> then
                        <port_name> <= <expression>;
                        end if;   ...
                        -- conditional state assignment
                        if <predicate_name> = <‘0’ or '1'> then
                                <state_name> <= <value_3>;
                        else    <state_name> <= <value_4>;
                        end if;
```

```
when <value_3> =>   ...
                    -- conditional output and state assignment
                    if <predicate_name> = <'0' or '1'> then
                            <port_name> <= <expression>;
                            <state_name> <= <value_3>;
                    else    <state_name> <= <value_4>;
                    end if;
            when <value_4> =>
                    ...     ...
end case;  end if;  end process;   end <architecture_name>;
```

Figure 66: *VHDL model skeleton*

5.1.2. State assignments

When using an explicit template as described above, automatic state assignment can be selected and six automatic state assignments can be invoked, namely the optimized compact, the one-hot, the Gray, the Johnson, the sequential and the random encoding. In addition to the six state assignments, the user can impose the encoding by using the code file option.

5.1.2.1. State assignment optimizations

The user may call automatically the optimized compact, the one-hot, the Gray, the Johnson, the sequential or the random encoding.

For the one-hot encoding the number of flip-flops is equal to the number of states. The one hot encoding is suitable and efficient for speed optimization or for targets having a large number of flip-flops.

The optimized compact encoding uses the minimal number of flip-flops to store the state variables. It is very sophisticated and assigns for instance nodes having the same predecessor on a face of a hyper cube. That means that their codes differ only by a minimal number of bits. This state assignment leads to a gain in area of next state and output logic of about 30% compared with average results using randomly generated codes.

The Gray encoding is suitable for controller exhibiting long paths without branching. It consists of identifying long paths and applying successive Gray codes on the nodes of the path. It is well known that such an encoding minimizes the hazards and glitches as only one bit changes when going from a state to the next one on a path. It is also a very efficient code when using T or JK flip-flops.

The Johnson encoding is suitable also for controller exhibiting long paths without branching. It consists of identifying long paths and applying successive Johnson codes on the nodes of the path. The same remark about hazard avoidance is valid for this code.

The sequential encoding consists of identifying long paths and applying successive radix two codes on the nodes of the path. It has been shown that a radix two code minimizes efficiently next state equation complexity on path. Thus area is minimized.

For random encoding, the system applies a random encoding.

5.1.2.2. User controlled state assignment

With this template, the user codes have to be written in a special code file which is read during the synthesis. For the user controlled state assignment, the designer has to choose the user encoding and to specify his code file.

5.1.3. Choice of the flip-flops

Using this template, the user can also choose the type of the flip-flops (D,T,JK).

Tables 1 and 2 give the number of product terms after synthesis according to the six encoding available in PLS: optimized compact, sequential, random, one-hot, Gray and Johnson, and the three types of flip-flops: D, T and JK. These examples have been synthesized on MACHs. Table 1 gives the results for the “bbara” example and table 2 for the “bbsse” example. For the first example, the minimal number of product terms (23) is obtained for the Gray encoding using T flip-flops. For the second one, the minimal number of product terms (48) is obtained for the optimized compact encoding using D flip-flops.

Table 1: *Number of Product Terms after synthesis of the “bbara” example*

Encoding	Nb. of PTs using D Flip-Flops	Nb. of PTs using T Flip-Flops	Nb. of PTs using JK Flip-Flops
OPT	27	28	28
SEQ	33	28	28
RAN	43	30	27
ONE	56	108	108
GRAY	28	23	23
JOHN	34	33	34

Table 2: *Number of Product Terms after synthesis of the “bbsse” example*

Encoding	Nb. of PTs using D Flip-Flops	Nb. of PTs using T Flip-Flops	Nb. of PTs using JK Flip-Flops
OPT	48	58	58
SEQ	52	57	57
RAN	76	71	61
ONE	86	299	299
GRAY	50	50	51
JOHN	64	65	66

5.2. Implicit VHDL FSM template

5.2.1. VHDL template

A finite state machine can be “hidden” in a VHDL description. Such a finite state machine description contains at least one sequential process declaration or at most two processes: a sequential one and a combinational one. The sensitivity list of the sequential process must contain at least one signal which is the clock signal and at most two signals which are the clock and the reset signals. For the clock a rising or a falling edge can be declared similarly to the first template. The reset is not mandatory. If it is declared, it has to be an asynchronous signal. In the sequential process, an “*if*” statement specifies an asynchronous reset if it exists and a synchronous part assigning the state variable.

5.2.1.1. State register and next state equations

• The state variables have to be assigned within the sequential process.

• The type of the state register can be integer, integer range, bit_vector, std_logic_vector or a user defined enumerated type.

• The next state equations must be described in the sequential process using a *"case"* statement or outside the process like the non latched outputs.

5.2.1.2. Latched and non latched outputs

• The non latched outputs must be described in a combinational process or using data flow conditional statements ("<output> <= <value> *when* <condition> *else* ..." or "*with* <condition> *select* <output> <= <value> *when* <condition_value>, ...").

• The latched outputs must be assigned within the sequential process like the state register.

• Note that presently the vectored outputs are not recognized as outputs of the FSM.

5.2.1.3. Latched inputs

• The latched inputs must be described using internal signals representing the output of the flip-flops. These internal signals have then to be assigned in a sequential process with the latched inputs. This sequential process must be the one where the state register is assigned as showed in figure 70. Figure 70 gives the VHDL description of the FSM described in figure 67 where the two inputs A and B are latched.

The figure 67 represents a Moore FSM. This FSM has two outputs: Z0 which is a latched output and Z1 which is a non latched output. This machine will be described twice. The first description uses a simple sequential process (cf. figure 68). The second description uses both a sequential process and a combinational one (cf. figure 69).

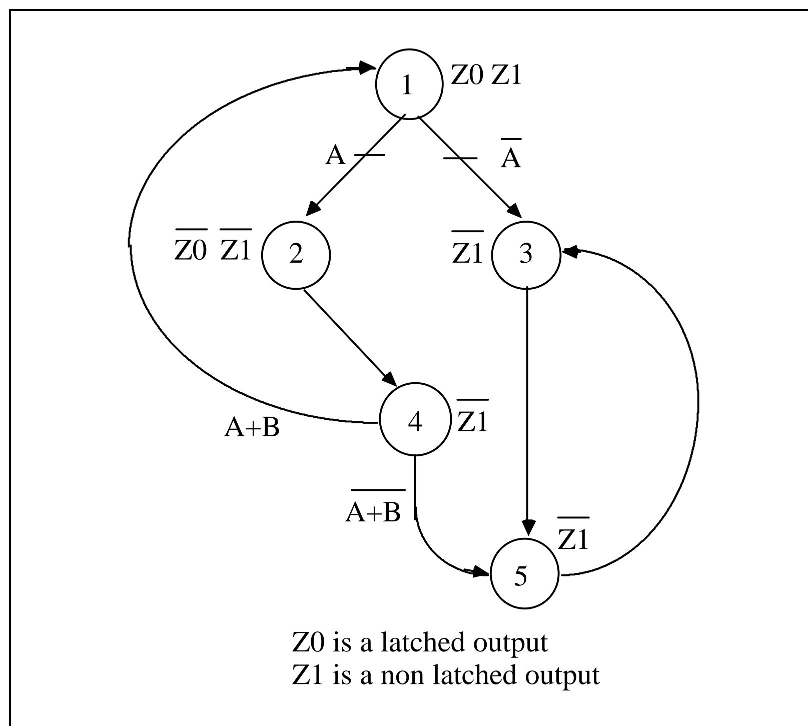

Figure 67: *Graphical representation of a Moore FSM*

The FSM represented in figure 67 is described in figure 68 using a sequential process for the state register, the latched output Z0 and the next state logic, and a conditional assignment for the output Z1.

```
entity FSM is
port (      RESET, CLOCK:      in BIT;
            A, B:              in BIT;
            Z0, Z1:            out BIT);
end FSM;

architecture SYNTHESIS of FSM is
      signal VALUE: INTEGER;
begin
P1: process (CLOCK, RESET)
begin
      if (RESET = '1') then
            VALUE <= 1;
      elsif (CLOCK'EVENT and CLOCK = '1') then
            case VALUE is
                  when 1 =>
                        Z0 <= '1';
                        if (A = '1') then  VALUE <= 2;
                        else               VALUE <= 3;
                        end if;
                  when 2 =>
                        Z0 <= '0';
                        VALUE <= 4;
                  when 3 =>
                        VALUE <= 5;
                  when 4 =>
                        if (A = '1' or B = '1') then
                              VALUE <= 1;
                        else  VALUE <= 5;
                        end if;
                  when 5 =>
                        VALUE <= 3;
                  when others =>
                        VALUE <= 1;
            end case;
      end if;
end process;
Z1 <= '1' when (VALUE = 1) else '0'; -- Z1 is not latched
end SYNTHESIS;
```

Figure 68: *VHDL description of a Moore FSM*

Note that in figure 68, the *"case"* statement describing the next state logic and the state register assignment in the sequential process has a *"when others"* branch. In this branch the state register must be assigned to the reset value and this value is used for simplification.

The FSM represented in figure 66 is described in figure 6 using a sequential process for the state register and the latched output Z0, and a combinational process for the output Z1 and the next state logic.

```
entity FSM is
port (          RESET, CLOCK:           in BIT;
                A, B:               in BIT;
                Z0, Z1:             out BIT);
end FSM;

architecture SYNTHESIS of FSM is
        signal VALUE: INTEGER;
        signal NEXTVALUE: INTEGER;
begin
P1: process (CLOCK, RESET)
begin
        if (RESET = '1') then
                VALUE <= 1;
        elsif (CLOCK'EVENT and CLOCK = '1') then
                VALUE <= NEXTVALUE;
                case VALUE is
                        when 1 =>
                                Z0 <= '1';
                        when 2 =>
                                Z0 <= '0';
                        when others =>
                                null;
                end case;
        end if;
end process;
P2: process (A, B, VALUE)
begin
        Z1 <= '0';  -- default value
        case VALUE is
                when 1 =>
                        Z1 <= '1';
                        if (A = '1') then
                                NEXTVALUE <= 2;
                        else    NEXTVALUE <= 3;
                        end if;
                when 2 =>
                        NEXTVALUE <= 4;
                when 3 =>
                        NEXTVALUE <= 5;
                when 4 =>
                        if (A = '1' or B = '1') then
                                NEXTVALUE <= 1;
                        else    NEXTVALUE <= 5;
                        end if;
                when 5 =>
                        NEXTVALUE <= 3;
                when others =>
                        NEXTVALUE <= 1;
        end case;
end process;
end SYNTHESIS;
```

Figure 69: *VHDL description of a Moore FSM*

Figure 70 gives the VHDL description of the FSM described in figure 67. In this description, the inputs A and B are latched.

```
entity FSM is
port (          RESET, CLOCK:        in BIT;
                A, B:            in BIT;
                Z0, Z1:          out BIT);
end FSM;

architecture SYNTHESIS of FSM is
        signal VALUE: INTEGER;
        signal A_FF, B_FF : BIT;
begin
P1: process (CLOCK, RESET)
begin
        if (RESET = '1') then
                VALUE <= 1;
        elsif (CLOCK'EVENT and CLOCK = '1') then
                case VALUE is
                        when 1 =>
                                Z0 <= '1';
                                if (A_FF = '1') then VALUE <= 2;
                                else            VALUE <= 3;
                                end if;
                        when 2 =>
                                Z0 <= '0';
                                VALUE <= 4;
                        when 3 =>
                                VALUE <= 5;
                        when 4 =>
                                if (A_FF = '1' or B_FF = '1') then
                                        VALUE <= 1;
                                else    VALUE <= 5;
                                end if;
                        when 5 =>
                                VALUE <= 3;
                        when others =>
                                VALUE <= 1;
                end case;
                A_FF <= A; -- A_FF is latched
                B_FF <= B; -- B_FF is latched
        end if;
end process;
Z1 <= '1' when (VALUE = 1) else '0'; -- Z1 is not latched
end SYNTHESIS;
```

Figure 70: *VHDL description of a Moore FSM with latched inputs*

5.2.2. State assignments

When using an implicit template as described above, commonly no automatic state assignment method is invoked. The state codes are explicitly given in the description. But in PLS, automatic state assignment can be selected and five automatic state assignments available for the first template can be invoked.

5.2.2.1. State assignment optimizations

The user may select the optimized compact or one-hot or Gray or Johnson or sequential state assignment. If the encoding is not specified then PLS uses the user state assignment.

5.2.2.2. User controlled state assignment

The user can use for the state identification an identifier, an integer or a bit string. This identification defines the code associated with the state.

If an identifier is used, the code is the binary value associated with the order of the identifier in the enumerated type declaration. For example, the binary value associated with the identifier "STATE3" defined in the enumerated type "STATE_TYPE" declared as follow:

"**type** STATE_TYPE **is** (STATE1,STATE2,STATE3,STATE4);"

is "10".

If the state identification is an integer, the code associated with the state is the binary string corresponding to the integer ("0..011" for 3 for example).

Finally, if a bit string value is used as state identification, it will be used as its code.

So, by default, if the encoding is not specified, the PLS encoding is the one written in the VHDL specification. Note that if the user gives one hot encoding string value as identification of the states ("00…100" for the third state), "false" one hot state assignment will be performed. This means that the code "00100" will generate the canonical product term !Y1.!Y2.Y3.!Y4.!Y5 instead of only Y3 to identify this state like it does in the one hot encoding.

5.3. Symbolic FSM identification

This original feature addresses the case where VHDL variables and signals can be identified to play the role of internal and output variables in a classical finite state machine definition. This means that a deterministic application "(State Variable) x (Inputs) -> (Next State Variable)" can be identified. For this purpose, all internal latched variables and signals are scanned. Their assignments are analyzed. If together with "input like variables" they define such an application, this template will be considered as a FSM template; of course the determinism propriety is checked and then all state assignments may be applied to this variable. Thus much looser templates can be identified. An example of such a description is given in figure 71.

```
entity FSM is
port (      CLOCK:          in BIT;
            A, B:       in BIT;
            Z1:         out BIT);
end FSM;
architecture SYNTHESIS of FSM is
      signal VALUE: INTEGER;
      signal NEXTVALUE: INTEGER;
begin
P1: process (CLOCK)
begin
      if (CLOCK'EVENT and CLOCK = '1') then
            VALUE <= NEXTVALUE;
      end if;
end process;
P2: process (A, B, VALUE)
begin
      Z1 <= '0';  -- default value
      if (VALUE = 1) then
            Z1 <= '1';
            if (A = '1') then
                  NEXTVALUE <= 2;
            else  NEXTVALUE <= 3;
            end if;
      elsif (VALUE = 2) then
            NEXTVALUE <= 4;
      elsif (VALUE = 3) then
            NEXTVALUE <= 5;
      elsif (VALUE = 4) then
            if (A = '1' or B = '1') then
                  NEXTVALUE <= 1;
            else  NEXTVALUE <= 5;
            end if;
      elsif (VALUE = 5) then
            NEXTVALUE <= 3;
      else  NEXTVALUE <= 1;
      end if;
end process;
end SYNTHESIS;
```

Figure 71: *VHDL description of a Moore FSM*

Such a description can also be synthesized by PLS using the five automatic state assignments or the user controlled state assignment.

5.4. Templates using multiple wait statements

A template using multiple *wait* statements is supported. For this template, only a synchronous reset can be described and the transition between the states must be unconditional. More details about the limitations on the use of multiple *wait* statements are given in section "3.2.3. Multiple *wait* statements descriptions".

Not all multiple *wait* templates are supported in PLS. A commonly used template not supported in PLS is the multiple *wait* statements description using several *loop* statements. An example is given in figure 72.

```
entity FSM is
port (          RESET, CLOCK:          in BIT;
                A, B:                  in BIT;
                Z0, Z1:                out BIT);
end FSM;

architecture SYNTHESIS of FSM is
begin
process
begin
wait until RESET = '1';
loop L0:
wait until CLOCK'EVENT and CLOCK = '1';
Z0 <= '1';     Z1 <= '1';
if A = '1' then
        wait until CLOCK'EVENT and CLOCK = '1';
        Z0 <= '0';          Z1 <= '0';
        wait until CLOCK'EVENT and CLOCK = '1';
        Z1 <= '0';
        if (A or B) = '1' then
                exit loop L0;
        else
                wait until CLOCK'EVENT and CLOCK = '1';
                Z1 <= '0';
        end if;
end if;
loop L1:
wait until CLOCK'EVENT and CLOCK = '1';
Z1<= '0';
wait until CLOCK'EVENT and CLOCK = '1';
Z1 <= '0';
end loop L1;
end loop L0;
end process;
end SYNTHESIS;
```

Figure 72: *Multiple wait statements description of a Moore FSM using several loop statements not supported in PLS*

5.5. Handling FSMs within your design

Commonly FSMs are embedded in a larger design. So the problem is how to handle them within a design. The user will have several options. He may want to handle them individually, under his control to optimize them specifically, or he wants just to let them embedded in the description. In this last case, he may ask to the synthesis tool to “recognize” or “extract” them.

5.5.1. Pre-processing or separate FSM handling

In this case, the user will write in different files. For each FSM, he creates a file and then he synthesizes each file separately specifying the encoding for each FSM and the type of flip-flops in case of explicit description. During synthesis, a netlist is synthesized for each VHDL file. The user has to assemble them. If the netlists are in EDIF format, the designer can use the textual command which allows the merge of EDIF netlists; if they are in XNF format (Xilinx format) and if the designer uses Xilinx tools, he can use the Xilinx command which allows the merge of XNF netlists; finally if they are in an other format, the user has to merge them manually.

5.5.2. Embedded FSMs

In this case the user does not want to declare separately the FSMs. Commonly, he will use a process style with no explicit declaration or a structural description where each FSM may be described using the implicit or explicit template. It may then be useful that the synthesis tool recognize and extract them. For this the user will ask for this automatic recognition in the VHDL by selecting the option “FSM Extraction”. Then three possibilities are offered:

a) The user does not give any information about encoding. If the optimization criterion is area, the optimized compact state assignment is automatically chosen for all the FSMs. If the optimization criterion is speed, the one-hot state assignment is automatically chosen for all the FSMs.

b) A single common encoding is specified by the designer; it will be identical for all the FSMs and it may be the optimized compact, the one-hot, the Gray, the Johnson or the sequential encoding. The user can also specify a global synthesis criterion and a global power for his design. Note that in this case the flip-flops used for the state register synthesis are the D flip-flops. The T and JK flip-flops are not supported in this case.

c) Specific options can be given for each FSM. For this purpose, a synthesis directive file is specified. This file defines the entity and the architecture names of the FSMs and a dedicated encoding, the choice of the flip-flops, a synthesis criterion and a power for each FSM. The specific options defined in the synthesis directive file overrides the global options. For example, if there are three FSMs embedded in the design, each FSM has to be described in a separated entity named FSM1, FSM2 and FSM3. The architecture corresponding to each FSM is named ARCH. If the designer wants to use the optimized compact encoding for FSM1 with an area optimization criterion, one-hot encoding for FSM2 with a speed optimization criterion and a power of 2, and Gray encoding with T flip-flops for FSM3, the directive file given in figure 73 can be used. Note that choosing T flip-flops for FSM3, FSM3 have to be described with the explicit template. This must be specified in the synthesis directive file with the option "-fl control".

```
directive -ent FSM1 -arch ARCH -c OPT -crit AREA
directive -ent FSM2 -arch ARCH -c ONE -crit SPEED -power 2
directive -ent FSM3 -arch ARCH -c GRAY -ff T -fl control
```

Figure 73: *Example of synthesis directive file*

Note that in this case, if no encoding has been declared for a given FSM, the global encoding value will be used. So, the global options are the default options. For example, if the designer wants to use the optimized compact encoding for FSM1 and FSM2 with an area optimization criterion, and the one-hot encoding for FSM3, he may select the optimized compact encoding for the global encoding and ask for an area global optimization criterion and he may give a directive file. This file must contain the following line: “directive -ent FSM3 -arch ARCH -c one”, giving the specific options for the FSM3 synthesis. For more details on the format of the synthesis directive file, see the grammar in appendix 1.

Note that the user can choose D, T or JK flip-flops for each FSM described with the explicit template. This choice is forbidden for FSM described with the implicit template.

Note that in this case, after synthesis, there is only one resulting netlist for all the design.

6. Communicating Finite State Machines Synthesis

6.1. Introduction

Two finite state machines communicate if the transition predicate of a FSM depends on the state or the output of the other one. Two communicating FSMs may be connected in a concurrent mode or in a hierarchical (master-slave) mode. In the last mode, one of the FSMs stays in the same state while the state of the other one changes.

For communicating FSMs, two composition techniques will be used. The first one considers a global entity connecting several FSMs described by processes with implicit descriptions. The second one connects in a structural mode the different FSMs. Each FSM can then be described using any accepted FSM description style.

6.2. Communicating FSMs

6.2.1. Concurrent communicating FSMs

Figure 73 shows two communicating FSMs: FSM1 (figure 74.a) and FSM2 (figure 74.b). FSM1 has three input signals ("A", "B", "Z0") and two outputs signals ("IsA", "IsB"). FSM2 has one input signal ("sig") and one output signal ("Z0"). These two FSMs communicate by state tests and by an output signal. For example, in figure 74, FSM1 goes from state "S1" to state "S2" if FSM2 is in state "SS3". These two FSMs communicate also by the output signal "Z0" sent by FSM2 to FSM1.

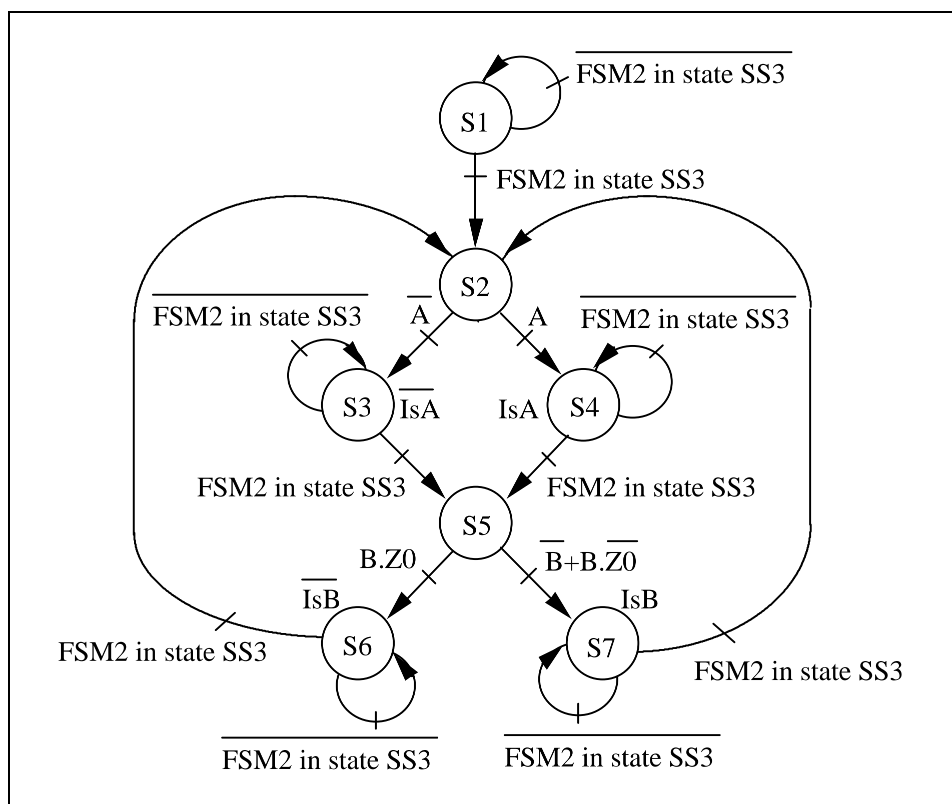

Figure 74.a: *FSMs communicating by states (FSM1)*

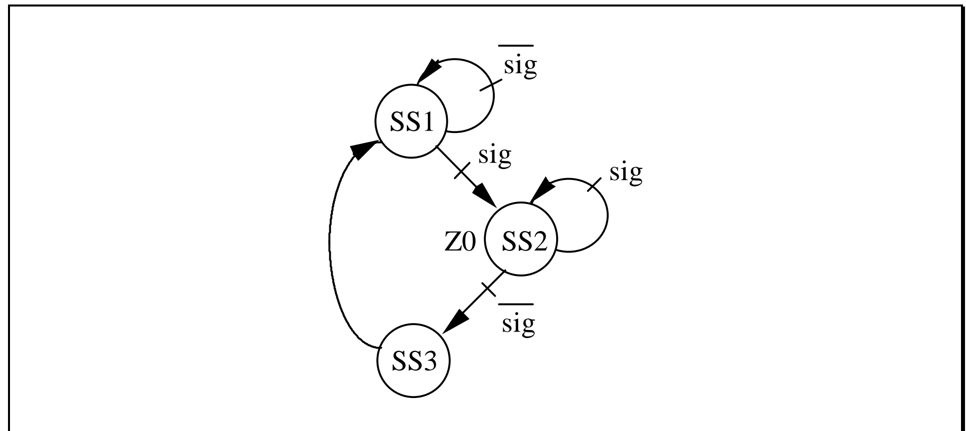

Figure 74.b: *FSMs communicating by states (FSM2)*

Communicating FSMs by state tests can be transformed into communicating FSMs by output signals. So, if a FSM is sensitive to a state of another FSM, a signal identifying this state has to be created. Figure 74 gives the same communicating FSMs than in figure 73, but in figure 74 they communicate by output signals instead of states. In FSM2 the output signal “got_one” identifying the state “SS3” has been created and in FSM1 the transition predicate “FSM2 in state SS3” has been replaced by “got_one”.

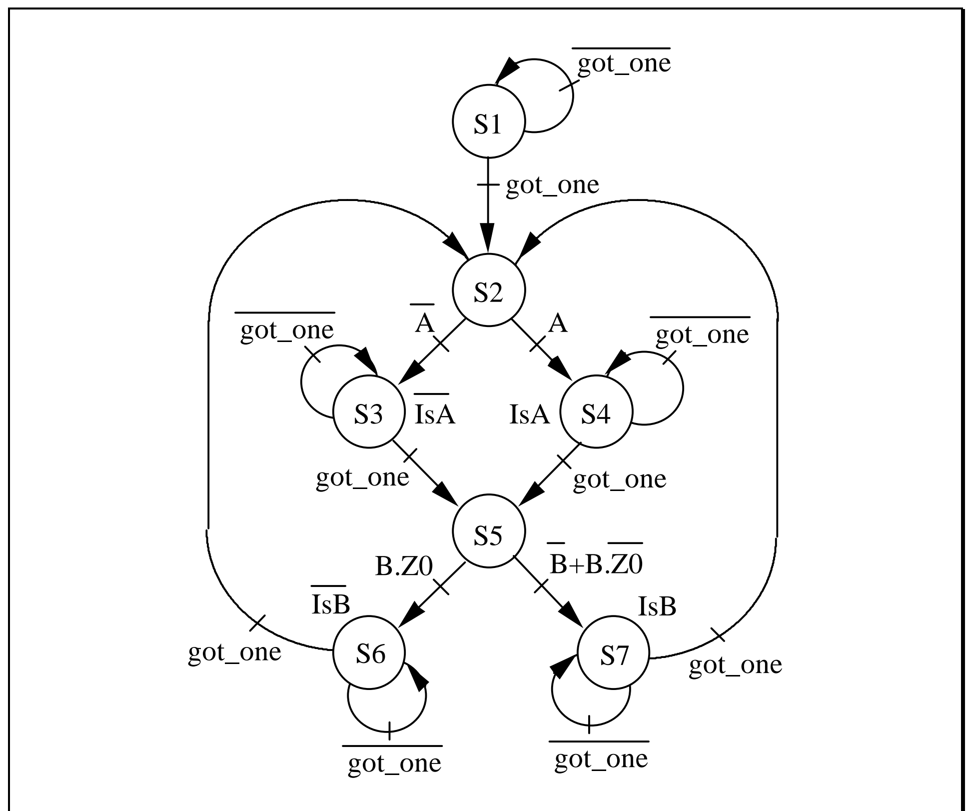

Figure 75.a: *FSMs communicating by output signals (FSM1)*

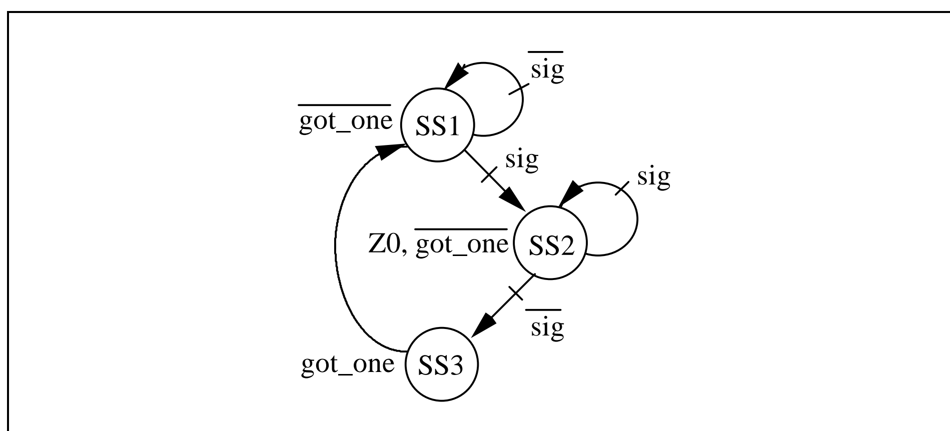

Figure 75.b: *FSMs communicating by output signals (FSM2)*

6.2.2. Hierarchical or master-slave communicating FSMs

Two FSMs communicate in a hierarchical (master-slave) mode if one of the FSMs stays in the same state while the state of the other one changes. As for the concurrent communicating FSMs, the communication is made by output signals or by states.

Let FSM1 and FSM2, two finite state machines communicating in a hierarchical (master-slave) mode. S1 and S2 are respectively a state of FSM1 and FSM2. A state S1 of FSM1 is said to be a “communication state” with respect to FSM2 if either:

- it exists a state S2 of FSM2, origin of at least one transition labeled by a predicate which is a function of S1. In this case, S1 is said to be a “call state” for FSM1, or

- it exists a state S1 of FSM1, origin of at least one arc labeled by a predicate which is a function of S2. In this case, S1 is called a “waiting state” for FSM1.

A self loop on a waiting state is called a “waiting transition”. The states destination of the other transitions (excluding the self loop) issued from a waiting state are referred as “return states”.

Figure 76 shows two communicating hierarchical FSMs. The master FSM in figure 76.a has three input signals (“A”, “B”, “got_one”) and three output signals (“IsA”, “IsB”, “get_sig”). The slave FSM in figure 76.b has two input signals (“sig”, “get_sig”) and one output signal (“got_one”). The two FSMs communicate by output signals: “got_one” is sent by the slave FSM to the master and “get_sig” is sent by the master FSM to the slave. Note that the states with double circles are the “wait” states. For the master FSM, the “wait” states are (“S1”, “S3”, “S4”, “S6”, “S7”) and the slave FSM has one “wait” state (“SS4”).

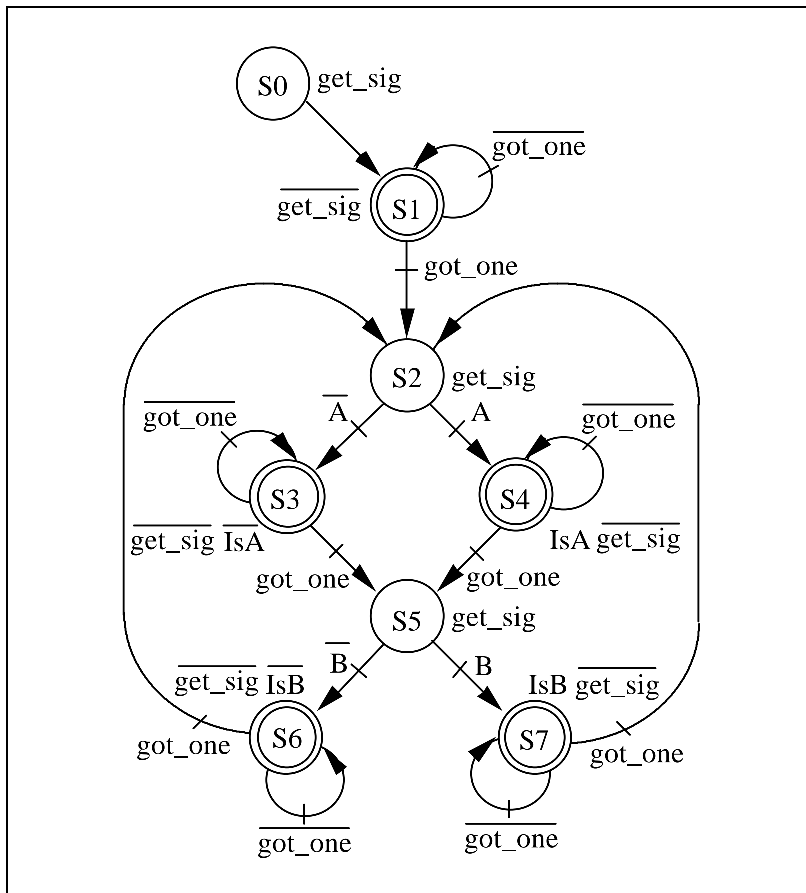

Figure 76.a: *Master FSM*

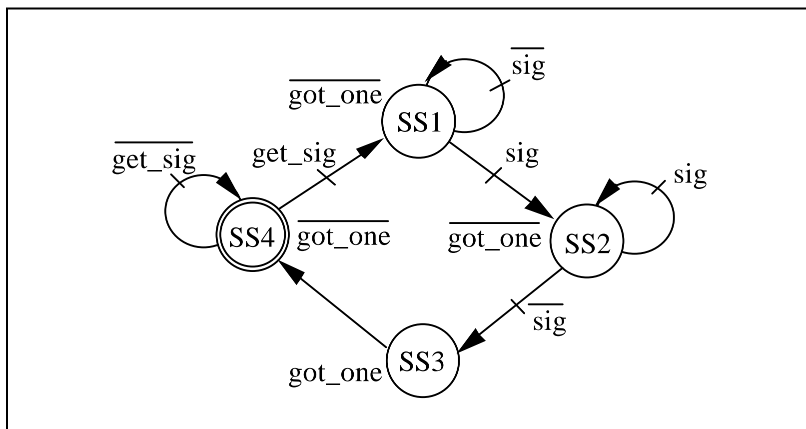

Figure 76.b: *Slave FSM*

Note that these two FSMs in fact communicate by states and the transformation through an output signal communication as explained above has been achieved. The output signal “got_one” identifies the state “SS3” and the output signal “get_sig” identifies the states “S0”, “S2” and “S5”. “S0”, “S2” and “S5” are “call” states for the master FSM and “ SS3” is a “call” state for the slave FSM.

6.3. Processes based description

6.3.1. Modeling

Communicating FSMs can be declared by an architecture containing different processes. In this type of description, both communicating FSMs by states and by output signals are accepted.

Figure 77 shows such a description for the example of figure 76. The master FSM is described by the process labeled “MASTER_FSM” and the slave FSM is described by the one labeled “SLAVE_FSM”. The internal signals “VALUE1” and “VALUE2” represent respectively the state register of the master FSM and the slave FSM. The communication between the two processes is described by using the state registers modeled by the “VALUE1” and VALUE2” signals. In the “MASTER_FSM” process, tests are made on the value of “VALUE2” (“if VALUE2 = 3 ...”) and in the “SLAVE_FSM” process a test is made on the values of “VALUE1“ (“if (VALUE1 = 0 or VALUE1 = 2 or VALUE1 = 5) ...”).

```
entity FSM_HIER is
        port (  RESET:          in      BIT;
                CLK:            in      BIT;
                A:              in      BIT;
                B:              in      BIT;
                SIG:            in      BIT;
                IS_A:           out     BIT;
                IS_B:           out     BIT );
end FSM_HIER;

architecture ARCH of FSM_HIER is
        signal VALUE1 : INTEGER range 0 to 7;
        signal VALUE2 : INTEGER range 1 to 4;
begin
MASTER_FSM : process (RESET,CLK)
begin
        if RESET = '1' then
                VALUE1 <= 0;
        elsif CLK'EVENT and CLK = '1' then
                case  VALUE1 is
                when 0 =>
                        VALUE1 <= 1;
                when 1 =>
                        if VALUE2 = 3 then VALUE1 <= 2;
                        else    VALUE1 <= 1;
                        end if;
                when 2 =>
                        if A = '0' then VALUE1 <= 3;
                        else    VALUE1 <= 4;
                        end if;
                when 3 =>
                        if VALUE2 = 3 then VALUE1 <= 5;
                        else    VALUE1 <= 3;
                        end if;
                        IS_A <= '0';
                when 4 =>
                        if VALUE2 = 3 then VALUE1 <= 5;
                        else    VALUE1 <= 4;
```

```
                end if;
                IS_A <= '1';
            when 5 =>
                if B = '0' then VALUE1 <= 6;
                else      VALUE1 <= 7;
                end if;
            when 6 =>
                if VALUE2 = 3 then VALUE1 <= 2;
                else      VALUE1 <= 6;
                end if;
                IS_B <= '0';
            when 7 =>
                if VALUE2 = 3 then VALUE1 <= 2;
                else      VALUE1 <= 7;
                end if;
                IS_B <= '1';
            end case;
    end if;
end process;
SLAVE_FSM : process (RESET,CLK)
begin
    if RESET = '1' then
            VALUE2 <= 4;
    elsif CLK'EVENT and CLK = '1' then
            case  VALUE2 is
            when 1 =>
                if SIG = '1' then VALUE2 <= 2;
                else      VALUE2 <= 1;
                end if;
            when 2 =>
                if SIG = '0' then VALUE2 <= 3;
                else      VALUE2 <= 2;
                end if;
            when 3 =>
                VALUE2 <= 4;
            when 4 =>
                if (VALUE1 = 0 or VALUE1 = 2 or
                    VALUE1 = 5) then VALUE2 <= 1;
                else      VALUE2 <= 4;
                end if;
            end case;
    end if;
end process;
end ARCH;
```

Figure 77: *VHDL description of 2 hierarchical FSMs communicating by states*

6.3.2. Synthesis

When using a processes based description, it may be useful for optimization that the synthesis tool recognizes and extracts the FSMs. For this the user will ask for this automatic recognition in the VHDL by selecting the option “FSM Extraction”. Two possibilities are then offered:

a) The user does not give any information about encoding. If the optimization criterion is area, the optimized compact state assignment is automatically chosen for

all the FSMs. If the optimization criterion is speed, the one-hot state assignment is automatically chosen for all the FSMs.

b) A single common encoding is specified by the designer; it will be identical for all the FSMs and it may be the optimized compact, the one-hot, the Gray, the Johnson or the sequential encoding. The user can also specify a global synthesis criterion and a global power for his design. Note that in this case the flip-flops used for the state register synthesis are the D flip-flops. The T and JK flip-flops are not supported in this case.

6.4. Structural composition of FSMs

6.4.1. Modeling

In this case, the global interconnection is a structural composition of the FSMs. The communication is restricted to output exchange signals. Therefore, if a transition of the slave FSM depends on a state of the master FSM, a signal identifying the state in the master FSM is sent to the slave FSM as explained in § 2.1. Figure 78 gives the VHDL description of the example of figure 76.

```
library ASYL;
use ASYL.ASYL_RTL.all;

entity MASTER_FSM is
        port (  A:              in      BIT;
                B:              in      BIT;
                GOT_ONE:        in      BIT;
                RESET:          in      BIT;
                CLK:            in      BIT;
                IS_A:           out     BIT;
                IS_B:           out     BIT;
                GET_SIG:        out     BIT     );
attribute WIR_TYP of IS_A, IS_B, GET_SIG : signal is

        OUT_CONTROL;
end MASTER_FSM;

architecture ARCH of MASTER_FSM is
        signal STATE : INTEGER;
        attribute WIR_TYP of STATE : signal is STATE_REG;
begin
GET_SIG <= '1' when STATE = 0 or STATE = 2
                or STATE = 5 else '0';
process (RESET,CLK)
begin
        if RESET = '1' then
                STATE <= 0;
        elsif CLK'EVENT and CLK = '1' then
                case  STATE is
                when 0 =>
                        STATE <= 1;
                when 1 =>
                        if GOT_ONE = '1' then STATE <= 2;
                        else    STATE <= 1;
                        end if;
```

```
                when 2 =>
                        if A = '0' then STATE <= 3;
                        else    STATE <= 4;
                        end if;
                when 3 =>
                        if GOT_ONE = '1' then STATE <= 5;
                        else    STATE <= 3;
                        end if;
                        IS_A <= '0';
                when 4 =>
                        if GOT_ONE = '1' then STATE <= 5;
                        else    STATE <= 4;
                        end if;
                        IS_A <= '1';
                when 5 =>
                        if B = '0' then STATE <= 6;
                        else    STATE <= 7;
                        end if;
                when 6 =>
                        if GOT_ONE = '1' then STATE <= 2;
                        else    STATE <= 6;
                        end if;
                        IS_B <= '0';
                when 7 =>
                        if GOT_ONE = '1' then STATE <= 2;
                        else    STATE <= 7;
                        end if;
                        IS_B <= '1';
                when others => null;
                end case;
        end if;
end process;
end ARCH;
```

Figure 78.a: *VHDL description of the master FSM*

```
library ASYL;
use ASYL.ASYL_RTL.all;

entity SLAVE_FSM is
        port (  CLK:            in      BIT;
                RESET:          in      BIT;
                SIG:            in      BIT;
                GET_SIG:        in      BIT;
                GOT_ONE:        out     BIT     );
attribute WIR_TYP of GOT_ONE : signal is
        OUT_CONTROL;
end SLAVE_FSM;

architecture ARCH of SLAVE_FSM is
        signal STATE : INTEGER;
        attribute WIR_TYP of STATE : signal is STATE_REG;
begin
GOT_ONE <= '1' when STATE = 3 else '0';
process (RESET,CLK)
begin
```

```
        if RESET = '1' then
                STATE <= 4;
        elsif CLK'EVENT and CLK = '1' then
                case  STATE is
                when 1 =>
                        if SIG = '1' then STATE <= 2;
                        else      STATE <= 1;
                        end if;
                when 2 =>
                        if SIG = '0' then STATE <= 3;
                        else      STATE <= 2;
                        end if;
                when 3 =>
                        STATE <= 4;
                when 4 =>
                        if GET_SIG = '1' then STATE <= 1;
                        else      STATE <= 4;
                        end if;
                when others => null;
                end case;
        end if;
end process;
end ARCH;
```

Figure 78.b: *VHDL description of the slave FSM*

```
library ASYL;
use ASYL.ASYL_RTL.all;

entity FSM_HIER is
        port (  RESET:          in      BIT;
                CLK:            in      BIT;
                A:              in      BIT;
                B:              in      BIT;
                SIG:            in      BIT;
                IS_A:           out     BIT;
                IS_B:           out     BIT );
end FSM_HIER;

architecture ARCH of FSM_HIER is
        component MASTER_FSM
                port (  A:              in      BIT;
                        B:              in      BIT;
                        GOT_ONE:        in      BIT;
                        RESET:          in      BIT;
                        CLK:            in      BIT;
                        IS_A:           out     BIT;
                        IS_B:           out     BIT;
                        GET_SIG:        out     BIT );
        end component;
        component SLAVE_FSM
                port (  CLK:            in      BIT;
                        RESET:          in      BIT;
                        SIG:            in      BIT;
                        GET_SIG:        in      BIT;
                        GOT_ONE:        out     BIT );
```

```
        end component;
        for all : MASTER_FSM
                use entity work.MASTER_FSM (ARCH);
        for all : SLAVE_FSM
                use entity work.SLAVE_FSM (ARCH);
        signal GET_SIG : BIT;
        signal GOT_ONE : BIT;
begin
        MASTER : MASTER_FSM
                port map (A,B,GOT_ONE,RESET,
                                CLK,IS_A,IS_B,GET_SIG);
        SLAVE : SLAVE_FSM
                port map (CLK,RESET,SIG,GET_SIG,
                                GOT_ONE);
end ARCH;
```

Figure 78.c: *VHDL description of the interconnection of the 2 FSMs*

6.4.2. Synthesis

Each FSM can be described either with the non explicit template or with the restrictive explicit template. For each FSM, all the synthesis options are available: the state assignment, the choice of the flip-flops, the optimization criterion and the power.

As for the first composition technique, it may be useful that the synthesis tool recognizes and extracts the FSMs. For this the user will ask for this automatic recognition in the VHDL by selecting the option “FSM Extraction”. Then, the three possibilities already offered for the embedded FSMs in the “Finite State Machine Synthesis” section are available:

a) The user does not give any information about encoding. If the optimization criterion is area, the optimized compact state assignment is automatically chosen for all the FSMs. If the optimization criterion is speed, the one-hot state assignment is automatically chosen for all the FSMs.

b) A single common encoding is specified by the designer; it will be identical for all the FSMs and it may be the optimized compact, the one-hot, the Gray, the Johnson or the sequential encoding. The user can also specify a global synthesis criterion and a global power for his design. Note that in this case the flip-flops used for the state register synthesis are the D flip-flops. The T and JK flip-flops are not supported in this case.

c) Specific options can be given for each FSM. For this purpose, a synthesis directive file is specified. This file defines the entity and the architecture names of the FSMs and a dedicated encoding, the choice of the flip-flops, a synthesis criterion and a power for each FSM. The specific options defined in the synthesis directive file overrides the global options. For example, if there are three FSMs embedded in the design, each FSM has to be described in a separated entity named FSM1, FSM2 and FSM3. The architecture corresponding to each FSM is named ARCH. If the designer wants to use the optimized compact encoding for FSM1 with an area optimization criterion, one-hot encoding for FSM2 with a speed optimization criterion and a power of 2, and Gray encoding with T flip-flops for FSM3, the directive file given in figure 79 can be used. Note that choosing T flip-flops for FSM3, FSM3 have to be described with the explicit template. This must be specified in the synthesis directive file with the option "-fl control".

```
directive -ent FSM1 -arch ARCH -c OPT -crit AREA
directive -ent FSM2 -arch ARCH -c ONE -crit SPEED -power 2
directive -ent FSM3 -arch ARCH -c GRAY -ff T -fl control
```

Figure 79: *Example of synthesis directive file*

Note that in this case, if no encoding has been declared for a given FSM, the global encoding menu will be used. So, the global options are the default options. For example, if the designer wants to use the optimized compact encoding for FSM1 and FSM2 with an area optimization criterion, and the one-hot encoding for FSM3, he may select the optimized compact encoding for the global encoding and ask for an area global optimization criterion and he may give a directive file. This file must contain the following line: “directive -ent FSM3 -arch ARCH -c one”, giving the specific options for the FSM3 synthesis. For more details on the format of the synthesis directive file, see the grammar in appendix 1.

Note that the user can choose D, T or JK flip-flops for each FSM described with the explicit template. This choice is forbidden for FSM described with the implicit template.

7. State Chart Synthesis

7.1. Explicit VHDL state chart template

7.1.1. Modeling

The explicit VHDL state chart specification describes conditional transitions between states in a synchronous process style similar to the one used for explicit VHDL FSM template. Within each state, register transfer level operations can be described.

These operations are defined using either conventional VHDL operators ("+", "-", "*", "/", "=", "=/=", ">", ">=", "<", "<=", plus all the Boolean operators) or using VHDL functions similarly to the macro blocks handling: MACRO+.

The results of the operations performed in a state are either latched in memory elements or sent directly to the external world, depending whether they are assigned within or outside the synchronous process. Intermediate results using variables are not latched they correspond to wires.

All signals assigned within the synchronous process are latched. The signals assigned outside the synchronous process are not latched in memory elements and are similar to the data flow part of an FSM declaration as explained in the previous section.

The state chart of an 8 bit multiplier is given in figure 80. The explicit VHDL description of this state chart is given in figure 81. In this example, after the initialization phase in the states 1 and 2, the multiplication algorithm is based on successive steps of additions and shifts of the operands (states 3 and 4). According to the value of the current bit of MQ, either an addition and a shift are made (MQ(i) = '1') or only a shift operation is performed (MQ(i) ='0'). The most significant bits of the result are fed into the output bus in the state 6 and the least significant bits in state 7. This is made by a concurrent signal assignment in the VHDL architecture description given in figure 81.

The skeleton of the explicit VHDL template is given in figure 82.

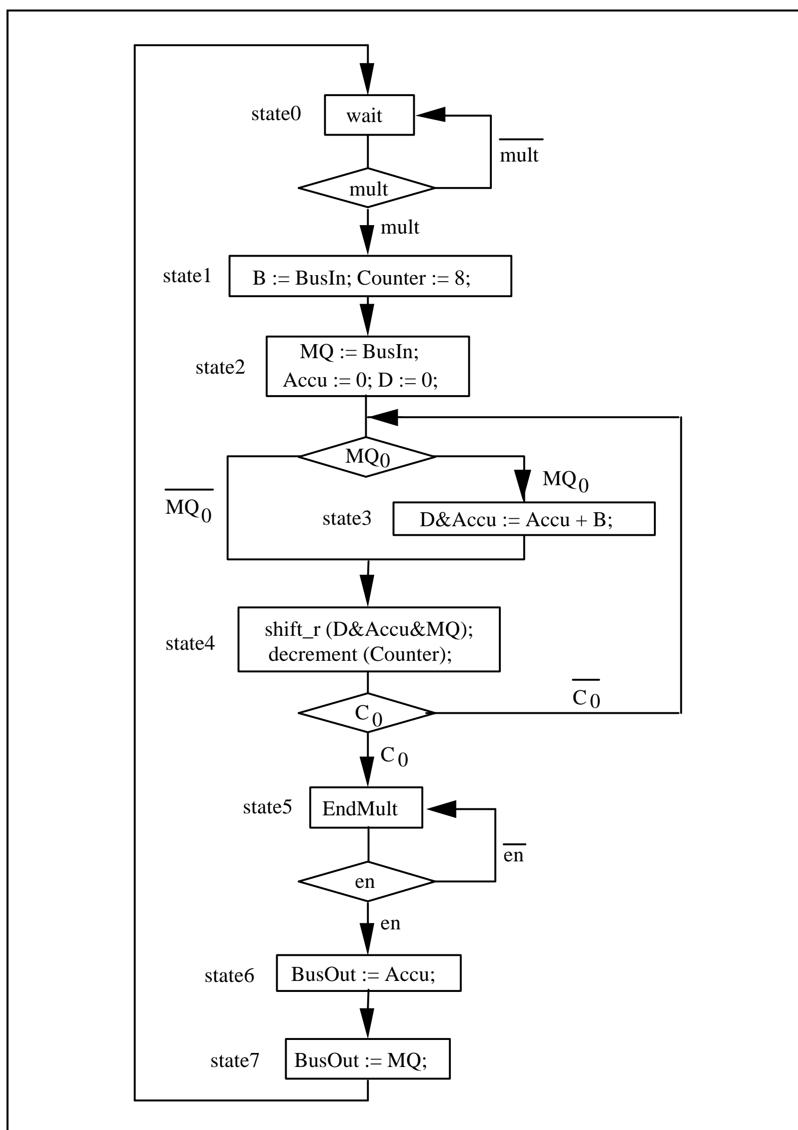

Figure 80: *State chart description of an 8 bits multiplier*

```
library ASYL;
use ASYL.ASYL_RTL.all;
use ASYL.ARITH.all;

entity MULT8 is
port (    RESET :         in BIT;
          CLOCK :         in BIT;
          MULT: in BIT;
          EN:             in BIT;
          FINMULT :       out BIT;
          BUSIN :         in BIT_VECTOR (7 downto 0);
          BUSOUT :        out BIT_VECTOR (7 downto 0) );
attribute WIR_TYP of FINMULT: signal is OUT_CONTROL;
end MULT8;

architecture ARCHI of MULT8 is
```

```
signal STATE: INTEGER;
signal ACCU, MQ, B, COUNTER: BIT_VECTOR (7 downto 0);
signal D : BIT;
attribute WIR_TYP of STATE : signal is STATE_REG;
begin
FINMULT <= '1' when STATE = 5 else
                '0';
BUSOUT <= ACCU when STATE = 6 else
        MQ when STATE = 7 else
        B"00000000";
process (CLOCK, RESET)
    variable AUX1 : BIT_VECTOR (8 downto 0);
    variable MQ_AUX : BIT_VECTOR (7 downto 0);
    variable COUNTER_AUX : BIT_VECTOR (7 downto 0);
begin
    if RESET = '1' then
        STATE <= 0;
    elsif (CLOCK'EVENT and CLOCK = '1') then
        case STATE is
        when 0 =>
                if MULT = '1' then STATE <= 1;
                else STATE <= 0;
                end if;
        when 1 =>
                B <= BUSIN;
                COUNTER <= B"00000000";
                STATE <= 2;
        when 2 =>
                MQ_AUX := BUSIN;
                MQ <= MQ_AUX;
                D<= '0';
                ACCU <= B"00000000";
                if MQ_AUX(0) = '1' then STATE <= 3;
                else STATE <= 4;
                end if;
        when 3 =>
                AUX1 := ('0'& ACCU) + ('0'& B);
                ACCU <= AUX1(7 downto 0);
                D <= AUX1(8);
                STATE <= 4;
        when 4 =>
                D<= '0';
                ACCU <= D & ACCU(7 downto 1);
                MQ_AUX := ACCU(0) & MQ(7 downto 1);
                MQ <= MQ_AUX;
                COUNTER_AUX := '1' & COUNTER(7 downto 1);
                COUNTER <= COUNTER_AUX;
                if COUNTER_AUX(0) = '1' then  STATE <= 5;
                elsif MQ_AUX(0) = '1' then STATE <= 3;
                else STATE <= 4;
                end if;
        when 5 =>
                if EN = '1' then STATE <= 6;
                else STATE <= 5;
                end if;
        when 6 =>
```

```
                STATE <= 7;
          when 7 =>
                STATE <= 0;
          when others =>
                null;
          end case;
    end if;
end process;
end ARCHI;
```

Figure 81: *VHDL description of an 8 bits multiplier*

```
library ASYL;
use ASYL.ASYL_RTL.all;
-- package ASYL_RTL contains attributes used for synthesis and
-- functions corresponding to operations performed by library
blocks
entity <entity_name> is
<port declaration>
<attribute specification>
end <entity_name>;

architecture <architecture_name> of <entity_name> is
<signal declaration>
<attribute specification>
begin
    -- concurrent signal assignments
    <port_name> <= <expression>;
    <port_name> <= <expression_1> when <condition_1> else
                    ....
                    <expression_N-1> when <condition_N-1> else
                    <expression_N>;
    -- <port_name> is the name of an output signal of the design.
    -- <expression> can be: a constant value, a Boolean or
arithmetic
    --      expression on input and internal signals (except the state
    --          register signal), a function call on input and internal
    --          signals (except the state register signal).
    -- <condition> can be a Boolean expression on input and/or
    --          internal signals (including the state register signal).
    process (<clock_name>, <reset_name>) -- sequential process
    <variable declaration>
    begin
        if <reset_name> = <'0' or '1'> then ...
            -- asynchronous reset, expressions are supported
        elsif <clock_name>'EVENT and
                        <clock_name> = <'0' or '1'> then
            case <state_name> is
                when <value_1> =>
                    <signal_name> <= <function_name>
                        (<signal_name>, <variable_name>,...);
                    -- function_name corresponds to an RTL
operation
                    -- name performed by a library block
                    <state_name> <= <value_2>;
                    -- unconditional state assignment.
```

```
                when <value_2> =>
                    <signal_name> <= <signal_name>
                        <VHDL_operator> <signal_name>;
                    -- VHDL_operator is synthesized either on a block
                    -- performing the requested operation or, if it does
                    -- not exist, is synthesized on basic cells.
                when <value_3> =>
                    <variable_name> <=
                        <function_name>(<signal_name>,...);
                    <signal_name> <=
                        <function_name>(<variable_name>,...);
                    -- chained operations can be described using
                    -- intermediate variables
                when <value_4> =>
                    if <predicate_name> = <'0' or '1'> then
                        -- Boolean expressions are supported
                        <signal_name> <=
                            <function_name>(<signal_name>,...);
                    end if;
                    -- operations as state assignments can be
conditional
                ...
            end case;
        end if;
    end process;
end <architecture_name>;
```

Figure 82: *Explicit VHDL model skeleton*

7.1.2. Synthesis process with FSM and data path extraction with maximal resource sharing

We consider here the option where a strict separation is asked for between the finite state machine and the data path. The synthesis process goes through two fundamental steps: the data path creation and a FSM extraction and synthesis. The synthesized circuit is made up of two separate parts: the finite state machine and the data path with an identified clocking scheme between both. Let us recall first that for the extracted FSM a large set of options are available including all the state assignments and the use of D, T or JK flip-flops when asking for the FSM extraction. The example given in figure 80, namely an 8 bits multiplier will illustrate the synthesis flow. The data path uses maximal sharing or minimal resources.

7.1.2.1. Data path generation

The data path generation consists in allocating the minimum number of operators from a block library. Therefore this implies extensive multiplexing. An optimized interconnection is created between the data path blocks. An important effort is done to fully take profit of the existence of commutative operators (adders, multipliers, …) and to perform guided port exchanges to minimize the interconnection cost. The goal is to assign a register only to one port of an operator and to favour multiplexor sharing between operators. Figure 83 gives the generated data path for the 8 bits multiplier example. All the command signals of the registers and multiplexors have not been drawn for legibility.

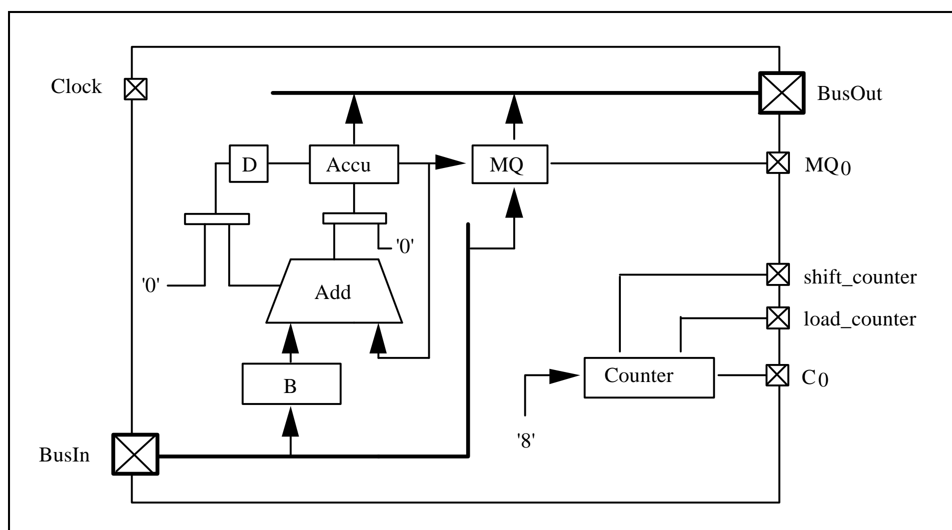

Figure 83: *The generated data path for the 8 bits multiplier example*

7.1.2.2. Controller specification extraction and synthesis

The FSM extracted from the initial specification is a Moore machine with parameterized outputs expressing conditional operations. The FSM extraction specifies the state transition characteristics i.e. the input predicates associated with the transitions as well as the outputs attached to a state. These outputs are the suitable control values of the interconnection devices (multiplexors, internal buses), the operators (functional units) and the memory elements. The FSM is automatically synthesized and connected to the data path; all available state assignments may be used.

The following figure depicts the control flow graph extracted after the data path generation. Input firing conditions associated with the transitions between states are explicitly given. Suitable values of the command signals of registers and multiplexors are depicted for initialization in states 1 and 2.

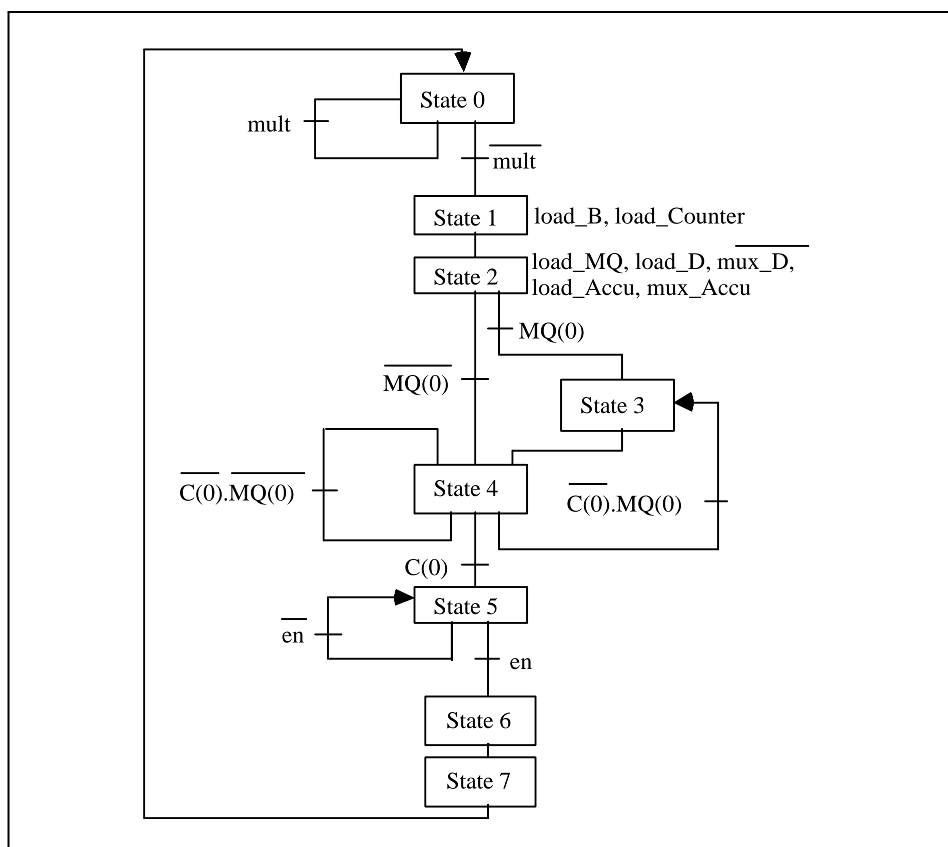

Figure 84: *The extracted control flow graph of the 8 bits multiplier*

7.1.2.3. Final circuit

The final circuit is implemented with the suitable connection of the data path block and the controller implementation block. The figure 85 shows the final circuit for the 8 bits multiplier example. All the connections between the controller and the data path have not be drawn for legibility as well as the connections of the clock and reset ports in the FSM and data path parts.

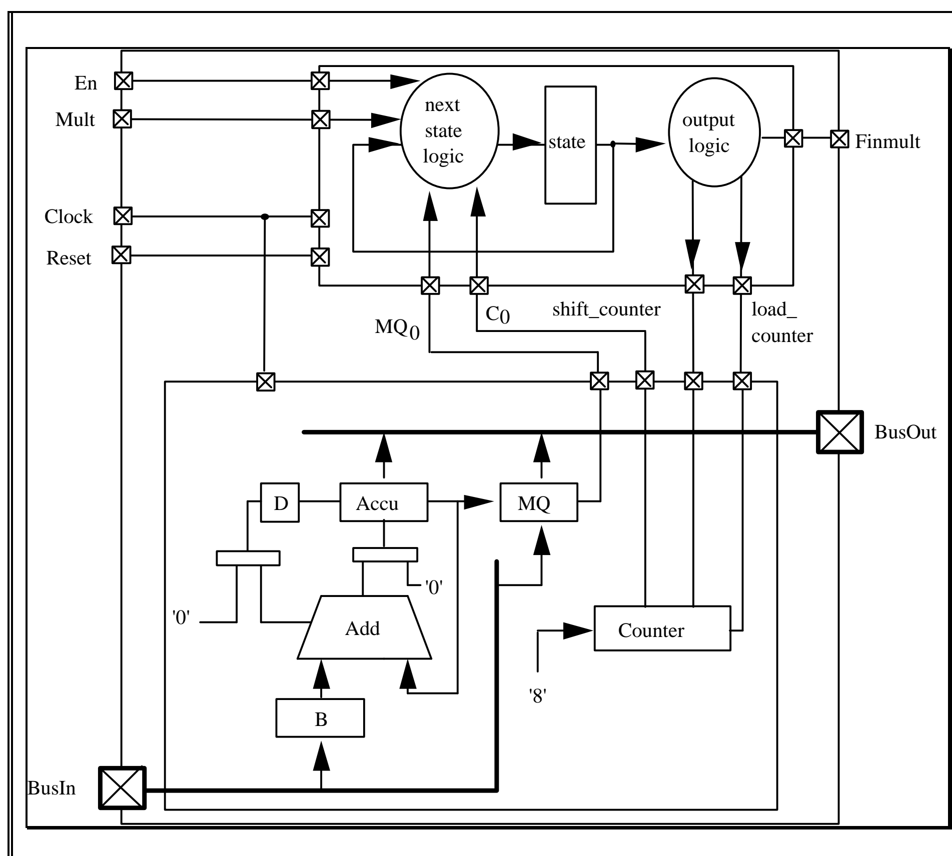

Figure 85: *Final circuit*

7.1.3. Operator or function call

As stated above the operations within the states may be:

- classical VHDL operators,

-VHDL functions.

Not all the possibilities announced in the VHDL macro block handling (MACRO+) are available. Macro blocks can only be inferred through VHDL operators or called through VHDL functions. These VHDL functions have to be combinational functions. VHDL procedures and structural calls are not allowed with this template. No directive file is required.

The implementation of a **VHDL operator** is performed as follows:

if there exists a block in the library performing the function of the VHDL operator then the block is instantiated
else the operator is synthesized by *PLS* macro generators (MACRO+). So the operator is expressed in terms of Boolean equations which is mapped on basic cells. As several sets of equations are stored, an optimal choice according to the user requirements is made as explained in the "Macro Block Handling: Macro+" part.

Note that all the *PLS* arithmetic macro generators are available: adder/substractor, incrementer/decrementer, comparator, multiplier of any bit width in all target technologies. These generators have to be called by inference through VHDL operators: +, -, *, =, /=, <, <=, >, >=.

The use of VHDL functions is interesting for the vendor macro block calls and the predesigned block calls. The use of a VHDL function allows a direct binding or a user controlled choice among a VHDL function and an operation of a library block.

The implementation of a **VHDL function** is performed as follows:

if there exists a block in the library performing the operation specified by the VHDL function then the block is instantiated
else, if no macro block library is specified, or if the library does not contain any
block performing the operation specified by the VHDL function, the VHDL function is synthesized as a black box.

The user has also the ability to describe **compositions of macro blocks** related to classical operation chaining. Figure 86 shows a composition made up of two blocks: 1 multiplier and 1 shifter. The output of the multiplier is directly connected to the input of the shifter.

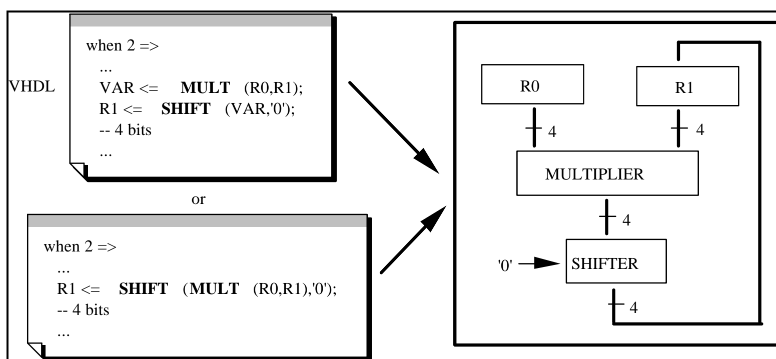

Figure 86: *Composition of macro blocks*

The MULTIPLIER and SHIFTER blocks can be either macro blocks, predesigned blocks from the library or black boxes if they do not exist in the library.

7.1.4. Data path optimization

Data path creation uses a minimal amount of resources. If N_{add} is the maximum number of additions used (in parallel) in a state, then N_{add} adders are instantiated and used if required in the other states; no resource folding is necessary. Data path optimization is performed by interconnection minimization. This interconnection optimization is done by selecting operator assignment and operator alignment.

Operator assignment means that the synthesis tool chooses which operator is chosen in a given state. Suppose that two adders, ADD1 and ADD2, have been instantiated for two additions to be performed in a given state. If an addition has to be performed in a next state, one of the adders (ADD1, ADD2) has to be used. The choice will be guided by connection optimization.

Operator alignment means that if an operator is commutative, an input register can be connected at a functional point to any of its input ports: the selected input port is the one minimizing the connection. Sophisticated techniques are used favoring multiplexor sharing.

7.1.5. Clocking scheme

When connecting a controller to a data path, a decision has to be made concerning what clock event the controller and the data path change state or latch registers. Among several clocking schemes, the one currently proposed in the synthesis tool samples the state register of the controller and the data path register on the same clock edge.

Assuming that the case construct of the VHDL process is activated on the rising edge of the clock, the FSM flip-flops and the data path flip-flops are latched on the rising edge of the clock. The VHDL simulation results of the initial specification and of the synthesized circuit are identical.

The clocking scheme used today between the controller and the data path is illustrated in the figure 87 .

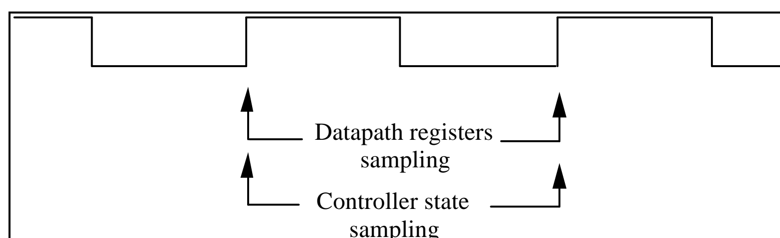

Figure 87 : *Clocking scheme*

7.2. Implicit VHDL state chart template

7.2.1. Modeling

This template is similar to the implicit finite state machine description. The logic corresponding to the operations described at the nodes will be instantiated at each node declaration with more conventional resource folding techniques. In no way, a maximal resource sharing is guaranteed. There is no strict separation between the controller and the data path logic and both logic are merged. The FSM can be extracted for state assignment option as described in the “Finite State Machine Synthesis” section. For the operations descriptions, all the possibilities offered in the MACRO+ generator are available.

Figure 88 gives the VHDL description of the 8 bits multiplier represented in figure 80.

```
library ASYL;
use ASYL.ARITH.all;

entity MULT8 is
port (   RESET :        in BIT;
         CLOCK :        in BIT;
         MULT:          in BIT;
         EN:            in BIT;
         FINMULT :      out BIT;
```

```
          BUSIN :          in BIT_VECTOR (7 downto 0);
          BUSOUT :         out BIT_VECTOR (7 downto 0) );
end MULT8;

architecture ARCHI of MULT8 is
signal VALUE: INTEGER;
signal NEXTVALUE: INTEGER;
signal ACCU, MQ, B, COUNTER: BIT_VECTOR (7 downto 0);
signal NEXTACCU, NEXTMQ: BIT_VECTOR (7 downto 0);
signal NEXTB, NEXTCOUNTER: BIT_VECTOR (7 downto 0);
signal D, NEXTD: BIT;
begin
FINMULT <= '1' when VALUE = 5 else
                 '0';
BUSOUT <= ACCU when VALUE = 6 else
                MQ when VALUE = 7 else
                B"00000000";
P1: process (CLOCK, RESET)
begin
    if RESET = '1' then
          VALUE <= 0;
    elsif (CLOCK'EVENT and CLOCK = '1') then
          VALUE <= NEXTVALUE;
    end if;
end process;
P2: process (VALUE, MULT, EN, NEXTMQ, NEXTCOUNTER)
begin
          case VALUE is
          when 0 =>
                    if MULT = '1' then NEXTVALUE <= 1;
                    else NEXTVALUE <= 0;
                    end if;
          when 1 =>
                    NEXTVALUE <= 2;
          when 2 =>
                    if NEXTMQ(0) = '1' then NEXTVALUE <= 3;
                    else NEXTVALUE <= 4;
                    end if;
          when 3 =>
                    NEXTVALUE <= 4;
          when 4 =>
                    if NEXTCOUNTER(0) = '1' then NEXTVALUE <=
5;
                    elsif NEXTMQ(0) = '1' then NEXTVALUE <= 3;
                    else NEXTVALUE <= 4;
                    end if;
          when 5 =>
                    if EN = '1' then NEXTVALUE <= 6;
                    else NEXTVALUE <= 5;
                    end if;
          when 6 =>
                    NEXTVALUE <= 7;
          when 7 =>
                    NEXTVALUE <= 0;
          when others =>
                    NEXTVALUE <= 0;
```

```
        end case;
end process;
P3: process (VALUE, BUSIN, ACCU, B, D, MQ, COUNTER)
    variable AUX1 : BIT_VECTOR (8 downto 0);
begin
    AUX1 := ('0'& ACCU) + ('0'& B);
    if VALUE = 1 then NEXTB <= BUSIN;
    else NEXTB <= B;
    end if;
    if VALUE = 1 then NEXTCOUNTER <= B"00000000";
    elsif VALUE = 4 then
        NEXTCOUNTER <= '1' & COUNTER(7 downto 1);
    else NEXTCOUNTER <= COUNTER;
    end if;
    if VALUE = 2 then NEXTMQ <= BUSIN;
    elsif VALUE = 4 then
        NEXTMQ <= ACCU(0) & MQ(7 downto 1);
    else NEXTMQ <= MQ;
    end if;
    if (VALUE = 2 or VALUE = 4) then NEXTD<= '0';
    elsif VALUE = 3 then NEXTD <= AUX1(8);
    else NEXTD <= D;
    end if;
    if VALUE = 2 then NEXTACCU <= B"00000000";
    elsif VALUE = 3 then NEXTACCU <= AUX1(7 downto 0);
    elsif VALUE = 4 then NEXTACCU <= D & ACCU(7 downto 1);
    else NEXTACCU <= ACCU;
    end if;
end process;
P4: process (CLOCK,VALUE)
begin
    if (CLOCK'EVENT and CLOCK = '1') then
        B <= NEXTB;
        COUNTER <= NEXTCOUNTER;
        MQ <= NEXTMQ;
        D<= NEXTD;
        ACCU <= NEXTACCU;
    end if;
end process;
end ARCHI;
```

Figure 88 : *VHDL description of an 8 bits multiplier*

7.2.2. Synthesis

The FSM can be extracted and synthesized as explained in the "Finite State Machine Synthesis" section. The user may select five automatic state assignments: the optimized compact, the one-hot, the Johnson or the sequential. If the encoding is not specified the *PLS* uses the user encoding given in the VHDL description.

Note that, for this template, the data path extraction option is not available. The synthesis of the example given in figure 88 will result in a single netlist where the control part and the data path are mixed; no separation is made between both during the synthesis process. Remember that all the possibilities offered in macro block handling: MACRO+ are available. Refer to that section for more information.

7.3. General solution space exploration

The designer has several alternatives to explore different solutions for the final implementation.

First according to the template the following possibilities are offered:

- alternatives on the controller synthesis using the different state assignments, the different types of flip-flops and the speed/area control options,

- alternatives on the data path extraction (extraction of the data path with maximal sharing or not).

Figure 89 illustrates these alternatives.

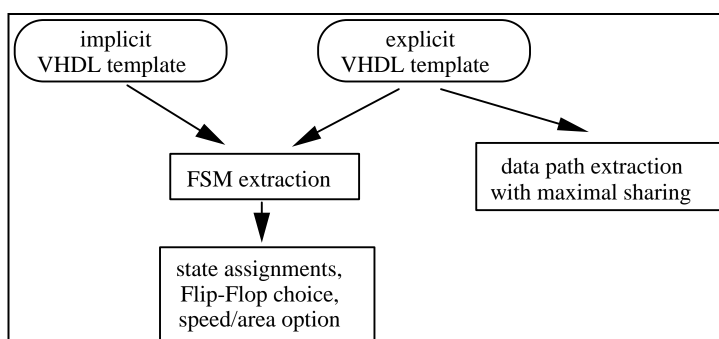

Figure 89: *Solution space exploration according to the VHDL template*

Other alternatives can be explored such as calls of different library blocks.

When using the explicit VHDL template, to help the designer in the selection of the best implementation, the system gives the critical paths of the data path and the controller as output. The resulting frequency of the circuit for the chosen clocking scheme is printed. The critical state and the longest corresponding data path operation are reported to the designer who can then optimize the circuit speed by changing operators used in this state.

8. Communicating State Charts Synthesis

Like for communicating FSMs, state charts can communicate in a concurrent or hierarchical (master-slave) mode. The communication can be made by states or by signals. For more explanations refer to the "Communicating FSMs Synthesis" section.

8.1. Structural composition of state charts

8.1.1. Modeling

Like for communicating FSMs, communicating state charts can be described in a structural VHDL style. In this case, the communication is restricted to output signal exchanges. An example of two communicating state charts is given in figure 90. This example implements the equation: Y = (((Z+T) x T) - 648). The multiplier is the 8 bits sequential multiplier presented in the "State chart synthesis" section. This state chart will be called SC2. The addition and the subtraction are performed by a second state chart which communicates with the multiplier to perform the multiplication. This second state chart will be called SC1.

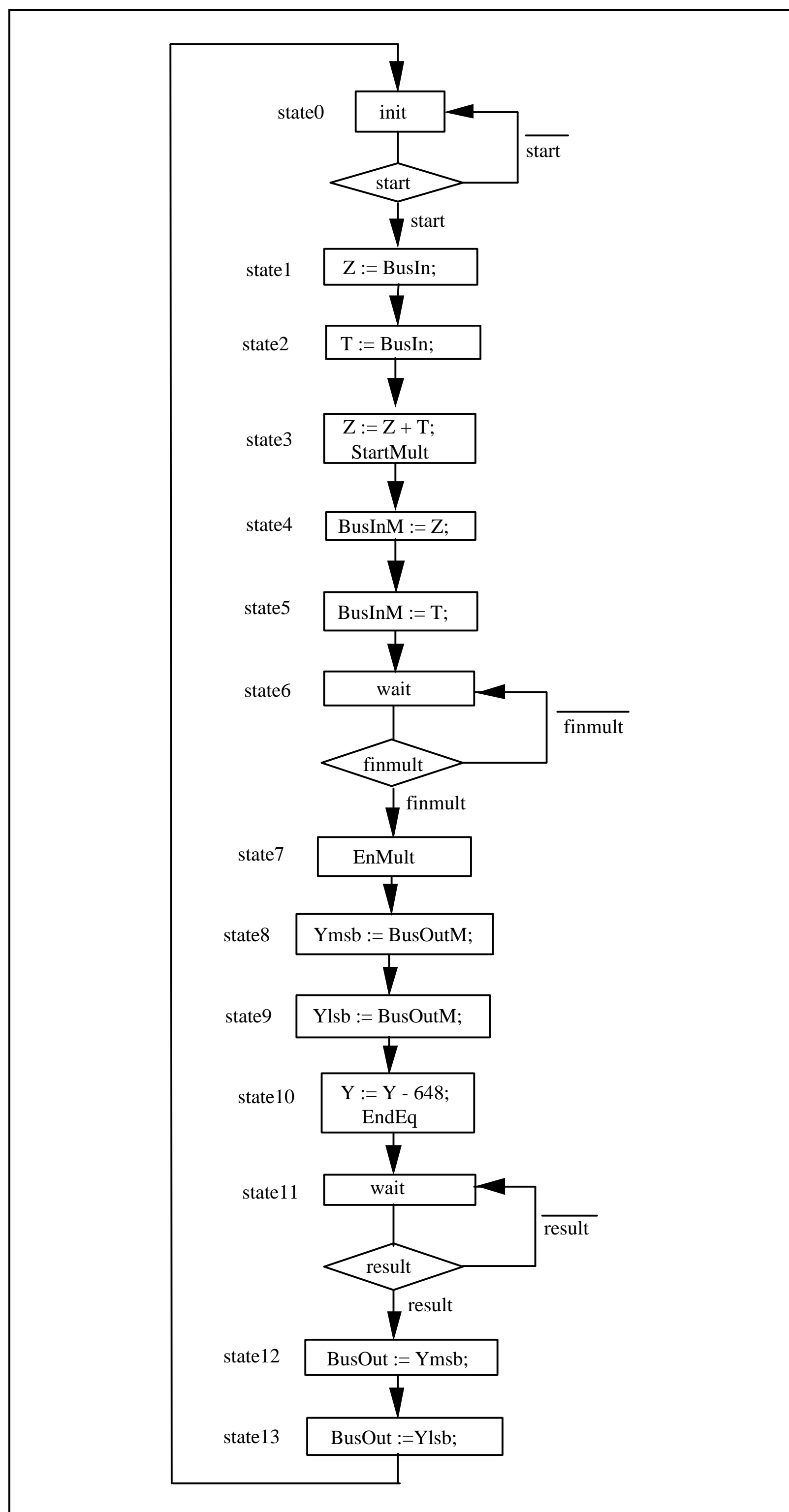

Figure 90.a: *SC1 state chart description*

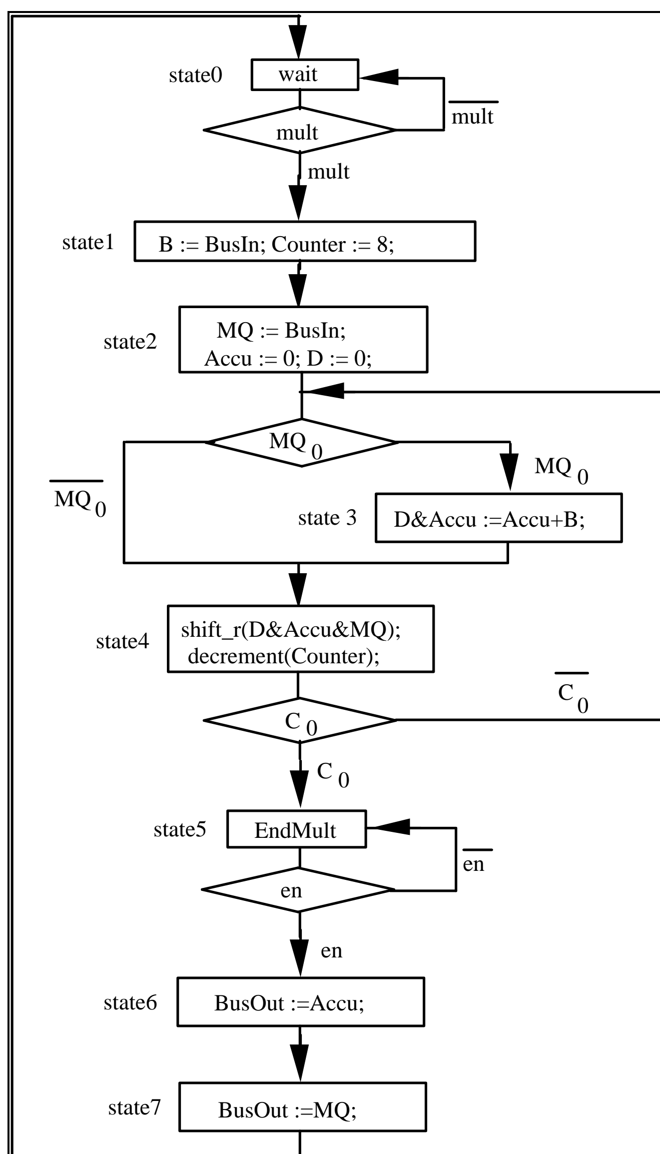

Figure 90.b: *SC2 state chart description*

These two state charts communicate by output signals. During the first states of SC1, the input data are stored in the Z and T registers and then the addition Z = Z+T is performed. In the following state, SC1 sends the "mult" signal to SC2 to start the multiplication. The data contained in the Z and T registers are successively transferred on the input bus of SC2 (multiplier). When the multiplication is finished, SC2 informs SC1 by sending the "finmult" signal and wait for the "en" signal sent by SC1, to put the result on the output bus. SC1 and SC2 communicates by control signals: "mult", "finmult" and "en" but also by data signals: "BusinM" and "BusoutM" which represent the input and output buses of the multiplier SC2.

These two communicating state charts can be described using a structural VHDL style. For this, each state chart must be specified by a pair entity-architecture which are then connected in a top level entity-architecture using component instantiation as shown in figure 90. Each state chart can be described either using the explicit VHDL template or the implicit one. For more details on these templates refer to the "State chart synthesis" section.

```
library ASYL;
use ASYL.ASYL_RTL.all;
use ASYL.ARITH.all;

entity SC1 is
port (  RESET :     in BIT;
        CLOCK :     in BIT;
        START :     in BIT;
        RESULT :    in BIT;
        FINMULT :   in BIT;
        MULT:       out BIT;
        EN:         out BIT;
        ENDEQ:      out BIT;
        BUSOUTM :   in BIT_VECTOR (7 downto 0);
        BUSIN :     in BIT_VECTOR (7 downto 0);
        BUSINM :    out BIT_VECTOR (7 downto 0);
        BUSOUT :    out BIT_VECTOR (7 downto 0) );
attribute WIR_TYP of MULT, EN, ENDEQ: signal is
OUT_CONTROL;
end SC1;

architecture ARCH of SC1 is
signal STATE: INTEGER;
signal Z, T, YMSB, YLSB: BIT_VECTOR (7 downto 0);
attribute WIR_TYP of STATE : signal is STATE_REG;
begin
MULT <= '1' when STATE = 3 else
          '0';
EN <= '1' when STATE = 7 else
        '0';
ENDEQ <= '1' when STATE = 10 else
           '0';
BUSINM <= Z when STATE = 4 else
            T when STATE = 5 else
            X"00";
BUSOUT <= YMSB when STATE = 12 else
            YLSB when STATE = 13 else
            X"00";
process (CLOCK, RESET)
   variable AUX1 : BIT_VECTOR (15 downto 0);
begin
   if RESET = '1' then
        STATE <= 0;
   elsif (CLOCK'EVENT and CLOCK = '1') then
        case STATE is
        when 0 =>
                if START = '1' then STATE <= 1;
                else STATE <= 0;
                end if;
        when 1 =>
                Z <= BUSIN;
                STATE <= 2;
        when 2 =>
                T <= BUSIN;
                STATE <= 3;
        when 3 =>
```

```
                    Z <= Z + T;
                    STATE <= 4;
            when 4 =>
                    STATE <= 5;
            when 5 =>
                    STATE <= 6;
            when 6 =>
                    if FINMULT = '1' then STATE <= 7;
                    else STATE <= 6;
                    end if;
            when 7 =>
                    STATE <= 8;
            when 8 =>
                    YMSB <= BUSOUTM;
                    STATE <= 9;
            when 9 =>
                    YLSB <= BUSOUTM;
                    STATE <= 10;
            when 10 =>
                    AUX1 := (YMSB & YLSB) - X"0288";
                    YMSB <= AUX1(15 downto 8);
                    YLSB <= AUX1(7 downto 0);
                    STATE <= 11;
            when 11 =>
                    if RESULT = '1' then STATE <= 12;
                    else STATE <= 11;
                    end if;
            when 12 =>
                    STATE <= 13;
            when 13 =>
                    STATE <= 0;
            when others =>
                    null;
            end case;
    end if;
end process;
end ARCH;
```

Figure 91.a: *VHDL description of SC1*

```
library ASYL;
use ASYL.ASYL_RTL.all;
use ASYL.ARITH.all;

entity SC2 is
port (  RESET :      in BIT;
        CLOCK :      in BIT;
        MULT:        in BIT;
        EN:          in BIT;
        FINMULT :    out BIT;
        BUSINM :     in BIT_VECTOR (7 downto 0);
        BUSOUTM :    out BIT_VECTOR (7 downto 0) );
attribute WIR_TYP of FINMULT: signal is OUT_CONTROL;
end SC2;

architecture ARCH of SC2 is
```

```
signal STATE: INTEGER;
signal ACCU, MQ, B, COUNTER: BIT_VECTOR (7 downto 0);
signal D : BIT;
attribute WIR_TYP of STATE : signal is STATE_REG;
begin
FINMULT <= '1' when STATE = 5 else
                '0';
BUSOUTM <= ACCU when STATE = 6 else
                MQ when STATE = 7 else
                X"00";
process (CLOCK, RESET)
    variable AUX1 : BIT_VECTOR (8 downto 0);
    variable MQ_AUX : BIT_VECTOR (7 downto 0);
    variable COUNTER_AUX : BIT_VECTOR (7 downto 0);
begin
    if RESET = '1' then
        STATE <= 0;
    elsif (CLOCK'EVENT and CLOCK = '1') then
        case STATE is
        when 0 =>
                if MULT = '1' then STATE <= 1;
                else STATE <= 0;
                end if;
        when 1 =>
                B <= BUSINM;
                COUNTER <= X"00";
                STATE <= 2;
        when 2 =>
                MQ_AUX := BUSINM;
                MQ <= MQ_AUX;
                D<= '0';
                ACCU <= X"00";
                if MQ_AUX(0) = '1' then STATE <= 3;
                else STATE <= 4;
                end if;
        when 3 =>
                AUX1 := ('0'& ACCU) + ('0'& B);
                ACCU <= AUX1(7 downto 0);
                D <= AUX1(8);
                STATE <= 4;
        when 4 =>
                D<= '0';
                ACCU <= D & ACCU(7 downto 1);
                MQ_AUX := ACCU(0) & MQ(7 downto 1);
                MQ <= MQ_AUX;
                COUNTER_AUX := '1' & COUNTER(7 downto 1);
                COUNTER <= COUNTER_AUX;
                if COUNTER_AUX(0) = '1' then  STATE <= 5;
                elsif MQ_AUX(0) = '1' then STATE <= 3;
                else STATE <= 4;
                end if;
        when 5 =>
                if EN = '1' then STATE <= 6;
                else STATE <= 5;
                end if;
        when 6 =>
```

```
                STATE <= 7;
        when 7 =>
                STATE <= 0;
        when others =>
                null;
        end case;       end if;   end process;
end ARCH;
```

Figure 91.b: *VHDL description of SC2*

```
entity TOPSC is
port (  RESET :         in BIT;
        CLOCK :         in BIT;
        START:          in BIT;
        RESULT:         in BIT;
        ENDEQ :         out BIT;
        BUSIN :         in BIT_VECTOR (7 downto 0);
        BUSOUT :        out BIT_VECTOR (7 downto 0) );
end TOPSC;
architecture ARCH of TOPSC is
   component SC1
        port (  RESET, CLOCK, START : in BIT;
                RESULT :        in BIT;
                FINMULT :       in BIT;
                MULT:           out BIT;
                EN:             out BIT;
                ENDEQ:          out BIT;
                BUSOUTM :       in BIT_VECTOR (7 downto 0);
                BUSIN :         in BIT_VECTOR (7 downto 0);
                BUSINM :        out BIT_VECTOR (7 downto 0);
                BUSOUT :        out BIT_VECTOR (7 downto 0) );
   end component;
   component SC2
        port (  RESET :         in BIT;
                CLOCK :         in BIT;
                MULT:           in BIT;
                EN:             in BIT;
                FINMULT :       out BIT;
                BUSINM :        in BIT_VECTOR (7 downto 0);
                BUSOUTM :       out BIT_VECTOR (7 downto 0) );
   end component;
   for all : SC1 use entity work.SC1 (ARCH);
   for all : SC2 use entity work.SC2 (ARCH);
   signal BUSINM, BUSOUTM: BIT_VECTOR (7 downto 0);
   signal MULT, EN, FINMULT : BIT;
begin
   ADDSUBSC : SC1 port map (RESET, CLOCK, START,
RESULT,
FINMULT, MULT, EN, ENDEQ, BUSOUTM, BUSIN, BUSINM,
BUSOUT);
   MULTSC : SC2 port map (RESET, CLOCK, MULT, EN,
FINMULT, BUSINM, BUSOUTM);
end ARCH;
```

Figure 91.c: *VHDL description of the interconnection of SC1 and SC2*

8.1.2. Synthesis

Each state chart can be described either using the explicit VHDL template or the implicit one. For each state chart, all the synthesis options are available: the optimization criterion, the power, the FSM encoding, the FSM flip-flops choice and the data path extraction with maximal resource sharing. Note that the data path extraction and the FSM Flip-Flop choice are available only if the state chart is described using the explicit VHDL template.

As for communicating FSMs, it may be useful that the synthesis tool recognizes and extracts the FSMs. For this the user will ask for this automatic recognition in the VHDL by selecting the option “FSM Extraction”. Then, three possibilities are offered:

a) The user does not give any information about encoding and data path extraction. If the optimization criterion is area, the optimized compact state assignment is automatically chosen for all the FSMs. If the optimization criterion is speed, the one-hot state assignment is automatically chosen for all the FSMs. The data paths are not extracted.

b) A single common encoding is specified by the designer; it will be identical for all the FSMs and it may be the optimized compact, the one-hot, the Gray, the Johnson or the sequential encoding. The user can also specify a global synthesis criterion and a global power for his design. The flip-flops used for the state register synthesis can be the D, T or JK flip-flops, but T or JK flip-flops choice is allowed only for state charts described with the explicit VHDL template. The user can also ask for a global data path extraction but the data path will be extracted only for state charts described with the explicit VHDL template.

c) Specific options can be given for each state chart. For this purpose, a synthesis directive file is specified. This file defines the entity and the architecture names of the state charts and a dedicated encoding, the choice of the flip-flops, a synthesis criterion, a power and a data path extraction for each state chart. The specific options defined in the synthesis directive file overrides the global options. For example, if there are three state charts embedded in the design, each state chart has to be described in a separated entity named SC1, SC2 and SC3. The architecture corresponding to each state chart is named ARCH. If the designer wants to use the optimized compact encoding for SC1 with an area optimization criterion and an extraction of the data path, one-hot encoding for SC2 with a speed optimization criterion and a power of 2, and Gray encoding with T flip-flops for SC3 with an extraction of the data path, the directive file given in figure 92 can be used. Note that choosing T flip-flops for SC3 and the extraction of the data path for SC1 and SC3, SC1 and SC3 have to be described with the explicit VHDL template. This must be specified in the synthesis directive file with the option "-fl control".

```
directive -ent SC1 -arch ARCH -c OPT -crit AREA -fl control
directive -ent SC2 -arch ARCH -c ONE -crit SPEED -power 2
directive -ent SC3 -arch ARCH -c GRAY -ff T -fl control
```

Figure 92: *Example of synthesis directive file*

Note that in this case, if no encoding has been declared for a given state chart, the global encoding will be used. So, the global options are the default options. For example, if the designer wants to use the optimized compact encoding for SC1 and SC2 with an area optimization criterion, and the one-hot encoding for SC3, he may select the optimized compact encoding for the global encoding and ask for an area global optimization criterion and he may give a directive file. This file must contain the following line: “directive -ent SC3 -arch ARCH -c one”, giving the specific

global optimization criterion and he may give a directive file. This file must contain the following line: “directive -ent SC3 -arch ARCH -c one”, giving the specific options for the SC3 synthesis. For more details on the format of the synthesis directive file, see the grammar in appendix 1.

Note that the user can choose D, T or JK flip-flops for each state chart described with the explicit template. This choice is forbidden for state chart described with the implicit template. This is also true for the data path extraction with maximal sharing option.

8.1.3. Synthesis of the example of §1.1 with FSMs and data paths with maximal sharing extraction

We consider here the option where a strict separation is asked for between the finite state machine and the data path with maximal resource sharing for each state chart. Each state chart is synthesized separately.

Figure 93.a and figure 94.a give the generated data path for respectively SC1 and SC2. Figure 92.b and figure 94.b give the extracted control flow graph of respectively SC1 and SC2. In the data path, all the control signals of the registers and multiplexors have not been drawn for more legibility. In the control flow graph, the command signals are given only for the first states for the same reason.

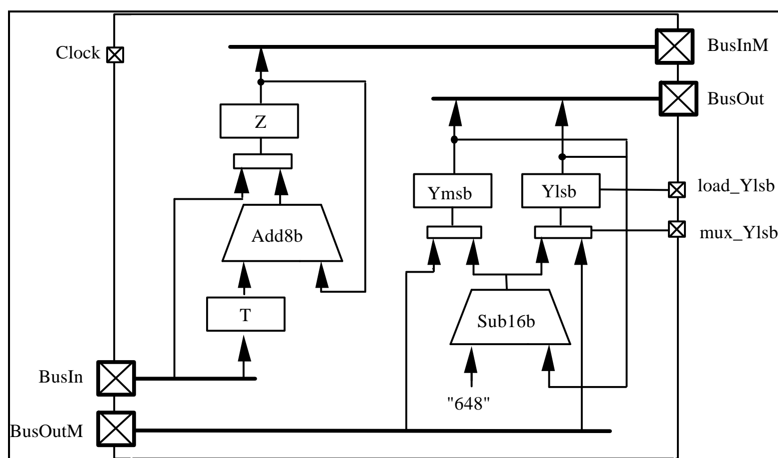

Figure 93.a: *The generated data path for SC1*

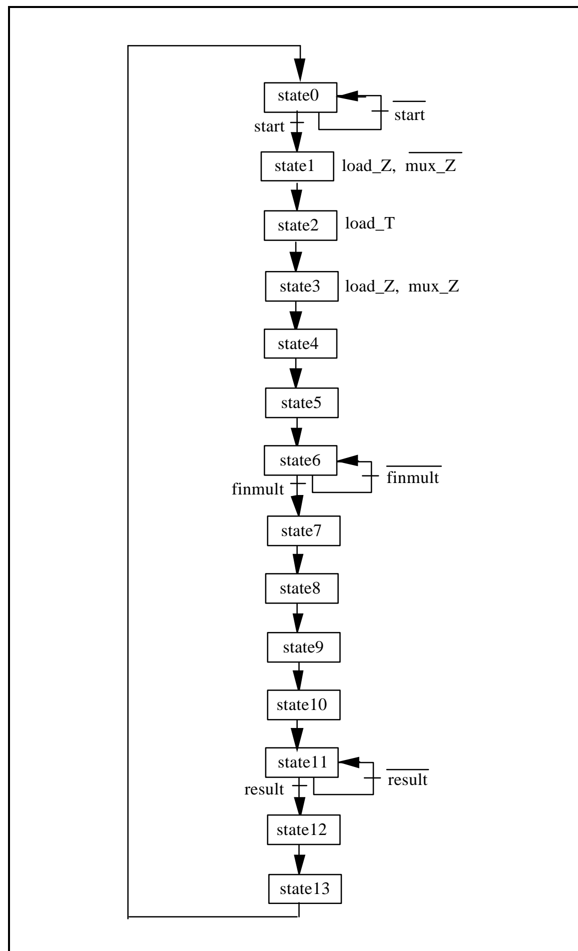

Figure 93.b: *The extracted control flow graph for SC1*

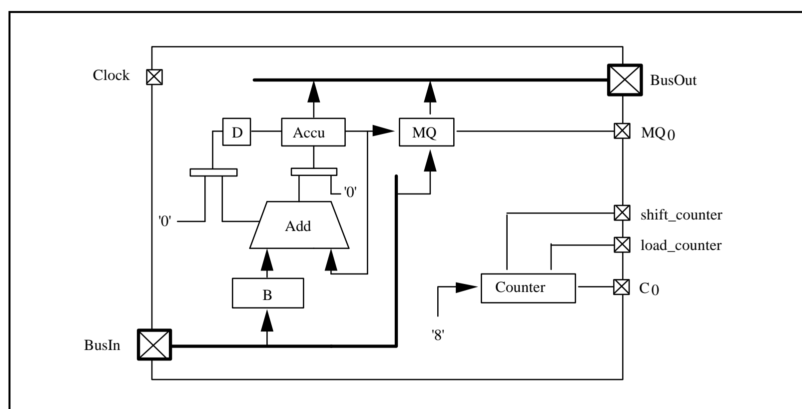

Figure 94.a: *The generated data path for SC2*

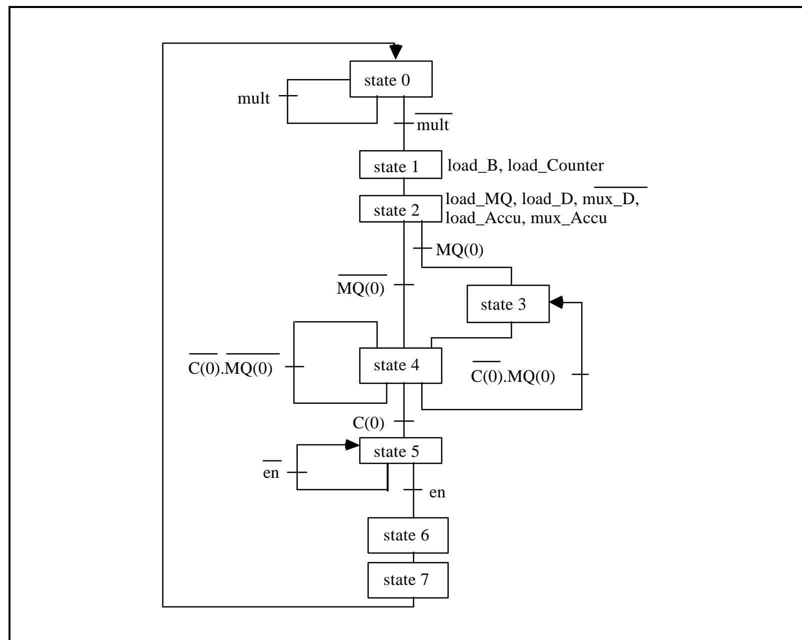

Figure 94.b: *The extracted control flow graph for SC2*

After the synthesis of each state chart, they are connected together. Figure 95 gives the final netlist: the interconnection of SC1 and SC2.

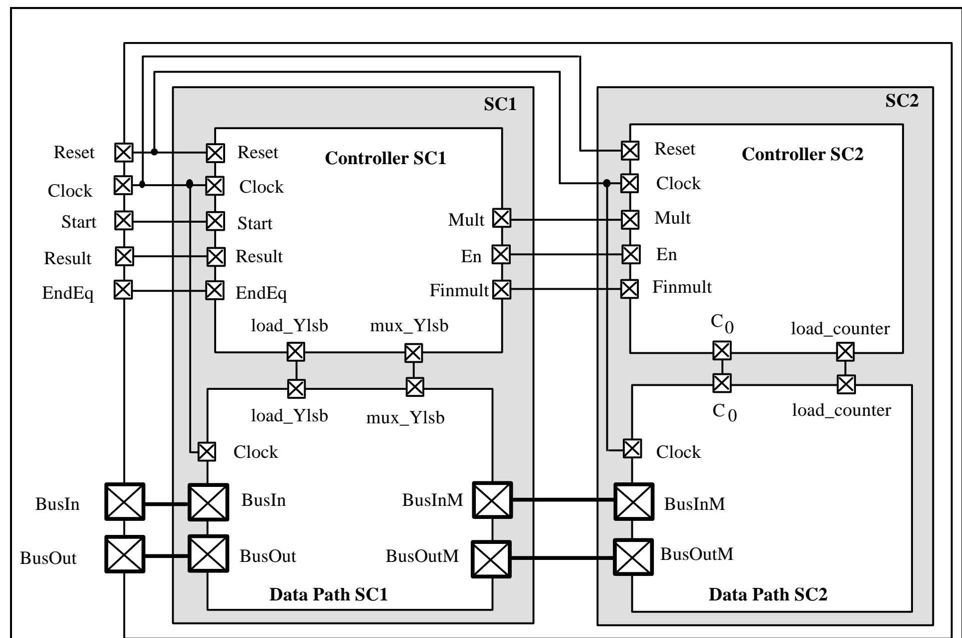

Figure 95: Final netlist: interconnection of SC1 and SC2

8.2. Processes based description

8.2.1. Modeling

As for communicating FSMs, communicating state charts can be described with several communicating processes in an architecture. The example given in figure 96 gives the VHDL description of the two communicating state charts represented in figure 90.

```
library ASYL;
use ASYL.ARITH.all;

entity TOPSC is
port (    RESET :       in BIT;
          CLOCK :       in BIT;
          START:        in BIT;
          RESULT:       in BIT;
          ENDEQ :       out BIT;
          BUSIN :       in BIT_VECTOR (7 downto 0);
          BUSOUT :      out BIT_VECTOR (7 downto 0) );
end TOPSC;

architecture ARCH of TOPSC is
signal VALUESC1: INTEGER;
signal NEXTVALUESC1: INTEGER;
signal Z, T, YMSB, YLSB: BIT_VECTOR (7 downto 0);
signal NEXTZ, NEXTT: BIT_VECTOR (7 downto 0);
signal NEXTYMSB, NEXTYLSB: BIT_VECTOR (7 downto 0);
signal MULT, EN, FINMULT: BIT;
signal BUSINM, BUSOUTM: BIT_VECTOR (7 downto 0);
signal VALUESC2: INTEGER;
signal NEXTVALUESC2: INTEGER;
signal ACCU, MQ, B, COUNTER: BIT_VECTOR (7 downto 0);
signal NEXTACCU, NEXTMQ: BIT_VECTOR (7 downto 0);
signal NEXTB, NEXTCOUNTER: BIT_VECTOR (7 downto 0);
signal D, NEXTD: BIT;
begin
MULT <= '1' when VALUESC1 = 3 else
         '0';
EN <= '1' when VALUESC1 = 7 else
       '0';
ENDEQ <= '1' when VALUESC1 = 10 else
          '0';
BUSINM <= Z when VALUESC1 = 4 else
           T when VALUESC1 = 5 else
           X"00";
BUSOUT <= YMSB when VALUESC1 = 12 else
           YLSB when VALUESC1 = 13 else
           X"00";
SC1_P1: process (CLOCK, RESET)
```

```
begin
    if RESET = '1' then
        VALUESC1 <= 0;
    elsif (CLOCK'EVENT and CLOCK = '1') then
        VALUESC1 <= NEXTVALUESC1;
    end if;
end process;
SC1_P2: process (VALUESC1, START, FINMULT, RESULT)
begin
        case VALUESC1 is
        when 0 =>
                if START = '1' then NEXTVALUESC1 <= 1;
                else NEXTVALUESC1 <= 0;
                end if;
        when 1 =>
                NEXTVALUESC1 <= 2;
        when 2 =>
                NEXTVALUESC1 <= 3;
        when 3 =>
                NEXTVALUESC1 <= 4;
        when 4 =>
                NEXTVALUESC1 <= 5;
        when 5 =>
                NEXTVALUESC1 <= 6;
        when 6 =>
                if FINMULT = '1' then NEXTVALUESC1 <= 7;
                else NEXTVALUESC1 <= 6;
                end if;
        when 7 =>
                NEXTVALUESC1 <= 8;
        when 8 =>
                NEXTVALUESC1 <= 9;
        when 9 =>
                NEXTVALUESC1 <= 10;
        when 10 =>
                NEXTVALUESC1 <= 11;
        when 11 =>
                if RESULT = '1' then NEXTVALUESC1 <= 12;
                else NEXTVALUESC1 <= 11;
                end if;
        when 12 =>
                NEXTVALUESC1 <= 13;
        when 13 =>
                NEXTVALUESC1 <= 0;
        when others =>
                NEXTVALUESC1 <= 0;
        end case;
end process;
```

```
SC1_P3: process (VALUESC1, BUSIN, BUSOUTM, Z, T, YMSB,
YLSB)
    variable AUX1: BIT_VECTOR (15 downto 0);
begin
    AUX1 := (YMSB & YLSB) - X"0288";
    if VALUESC1 = 1 then NEXTZ <= BUSIN;
    elsif VALUESC1 = 3 then NEXTZ <= Z + T;
    else NEXTZ <= Z;
    end if;
    if VALUESC1 = 2 then NEXTT <= BUSIN;
    else NEXTT <= T;
    end if;
    if VALUESC1 = 8 then NEXTYMSB <= BUSOUTM;
    elsif VALUESC1 = 10 then NEXTYMSB <= AUX1(15 downto 8);
    else NEXTYMSB <= YMSB;
    end if;
    if VALUESC1 = 9 then NEXTYLSB <= BUSOUTM;
    elsif VALUESC1 = 10 then NEXTYLSB <= AUX1(7 downto 0);
    else NEXTYLSB <= YLSB;
    end if;
end process;

SC1_P4: process (CLOCK,VALUESC1)
begin
    if (CLOCK'EVENT and CLOCK = '1') then
        Z <= NEXTZ;
        T <= NEXTT;
        YMSB <= NEXTYMSB;
        YLSB<= NEXTYLSB;
    end if;
end process;

FINMULT <= '1' when VALUESC2 = 5 else
           '0';
BUSOUTM <= ACCU when VALUESC2 = 6 else
           MQ when VALUESC2 = 7 else
           X"00";

SC2_P1: process (CLOCK, RESET)
begin
    if RESET = '1' then
        VALUESC2 <= 0;
    elsif (CLOCK'EVENT and CLOCK = '1') then
        VALUESC2 <= NEXTVALUESC2;
    end if;
end process;
```

```
SC2_P2: process (VALUESC2, MULT, EN, NEXTMQ,
NEXTCOUNTER)
begin
        case VALUESC2 is
        when 0 =>
                if MULT = '1' then NEXTVALUESC2 <= 1;
                else NEXTVALUESC2 <= 0;
                end if;
        when 1 =>
                NEXTVALUESC2 <= 2;
        when 2 =>
                if NEXTMQ(0) = '1' then NEXTVALUESC2 <= 3;
                else NEXTVALUESC2 <= 4;
                end if;
        when 3 =>
                NEXTVALUESC2 <= 4;

        when 4 =>
                if NEXTCOUNTER(0) = '1' then
                        NEXTVALUESC2 <= 5;
                elsif NEXTMQ(0) = '1' then NEXTVALUESC2 <= 3;
                else NEXTVALUESC2 <= 4;
                end if;
        when 5 =>
                if EN = '1' then NEXTVALUESC2 <= 6;
                else NEXTVALUESC2 <= 5;
                end if;
        when 6 =>
                NEXTVALUESC2 <= 7;
        when 7 =>
                NEXTVALUESC2 <= 0;
        when others =>
                NEXTVALUESC2 <= 0;
        end case;
end process;

SC2_P3: process (VALUESC2, BUSINM, ACCU, B, D, MQ,
COUNTER)
    variable AUX1 : BIT_VECTOR (8 downto 0);
begin
    AUX1 := ('0'& ACCU) + ('0'& B);
    if VALUESC2 = 1 then NEXTB <= BUSINM;
    else NEXTB <= B;
    end if;
    if VALUESC2 = 1 then NEXTCOUNTER <= X"00";
    elsif VALUESC2 = 4 then
        NEXTCOUNTER <= '1' & COUNTER(7 downto 1);
    else NEXTCOUNTER <= COUNTER;
    end if;
    if VALUESC2 = 2 then NEXTMQ <= BUSINM;
    elsif VALUESC2 = 4 then
        NEXTMQ <= ACCU(0) & MQ(7 downto 1);
    else NEXTMQ <= MQ;
    end if;
    if (VALUESC2 = 2 or VALUESC2 = 4) then NEXTD<= '0';
    elsif VALUESC2 = 3 then NEXTD <= AUX1(8);
```

```
    else NEXTD <= D;
    end if;
    if VALUESC2 = 2 then NEXTACCU <= X"00";
    elsif VALUESC2 = 3 then NEXTACCU <= AUX1(7 downto 0);
    elsif VALUESC2 = 4 then
        NEXTACCU <= D & ACCU(7 downto 1);
    else NEXTACCU <= ACCU;
    end if;
end process;

SC2_P4: process (CLOCK,VALUESC2)
begin
    if (CLOCK'EVENT and CLOCK = '1') then
        B <= NEXTB;
        COUNTER <= NEXTCOUNTER;
        MQ <= NEXTMQ;
        D<= NEXTD;
        ACCU <= NEXTACCU;
    end if;
end process;
end ARCH;
```

Figure 96: *VHDL description of two communicating state charts (SC1 and SC2)*

8.2.2. Synthesis

When using a processes based description, the data path with minimal resources will not be extracted and synthesized separately. Nevertheless, it may be useful for optimization that the synthesis tool recognizes and extracts the FSMs. For this the user will ask for this automatic recognition in the VHDL by selecting the option “FSM Extraction”. Two possibilities are then offered:

a) The user does not give any information about encoding. If the optimization criterion is area, the optimized compact state assignment is automatically chosen for all the FSMs. If the optimization criterion is speed, the one-hot state assignment is automatically chosen for all the FSMs.

b) A single common encoding is specified by the designer; it will be identical for all the FSMs and it may be the optimized compact, the one-hot, the Gray, the Johnson, the sequential or the random encoding. The user can also specify a global synthesis criterion and a global power for his design. Note that in this case the flip-flops used for the state register synthesis are the D flip-flops. The T and JK flip-flops are not supported in this case.

9. Compatibility Between Synopsys and PLS/VHDL

This section lists the differences between Synopsys and *PLS* VHDL subsets today. Some VHDL constructions which are supported by Synopsys are not supported by *PLS* now. Some others constructions, specific to Synopsys (like the ones using the names of the libraries, the names of the packages or the names of the conversion functions) have to be changed to *PLS* VHDL specific constructs.

Each example is given twice. In the first, the bold words indicate the constructions which are not supported by *PLS* VHDL. In the second, the equivalent *PLS* description is given as a work around.

9.1. Configuration unit

Configuration units are not supported by *PLS* VHDL. The work around consists in replacing the configuration unit by a configuration clause in the declarative part of the architecture where the components are declared.

```
entity EXAMPLE is
        port (     ...          );
end EXAMPLE;
architecture ARCH of EXAMPLE is
        component SMALL
        port (     A,B : in BIT;      S : out BIT         );
        end component;
begin
        ...
end ARCH;

configuration CONF of EXAMPLE is
for ARCH
        for all : SMALL use entity MY_LIB.SMALL(ARCH);
        end for;
end for;
end CONF;
```

Figure 97: *Example configured for Synopsys*

To be accepted by *PLS* VHDL, this example has to be transformed into :

```
entity EXAMPLE is
        port (     ...          );
end EXAMPLE;
architecture ARCH of EXAMPLE is
        component SMALL
        port (     A,B : in BIT;      S : out BIT         );
        end component;
        for all : SMALL use entity MY_LIB.SMALL(ARCH);
begin
        ...
end ARCH;
```

Figure 98: *Example configured for PLS VHDL*

9.2. Recursive functions and procedures

Recursivity is not a "natural" concept in logic synthesis. Some synthesis tools accept a "finite recursivity" with a maximum number of calls. In *PLS* VHDL, recursive functions and procedures are not supported. Remove the recursivity in the VHDL description before synthesizing it using *PLS* VHDL.

9.3. Expressions

9.3.1. Expressions used in "port map" of component instantiations:

Expressions cannot be used in the "port map" clause of a component instantiation. An example is given below:

```
architecture ARCH of ENT is
...
component COMP
port (          A,B : in BIT_VECTOR (3 downto 0);
                S : out BIT_VECTOR (3 downto 0) );
end COMP;
for all : COMP use entity work.COMP(ARCH);
begin
...
      C1 : COMP port map (A1 and B1, C1, S1);
...
end ARCH;
```

Figure 105: *Supported by Synopsys and not accepted by PLS VHDL*

To be accepted by *PLS*, intermediate signals have to be used to compute the expressions, as shown below:

```
architecture ARCH of ENT is
...
component COMP
port (          A,B : in BIT_VECTOR (3 downto 0);
                S : out BIT_VECTOR (3 downto 0) );
end COMP;
for all : COMP use entity work.COMP(ARCH);
signal E1 : BIT_VECTOR (3 downto 0);
begin
...
      E1 <= A1 and B1;
      C1 : COMP port map (E1, C1, S1);
...
end ARCH;
```

Figure 106: *Work around to be accepted by PLS VHDL*

9.3.2. Expressions used with the "/", "mod" and "rem" operators:

The "/", "mod" and "rem" operators cannot be used with expressions as operands. An example is given below:

```
architecture ARCH of ENT is
...
begin
...
        S1 <= A1+B1 mod 4;
...
end ARCH;
```

Figure 107: *Supported by Synopsys and not accepted by PLS VHDL*

To be accepted by *PLS*, an intermediate signal (or variable) has to be used as shown below:

```
architecture ARCH of ENT is
...
signal E : BIT_VECTOR (3 downto 0);
begin
...
        E <= A1 + B1;
        S1 <= E mod 4;
...
end ARCH;
```

Figure 108: *Work around to be accepted by PLS VHDL*

9.4. Predefined packages and conversion functions

UNSIGNED and SIGNED types are supported; they are defined as an array of STD_LOGIC in the package SL_ARITH.

The predefined packages and conversion functions are specific to each synthesis tool. In order to run a "Synopsys like" VHDL description in *PLS* VHDL, replace the Synopsys libraries and packages by the equivalent *PLS* libraries and packages.

9.4.1. Using the vhdl option vhdl style synopsys

This option allow the users to run their design written for Synopsys with PLS whitout any modification in their VHDL source code. This function will automaticaly replace the synopsys type conversion functions with the PLS ones.

conv_integer => to_integer
conv_signed => to_signed
...

We recommand to use this option only to facilitate the translation of a design from Synopsys to PLS and not for real design building.

9.4.2. Using package "ARITHMETIC" of library "Synopsys":

The following lines used in Synopsys VHDL descriptions

```
library SYNOPSYS;
use SYNOPSYS.ARITHMETIC.all;
```

must be replaced in *PLS* VHDL descriptions by:

```
library ASYL;
use ASYL.ARITH.all;
```

The following types and conversion functions defined in the package "arithmetic" of the "Synopsys" library must be replaced by the corresponding types and conversion functions defined in the package "arith" of the "Asyl" library.

Synopsys types	Corresponding *PLS* types
UNSIGNED	BIT_VECTOR
SIGNED	BIT_VECTOR
Synopsys functions	Corresponding *PLS* functions
CONV_INTEGER	TO_INTEGER
CONV_SIGNED	TO_BITVECTOR
CONV_UNSIGNED	TO_BITVECTOR

Note that the UNSIGNED and SIGNED types have to be replaced by the BIT_VECTOR type because the UNSIGNED and SIGNED types are defined as arrays of BIT in the "arithmetic" package of the "Synopsys" library.

9.4.3. Using package "BV_ARITHMETIC" of library "Synopsys":

The following lines used in Synopsys VHDL descriptions

```
library SYNOPSYS;
use SYNOPSYS.BV_ARITHMETIC.all;
```

must be replaced in *PLS* VHDL descriptions by:

```
library ASYL;
use ASYL.ARITH.all;
```

The following conversion functions defined in the package "bv_arithmetic" of the "Synopsys" library must be replaced by the corresponding conversion functions defined in the package "arith" of the "Asyl" library.

Synopsys functions	Corresponding *PLS* functions
BVTOI	TO_INTEGER
SBVTOI	TO_BITVECTOR
ITOBV	TO_BITVECTOR
EXT	EXTEND0

9.4.4. Using package "ARITHMETIC" of library "MVL_7":

The following lines used in Synopsys VHDL descriptions

```
library MVL_7;
use MVL_7.TYPES.all;
use MVL_7.ARITHMETIC.all;
```

must be replaced in *PLS* VHDL descriptions by :

```
library ASYL;
use ASYL.SL_ARITH.all;
```

The following types and conversion functions defined in the package "arithmetic" of the "mvl_7" library must be replaced by the corresponding types and conversion functions defined in the package "sl_arith" of the "Asyl" library.

Synopsys types	Corresponding *PLS* types
UNSIGNED	UNSIGNED
SIGNED	SIGNED
Synopsys functions	Corresponding *PLS* functions
CONV_INTEGER	TO_INTEGER
CONV_UNSIGNED	TO_UNSIGNED
CONV_SIGNED	TO_SIGNED

9.4.5. Using package "STD_LOGIC_ARITH" of "Synopsys" library:

The following lines used in Synopsys VHDL descriptions

```
library SYNOPSYS;
use SYNOPSYS.STD_LOGIC_ARITH.all;
```

must be replaced in *PLS* VHDL descriptions by :

```
library ASYL;
use ASYL.SL_ARITH.all;
```

The following types and conversion functions defined in the package "std_logic_arith" of the "Synopsys" library must be replaced by the corresponding types and conversion functions defined in the package "sl_arith" of the "Asyl" library.

Synopsys types	Corresponding *PLS* types
UNSIGNED	UNSIGNED
SIGNED	SIGNED
Synopsys functions	Corresponding *PLS* functions
CONV_INTEGER	TO_INTEGER
CONV_UNSIGNED	TO_UNSIGNED
CONV_SIGNED	TO_SIGNED
CONV_STD_LOGIC_VECTOR	TO_STDLOGICVECTOR
EXT	EXTEND0

9.5. Sensitivity list in a process

Synopsys accepts a process description without or with an incomplete sensitivity list. In this case a warning is issued during the Synopsys compilation of the VHDL description. But the synthesis doesn't take into account the sensitivity list of a process and the corresponding hardware is generated considering that all signals read in the process are active.

On the other hand, *PLS* takes into account the sensitivity lists of the processes during synthesis, to guarantee the compatibility of the simulation results before and after synthesis. So, when translating a Synopsys VHDL description into an *PLS* VHDL description, a particular attention must be given to the sensitivity lists in processes. More precisely, to infer combinatorial logic, the sensitivity list of a process must contain all condition signals and any signal appearing in the left part of an assignment. In general, to verify the content of the sensitivity list, it is suitable to simulate the VHDL description before synthesizing it.

9.6. Indexed and sliced signals in a sensitivity list of a process

Use internal signals to replace indexed or sliced signals in a sensitivity list of a process. An example is given below:

```
entity EXAMPLE is
        port (   A : inout BIT_VECTOR (15 downto 0);
                 B : in BIT_VECTOR (15 downto 0);
                 S1 : out BIT_VECTOR (6 downto 0);
                 S2 : out BIT;
                 S3 : in BIT;
                 S4 : out BIT_VECTOR (15 downto 0)      );
end EXAMPLE;

architecture ARCH of EXAMPLE is
begin
        process ( A(7), B(6 downto 0), S3 )
        begin
                 ...
                 S1 <= B(6 downto 0);
                 S2 <= A(7);
                 A(7) <= S3;
                 ...
        end process;
end ARCH;
```

Figure 109: *Supported by Synopsys and not accepted by PLS VHDL*

To be accepted by *PLS* VHDL, this example has to be transformed into:

```
entity EXAMPLE is
	port (	A : inout BIT_VECTOR (15 downto 0);
		B : in BIT_VECTOR (15 downto 0);
		S1 : out BIT_VECTOR (6 downto 0);
		S2 : out BIT;
		S3 : in BIT;
		S4 : out BIT_VECTOR (15 downto 0)	);
end EXAMPLE;

architecture ARCH of EXAMPLE is
	signal AUX1 : BIT;
	signal AUX2 : BIT_VECTOR (6 downto 0);
begin
	AUX1 <= A(7);
	AUX2 <= B(6 downto 0);
	process ( AUX1, AUX2, S3)
	begin
		...
		S1 <= B(6 downto 0);
		S2 <= A(7);
		A(7) <= S3;
		...
	end process;
end ARCH;
```

Figure 110: *Work around to be accepted by PLS VHDL*

9.7. Synopsys predefined attributes

Synopsys has defined some specific attributes in their packages. These attributes will pass information to the synthesis tool of Synopsys. Because the Synopsys packages defining theses attributes is not included in the VHDL description, these attributes must be put in comments to compile using *PLS*.

9.8. Compilation directives

PLS accepts the following compilation directives:

```
-- <name> synthesis_on
-- <name> synthesis_off

-- <name> translate_on
-- <name> translate_off
```

Therefore, the Synopsys compilation directives:

```
-- Synopsys synthesis_on
-- Synopsys synthesis_off
```

are accepted by *PLS*.

The VHDL lines included between these two directives are ignored by the synthesis tool. These lines are not synthesized.

Appendix 1:

VHDL Subset

Design Entities and Configurations

Entity Header

Generics	supported
Ports	supported
Entity Declarative Part	supported
Entity Statement Part	unsupported

Architecture Bodies

Architecture Declarative Part	supported
Architecture Statement Part	supported

Configuration Declarations

Block Configuration	ignored
Component Configuration	ignored

Subprograms

Functions	supported (no initial value for parameters, one and only one “return” by function in last statement)
Procedures	supported (no “return” statement)

Packages

package	Types TIME and Real from the STANDARD are notsupported and the TEXTIO package is not supported. The STD_LOGIC_1164 package defined by the IEEE is supported. Global signals must not be declared in a package.

Types

Scalar Types

Enumeration Types:

BOOLEAN, BIT	supported
STD_ULOGIC, STD_LOGIC	supported
X01, UX01, X01Z, UX01Z	supported

Integer Types:

INTEGER	supported
POSITIVE subtype	supported
NATURAL subtype	supported
	Any integer type defined by the user with a range is supported.

Physical Types	unsupported

Floating Point Types	unsupported

Composite Types

BIT_VECTOR	supported
STD_ULOGIC_VECTOR	supported
STD_LOGIC_VECTOR	supported
UNSIGNED	supported
SIGNED	supported
Record Types	unsupported
Access Types	unsupported
File Types	unsupported

Mode

In, Out, Inout:	supported
Buffer	supported
Linkage:	unsupported

Declarations

Type Declarations	Supported for enumerated types, types with positive range having constant bounds, bit vector types and bidimentional arrays.
Subtype Declarations	supported

Objects

Constant Declaration	supported (deferred constants are not supported)
Signal Declaration	supported (no initial value, “register” or “bus” type signals are not supported)
Variable Declaration	supported (no initial value, except in function and procedure)
File Declaration	unsupported
Alias Declaration	supported
Attribute Declarations	ignored
Component Declarations	supported

Specifications

Attribute Specification HIGH, REVERSE_RANGE,	Only supported for some predefined attributes : LOW, LEFT, RIGHT, RANGE, EVENT, STABLE. Otherwise, ignored.
Configuration Specification	Supported only with the “all” clause for instances list.
Disconnection Specification	unsupported

Names

Simple Names	supported
Selected Names	supported
Indexed Names	supported
Slice Names	Supported for constant ranges (ranges with constant bounds).
Attribute Names	Supported for some predefined attributes (LEFT, RIGHT, HIGH, LOW, RANGE, REVERSE_ RANGE, EVENT, STABLE).

Expressions

Operators

Logical Operators:

and, or, nand, nor, xor, xnor, not	supported

Relational Operators:

=, /=,<, <=, >, >=	supported

Adding Operators:

&(concatenation)	supported
+, -	supported

Multiplying Operators:

*	supported
/, mod, rem	Supported if 1 operand is a power of 2.

Shift Operators:

sll, srl, sla, sra, rol, ror	supported

Miscellaneous Operators:

abs	Supported for constant operands.
**	Only supported if the left operand is 2.
Sign: +, -	supported

Operands

Abstract Literals	Only integer literals are supported.
Physical Literals	ignored
Enumeration Literals	supported.
String Literals	supported
Bit String Literals	supported
Record Aggregates	unsupported
Array Aggregates clause,	supported (with association or with the “other” but the two methods must not be mixed). Each value must be a constant.
Function Call	supported
Qualified Expressions	supported for accepted predefined attributes
Types Conversions	unsupported
Allocators	unsupported
Static Expressions	supported

Sequential Statements

Wait statement

Wait on *sensitivity_list* until *Boolean_expression*	Supported with one signal in the *sensitivity list* and in the *Boolean expression*. In case of multiple *wait* statements, the *sensitivity list* and the *Boolean expression* must be the same for each *wait* statement.
Wait for *time_expression*	unsupported
Assertion Statement	ignored
Signal Assignment Statement	supported (delay is ignored).
Variable Assignment Statement	supported
Procedure Call Statement	supported
If Statement	supported
Case Statement	supported

Loop Statement

“for ... loop ... end loop”	supported for constant bounds only
“while ... loop ... end loop”	unsupported
“loop ... end loop”	Only supported in the particular case of multiple *wait* statements
Next Statement multiple	Only supported in a *loop* statement in case of *wait* statements
Exit Statement multiple	Only supported in a *loop* statement in case of *wait* statements
Return Statement	Only supported in function bodies.
Null Statement	supported

Concurrent Statements

Block Statement	supported. “Guarded” blocks are not supported. No port or generic clauses
Process Statement	supported
Concurrent Procedure Call	supported
Concurrent Assertion Statement	ignored
Concurrent Signal Assignment Statements	supported (no “after” clause, no “transport” or “guarded” options, no multiples waveforms)
Component Instantiation Statement	supported
“For ... Generate” Statement	supported for constant bounds only
“If ... Generate” Statement	supported

Appendix 2:

Synthesis Directive File Syntax

<Directive> --> **DIRECTIVE** <Entity Specification>
<Architecture Specification>
[<Optimization Criterion>]
[<Power>]
[<FSM Encoding>]
[<FSM Flip-Flops Choice>]
[<Explicit Template Specification>]

<Entity Specification> --> **-ENT** <Entity Name>

<Entity Name> --> <Idf>

<Architecture Specification> --> **-ARCH** <Architecture Name>

<Architecture Name> --> <Idf>

<Optimization Criterion> --> **-CRIT** <Criterion>

<Criterion> --> **SPEED | MAX_SPEED_AREA | SPEED_MAX_AREA |**

AREA

<Power> --> **-POWER** <Power Value>

<Power Value> --> **1 | 2**

<FSM Encoding> --> **-C** <Encoding Value>

<Encoding Value> --> **ONE | OPT | GRAY | JOHN | SEQ | RAN | USER** <Encoding File> | **BEST**

<Encoding File> --> **-IFC** <Encoding File Name>

<Encoding File Name> --> <Idf>

<FSM Flip-Flops Choice> --> **-FF** <Flip-Flop Type>

<Flip-Flop Type> --> **D** | **T** | **JK**

[<Explicit Template Specification>] **-FL CONTROL**

<Idf> --> <Letter> { <Letter> | <Digit> }*

<Number> --> <Digit>+

<Letter> --> A | ... | Z | a | ... | z | _

<Digit> --> 0 | ... | 9

Where:

<...> : Terminal name

--> : Is derived in

| : Alternatives

[...] : Optional element

... * : 0, 1 or several occurrence(s)