PLS

Verilog

Table of Contents

Verilog
1. Introduction 1
2. How to declare a circuit in Verilog 2
2.1. General declaration 2
2.1.1. Module declaration 2
2.1.2. Accepted Verilog types 2
2.2. Hierarchical description 2
3.Data flow Verilog descriptions 4
3.1. How to describe boolean equations 4
3.1.1. Constants 4
3.1.2. Truth Table 4
3.1.3. Don't care 5
3.1.4. How the logic is synthesized 6
3.2. How to describe multilevel logic 6
3.2.1. Gate netlist 6
3.2.2. Netlist using arithmetic operators 7
3.2.3. Optimizations 8
3.2.3.1. Resource folding and minimization of the number of multiplexers 8
3.2.3.2. Recognition of common sub-expressions 8
3.2.3.3. Synthesis of well-balanced trees 9
3.2.3.4. Expression simplification 10
3.3. How to include memory elements using PLS prestored library 11
4. Behavioral Verilog descriptions 13
4.1. Combinational circuits descriptions using always blocks functions and tasks 13
4.1.1. Combinational always blocks 13
4.1.2. Truth tables 14
4.1.3. Netlist declaration 16
4.1.4. Repetitive or bit slice structure 18
4.2. Sequential circuits descriptions using always blocks 19
4.2.1 Description styles 19
4.2.2. Examples: register and counter descriptions 20
4.3. Hierarchy handling through functions and tasks 21
5. General examples using all the Verilog styles 23
5.1. Example 1: timer/counter (prepbenchmark 2) 23
5.2. Example 2: memory map (prepbenchmark 9) 27
6. Finite State Machine Synthesis 31
6.1. Verilog template 31
6.1.1. State register and next state equations 31
6.1.2. Latched and non latched outputs 31
6.1.3. Latched inputs 31
6.2. State assignments 35
6.2.1. State assignment optimizations 35
6.2.2. User controlled state assignment 35
6.3. Symbolic FSM identification 35
6.4. Handling FSMs within your design 36
6.4.1. Pre-processing or separate FSM handling 36
6.4.2. Embedded FSMs 37
7. Communicating Finite State Machines Synthesis 38
7.1. Introduction 38
7.2. Communicating FSMs 38
7.2.1. Concurrent communicating FSMs 38
7.2.2. Hierarchical or master-slave communicating FSMs 40

7.3. Always blocks based description ____ 42
7.3.1. Modeling ____ 42
7.3.2. Synthesis ____ 43
7.4. Structural composition of FSMs ____ 44
7.4.1. Modeling ____ 44
7.4.2. Synthesis ____ 46
8. Verilog Subset for synthesis ____ 47
8.1. Limited Verilog Language Constructs ____ 47
8.1.1. *always* statement ____ 47
8.1.2. *for* statement ____ 47
8.1.3. *repeat* statement ____ 47
8.2. Ignored Verilog Language Constructs ____ 47
8.2.1. Ignored Statements ____ 47
8.2.2. Ignored Miscellanous Constructs ____ 47
8.3. Unsupported Verilog Language Constructs ____ 48
8.3.1. Unsupported Definitions and Declarations ____ 48
8.3.2. Unsupported Statements ____ 48
8.3.3. Unsupported Operators ____ 48
8.3.4. Unsupported Gate-Level constructs ____ 48

Verilog

1. Introduction

Complex circuit or board are commonly designed using a top down methodology. Different specification are required at each level of the design process. At an architectural level, a specification corresponds to a block diagram. A block corresponds to a register transfer block (register, adder, counter, multiplexer, glue logic, finite state machine,…). The connections are N-bit wires. The PLS Verilog provides an efficient way to describe both the circuit globally and each block according to the most efficient "style". The synthesis is then performed with the best synthesis flow for each block. It is obvious that different synthesis methods will be used for arithmetic blocks, glue logic, finite state machines, …. The quality of the synthesis result will depend on the optimization afforded for each block.

The description of the PLS Verilog is design-oriented; the user learns first how to declare a circuit and how to connect blocks. Structural Verilog allowing a hierarchical description is introduced in the first part.

The second part focuses first on combinational blocks given by equations or netlist connecting gates or arithmetic blocks. The corresponding Verilog style is called "data-flow style".

The third part addresses the more complicated Verilog behavioral style which is mainly based on the notion of always block. Even if an always block can also be used to describe combinational blocks its main goal is to declare sequential blocks sensitive to clock events or levels; of course registers and counters are the first examples illustrating this style. A powerful hierarchy handling will be afforded through function and task calls. Indication about PLS optimization is given to the user so that he understands better the obtained results.

Finally, the description styles and the synthesis of a FSM is presented in the fourth part.

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

2. How to declare a circuit in Verilog

2.1. General declaration

2.1.1. Module declaration

A circuit description is declared using a module which contains two parts: the I/O ports and the body. The I/O ports of the circuits are declared. Each port has a name, a mode (input, output or inout) and a type (See ports A,B,C,D,E in the figure 1). In the module body, 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 Verilog language and are mandatory in the description.

```
module EXAMPLE ( A, B, C, D, E);
        input A, B, C;
        output D, E;
        wire T ;
        ...
endmodule
```

Figure 1: *module example*

2.1.2. Accepted Verilog types

The accepted types in Verilog are:

- the bit types (‘0’, ‘1’, ‘X’, ‘Z’) **wire**, **wand**, **wor**, **tri**, **reg**.
- the bit vector type.
- the integer type.
- the memory type (array of registers or integers).
- the supply types **supply0** and **supply1**.

2.2. Hierarchical description

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 net. The component is the building or basic block. A port is a component I/O connector. A net corresponds to a wire between components

In Verilog, a component is represented by a module instantiation. The connections between components are specified within *module instantiation statements*. Such a statement specifies an instance of a component occurring inside of another module of the circuit. Each module instantiation statement must be given a name. Beside the name, a module instantiation statement contains an association list that specifies which actual nets or ports are associated with which local ports of the module 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.

```
module NAND2 ( A, B, Y);
        input A, B;
        output Y;

        assign Y = ~(A & B);
endmodule

module HALFADDER ( X, Y, C, S);
        input X, Y;
        output C, S;
        wire S1, S2, S3;

        NAND2 NANDA ( X, Y, S3);
        NAND2 NANDB ( X, S3, S1);
        NAND2 NANDC ( S3, Y, S2);
        NAND2 NANDD ( S1, S2, S);
        assign C = S3;
endmodule
```

Figure 2: *Structural description of a half adder*

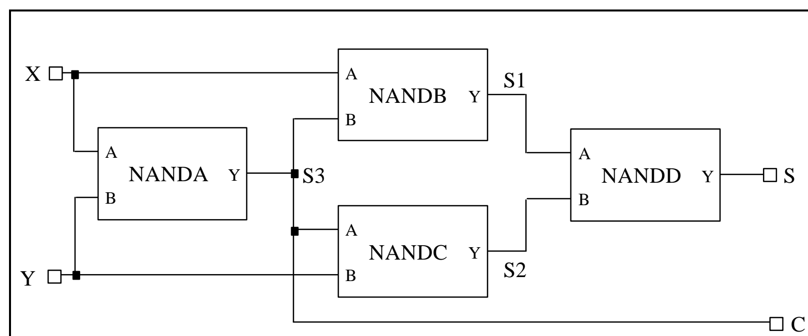

Figure 3: *Synthesized top level netlist*

3.Data flow Verilog descriptions

3.1. How to describe boolean equations

3.1.1. Constants

Constant values may be assigned to signals. In the example of figure 4, the output ports ZERO, ONE and TWO have 2 bits width and are assigned respectively to the constant binary values : 2'b00, 2'b01 and 2'b10. (ZERO[0]=0, ZERO[1]=0, ONE[0]=1, ONE[1]=0, TWO[0]=0, TWO[1]=1).

```
module EXAMPLE ( ZERO, ONE, TWO);
        output ZERO, ONE, TWO;
        wire [1:0] ZERO, ONE, TWO;

        assign ZERO     = 2'b00;
        assign ONE      = 2'b01;
        assign TWO      = 2'b10;
endmodule
```

Figure 4: *Constants example*

3.1.2. 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 5. 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 decodings of the input variables.

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

Figure 5: Truth table

The figures 6 and 7 give two equivalent descriptions of the truth table of figure 5.

```
module EXAMPLE ( A, B, S);
	input A, B;
	output S;

	assign S = ((A == 1'b1) && (B == 1'b0)) ? 1'b0 : 1'b1;
endmodule
```

Figure 6: *Truth table example*

```
module EXAMPLE ( A, B, S);
	input A, B;
	output S;

	assign S = !((A == 1'b1) && (B == 1'b0));
endmodule
```

Figure 7: *Truth table example*

3.1.3. 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.

Figures 9 and 10 describe the truth table of figure 8. In figure 9, the output value is given by successive decodings of the input variables. In figure 10, the operator "{}" is the concatenation operator. For example, in figure 8, 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 8: *Truth table with don't care*

```
module EXAMPLE ( A, B, C, S);
	input A, B, C;
	output S;

	assign S = (A == 1'b0) ? 1'b1 : ((B == 1'b0) ? 1'b0 : 1'bx);
endmodule
```

Figure 9: *Truth table example with don't care*

```
module EXAMPLE ( A, B, C, S);
        input A, B, C;
        output S;
        wire [2:0] S1;

        assign S1 = {A, B, C};
        assign S = ((S1 == 3'b000) || (S1 == 3'b001) ||
                          (S1 == 3'b010) || (S1 == 3'b011))
                                ? 1'b1
                                : (((S1 == 3'b100) || (S1 == 3'b101))
                                            ? 1'b0
                                            : 1'bx);
endmodule
```

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

3.1.4. How the logic is synthesized

The logic is synthesized by using the basic capabilities of PLS. This means that the boolean expressions are first minimized. Then 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 Verilog command (see later on).

3.2. How to describe multilevel logic

3.2.1. Gate netlist

Connection of gates are declared by using Verilog logical operators which are: ~, &, |, ^, ^~, ~^. This is illustrated in the example of figure 11.

```
module EXAMPLE ( A, B, C, S);
        input A, B, C;
        output S;

        assign S = (A & B) | (~C);
endmodule
```

Figure 11: *Gate netlist description*

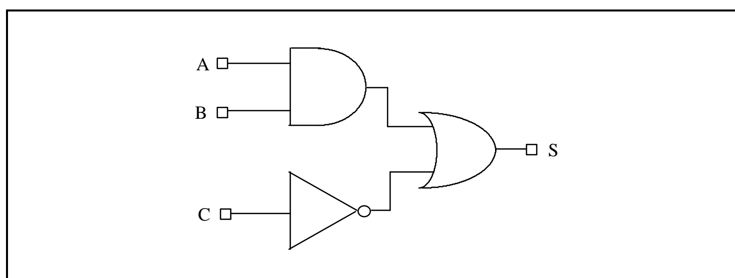

Figure 12: *Possible synthesized netlist*

3.2.2. Netlist using arithmetic operators

Arithmetic operators can be used instead of gates. These operators are the Verilog arithmetic operators: + (addition), - (substraction), * (multiplication), << (left shift), >> (right shift), as well as the Verilog relational operators : == (equality), != (inequality), <, <=, > and >= (ordering).

The number of bits of the operators is given by the width of the operands. In the example of figure 13, the two adders and the substractor operators have 8 bits. The synthesized netlist from the previous description is given in figure 14.

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 (Carry Skip Adder, Carry Look Ahead Adder, Conditional Sum Adder, Sequential Adder),

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

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

-1 multiplier (Braun multiplier).

These equations are carefully synthesized. If one of the operands is a constant, the iterative structure is left for a basic gate optimization.

```
module EXAMPLE ( A, B, C, D, S);
    input [7:0] A, B, C, D;
    output [7:0] S;

    assign S = (A + B) - (C + D);
endmodule
```

Figure 13: *Example using arithmetic operators*

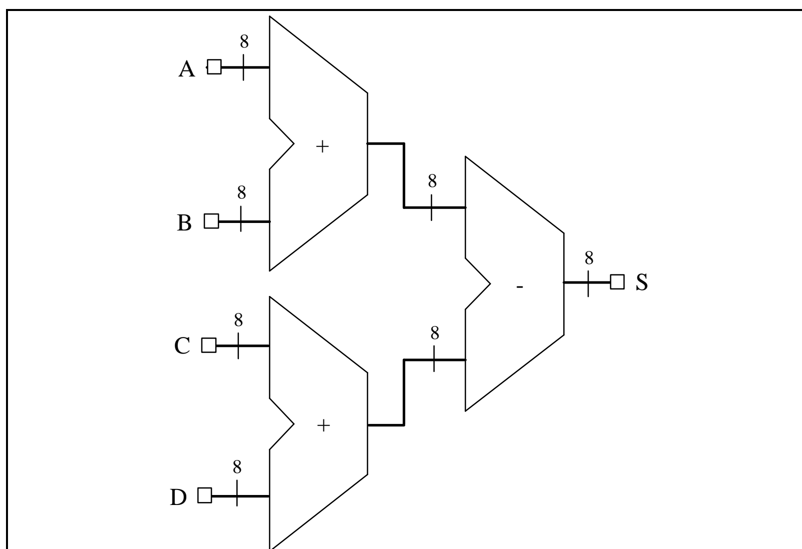

Figure 14: *Synthesized netlist*

3.2.3. Optimizations

3.2.3.1. Resource folding and minimization of the number of multiplexers

The optimizer first shares the operators and then reduces the number of required multiplexers by permuting the operands of the commutative operators. The example given in figure 15 illustrates the resource folding and the minimization of the number of multiplexers. The resulting netlist corresponding to the example described in figure 15 is shown in the figure 16. It contains 1 adder and 1 multiplexer instead of 2 adders or 1 adder and 2 multiplexers which may have been instantiated by direct reading.

```
module EXAMPLE ( A, B, C, E, S);
	input [7:0] A, B, C;
	input E;
	output [7:0] S;

	assign S = E ? A + B : C + A;
endmodule
```

Figure 15: *Example of resource folding*

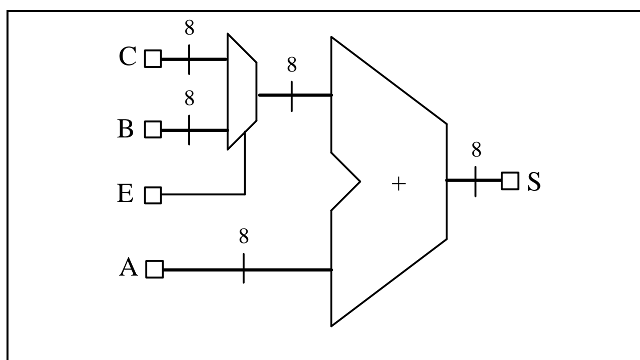

Figure 16: *Synthesized netlist*

3.2.3.2. Recognition of common sub-expressions

The optimizer recognizes common sub-expressions. In the example of figure 17, 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 18. For the sub-expressions "A * C" and "C * D", the optimizer shares the multiplier and instantiates one multiplexer. The adder is also shared . The final netlist is given in figure 18 and contains 1 adder, 2 multipliers and 1 multiplexer instead of 2 adders and 4 multipliers for an unoptimized synthesis.

```
module EXAMPLE ( A, B, C, D, E, F, S);
	input [7:0] A, B, C, D, E, F;
	output [15:0] S;

	assign S = B ? C * D + E * F : E * F + A * C;
endmodule
```

Figure 17: *Example of standard sub-expressions*

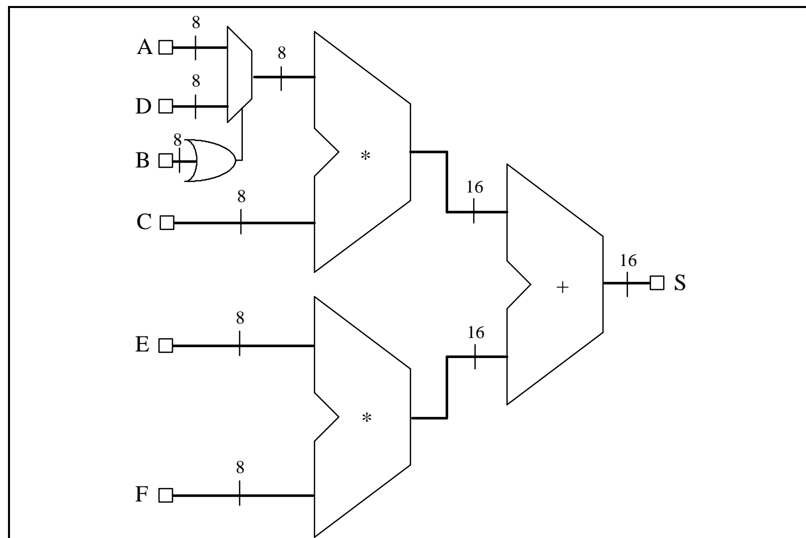

Figure 18: *Synthesized netlist*

3.2.3.3. 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 19 when recognizing the sub-expression "A + B + D + E", a multiplexer is instantiated allowing to add C or F to the sub-expression according to the value of G. The final netlist given in figure 20 contains 4 adders and 1 multiplexer. 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.

```
module EXAMPLE ( A, B, C, D, E, F, G, S);
        input [7:0] A, B, C, D, E, F;
        input G;
        output [7:0] S;

        assign S = G ? E + B + D + A + F : A + B + C + D + E;
endmodule
```

Figure 19: *Example of standard sub-expressions and well-balanced trees*

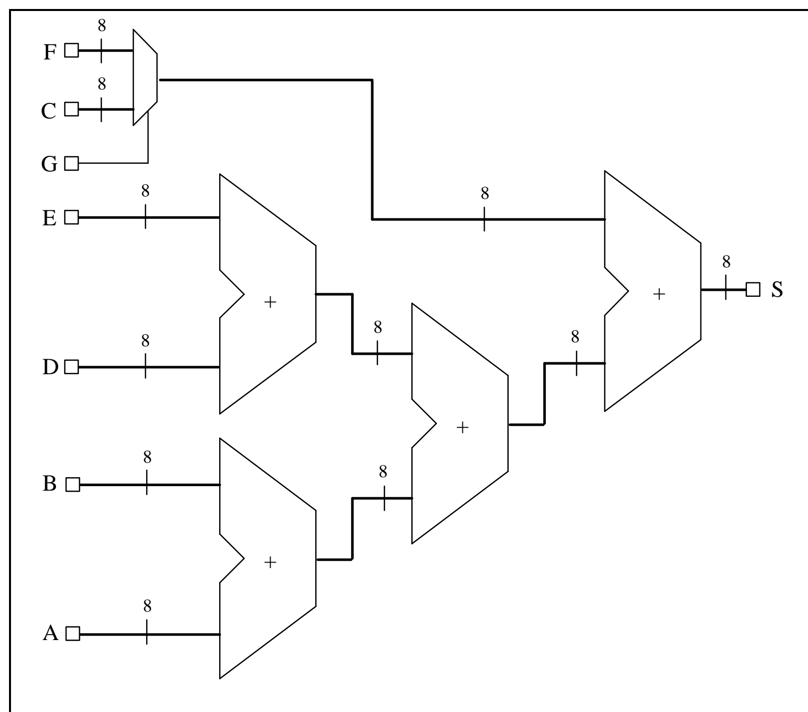

Figure 20: *Synthesized netlist*

3.2.3.4. Expression simplification

The optimizer is able to simplify expressions. In the example given in figure 21, it replaces the expressions [A - 8'b0000_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, substracter or multiplier is instantiated.

```
module EXAMPLE ( A, B, S);
        input [7:0] A;
        input B;
        output [7:0] S;

        assign S = B ? A - A : A -  8’b0000_0010 * A + A;
endmodule
```

Figure 21: *Example of expressions which are simplified*

3.3. How to include memory elements using PLS prestored library

The instantiation of memory elements is made by prestored module instantiations. PLS provides predefined modules corresponding to single bit 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 modules are located in the library "asyllib.v".

These modules are :

- DFF (DATA,CLOCK,Q) :
  D flip-flop,
- DFFC (DATA,CLEAR,CLOCK,Q) :
  D flip-flop with an active high asynchronous Clear,
- DFFP (DATA,PRESET,CLOCK,Q) :
  D flip-flop with an active high asynchronous Preset,
- DFFPC (DATA,PRESET,CLEAR,CLOCK,Q) :
  D flip-flop with active high asynchronous Clear and Preset,

- DLATCH (DATA,ENABLE,Q) :
  D Latch,
- DLATCHC (DATA,CLEAR, ENABLE,Q) :
  D Latch with an active high asynchronous Clear,
- DLATCHP (DATA,PRESET, ENABLE,Q) :
  D Latch with an active high asynchronous Preset,
- DLATCHPC (DATA,PRESET, CLEAR, ENABLE,Q) :
  D Latch with active high asynchronous Clear and Preset,

The full definition of these modules is available for in the file "asyllib.v". You must include this file in your Verilog desccription if you want to use one of these modules.

By default, all these memory elements are of size one but for vectored flip-flops and latches, it is possible to override the size of the module when it is instantiated. An example is given in figure 22. The synthesized netlist is shown in figure 23.

```
'include "asyllib.v"

module EXAMPLE ( DI, CLK, DO) ;
	input [7:0] DI ;
	input CLK ;
	output [7:0] DO ;

	DFF #(8) DFF ( DI, CLK, DO);
endmodule
```

Figure 22: *Example of sequential netlist*

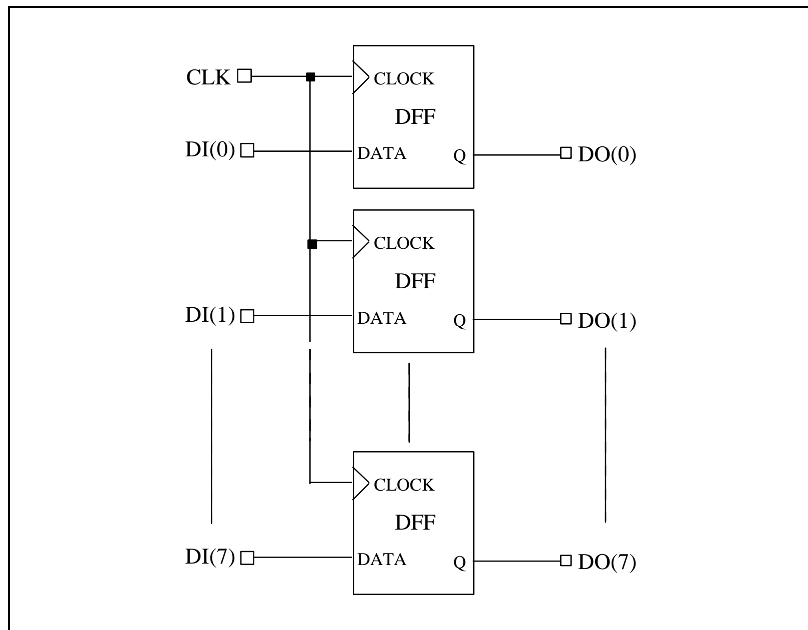

Figure 23: *Synthesized netlist*

4. Behavioral Verilog descriptions

The behavioral Verilog supported in this version includes combinational functions, combinational and sequential tasks, and combinational and sequential always blocks.

4.1. Combinational circuits descriptions using always blocks functions and tasks

4.1.1. Combinational always blocks

A combinational always block assigns values to output boolean functions called registers 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 24. First the register S is assigned to 0, but later on for (A & B) == 1’b1 the value for S is changed in 1’b1.

```
module EXAMPLE ( A, B, S);
    input A;
    input B;
    output S;
    reg S;

    always @( A or B)
    begin
        S = 1’b0;
        if (A & B)
            S = 1’b1;
    end
endmodule
```

Figure 24: *Combinational always block*

The example of figure 24 corresponds to the truth table of figure 25.

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

Figure 25: *Truth table*

At the end of the always block, 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 & B == 1’b1.

A combinational always block has a sensitivity list appearing within parenthesis after the word "always @". A always block is activated if an event (value change or edge) 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 26, the sensitivity list contains three signals which are the A, B and ADD_SUB signals. Figure 26 and figure 27 give two examples of combinational always blocks.

```
module ADD_SUB ( A, B, ADD_SUB, S);
	input [3:0] A, B;
	input ADD_SUB;
	output [3:0] S;
	reg [3:0] S;

	always @( A or B or ADD_SUB)
		if (ADD_SUB)
			S = A + B;
		else
			S = A - B;
endmodule
```

Figure 26: *Combinational always block*

```
module EXAMPLE ( A, B, S);
	input A, B;
	output S;
	reg S, X, Y;

	always @( A or B)
	begin
		X = A & B;
		Y = B & A;
		if (X == Y)
			S = 1'b1;
	end
endmodule
```

Figure 27: *Combinational always block*

Some examples described above by "Data flow Verilog descriptions", will be described again by always blocks.

4.1.2. Truth tables

The truth table of figure 28 is recalled using the data flow style in figure 29 and for comparison using the behavioral style in figure 30. The conditional assignment in figure 29 is replaced by an if statement within a always block in figure 30.

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

8: Truth table

```
module EXAMPLE ( A, B, S);
    input A, B;
    output S;

    assign S = ((A == 1'b1) && (B == 1'b0)) ? 1'b0 : 1'b1;
endmodule
```

Figure 29 : *Data flow description of a truth table*

```
module EXAMPLE ( A, B, S);
    input A, B;
    output S;
    reg S;

    always @( A or B)
    begin
        if ((A == 1'b1) && (B == 1'b0))
            S = 1'b0;
        else
            S = 1'b1;
    end
endmodule
```

Figure 30: *Behavioral description of a truth table*

Similarly don't care values are supported in behavioral style. The truth table of figure 31 is described using the data flow style in figure 32 and using the behavioral style in figure 33. The selected assignment in figure 32 is replaced by a case statement within a always block in figure 33.

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 31: *Truth table with don't care*

```
module EXAMPLE ( A, B, C, S);
        input A, B, C;
        output S;
        wire [2:0] S1;

        assign S1 = {A, B, C};
        assign S= ((S1 == 3'b000) || (S1 == 3'b001) ||
                  (S1 == 3'b010) || (S1 == 3'b011))
                        ? 1'b1
                        : (((S1 == 3'b100) || (S1 == 3'b101))
                                ? 1'b0
                                : 1'bx);
endmodule
```

Figure 32: *Data flow description of a truth table with don't care*

```
module EXAMPLE ( A, B, C, S);
        input A, B, C;
        output S;
        reg S;
        reg [2:0] S1;

        always @(A or B or C)
        begin
                S1 = {A, B, C};
                case (S1)
                        3'b000,
                        3'b001,
                        3'b010,
                        3'b011:  S = 1'b1;
                        3'b100,
                        3'b101:   S = 1'b0;
                        default: S = 1'bx;
                endcase
        end
endmodule
```

Figure 33: *Behavioral description of a truth table with don't care*

4.1.3. Netlist declaration

For netlist declaration, the always block style does not alter at all the data flow description. The always block and its sensitivity list are just put ahead. Figure 34 recalls the description of a gate netlist using the data flow style and figure 35 gives the same description using the behavioral style. The synthesized netlist stays the same.

```
module EXAMPLE ( A, B, C, S);
        input A, B, C;
        output S;

        assign S = (A & B) | (~C);
endmodule
```

Figure 34: *Gate netlist description using data flow style*

```
module EXAMPLE ( A, B, C, S);
        input A, B, C;
        output S;
        reg S;

        always @(A or B or C)
                S = (A & B) | (~C);
endmodule
```

Figure 35: *Gate netlist description using behavioral style*

The Verilog operators supported in the section 3 : "Data flow Verilog descriptions", are also supported in always blocks with the same restrictions and the same optimizations. Please refer to section 3.2 and 3.3 of the section 3 for more details. Figure 36 recalls an example using Verilog arithmetic operator. The synthesized netlist is identical. Similarly, for the adder, the 4 types of adders may be instantiated according to the user requirement.

```
module EXAMPLE ( A, B, C, D, S);
        input [7:0] A, B, C, D;
        output [7:0] S;
        reg [7:0] S;

        always @( A or B or C or D)
                S = (A + B) - (C + D);
endmodule
```

Figure 36: *Example using arithmetic operators*

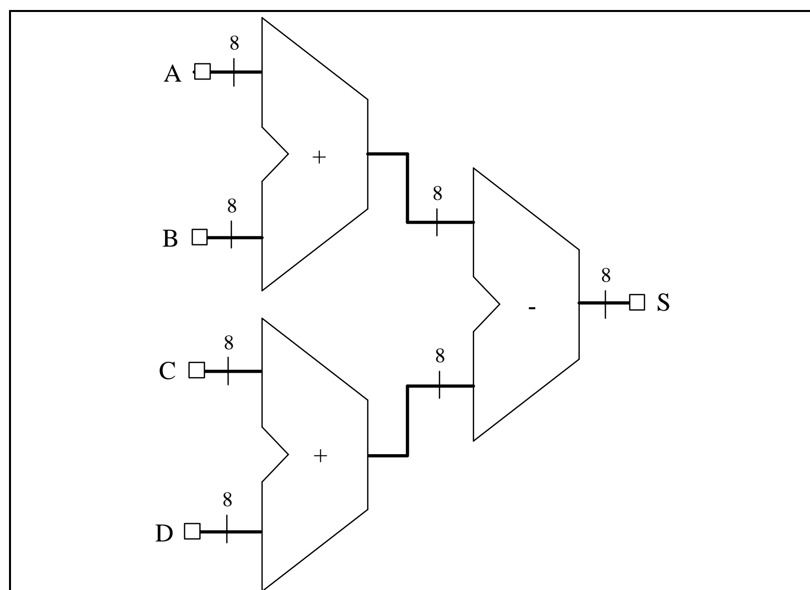

Figure 37: *Synthesized netlist*

4.1.4. Repetitive or bit slice structure

When using always blocks, repetitive or bit slice structure can also be described using the *"for"* statement or the *"repeat"* statement. Figures 38a and 38b give an example of a 8 bits adder described with such statements.

The *"for"* statement is supported for constant bounds, stop test condition using operators <, <=, > or >= and next step computation falling in one of the following specifications: <var> = <var> + step or <var> = <var> - <step> (where <var> is the loop variable and <step> is a constant value).

The *"repeat"* statement is only supported for constant values.

```
module EXAMPLE ( A, B, CIN, SUM, COUT);
        input [0:7] A, B;
        input CIN;
        output [0:7] SUM;
        output COUT;
        reg [0:7] SUM;
        reg COUT;
        reg [0:8] C;
        integer I;

        always @(A or B or CIN)
        begin
                C[0] = CIN;
                for (I = 0 ; I <= 7 ; I = I+1)
                begin
                   SUM[I] = A[I] ^ B[I] ^ C[I];
                   C[I+1] = (A[I]&B[I])|(A[I]&C[I])|(B[I]&C[I]);
                end
                COUT = C[8];
        end
endmodule
```

Figure 38a: *8 bit adder described with a "for" statement*

```
module EXAMPLE ( A, B, CIN, SUM, COUT);
        input [0:7] A, B;
        input CIN;
        output [0:7] SUM;
        output COUT;
        reg [0:7] SUM;
        reg COUT;
        reg [0:8] C;
        integer I;

        always @(A or B or CIN)
        begin
                C[0] = CIN;
                I = 0;
                repeat (8)
                begin
                   SUM[I] = A[I] ^ B[I] ^ C[I];
                   C[I+1]  = (A[I]&B[I])|(A[I]&C[I])|(B[I]&C[I]);
                   I = I+1;
                end
                COUT = C[8];
        end
endmodule
```

Figure 38b: *8 bit adder described with a "repeat" statement*

4.2. Sequential circuits descriptions using always blocks

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

4.2.1 Description styles

The sensitivity list contains a maximum of three edge-triggered events: the clock signal event wich is mandatory and possibly a reset signal event and a set signal event. One and only one "if-else" statement is accepted in such a always block. An asynchronous part may appear before the synchronous part in the first and the second branch of the "if-else" statement. Signals assigned in the asynchronous part must be assigned to constant values which must be ‘0’, ‘1’, 'X' or ‘Z’ or any vector composed of these values. These signals must also be assigned in the synchronous part which corresponds to the last branch of the "if-else" statement. The clock signal condition is the condition of the last branch of the "if-else" statement. Figure 39 shows the skeleton of such a always block. Complete examples are shown in part 2.2.

```
always @(    <posedge | negedge> CLK or
             <posedge | negedge> RST or
             <posedge | negedge> SET)
   ...
begin
   if (RST == <1'b0 | 1'b1>)
      // an asynchronous part may appear here
      // signals must be assigned to constant values ('0', '1', 'X'
         // or 'Z' or any vector composed of these values)
   else
   if (SET == <1'b0 | 1'b1>)
      // an asynchronous part may appear here
      // signals must be assigned to constant values ('0', '1', 'X'
         // or 'Z' or any vector composed of these values)
   else
      // synchronous part
      // signals assigned in the asynchronous part must also be
      // assigned here
end
```

Figure 39: *Sequential always block with an asynchronous reset and set*

4.2.2. Examples: register and counter descriptions

The example of figure 40 gives the description an 8 bit register using a always block.

```
module EXAMPLE ( DI, CLK, DO);
        input [7:0] DI;
        input CLK;
        output [7:0] DO;
        reg [7:0] DO;

        always @( posedge CLK)
                DO = DI ;
endmodule
```

Figure 40: *8 bit register using an always block*

The example of figure 41 gives the description of an 8 bit register with a clock signal and an asynchronous reset signal and figure 42 describes an 8 bit counter.

```
module EXAMPLE ( DI, CLK, RST, DO);
        input [7:0] DI;
        input CLK, RST;
        output [7:0] DO;
        reg [7:0] DO;

        always @( posedge CLK or posedge RST)
                if (RST == 1'b1)
                        DO = 8'b00000000;
                else
                        DO = DI;
endmodule
```

Figure 41: *8 bit register with asynchronous reset using an always block*

```
module EXAMPLE ( CLK, RST, DO);
	input CLK, RST;
	output [7:0] DO;
	reg [7:0] DO;

	always @( posedge CLK or posedge RST )
		if (RST == 1'b1) then
			DO = 8'b00000000;
		else
			DO = DO + 8'b00000001;
endmodule
```

Figure 42: 8 bit counter with asynchronous reset using an always block

4.3. Hierarchy handling through functions and tasks

The declaration of a function or a task aims at handling blocks used several times in a design. They must be declared and used in a module. The heading part contains the parameters: input parameters for functions and input, output and inout parameters for tasks. The content is similar to the combinational always block content. Recursive function and task calls are not supported.

Example of figure 43 shows a function declared within a module. 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 task is shown in figure 44.

```
module EXAMPLE ( A, B, CIN, S, COUT);
	input [3:0] A, B;
	input CIN;
	output [3:0] S;
	output COUT;
	wire [1:0] S0, S1, S2, S3;

	function [1:0] ADD;
		input A, B, CIN;
		reg S, COUT;

		begin
			S = A ^ B ^ CIN;
			COUT = (A&B) | (A&CIN) | (B&CIN);
			ADD = {COUT, S};
		end
	endfunction

	assign 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];
endmodule
```

Figure 43: *Function declaration and function call*

```
module EXAMPLE ( A, B, CIN, S, COUT);
        input [3:0] A, B;
        input CIN;
        output [3:0] S;
        output COUT;
        reg [3:0] S;
        reg COUT;
        reg [1:0] S0, S1, S2, S3;

        task ADD;
                input A, B, CIN;
                output [1:0] C;
                reg [1:0] C;
                reg S, COUT;

                begin
                        S = A ^ B ^ CIN;
                        COUT = (A&B) | (A&CIN) | (B&CIN);
                        C = {COUT, S};
                end
        endtask

        always @( A or B or CIN)
        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
endmodule
```

Figure 44: *Task declaration and task enable*

5. General examples using all the Verilog styles

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

5.1. 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 45.

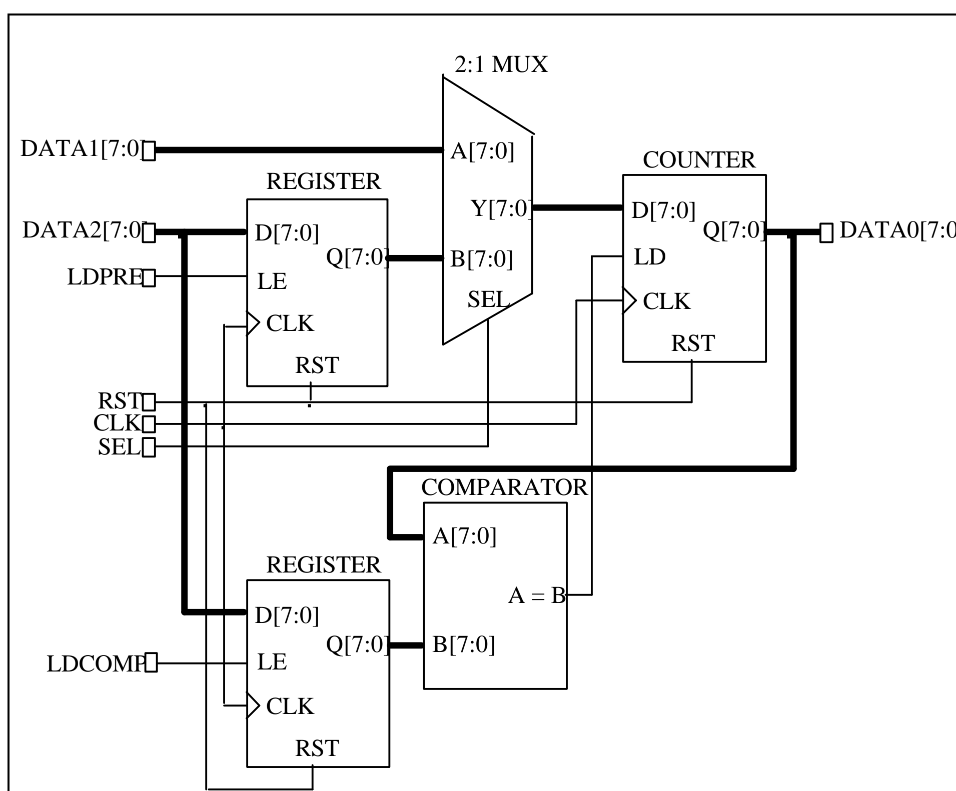

Figure 45: *Example 1: an 8-bit timer/counter*

Figure 52 gives the Verilog description of this example. It is a mixture of structural and data flow styles.

In the top level module named “PREP2” the register, counter and comparator blocks are declared as modules and their descriptions are given at the beginning of the Verilog file. The multiplexor is described in a data flow style using the *“assign”* statement. Figure 46 gives the top level Verilog description of the 8-bit timer/counter.

```
module TOP_LEVEL (  CLK, RST, SEL, LDCOMP, LDPRE,
                    DATA1, DATA2, DATA0);
   input CLK, RST, SEL, LDCOMP, LDPRE;
   input [7:0] DATA1, DATA2;
   output [7:0] DATA0;
   wire [7:0] QPRE, QCOMP, QX, YX;
   wire LD;

   PREP2_REG ONE (CLK, RST, LDPRE, DATA2, QPRE);
   PREP2_REG   TWO   (CLK,   RST,   LDCOMP,   DATA2,
QCOMP);
   PREP2_COUNT THREE (CLK, RST, LD, YX, QX);
   PREP2_COMP FOUR (QX, QCOMP, LD);
   assign YX = (SEL == 1'b0) ? DATA1 : QPRE;
   assign DATA0 = QX;
endmodule
```

Figure 46: *The Verilog top level description of the 8-bit timer/counter*

The module "PREP2_REG" describes the register used in this example. The block diagram is shown in figure 47.

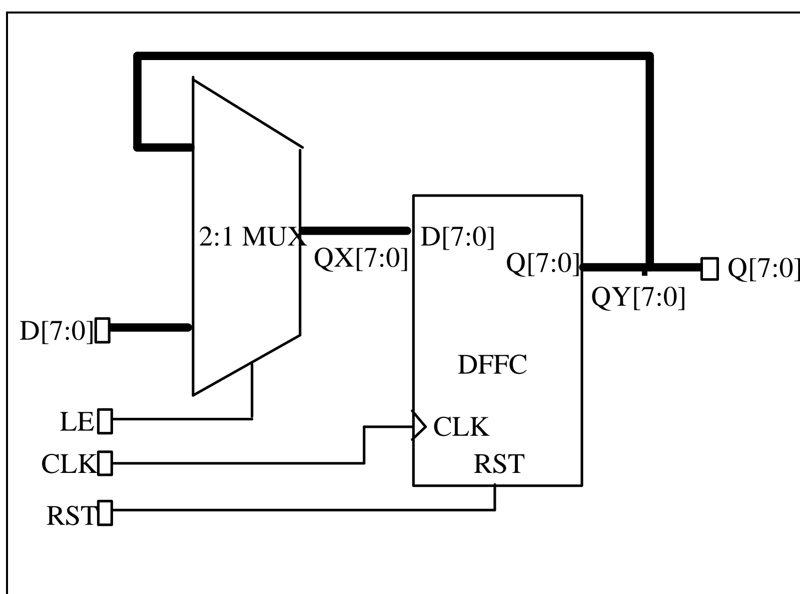

Figure 47: *The register block diagram*

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 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 Verilog description, an internal signals: QX is declared. QX is used to connect the output data of the multiplexor to the input data of the simple register without enable.

Figure 48 gives the Verilog description of the register.

```
module PREP2_REG (   CLK, RST, LE, D, Q);
    input CLK, RST, LE;
    input [7:0] D;
    ouput [7:0] Q;
    reg [7:0] Q;
    wire [7:0] QX;

    assign QX = (LE == 1'b0) ? Q : D;
    always @(posedge CLK or posedge RST)
       if (RST == 1'b1)
          Q = 8'b0 ;
       else
          Q = QX ;
endmodule
```

Figure 48: *The Verilog register description*

The module "PREP2_COUNT" describes the counter. Figure 49 gives the block diagram.

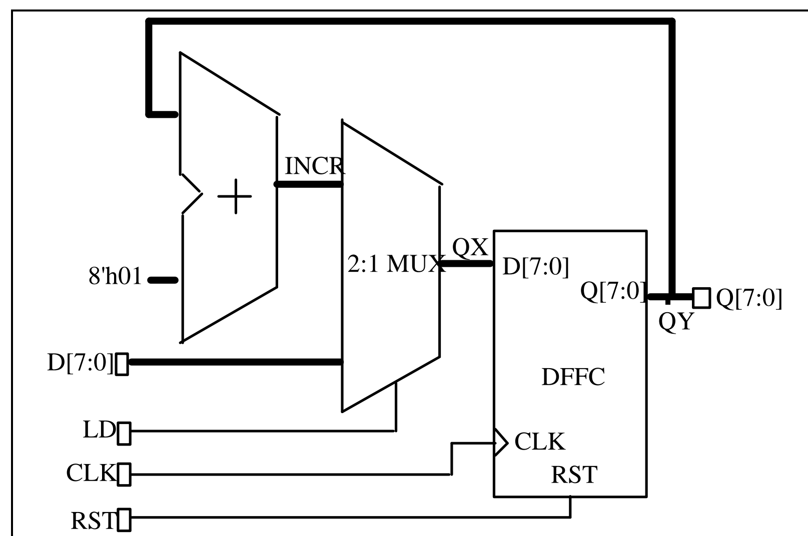

Figure 49: *The counter block diagram*

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 incrementation is described using the Verilog operator "+". 8'h01 is the notation used to express 1 in hexadecimal mode on 8 bits. Figure 50 gives the Verilog representation.

```
module PREP2_COUNT ( CLK, RST, LD, D, Q);
    input CLK, RST, LD;
    input [7:0] D;
    output [7:0] Q;
    reg [7:0] Q ;
    wire [7:0] QX, INCR;

    assign QX = (LD == 1'b1) ? INCR : D;
    assign INCR = Q + 8'h01;
    always @(posedge CLK or posedge RST)
        if (RST == 1'b1)
            Q = 8'b0 ;
        else
            Q = QX ;
endmodule
```

Figure 50: *The Verilog counter description*

The module “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. Figure 51 gives the Verilog description.

```
module PREP2_COMP ( A, B, EQ);
    input [7:0] A, B;
    output EQ;

    assign EQ = (A == B);
endmodule
```

Figure 51: *The Verilog comparator description*

The complete description is given below in figure 52.

```
module PREP2_REG ( CLK, RST, LE, D, Q);
    input CLK, RST, LE;
    input [7:0] D;
    output [7:0] Q;
    reg [7:0] Q;
    wire [7:0] QX;

    assign QX = (LE == 1'b0) ? Q : D;
    always @(posedge CLK or posedge RST)
        if (RST == 1'b1)
            Q = 8'b0 ;
        else
            Q = QX ;
endmodule

module PREP2_COUNT ( CLK, RST, LD, D, Q);
    input CLK, RST, LD;
    input [7:0] D;
    output [7:0] Q;
    reg [7:0] Q;
    wire [7:0] QX, INCR;
```

```
    assign QX = (LD == 1'b1) ? INCR : D;
    assign INCR = Q + 8'h01;
    always @(posedge CLK or posedge RST)
        if (RST == 1’b1)
            Q = 8’b0 ;
        else
            Q = QX ;
endmodule

module PREP2_COMP ( A, B, EQ);
    input [7:0] A, B;
    output EQ;

    assign EQ = (A == B);
endmodule

module TOP_LEVEL (  CLK, RST, SEL, LDCOMP, LDPRE,
                    DATA1, DATA2, DATA0);
    input CLK, RST, SEL, LDCOMP, LDPRE;
    input [7:0] DATA1, DATA2;
    output [7:0] DATA0;
    wire [7:0] QPRE, QCOMP, QX, YX;
    wire LD;

    PREP2_REG ONE (CLK, RST, LDPRE, DATA2, QPRE);
    PREP2_REG TWO (CLK, RST, LDCOMP, DATA2, QCOMP);
    PREP2_COUNT THREE (CLK, RST, LD, YX, QX);
    PREP2_COMP FOUR (QX, QCOMP, LD);
    assign YX = (SEL == 1'b0) ? DATA1 : QPRE;
    assign DATA0 = QX;
endmodule
```

Figure 52: *Complete Verilog representation of the 8-bit timer/counter*

5.2. 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 53 represents the block diagram of this example. Figure 54 gives the outputs value according to the inputs value and figure 55 gives the Verilog description.

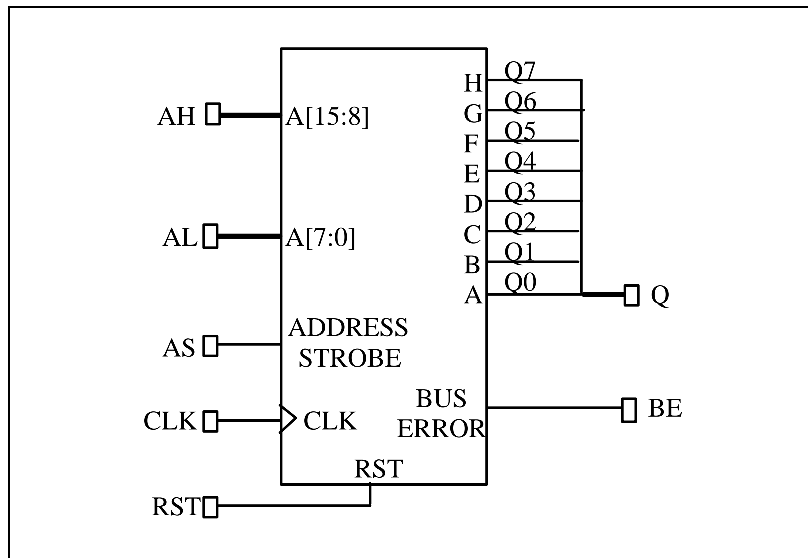

Figure 53: *Example 2: a memory map*

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 54: *Example 2: table*

This example is described using the behavioral style. It uses a single always block 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"* statement are used to assigned the output according to the values of the AS, AH and AL inputs.

```
module PREP9 ( CLK, RST, AS, AL, AH, BE, Q);
   input CLK, RST, AS;
   input [7:0] AL, AH;
   output BE;
   output [7:0] Q;
   reg [7:0] Q;
   reg BE;

   always @(posedge CLK or posedge RST)
   begin
      if (RST == 1'b1)
      begin
         Q = 8'h00;
         BE = 1'b0;
      end
      else
         if (AS == 1'b1)
         begin
            BE = 1'b0;
            if ((AH >= 8'hF0) && (AH <= 8'hFF))
               Q = 8'h80;
            else if ((AH <= 8'hEF) && (AH >= 8'hE8))
               Q = 8'h40;
            else if ((AH <= 8'hE7) && (AH >= 8'hE4))
               Q = 8'h20;
            else if (AH == 8'hE3)
               Q = 8'h10;
            else if (AH = 8'hE2)
               if ((AL <= 8'hFF) && (AL >= 8'hC0))
                  Q = 8'h08;
               else if ((AL <= 8'hBF) && (AL >= 8'hB0))
                  Q = 8'h04;
               else if ((AL <= 8'hAF) && (AL >= 8'hAC))
                  Q = 8'h02;
               else if (AL = 8'hAB)
                  Q = 8'h01;
               else
               begin
                  Q = 8'h00;
                  BE = 1'b1;
               end
            else
            begin
               Q = 8'h00;
               BE = 1'b1;
            end
         end
         else
         begin
```

```
            Q = 8'h00;
            BE = 1'b0;
        end
endmodule
```

Figure 55: *Example 2: Verilog description of a memory map*

6. Finite State Machine Synthesis

6.1. Verilog template

A finite state machine can be “hidden” in a Verilog description. Such a finite state machine description contains at least one sequential always block declaration or at most two always blocks: a sequential one and a combinational one. The sensitivity list of the sequential always block 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. The reset is not mandatory. If it is declared, it has to be an asynchronous signal. In the sequential always block, an *“if”* statement specifies an asynchronous reset if it exists and a synchronous part assigning the state variable.

6.1.1. State register and next state equations

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

• The type of the state register can be integer or bit_vector.

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

6.1.2. Latched and non latched outputs

• The non latched outputs must be described in a combinational always block or using data flow conditional statements ("assign <output> = <condition> ? <value>: ...").

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

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

6.1.3. 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 always block with the latched inputs. This sequential always block must be the one where the state register is assigned as showed in figure 58. Figure 58 gives the Verilog description of the FSM described in figure 56 where the two inputs A and B are latched.

The figure 56 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 always block (cf. figure 57). The second description uses both a sequential always block and a combinational one (cf. figure 58).

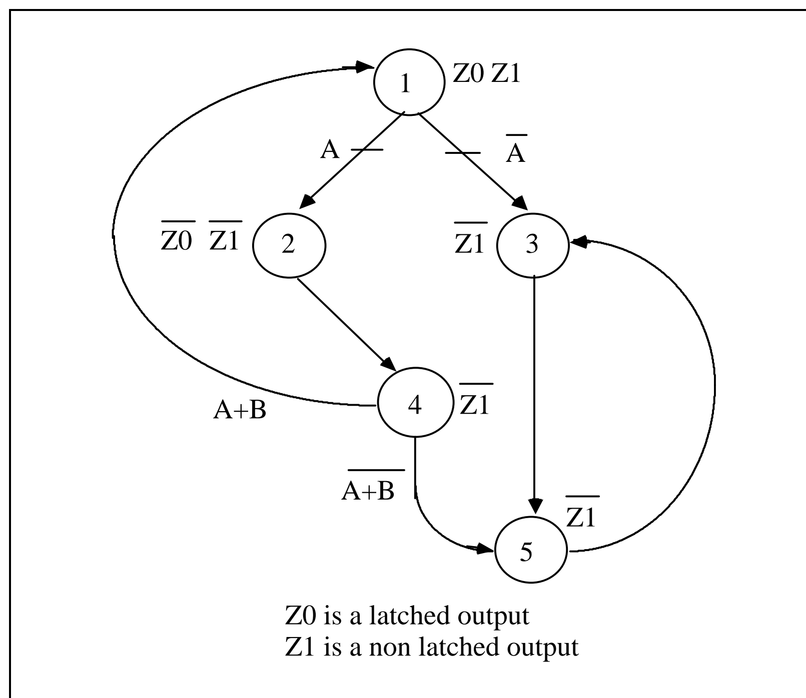

Figure 56: *Graphical representation of a Moore FSM*

The FSM represented in figure 56 is described in figure 57 using a sequential always block for the state register, the latched output Z0 and the next state logic, and a conditional assignment for the output Z1.

```
module FSM ( RESET, CLOCK, A, B, Z0, Z1);
        input RESET, CLOCK, A, B;
        output Z0, Z1;
        reg Z0;
        integer STATE;

        always @(posedge CLOCK or posedge RESET)
                if (RESET == 1'b1)
                        STATE = 1;
                else
                        case (STATE)
                                1: begin
                                        Z0 = 1'b1;
                                        if (A)
                                                STATE = 2;
                                        else
                                                STATE = 3;
                                end
                                2: begin
                                        Z0 = 1'b0;
                                        STATE = 4;
                                end
                                3: STATE = 5;
                                4: if (A | B)
                                        STATE = 1;
                                   else
                                        STATE = 5;
                                5:      STATE = 3;
```

```
                        default: STATE = 1;
                endcase
        assign Z1 = (STATE == 1); // Z1 is not latched
endmodule
```

Figure 57: *Verilog description of a Moore FSM*

Note that in figure 57, the *"case"* statement describing the next state logic and the state register assignment in the sequential always block has a *"default"* 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 56 is described in figure 58 using a sequential always block for the state register and the latched output Z0, and a combinational always block for the output Z1 and the next state logic.

```
module FSM ( RESET, CLOCK, A, B, Z0, Z1);
        input RESET, CLOCK, A, B;
        output Z0, Z1;
        reg Z0, Z1;
        integer STATE, NEXTSTATE;

        always @( posedge CLOCK or posedge RESET)
                if (RESET == 1'b1)
                        STATE = 1;
                else
                        begin
                                STATE = NEXTSTATE;
                                case (STATE)
                                        1: Z0 = 1'b1;
                                        2: Z0 = 1'b0;
                                endcase
                        end
        always @( A or B or STATE)
        begin
                Z1 = 1'b0; // default value
                case (STATE)
                        1: begin
                                Z1 = 1'b1;
                                if (A)
                                        NEXTSTATE = 2;
                                else
                                        NEXTSTATE = 3;
                        end
                        2: NEXTSTATE = 4;
                        3: NEXTSTATE = 5;
                        4: if (A | B)
                                NEXTSTATE = 1;
                          else
                                NEXTSTATE = 5;
                        5: NEXTSTATE = 3;
                        default: NEXTSTATE = 1;
                endcase
        end
endmodule
```

Figure 58: *Verilog description of a Moore FSM*

Figure 59 gives the Verilog description of the FSM described in figure 56. In this description, the inputs A and B are latched.

```
module FSM ( RESET, CLOCK, A, B, Z0, Z1);
    input RESET, CLOCK, A, B;
    output Z0, Z1;
    reg Z0;
    integer VALUE;
    reg A_FF, B_FF;
    always @( posedge CLOCK or posedge RESET)
    begin
        if (RESET == 1'b1)
            VALUE = 1;
        else
        begin
            case (VALUE)
                1: begin
                    Z0 = 1'b1;
                    if (A_FF == 1'b1)
                        VALUE = 2;
                    else
                        VALUE = 3;
                end
                2: begin
                    Z0 = 1'b0;
                    VALUE = 4;
                end
                3: VALUE = 5;
                4: if (A_FF | B_FF)
                    VALUE = 1;
                  else
                    VALUE = 5;
                5: VALUE = 3;
                default: VALUE = 1;
            endcase
            A_FF = A;       // A_FF is latched
            B_FF = B;       // B_FF is latched
        end
    end
    assign Z1 = (VALUE == 1); // Z1 is not latched
endmodule
```

Figure 59: *Verilog description of a Moore FSM with latched inputs*

6.2. State assignments

When using on of the templates which are 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 are available.

6.2.1. 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.

6.2.2. User controlled state assignment

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

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 Verilog 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.

6.3. Symbolic FSM identification

This original feature addresses the case where Verilog 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 60.

```
module FSM (CLOCK, A, B, Z1);
	input CLOCK, A, B ;
	output Z1 ;
	reg Z1;
	integer VALUE, NEXTVALUE;

	always @(posedge CLOCK)
		VALUE = NEXTVALUE;

	always @(A or B or VALUE)
	begin
		Z1 = 1'b0; // default value
		if (VALUE == 1)
		begin
			Z1 = 1'b1;
			if (A == 1'b1)
				NEXTVALUE = 2;
			else
				NEXTVALUE = 3;
		end
		else if (VALUE == 2)
			NEXTVALUE = 4;
		else if (VALUE == 3)
			NEXTVALUE = 5;
		else if (VALUE == 4)
			if ((A == 1'b1) || (B == 1'b1))
				NEXTVALUE = 1;
			else
				NEXTVALUE = 5;
		else if (VALUE == 5)
			NEXTVALUE = 3;
		else NEXTVALUE = 1;
endmodule
```

Figure 60: *Verilog 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.

6.4. 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.

6.4.1. 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 Verilog 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.

6.4.2. Embedded FSMs

In this case the user does not want to declare separately the FSMs. Commonly, he will use a always block style 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 Verilog 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.

c) Specific options can be given for each FSM. For this purpose, a synthesis directive file is specified. This file defines the module name of the FSMs and a dedicated encoding, 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 for FSM3, the directive file given in figure 61 can be used.

directive -module FSM1 **-c** OPT **-crit** AREA
directive -module FSM2 **-c** ONE **-crit** SPEED **-power** 2
directive -module FSM3 **-c** GRAY

Figure 61: *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 -module FSM3 -c one”, giving the specific options for the FSM3 synthesis.

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

7. Communicating Finite State Machines Synthesis

7.1. 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 module connecting several FSMs described by always blocks. The second one connects in a structural mode the different FSMs. Each FSM can then be described using any accepted FSM description style.

7.2. Communicating FSMs

7.2.1. Concurrent communicating FSMs

Figure 62 shows two communicating FSMs: FSM1 (figure 62.a) and FSM2 (figure 62.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 62, 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.

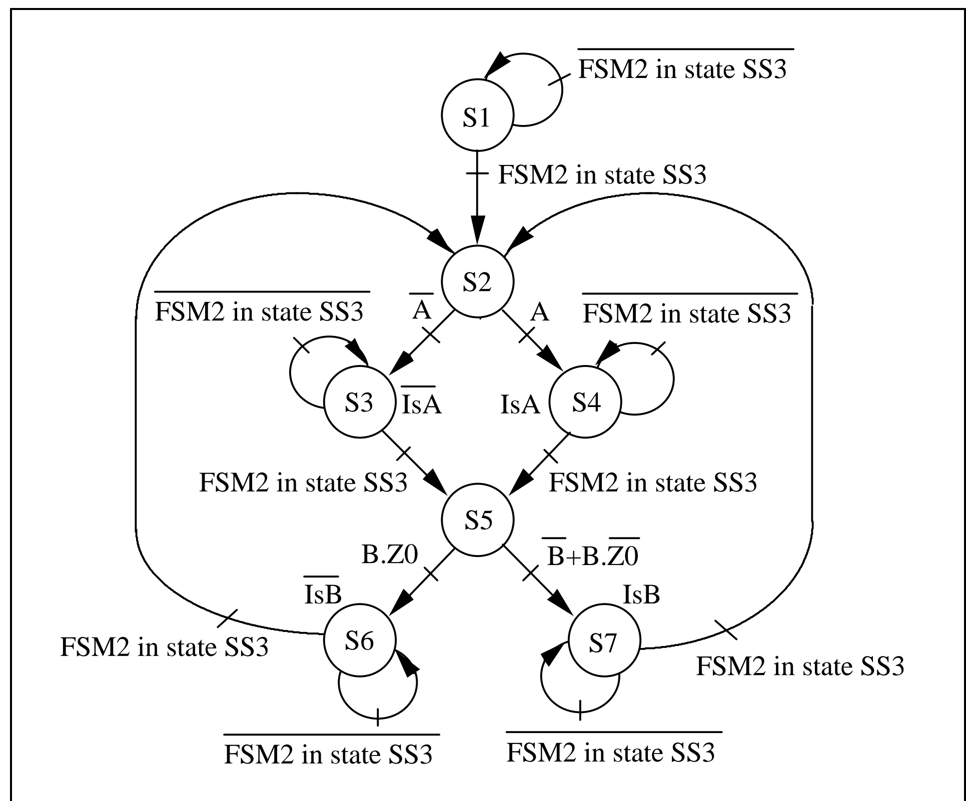

Figure 62.a: *FSMs communicating by states (FSM1)*

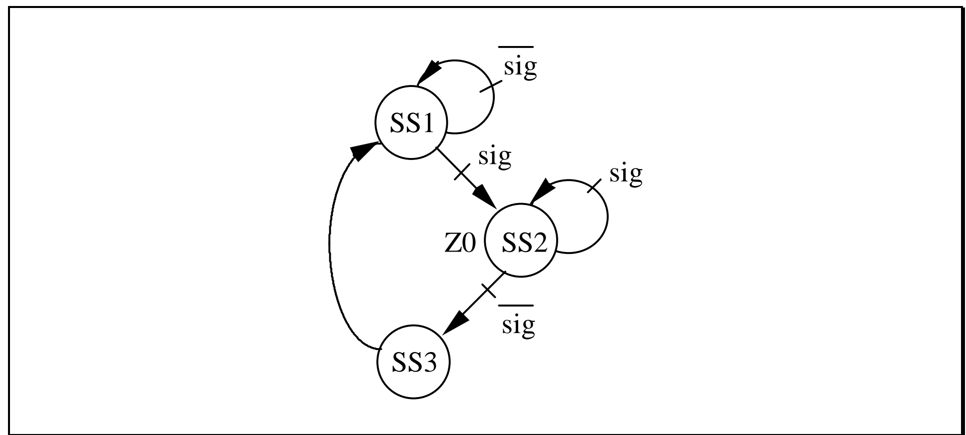

Figure 62.b: *FSMs communicating by states (FSM2)*

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 63 gives the same communicating FSMs than in figure 62, but in figure 63 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”.

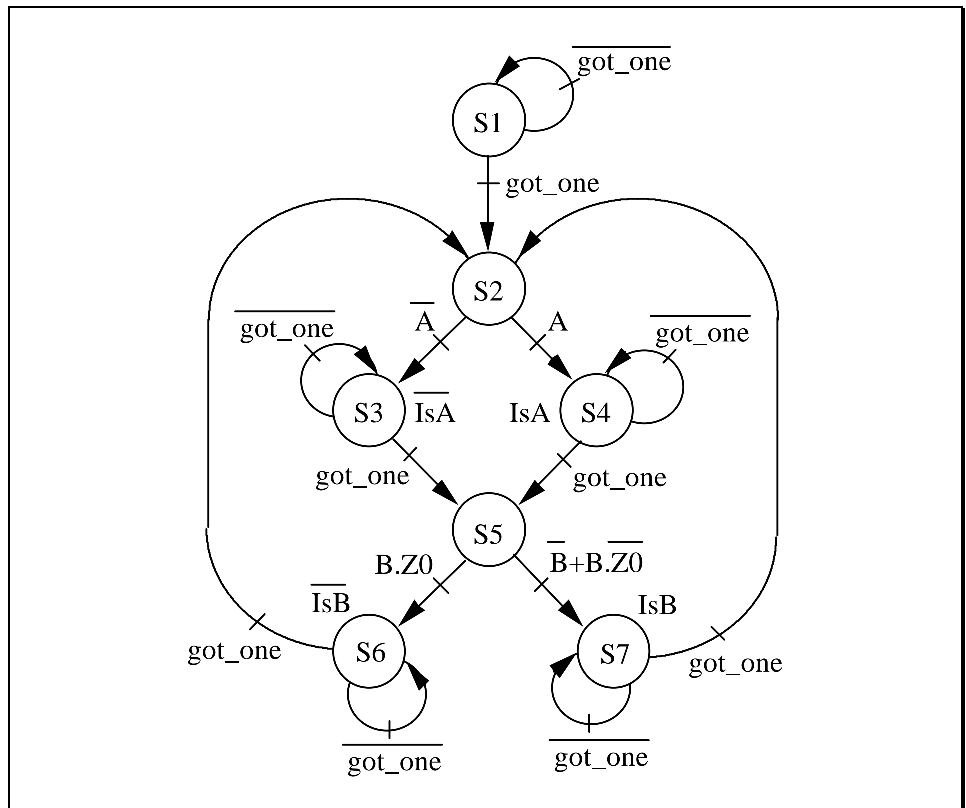

Figure 63.a: *FSMs communicating by output signals (FSM1)*

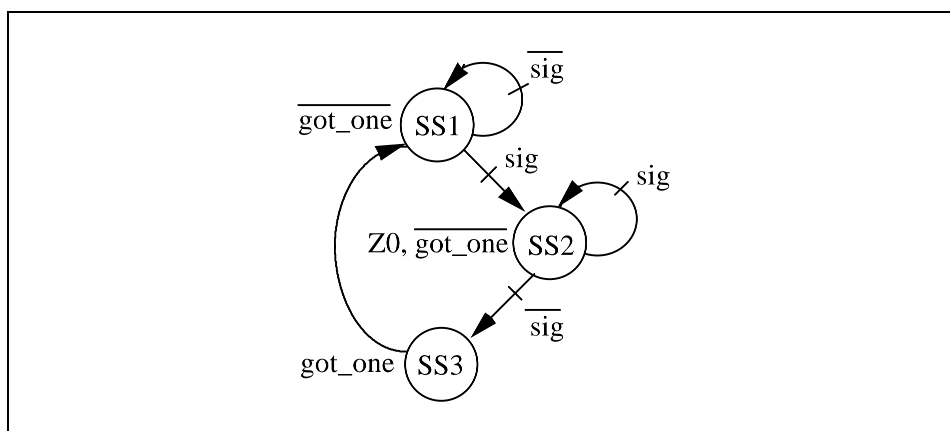

Figure 63.b: *FSMs communicating by output signals (FSM2)*

7.2.2. 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 labelled 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 labelled 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 64 shows two communicating hierarchical FSMs. The master FSM in figure 64.a has three input signals ("A", "B", "got_one") and three output signals ("IsA", "IsB", "get_sig"). The slave FSM in figure 64.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 waiting states. For the master FSM, the waiting states are ("S1", "S3", "S4", "S6", "S7") and the slave FSM has one waiting state ("SS4").

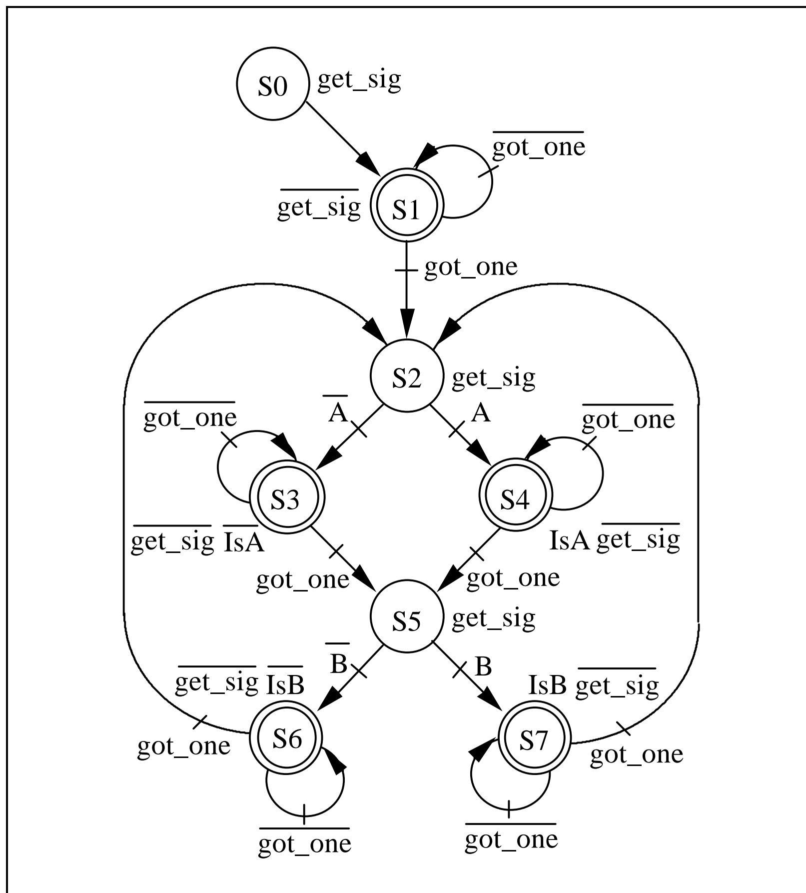

Figure 64.a: *Master FSM*

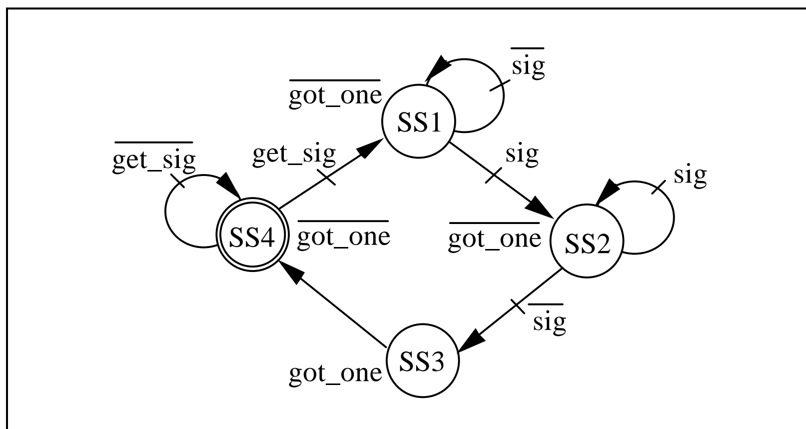

Figure 64.b: *Slave FSM*

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.

7.3. Always blocks based description

7.3.1. Modeling

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

Figure 65 shows such a description for the example of figure 64. The master FSM is described by the first always block and the slave FSM is described the second one. The internal signals "VALUE1" and "VALUE2" represent respectively the state register of the master FSM and the slave FSM. The communication between the two always block is described by using the state registers modeled by the "VALUE1" and VALUE2" signals. In the "MASTER_FSM" always block, tests are made on the value of "VALUE2" ("if (VALUE2 == 3) ...") and in the "SLAVE_FSM" always block a test is made on the values of "VALUE1" ("if ((VALUE1 == 0) || (VALUE1 == 2) || (VALUE1 == 5)) ...").

```
module FSM_HIER ( RESET, CLK, A, B, SIG, IS_A, IS_B);
        input RESET, CLK, A, B, SIG ;
        output IS_A, IS_B ;
        reg IS_A, IS_B ;
        integer VALUE1, VALUE2;

        // MASTER_FSM
        always @(posedge RESET or posedge CLK)
                if (RESET == 1'b1)
                        VALUE1 = 0;
                else
                        case (VALUE1)
                                0: VALUE1 = 1;
                                1: if (VALUE2 == 3)
                                        VALUE1 = 2;
                                  else
                                        VALUE1 = 1;
                                2: if (A == 1'b0)
                                        VALUE1 = 3;
                                  else
                                        VALUE1 = 4;
                                3: begin
                                        if (VALUE2 == 3)
                                                VALUE1 = 5;
                                        else
                                                VALUE1 = 3;
                                        IS_A = 1'b0;
                                end
                                4: begin
                                        if (VALUE2 == 3)
                                                VALUE1 = 5;
                                        else
                                                VALUE1 = 4;
                                        IS_A = 1'b1;
                                end
                                5: if (B == 1'b0)
                                        VALUE1 = 6;
                                  else
                                        VALUE1 = 7;
```

```
                        6: begin
                            if (VALUE2 == 3)
                                VALUE1 = 2;
                            else
                                VALUE1 = 6;
                            IS_B = 1'b0;
                        end
                        7: begin
                            if (VALUE2 == 3)
                                VALUE1 = 2;
                            else
                                VALUE1 = 7;
                            IS_B = 1'b1;
                        end
                    endcase

    // SLAVE_FSM
    always @(posedge RESET or posedge CLK)
            if (RESET == 1'b1)
                    VALUE2 = 4;
            else
                    case (VALUE2)
                        1: if (SIG == 1'b1)
                                VALUE2 = 2;
                           else
                                VALUE2 = 1;
                        2: if (SIG == 1'b0)
                                VALUE2 = 3;
                           else
                                VALUE2 = 2;
                        3: VALUE2 = 4;
                        4: if ((VALUE1 == 0) ||
                               (VALUE1 == 2) ||
                                (VALUE1 == 5))
                                VALUE2 = 1;
                           else
                                VALUE2 = 4;
                    endcase
endmodule
```

Figure 65: *Verilog description of 2 hierarchical FSMs communicating by states*

7.3.2. Synthesis

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 Verilog 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.

7.4. Structural composition of FSMs

7.4.1. 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 § V.2.1. Figure 66 gives the Verilog description of the example of figure 64.

```
module MASTER_FSM ( A, B, GOT_ONE, RESET, CLK,
                              IS_A, IS_B, GET_SIG);
        input A, B, GOT_ONE, RESET, CLK;
        output IS_A, IS_B, GET_SIG;
        reg IS_A, IS_B;
        integer STATE;

        assign GET_SIG = (STATE == 0) || (STATE == 2) ||
                                   (STATE == 5);
        always @(posedge RESET or posedge CLK)
                if (RESET == 1'b1)
                        STATE = 0;
                else
                        case (STATE) is
                                0: STATE = 1;
                                1: if (GOT_ONE == 1'b1)
                                        STATE = 2;
                                  else
                                        STATE = 1;
                                2: if (A == 1'b0)
                                        STATE = 3;
                                  else
                                        STATE = 4;
                                3: begin
                                        if (GOT_ONE == 1'b1)
                                                STATE = 5;
                                        else
                                                STATE = 3;
                                        IS_A = 1'b0;
                                end
                                4: begin
                                        if (GOT_ONE == 1'b1)
                                                STATE = 5;
                                        else
                                                STATE = 4;
                                        IS_A = 1'b1;
                                end
                                5: if (B == 1'b0)
                                        STATE = 6;
                                  else
                                        STATE = 7;
                                6: begin
                                        if (GOT_ONE == 1'b1)
                                                STATE = 2;
                                        else
                                                STATE = 6;
```

```
                              IS_B = 1'b0;
                         end
                         7: begin
                              if (GOT_ONE == 1'b1)
                                   STATE = 2;
                              else
                                   STATE = 7;
                              IS_B = 1'b1;
                         end
                    endcase
endmodule
```

Figure 66.a: *Verilog description of the master FSM*

```
module SLAVE_FSM ( CLK, RESET, SIG,
                    GET_SIG, GOT_ONE);
     input CLK, RESET, SIG, GET_SIG;
     output GOT_ONE;
     integer STATE;

     assign GOT_ONE = (STATE == 3);
     always @(posedge RESET or posedge CLK)
          if (RESET == 1'b1)
               STATE = 4;
          else
               case (STATE)
                    1: if (SIG == 1'b1)
                              STATE = 2;
                       else
                              STATE = 1;
                    2: if (SIG == 1'b0)
                              STATE = 3;
                       else
                              STATE = 2;
                    3: STATE = 4;
                    4: if (GET_SIG == 1'b1)
                              STATE = 1;
                       else
                              STATE = 4;
               endcase
endmodule
```

Figure 66.b: *Verilog description of the slave FSM*

```
module FSM_HIER ( RESET, CLK, A, B, SIG, IS_A, IS_B);
     input RESET, CLK, A, B, SIG;
     output IS_A, IS_B;
     wire GET_SIG;
     wire GOT_ONE;

     MASTER_FSM MASTER ( A, B, GOT_ONE, RESET,
                              CLK, IS_A, IS_B, GET_SIG);
     SLAVE_FSM  SLAVE ( CLK, RESET, SIG, GET_SIG,
                              GOT_ONE);
endmodule
```

Figure 66.c: *Verilog description of the interconnection of the 2 FSMs*

7.4.2. Synthesis

For each FSM, all the synthesis options are available: the state assignment, 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 Verilog 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.

c) Specific options can be given for each FSM. For this purpose, a synthesis directive file is specified. This file defines the module 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 module named FSM1, FSM2 and FSM3. 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 for FSM3, the directive file given in figure 67 can be used.

```
directive -module FSM1 -c OPT -crit AREA
directive -module FSM2 -c ONE -crit SPEED -power 2
directive -module FSM3 -c GRAY
```

Figure 67: *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 -module FSM3 -c one”, giving the specific options for the FSM3 synthesis.

8. Verilog Subset for synthesis

8.1. Limited Verilog Language Constructs

This section describes the Verilog constructs which are restricted by PLS.

8.1.1. *always* statement

PLS Verilog supports two kinds of always block, one using only edge-triggered events (a maximum of three) and the other using only value-change events.

8.1.2. *for* statement

PLS Verilog only supports *for* loop which are bounded by static variables.

8.1.3. *repeat* statement

PLS Verilog only supports *repeat* loop using a constant value.

8.2. Ignored Verilog Language Constructs

This section describes the Verilog constructs which are parsed and ignored by PLS.

8.2.1. Ignored Statements

- *specify* statement.
- *initial* statement.
- system function call.
- system task enable.

8.2.2. Ignored Miscellanous Constructs

- delay specifications.
- *scalared* and *vectored* declarations.
- *small*, *large* and *medium* charge storage strengths.
- *weak0*, *weak1*, *highz0*, *highz1*, *pull0*, *pull1* driving strengths.

8.3. Unsupported Verilog Language Constructs

This section describes the Verilog constructs which are unsupported by PLS.

8.3.1. Unsupported Definitions and Declarations

- *primitive* definition.
- *macromodule* definition.
- *time* declaration.
- *event* declaration.
- *triand*, *trior*, *tri0*, *tri1*, and *trireg* net types.

8.3.2. Unsupported Statements

- *deassign* statement.
- *defparam* statement.
- *disable* statement.
- *event* control.
- *force* statement.
- *release* statement.
- *fork* statement.
- *forever* statement.
- *while* statement.

8.3.3. Unsupported Operators

- case equality and inequality operators (=== and !==).
- division and modulus operators (/ and %) when the second operand is ot a power of two.

8.3.4. Unsupported Gate-Level constructs

- *nmos*, *pmos*, *rnmos* and *rpmos* MOS Switches.
- *tran*, *tranif0*, *tranif1*, *rtran*, *rtranif0* and *rtranif1* Bidirectional Pass Switches.
- *cmos* and *rcmos* CMOS Gates.
- *pullup* and *pulldown* Sources.