System Verilog through Examples

Author Name Here

August 31, 2013

Contents

Preface

My interest in `System Verilog` started during late 2004, when the language was being defined. Back then the only reference available was the LRM itself, as primitive implementations of the language started to surface in commercial tools. As I learnt the language, I started understanding the language from inside-out which made a big difference to the code I was writing. Though many texts on the subject are effective in conveying concepts of the language as explaination of limited examples, the focus of this book would be to teach nuances of `System Verilog` with utmost examples as possible. Such a method would aid even the novice of engineers to pick up the language without much difficuilty.

Acknowledgements

Introduction

System Verilogas a language extends the **Verilog**language and is a superset. Hence **Verilog**language is in its entirety should be supported by **System Verilog**unless explicitly stated. The focus of the book would be to teach **System Verilog**and the assumption is that the reader is adequately knowledgeable in **Verilog**. There exists adequate texts for **Verilog**.

Chapter 1

Literals

Literals are fixed values in source code and normally does not include spaces. **System Verilog**language adds few literals including time literal, array literal, structure literal, and enhancements to string literal.

1.1 Integer and Logic

SV introduces unsized single bit state values '0, '1, 'z, 'Z, 'x, 'X. Important feature to notice is that there is no format specifier in the literal and it is devoid of width specification as well. When assigned to a multibit data-type, they automatically size to the vector's size. The feature aids portable code.

Listing 1.1: Unsized literal usage

```
1 logic [47:0] val48;

3 initial begin
    val48 = 'x;
5   val48 = 48'hxxxxxxxxxxxx;
    val48 = 48'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
7 end
```

Listing 1.1 shows three different ways of assigning a 48bit logic vector with unknown state value 'x'. It is very obvious that the unsized single bit format is more readable and is not error prone.

Listing 1.2: Array Literal examples

```
1 int n[1:2][1:3] = '{'{0,1,2},'{3{4}}};
  int n[1:2][1:6] = '{2{'{3{4, 5}}}};
```

```
Line 1: becomes '{'{0,1,2}, '{4, 4, 4}}
Line 2: becomes '{'{4,5,4,5,4,5},'{4,5,4,5,4,5}}
```

Listing 1.3: Array Literal with default value

```
  int b[0:3] = '{1:1, default:0};
2 int b[0:3] = '{1:1, int:0};
```

```
Line 1, 2: both becomes [0, 1, 0, 0]
```

1.2 Time

SV introduces accurate time specifications. Earlier method of specification in **Verilog**was prone to errors due to compiler switches such as **-override_timescale** and the ordering of **`timescale** directives. Accurate time specification literal is a real-value followed by time unit, without a space. Time unit is one of `fs ps ns us ms s`. Examples of valid specifications are `0.1ns, 1s, 3.141us`.

1.3 Array

Array literals are syntactically similar to C initializers, but with the replicate operator ({{ }}) allowed. The nesting of braces must follow the number of dimensions, unlike in C. However, replicate operators can be nested.

Array literals can also use their index or type as a key and use a default key value. Both lines of Listing 1.3 are equivalent.

Listing 1.4: Structure Literals

```
  struct {int a; shortreal b;} ab, ab2[2];
2 ab = '{0, 0.0};
  ab2 = '{'{1, 1.1}, '{2, 2.2}};
4 ab2 = '{1, 1.1, 2, 2.2};
  ab = '{int:1, shortreal:1.1};
6 ab = '{default:0};

8 struct {int a, b[4];} ab [2][3] =
                  '{2{'{3{a,'{2{b,c}}}}}};
```

```
Line 4: Error: need to brace array elements
Line 5: becomes '{1, 1.1}
Line 6: becomes '{0, 0.0}
Line 8:
 '{2{ '{3{ a, '{b, c, b, c} }} }}
 '{2{ '{a, '{b,c,b,c}}, '{a, '{b,c,b,c}},
                        '{a, '{b,c,b,c}} }}
 '{ '{ '{a, '{b,c,b,c}}, '{a, '{b,c,b,c}},
                        '{a, '{b,c,b,c}} },
    '{ '{a, '{b,c,b,c}}, '{a, '{b,c,b,c}},
                        '{a, '{b,c,b,c}} } }
```

Listing 1.5: String Literal Examples

```
1 byte c3 [0:7] = "Hello World\n";
  byte c4 [7:0] = "Hello World\n";
3 bit [10:0] a = "\x41";
  bit [1:4][7:0] h = "hello";
```

```
Line 1: c3[0] = "H", c3[7] = "o"
Line 2: c4[0] = "H", c4[7] = "H"
Line 3: assigns to a 'b000_0100_0001
Line 4: assigns to h "ello"
```

1.4 Structure

A structure literal must have a type, which may be either explicitly indicated with a prefix or implicitly indicated by an assignment-like context. Nested braces should reflect the structure as in Listing 1.4. Unordered values as in line 4, would be an error.

1.5 String

System Verilogadds few special characters to string literals such as `\a, \v, \f, \x`. These special characters are intrepretted inside a string literal enclosed within “”. They have the same definition as C quoted strings such as “bell”, “vertical feed”, “form feed” and “hex number”. `\x` should be followed by hexadecimal value corresponding to the desired character.

A string literal can be assigned to an unpacked array of bytes. If the size

differs, it is left justified. In Listing 1.5, `c3[0]` contains character 'H' and `c3[7]` contains ASCII value of character 'o'.

Chapter 2

Data Types

System Verilogadds multiple integer data types, string, chandle, and class data types and enhances the Verilog event type.

2.1 Integer data types

Integer data types could be either 2-state or 4-state types. Complete list of integer data types are listed in Table 2.1. All integer data types are integral as they can be converted to or assigned from a one-dimensional bit-vector. Other data types such as **packedarray**, **packedstruct**, **packedunion**, **enum**and **time**do also exhibit the integral property.

Every bit of a 2-state data-type could hold the value '0' or '1', whereous each bit of a 4-state data-type could hold values '0', '1', 'X' or 'Z'.

All integer data types could be either **signed**or **unsigned**. It becomes important to understand sign-ness to understand the boundaries of each data-type. When an integer data type is declared to be **unsigned**, it would have range from 0 to $2^n - 1$, where n is the number of bits in the type.

Table 2.1: Integer data types

Data type	2/4 state	Default Sign-ness	lvalue	rvalue
shortint	2-state	signed	-2^{15}	$2^{15}-1$
int	2-state	signed	-2^{31}	$2^{31}-1$
longint	2-state	signed	-2^{63}	$2^{63}-1$
byte	2-state	signed	-2^{7}	$2^{7}-1$
bit	2-state	unsigned	user-defined vector size	
logic	4-state	unsigned	user-defined vector size	
reg	4-state	unsigned	user-defined vector size	
integer	4-state	signed	-2^{31}	$2^{31}-1$
time	4-state	unsigned	0	$2^{64}-1$

2.2 Floating-point data types

Verilogalready defines **real**, which is equivalent to **C**'s **double** data-type. **System Verilog**defines **shortreal**data-type, that is equivalent to **C**'s **float**. In general implementations of **C** compilers, **float** has 7 decimal precision, whereous **double** has a precision of 15.

2.3 void

The **void**data type represents nonexistent data. This type can be specified as the return type of functions to indicate no return value. This type can also be used for members of tagged unions.

2.4 chandle

The **chandle**data type represents storage for pointers. Its only application is to store DPI returned pointers. **System Verilog**can neither create nor destroy **C** pointers. **chandle**have initial value as **null**. **chandle**could be used in boolean equations to test for equality ==, !=, ===, !==. In boolean context, a **chandle**evaluates to 0 if **null**or to 1 otherwise. It should be obvious that **chandle**could be used as arguments to **function**or **task**but not as ports.

2.5 string

System Verilog's **string**is a built-in data-type. It is dynamic as its length may vary during simulation. It could be assigned from a string literal. **string**data-type allows indexing single or a group of characters. Unlike **C**, **System Verilog**'s **string**doesn't contain the special character '\0'.

Special string operators exists for **string**data type. These operators are invoked only if either of operands is a **string**data type. Otherwise, when both operands are string literals, the behaviour would be comparable with Verilog, where integer operators are invoked in most cases. If the result of the operator is used in another expression involving string types, it is implicitly converted to the string type.

Equality operator ==
: Checks whether the two strings are equal. Result is 1 if they are equal and 0 if they are not.

Inequality operator !=
: Logical negation of Equality operator.

Comparison operators $<, >, <=, >=$
: Relational operators return 1 if the corresponding condition is true using the lexicographical ordering of the two strings under consideration.

Concatenation { }
: The string concatenation operator evaluates each constituent expressions to string before concatenating it. The result is of type string.

Replication {M{ }}**
: Multiplier must be of integral type and can be non-constant. If multiplier is non-constant or if the expression is of type string, the result is a string containing M concatenated copies of expression. If expression is a literal and the multiplier is constant, the expression behaves like numeric replication in Verilog.

Indexing [ ]

Listing 2.1: Structure Literals

```
  string s1;
2 string s2 =           ;
  string s3 =           ;
4 string s4 =        ;
  bit [11:0] b = 12'ha41;
6 string s2 = string'(b);

8 typedef reg [15:0] r_t;
  r_t r;
10 integer i = 1;
  string s5 =   ;
12 string s6 = {    , s5};

14 r = r_t'(s6);
  s5 = string'(r);
16 s5 =     ;
  s5 = {5{    }};
18 s6 = {i{    }};
  r = {i{    }};
20 s6 = {i{s5}};
  s6 = {s6,s5};
22 s6 = {    ,s5};
  r = {   ,  };
24 b = {   ,  };
  s2[0] =    ;
26 s2[0] =        ;
  s2[1] =     ;
```

```
Line 1: initialised to ""
Line 2: OK
Line 3: '\0' is removed from string
Line 4: OK
Line 5: OK
Line 6: converted to 16'h0a41
  Note the typecasting operator

Line 9: reg init with 'x
Line 11: explicit string init to ""
Line 12: OK value "Hi"
Line 14: OK value 16'h4869
Line 15: OK value "Hi"
Line 16: OK value "Hi"
Line 17: OK value "HiHiHiHiHi"
Line 18: OK (non constant replication)
Line 19: Invalid (non constant replication)
Line 20: OK value "HiHiHiHiHiHi"
Line 21: OK
Line 22: OK value "HiHiHiHiHiHiHi"
Line 23: 16'4800 ("" is converted to 8'b0)
Line 24: OK value 12'h048
Line 25: OK
Line 26: OK, same as Line 25
Line 27: s2[1] unchanged, "\0" is not assigned
```

Returns a byte, the ASCII code at the given index. Indexes range from 0 to $N-1$, where N is the number of characters in the string. If index is out of range, the operator returns 0.

The methods of **`string`**data-type are

len

```
function int len()
```

Return value is the length of the string, excluding any termination character. For an empty string "", the return value is 0.

putc

```
task putc(int i, byte c)
```

Task replaces the i^{th} character in **string**with the given value 'c'. 'c' is checked for non-zero before action is taken. If index i is out of range, then string is un-affected.

getc

```
function byte getc(int i)
```

Returns the ASCII code of the i^{th} character in **string**.

toupper

```
function string toupper()
```

Returns a string with characters converted to uppercase, without affecting the original **string**.

tolower

```
function string tolower()
```

Returns a string with characters converted to lowercase, without affecting the original **string**.

compare

```
function int compare(string s)
```

Returns value of **ANSI C strcmp** function.

icompare

```
function int icompare(string s)
```

Returns value of **ANSI C strcmp** function, but is case in-sensitive.

substr

```
function string substr(int i, int j)
```

Returns a new **string**that is a substring formed by characters in position i through j. An empty-string is returned if either $i < 0$ or $j < i$ or $j >= str.len()$.

atoi, atohex, atooct, atobin, atoreal

```
function integer atoi()
function integer atohex()
function integer atooct()
function integer atobin()
function real    atoreal()
```

Returns the number after interpretting the string as either decimal, hexadecimal, octal, binary or real formats (respective functions). These functions expect raw strings and are not **Verilog**integer literal compliant (cannot interepret **Verilog**int literal).

itoa, hextoa, octtoa, bintoa, realtoa

```
task itoa(integer i)
task hextoa(integer i)
task octtoa(integer i)
task bintoa(integer i)
task realtoa(real r)
```

Family of method to store the ASCII real representation of argument into the string in the implied format.