Change Section 3.7, Page 12

Variables of type **`string`** can be indexed from 0 to N-1 (the last element of the array), and they can take on the special value “”, which is the empty string.Reading an element of a string yields a byte. A string shall not contain the special character “\0”.

Change Section 3.7, Page 12

If an initial value is not specified in the declaration, the variable is initialized to “”, the empty string. An empty string has zero length.

Change Section 3.7, Page 12

A string~~,~~ or a string literal~~, or packed array~~ can be assigned directly to a string variable. Integral types can be assigned to a string variable, but require a cast. When casting an integral value to a string, the ~~The~~ **`string`** variable shall grow or shrink to accommodate the size of the integral value ~~packed array~~. If the size (in bits) of the integral value ~~packed array~~ is not a multiple of 8, then the integral value ~~packed array~~ is zero filled on the left.

A string literal assigned to a string variable is converted according to the following steps:

- Any leading “\0” characters in the string literal are ignored.
- If the result of the first step is an empty string literal, the string is assigned the empty string.
- Otherwise, the string is assigned the remaining charactes in the string literal until reaching either the end of the string literal, or the first character “\0”, whichever comes first.

Thus, leading “\0” characters are ignored, and subsequent “\0” characters truncate the result.

Casting an integral value to a string variable proceeds in the following steps:

- If the size (in bits) of the integral value is not a multiple of 8, the integral value is left extended and filled with zeros until its bit-size is a multiple of 8. The extened value is then treated the same as a string literal, where each successive eight-bits represent a character.
- The steps described above for string literal conversion are applied to the extended value.

For example:

```
string s1 = "hello"; // sets s1 to "hello"
bit [11:0] b = 12'ha41;
~~string s2 = b; // sets s2 to 'h0a41~~
string s2 = string'(b); // sets s2 to 'h0a41
```

As a second example:

```
typedef reg [15:0] r_t;
```

```
r_t r;
integer i = 1;
string b = "";
string a = {"Hi", b};
r = a r_t'(a) ; // OK
b = r string'(r) ; // OK (implicit cast, implementations can issue a warning)
b = "Hi"; // OK
b = {5{"Hi"}}; // OK
a = {i{"Hi"}}; // OK (non constant replication)
r = {i{"Hi"}}; // invalid (non constant replication)
a = {i{b}}; // OK
a = {a,b}; // OK
a = {"Hi",b}; // OK
r = {"H",""}; // yields "H\0" "" is converted to 8'b0
b = {"H",""}; // yields "H" as "" is the empty string
a[0] = "h"; // OK same as a[0] = "hi" )
```

Change Table 3-2, Page 13

Operator	Semantics
`Str1 == Str2`	Equality. Checks if the two strings are equal. Result is 1 if they are equal and 0 if they are not. Both strings can be of type **`string`**. Or one of them can be a string literal. If both operands are string literals, the expression is the same Verilog equality operator for integer types. The special value `" "` is allowed.
`Str1 != Str2`	`Str1 != Str2` Inequality. Logical Negation of ==

`Str1 < Str2`
`Str1 <= Str2`
`Str1 > Str2`
 	
`Str1 >= Str2`
	Comparison. Relational operators return 1 if the corresponding condition is true using the lexicographical ordering of the two strings `Str1` and `Str2`. The comparison uses the `compare` string method ~~behaves like the ANSI C strcmp function (or the compare string method) (with regard to the lexical ordering) and embedded null bytes are included~~. Both operands can be of type **`string`**, or one of them can be a string literal.

Change Section 3.7.2, Page 14

3.7.2 putc()

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

— `str.putc(i, c)` replaces the *i*th character in *str* with the given integral value.

— `str.putc(i, s)` replaces the *i*th character in *str* with the first character in *s*.

— `s` can be any expression that can be assigned to a string.

— `putc` does not change the size of `str`: If `i` < 0 or `i` >= `str.len()`, then *str* is unchanged.

— If the second argument to this method is byte 0 or the empty string, the string variable shall be unaffected. An attempt to embbed a 0-valued character in the string shall be considered an out-of-bounds access.

Note: `str.putc(j, x)` is semantically equivalent to `str[j] = x`

Change Section 3.7.6, page 14

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

— `str.compare(s)` compares `str` and `s`, as in the ANSI C `strcmp` function ~~(~~with regard to lexical ordering and return value~~), and embedded null bytes are included~~.

See the relational string operators in Section 3.7, Table 3-2.

Change Section 3.7.7, page 14

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

— `str.icompare(s)` compares `str` and `s`, like the ANSI C `strcmp` function ~~(~~with regard to lexical ordering and return value~~)~~, but the comparison is case insensitive ~~and embedded null bytes are included~~.