System Verilog through Examples

Author Name Here

September 3, 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'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
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.

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.

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

In Listing 1.2, line 1 gets expanded to `'{'{0,1,2}, '{4, 4, 4}}` and line 2 gets expanded to `'{'{4,5,4,5,4,5},'{4,5,4,5,4,5}}`. Array literals can also use their index or type as a key and use a default key value. Both lines of Listing 1.2 are equivalent. Line 3, 4: both become `[0, 1, 0, 0]`.

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 1.3. Unordered values as in line 4, would be an error.

Listing 1.3: Structure Literals

```
struct {int a; shortreal b;} ab, ab2[2];
ab = '{0, 0.0};
ab2 = '{'{1, 1.1}, '{2, 2.2}};
ab2 = '{1, 1.1, 2, 2.2};
ab = '{int:1, shortreal:1.1};
ab = '{default:0};
struct {int a, b[4];} ab [2][3] = '{2{'{3{a,'{2{b,c}}}}}};
```

```
Line 4: Error: Only Line 3 format supported
Line 5: becomes '{1, 1.1}
Line 6: becomes '{0, 0.0}
Line 7: is expanded as...
 '{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}} } }
```

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 1.4, `c3[0]` contains character ‘H’ and `c3[7]` contains ASCII value of character ‘o’.

Listing 1.4: String Literal Examples

```
byte c3 [0:7] = "Hello World\n";
byte c4 [7:0] = "Hello World\n";
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"
```

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 [ ]
: 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.

Listing 2.1: Structure Literals

```
string s1;
string s2 = "abcdef\n";
string s3 = "ghijkl\0";
string s4 = "hello";
bit [11:0] b = 12'ha41;
string s2 = string'(b);

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

r = r_t'(s6);
s5 = string'(r);
s5 = "Hi";
s5 = {5{"Hi"}};
s6 = {i{"Hi"}};
r = {i{"Hi"}};
s6 = {i{s5}};
s6 = {s6,s5};
s6 = {"Hi",s5};
r = {"H",""};
b = {"H",""};
s2[0] = "h";
s2[0] = "cough";
s2[1] = "\0";
```

```
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
```

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

```
event e1, e2;
e1 = e2;
e1 - null;
```

event variables could be assigned to each other or null. After declaration both e1 and e2 are distinct event objects capable handling distinct simulation events. When e1 is assigned from e2, e1's object is lost and both are referencing to the same object that was e2. When e1 is assigned to 'null', there exists no event object for this and all threads waiting on e1 would never be triggered further.

```
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.

2.6 event

eventdata-type is enhanced over **Verilog**'s definition of the same.

```
typedef int IntP;
IntP    ia, ib;

typedef foo;
foo f = 1;
typedef int foo;
```

In the first block, two 'int' variables ia, ib are created as user-defined type.
In the second block, deferred declaration of variable 'f' occurs. This feature is usefull to overcome circular declarations.

```
interface intf_i;
  typedef int data_t;
endinterface

module sub(intf_i p);
  typedef p.data_t my_d
  my_data_t data;
endmodule
```

User-defined data-types have same scoping rules as variables. They can be accessed via hierarchial scope specifier as shown in tis example.

2.7 User-Defined-types

As in `C`, **`Verilog`**also provides **`typedef`**constructs that creates new data-types. They exhibit scoping rules as in **`C++`**

2.8 enum

Enumerated types define a set of named constants of any type. Enumerated variable names could be declared in singular or as an array and could have a constant to be initialized with. Valid specification include `name, name=C, name[N], name[N]=C, name[M:N], name[M:N]=C`, where N, M are positive integer constants. When declared as an array, an array of enumerated constants are created with incrementing values starting with the provided constant C or previous value as applicable. Names are suffixed with the number like `name0, name1` etc.... Range of enumerations are $0..N-1$ or $M..N$ as applicable.

Enumerated variables are strongly typed and need a cast to be assigned if not assigned from the enumerated constant itself. The cast does a dynamic value check and asserts error when value is outside the enumeration's range. When used in expressions they take their literal value and act like literal constants.

Enumerated variable names should be unique in the scope of its existence wherous the values should be unique within itself. Values could be overlapping with other enumerated types and is not of concern.

```
enum {red, yellow, green=3} light1, light2;
enum {IDLE, XX='x, S1=2'b01, S2=2'b10} state, next;
enum integer {IDLE, XX='x, S1=2'b01, S2=2'b10} state, next;
```

light1, light2 are of int data type since type is not specified by default.
Declaration could value constants to which they represent. 'red' has a value of 0, 'yellow' has a value of 1 and so on. 'green' has a value 3 as it is specified that way.
It is an error in next line when literal 'x is attempted to assigned, when no data type is specified as the default data type is 'int'. The correct way for such a declaration is as shown in the last line.

```
enum {bronze=3, silver,
enum {a, b=7, c, d=8} a
enum bit[2:0] {oa=1, ob
```

'silver' takes value 4 and 'gold' takes 5 as every next value is obtained by incrementing the previous value when not specified.
'a' takes value 0. 'c' and 'd' both would take the value 8, and hence it would be an error.
When value of 'oc' is computed it is overflown, and hence it would be an error.

```
enum bit [3:0] {bronze='h3, silver, gold='h5} medal4;
enum bit [3:0] {bronze=4'h3, silver, gold=4'h5} medal4;
enum bit [3:0] {bronze=5'h13, silver, gold=3'h5} medal4;
enum bit [0:0] {a,b,c} alphabet;
```

Line 1: Valid. Assignment of unsized value.
Line 2: Valid. bronze and gold sizes are redundant
Line 3: Error in the bronze and gold member declarations, as size of bit-vector doesn't match
Line 4: overflow in value, needs to be 2 bits

```
typedef enum {NO, YES}
boolean myvar;
```

'boolean' is an enumerated type and so it could be used in multiple places.

```
typedef enum { add=10, sub[5], jmp[6:8] } E1;
```

Following enumerated values are objtained {add=10, sub0=11, sub1=12, sub2=13, sub3=14, sub4=15, jmp[6]=16, jmp[7]=17, jmp[8]=18}

```
typedef enum {red, gree
int a;
Colors c;
c = green;
c = 1;
c = Colors'(1);
a = c;
c++;
c+=2;
```

In this example the assignment 'c=1' generates an error as it is invalid to assign a non-enumerated value to enumerated variable. The correct assignment is shown in the next line. When assigned to a int, it gets type-cast into int-literal automatically. They could also be used in expressions where corresponding literal constants could be used.
Both expressions 'c++' and 'c+=2' are illegal as they contain assignment without type-value checking.