APPLICATION NOTE

Mentor Synthesis Group

Synthesizing FPGAs with Verilog 2001

Precision Synthesis V2003a

Mentor Graphics®

Table of Contents

TABLE OF CONTENTS **2**

INTRODUCTION **3**

SUPPORTED VERILOG 2001 CONSTRUCTS **3**

DETAILED DESCRIPTION **3**

- COMBINATIONAL LOGIC SENSITIVITY TOKEN 3
- COMBINED PORT / DATA TYPE OPERATOR 3
- POWER OPERATOR 4
- SIGNED ARITHMETIC EXTENSIONS 4
- COMMA-SEPARATED SENSITIVITY LIST 5
- ANSI-STYLE PORT LISTS 5
- AUTOMATIC WIDTH EXTENSION PAST 32 BITS 5
- REGISTER DECLARATION WITH INITIALIZATION 6
- IN-LINE PARAMETER PASSING 6
- FILE AND LINE COMPILER DIRECTIVES 6
- ATTRIBUTES 7

REFERENCES **7**

Introduction

The Verilog-2001 Standard was approved by the IEEE in June of 2000 and includes enhancements to the previously existing Verilog standard in the following areas:

- *Enhance the Verilog language to help with today's deep-submicron and intellectual property modeling issues*
- *Ensure that all enhancements were both useful and practical, and that simulator and synthesis vendors would implement Verilog-2001 in their products*
- *Correct any ambiguities in the Verilog-1995 standard (IEEE 1364-1995)*

Verilog 2001 expands the construct set of Verilog – 1995 but does not replace or change the behavior of previously implemented constructs. Users may seamlessly mix new and previously existing constructs.

Supported Verilog 2001 Constructs

Precision Synthesis V2003a offers support for the following Verilog-2001 constructs

- Signed arithmetic extensions
- Comma-separated sensitivity list
- Combinational logic sensitivity token
- Combined port / data type declarations
- ANSI-style port lists
- Power Operator
- Automatic width extension past 32 bits
- Register declaration with initialization

Detailed Description

Combinational logic sensitivity token

To properly model combinational logic using a Verilog always procedure, the sensitivity list must include all input signals used by that block of logic. In large, complex blocks of combinational logic, it is easy to inadvertently omit an input from the sensitivity list, which can lead to simulation and synthesis mismatches.
Verilog-2000 adds a new wild card token, **@***, which represents a combinational logic sensitivity list. The @* token indicates that the simulator or synthesis tool should automatically be sensitive to any values used by the procedure in decisions or in expressions on the right-hand side of assignment statements. In the following example, the @* token will cause the procedure to automatically be sensitive to changes on sel, a or b.

```
always @* //combinational logic sensitivity
if (sel)
y = a;
else
y = b;
```

Combined port / data type operator

Verilog requires that signals connected to the input or outputs of a module have two declarations: the direction of the port, and the data type of the signal. In Verilog-1995, these two declarations had to be done as two separate statements. Verilog-2000 adds a simpler syntax, by combining the declarations into one statement.

```
module mux8 (y, a, b, en);
output reg [7:0] y;
input wire [7:0] a, b;
input wire en;
```

Power operator

Verilog-2001 adds a power operator, represented by an ** token. This operator preforms similar functionality as the C pow() function. It will return a real number if either operand is a real value, and an integer value if both operands are integer values. One practical application of the power operator is to calculate values such as 2^n. For

```
example:
always @(posedge clock)
result = base ** exponent;
```

Signed Arithmetic Extensions

For integer math operations, Verilog uses the data types of the operands to determine if signed or unsigned arithmetic should be performed. If either operand is unsigned, unsigned operations are performed. To perform signed arithmetic, both operands must be signed. In Verilog-1995, the integer data type is signed, and the reg and net data types are unsigned. A limitation in Verilog- 1995 is that the integer data type has a fixed vector size, which is 32-bits in most Verilog simulators. Thus, signed integer math in Verilog-1995 is limited to 32-bit vectors. The Verilog- 2000 standard adds five enhancements to provide greater signed arithmetic capability:

- Reg and net data types can be declared as signed
- Function return values can be declared as signed
- Integer numbers in any radix can be declared as signed
- Operands can be converted from unsigned to signed
- Arithmetic shift operators have been added

The Verilog-1995 standard has a reserved keyword, **signed**, but this keyword was not used in Verilog-1995. Verilog-2000 uses this keyword to allow reg data types, net data types, ports, and functions to be declared as signed types. Some example declarations are:

```
reg signed [63:0] data;
wire signed [7:0] vector;
input signed [31:0] a;
function signed [128:0] alu;
```

In Verilog-1995, a literal integer number with no radix specified is considered a signed value, but a literal integer with a radix specified is considered an unsigned value. Verilog-2000 adds an additional specifier, the letter '**s**', which can be combined with the radix specifier, to indicate that the literal number is a signed value.

```
16'hC501 //an unsigned 16-bit hex value
16'shC501 //a signed 16-bit hex value
```

In addition to being able to declare signed data types and values, Verilog-2000 adds two new system functions, **$signed** and **$unsigned**. These system functions are used to convert an unsigned value to signed, or vice-versa.

```
reg [63:0] a; //unsigned data type
always @(a) begin
result1 = a / 2; //unsigned arithmetic
result2 = $signed(a) / 2; //signed arithmetic
end
```

One more signed arithmetic enhancement in Verilog-2000 is arithmetic shift operators, represented by >>> and <<< tokens. An arithmetic right-shift operation maintains the sign of a value, by filling with the sign-bit value as it shifts. For example, if the 8-bit variable D contained 8'b10100011, a logical right shift and an arithmetic right shift by 3 bits would yield the following:

```
D >> 3 //logical shift yields 8'b00010100
D >>> 3 //arithmetic shift yields 8'b11110100
```

Comma-separated sensitivity list

Verilog-2000 adds a second way to list signals in a sensitivity list, by separating the signal names with commas instead of the **or** keyword. The following two sensitivity lists are functionally identical:

```
always @(a or b or c or d or sel)
always @(a, b, c, d, sel)
```

The new, comma-separated sensitivity list does not add any new functionality. It does, however, make Verilog syntax more intuitive, and more consistent with other signal lists in Verilog.

ANSI-style port lists

Verilog-1995 uses the older Kernighan and Ritchie C language syntax to declare module ports, where the order of the ports is defined within parentheses, and the declarations of the ports are listed after the parentheses. Verilog-1995 tasks and functions omit the parentheses list, and use the order of the input and output declarations to define the input/output order. Verilog-2000 updates the syntax for declaring inputs and outputs of modules, tasks, and functions to be more like the ANSI C language. That is, the declarations can be contained in the parentheses that show the order of inputs and outputs.

```
module mux8 ( output reg [7:0] y,
input wire [7:0] a,
input wire [7:0] b,
input wire en );
function [63:0] alu (
input [63:0] a,
input [63:0] b,
input [7:0] opcode );
```

Automatic Width Extension past 32 bits

With Verilog-1995, assigning an unsized high-impedance value (e.g.: ‘bz) to a bus that is greater than 32 bits would only set the lower 32 bits to high-impedance. The upper bits would be set to
0. To set the entire bus to high-impedance requires explicitly specifying the number of high impedance bits. For example:

```
Verilog-1995:
parameter WIDTH = 64;
reg [WIDTH-1:0] data;
data = 'bz; //fills with 'h00000000zzzzzzzz
data = 64'bz; //fills with 'hzzzzzzzzzzzzzzzz
```

The fill rules in Verilog-1995 make it difficult to write models that are easily scaled to new vector sizes. Redefinable parameters can be used to scale vector widths, but the Verilog source code must still be modified to alter the literal value widths used in assignment statements.
Verilog-2000 changes the rule for assignment expansion so that an unsized value of Z or X will automatically expand to fill the full width of the vector on the left-hand side of the assignment.

```
Verilog-2000:
parameter WIDTH = 64;
reg [WIDTH-1:0] data;
data = 'bz; //fills with 'hzzzzzzzzzzzzzzzz
```

This Verilog-2001 enhancement is not backward compatible with Verilog-1995. However, the IEEE standards group felt the Verilog-1995 behavior was a bug in the standard that needed to be corrected. It is expected that all existing models with greater than 32-bit busses have avoided this bug by explicitly specifying the vector sizes. Therefore, there should not be any compatibility problems with existing models.

Register declaration with initialization

Verilog-2000 adds the ability to initialize variables at the time they are declared, instead of requiring a separate initial procedure to initialize variables. The initial value assigned to the variable will take place within simulation time zero, just as if the value had been assigned within an initial procedure.

```
Verilog-1995:
reg clock;
initial
clk = 0;
Verilog-2000:
reg clock = 0;
```

In-Line Parameter Passing

Verilog-1995 has two methods of redefining parameters within a module instance: *explicit redefinition* using defparam statements, and *in-line implicit redefinition* using the # token as part of the module instance. The latter method is more concise, but because it redefines parameter by their declaration position, it is error-prone and is not self-documenting. The following example illustrates the two Verilog-1995 methods for parameter redefinition.

```
module ram (...);
   parameter WIDTH = 8;
   parameter SIZE = 256;
...
endmodule

module my_chip (...);
   ...
   //Explicit parameter redefinition by name
   RAM ram1 (...);
   defparam ram1.SIZE = 1023;
   //Implicit parameter redefintion by position
   RAM #(8,1023) ram2 (...);
endmodule
```

Verilog-2001 adds a third method to redefine parameters, ***in-line explicit redefinition***. This new method allows inline parameter values to be listed in any order, and document the parameters being redefined.

```
   //In-line explicit parameter redefintion
RAM #(.SIZE(1023)) ram2 (...);
```

File and Line Compiler Directives

Verilog tools need to keep track of the line number and the file name of Verilog source code. This information can be used for error messages, and can be accessed by the Verilog PLI. If Verilog source is pre-processed by some other tool, however, the line and file information of the original source code can be lost. Verilog-2001 adds a **`line** compiler directive, which can be used to specify the original source code line number and file name. This allows the location in an original file to be maintained if another process modifies the source, such as by adding or removing lines of source text.

Attributes

The Verilog language was originally created as a hardware description language for digital simulation. As tools other than simulation have adopted Verilog as a source input, there has been a need for these tools to be able add tool specific information to the Verilog language. In Verilog- 1995, there was no mechanism for adding tool-specific information, which led to non-standard methods, such as hiding synthesis commands in Verilog comments. Verilog-2001 adds a mechanism for specifying properties about objects, statements and groups of statements in the HDL source. These properties are referred to as ***attributes***. Attributes are used by Precision synthesis to apply optimization directives and thus control the results. An attribute is contained within the tokens **(*** and ***)**. Attributes can be associated with all instances of an object, or with a specific instance of an object. Attributes can be assigned values, including strings, and attribute values can be re-defined for each instance of an object. Verilog-2001 does not define any standard attributes. A complete list of supported attributes is provided in the Precision RTL Synthesis Reference Manual. An example is provided below:

```
(* parallel case *) case (1'b1) //1-hot FSM
state[0]: ...
state[1]: ...
state[2]: ...
endcase
```

References

1. *"Getting the Most out of the new Verilog 2000 Standard"*. Stuart Sutherland, Sutherland HDL, Inc, Don Mills, LCDM Engineering
2. *" The IEEE Verilog 1364-2001 Standard – What's New, and why you need it"*. Stuart Sutherland, Sutherland HDL, Inc. Presentation presented at the HDLCON-2000 Conference, March 10, 2000, San Jose, California.
3. *"IEEE Std p1364-2001, IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language"*. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017-2394, USE