Preprocessor Specification

Grammar Reference

This document specifies a preprocessor, primarily intended to supersede the existing Verilog/SystemVerilog one with a Turing-complete language. It is however compatible with any language that does not make use of the backtick.

The main text uses the following conventions:

- Bold is used for chapters and sections title formatting, and to highlight important information and document-specific terms.
- Monospace font is used for syntax descriptions and source code examples.
- Bold-red monospace characters denote reserved keywords, operators, and punctuation marks as a required part of the syntax.

The formal syntax is described using Backus-Naur Form (BNF). The following conventions are used:

- A vertical bar that is not in bold-red separates alternative items.
- Square brackets that are not in bold-red enclose optional items. The item may appear only once.
- Braces that are not in bold-red enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule.

TABLE OF CONTENTS

1 DELIMITATION

The preprocessor uses the backtick as a code delimiter, while the Verilog/SystemVerilog is seen as plain text. Except for evaluations, all preprocessor statements start at the backtick and end at the next, ending newline. Newlines appearing in between parentheses, brackets and curly braces, are not ending newlines.

A pass-through is given for Verilog/SystemVerilog directives by appending a backtick before theirs. The following character sequence is then seen as plain text, and reproduced in the output file. In that case, the extra backtick is removed.

Example:

```
``timescale 1ns/1ps
```

Here, the timescale directive is ignored by this preprocessor, and copied to the output file.

2 KEYWORDS

Keywords are predefined, non-escaped identifiers that are used to define the language constructs. All keywords are defined in lowercase only:

```
begin
else
elseif
end
endfor
endif
false
for
global
if
include
inside
local
log2
mut
return
size
true
```

3 EXPRESSIONS

Expressions are sequences of operators and their operands that specify a computation. Operators, in the form of non-alphanumeric characters, always produce a result. Those symbols are listed below in the order of decreasing precedence; the highest precedence is listed first.

Category	Symbol	Meaning	Associativity
Postfix	`expression[expression]` `expression([expr {, expr}])`	Subscript Function call	Left to right

Prefix	`+expression`
	`-expression`
	`~expression`
	`!expression`
	`$expression`
	`@expression`
		Arithmetic identity
		Arithmetic negation
		Bitwise NOT
		Logical NOT
		String interpolation
		Dynamic naming
			Right to left

	`expression ** expression`	Exponentiation	Right to left

Multiplicative	`expression * expression`
	`expression / expression`
	`expression % expression`
		Arithmetic product
		Arithmetic quotient
		Arithmetic remainder
			Left to right

Additive	`expression + expression` `expression - expression`	Arithmetic sum Arithmetic difference	Left to right

Shift	`expression << expression`
	`expression >> expression`
	`expression >>> expression`
		Left shift
		Logical right shift
		Arithmetic right shift
			Left to right

Relational	`expression inside expression`
	`expression == expression`
	`expression != expression`
	`expression < expression`
	`expression <= expression`
	`expression > expression`
	`expression >= expression`
		Set membership
		Equal
		Not equal
		Less-than
		Less-than or equal
		Greater-than
		Greater-than or equal
			Not chainable

Bitwise	`expression & expression`	Bitwise AND	Left to right
	`expression ^ expression`	Bitwise XOR	Left to right
	`expression | expression`	Bitwise OR	Left to right
Logical	`expression && expression`	Logical AND	Left to right
	`expression || expression`	Logical OR	Left to right
	`expression ? expr : expr`	Ternary	Right to left

4 VARIABLES

A variable is a storage paired with an identifier, which contains a quantum of information referred to as a value. An identifier is a sequence of one or more lowercase letters, digits and underscores, albeit the first character must not be a digit. A variable can be defined at most once. A definition binds an identifier to a type and assigns an initial value. Variables must be defined prior to their use in expressions.

4.1 Variables definition

The syntax for defining a variable is:

```
`[global|local] expression = expression \n
```

with the left expression being declarable, that is, either an identifier or a dynamic name. Each variable is only valid in a continuous portion of the code called its scope. The scope of a variable, introduced by a definition in a block, begins at the point of definition and ends at the end of the block. If there is a nested block with a definition that introduces identical identifier, the entire scope of the nested definition is excluded from the scope of the outer definition. Variables can be made global, in which case the scope extends from the point of definition to the end of the file in which they are defined.

Example:

```
`local foo = 42
`global bar = "Hello"
```

In this example, variable “foo” is local while “bar” is made global.

4.2 Variable mutation

Variables can only be defined once in a given scope, but their value can be changed with the mutation operator:

```
`mut expression (= | +=) expression \n
```

with the left expression being addressable, that is, either an identifier, a dynamic name, an array element, or a dictionary entry. The mutation is performed on the variable visible in the current scope. It does not affect other variables with the same name in a parent scope of the visible one.

Example:

```
`mut foo = 123
`mut bar += "World"
```

4.3 Dynamic naming

Variables can be dynamically named with the value stored in another variable. This is done with the at symbol:

```
@expression
```

The expression must be of string type. This operator is useful for deferred string interpolation in functions, where a name can be passed as argument.

Example:

```
`local variable_name = "foo"
`local @variable_name = 123
```

On the second line, a variable named “foo” containing the value “123” will be declared.

4.4 String interpolation

String interpolation is a process substituting values of variables into placeholders in a string. String interpolation allows users to embed variable references directly in processed string literals:

```
$expression
```

The expression must be of string type. A string is returned, following the same resolution process as for the plain text. Note that no implicit interpolation is performed on string creation; they must be explicitly interpolated so that variables are replaced.

Example:

```
`local who = "world"
`local message = $"Hello `who!"
```

The variable “who” in the string will be substituted by its value “world”. The “message” variable will then contain “Hello world!”.

5 DATATYPES

The language supports the four base datatypes: integers, booleans, strings and arrays.

5.1 Integers

Integers are literals made of a non-zero decimal digit followed by zero or more decimal digits. Presently, only decimal numbers are supported.

5.2 Booleans

Boolean literals are the keywords `true` and `false`.

5.3 Strings

Strings are literals made of any valid ASCII character, except the double-quote, and enclosed in between double-quotes. Special characters can be escaped with a backslash:

Escape sequence	Description	ASCII representation
\'	Single quote	0x27
\"	Double quote	0x22
\\	Backslash	0x5c
\a	Audible bell	0x07
\b	Backspace	0x08
\f	Form feed	0x0c
\n	Line feed	0x0a
\r	Carriage return	0x0d
\t	Horizontal tab	0x09
\v	Vertical tab	0x0b

5.4 Arrays

An array is an ordered set of items, which can be of different types. The array is identified with the name of its base variable. The syntax for defining an array is:

```
[[expression {, expression}]]
```

A continuous set of integers can be defined with the range operator:

```
expression .. expression
```

where the left expression is the lower inclusive end of the range and the right expression is the upper inclusive end of the range. This operator can only appear inside an array definition as an expression.

Example:

```
`local array = [1..3, true, "Daffy Duck"]
```

Arrays elements are accessed through the subscript operator, with an integer index.

5.5 Dictionaries

A dictionary is set of key-pair values, such that each possible key appears at most once in the collection. The dictionary is ordered against the key, not the insertion order like an array. The syntax for defining and array is:

```
{[expression : expression {, expression : expression}]}
```

with the expressions on the left of the colon being a string. Dictionaries elements are accessed through the subscript operator, with a string index.

Example:

```
`local money = {"Henry": 42, "Tom": 123}
`mut money["Tom"] += 10
```

6 STATEMENTS

Statements are fragments of the code that are executed in sequence. The body of a program is a sequence of statements, though without explicit delimiters.

For further use, we define the following equivalence:

```
statements = {statement}
```

6.1 Selection statement

Selection statements choose between one of several flows of control. The syntax is:

```
`if (expression) \n
  statements
{`elseif (expression) \n
  statements}
[`else \n
  statements]
`endif \n
```

Statements are executed if the associated condition evaluates to a non-zero value. In that case, statements of the following conditions are skipped. If the value is zero, then the statements are skipped and the following, if present, are evaluated the same way. The “else” block has no condition and is always executed if not skipped. There can be zero or more “elseif” blocks followed by zero or one “else” block in each selection statement.

6.2 Iteration statement

Iteration statements repeatedly execute the encompassed statements while successively assigning the values of the expression list to the variable defined by the identifier.

```
`for ([expression ,] expression : expression) \n
  statements
`endfor \n
```

with the left expressions being declarable, that is, either identifiers or dynamic names; and the right expression being an array. The left-most expression allows an optional record of the current iteration index, starting from 0.

6.3 Expression statement

A single backtick followed by a keyword is a statement. A single backtick followed by an identifier is an expression statement. This statement evaluates the expression and substitutes it to the original text in the output file. While other statements end at the end-of-line, an expression statement ends at the first non-alphanumeric or underscore character, outside of any parentheses, brackets or curly braces group. The syntax is:

```
`expression
```

To allow for function calls, if that character is an opening parenthesis, it is considered part of the expansion.

To prevent the preprocessor to consider an opening parenthesis, to expand in between an alphanumeric or underscore Verilog/SystemVerilog identifier, or to expand complex expressions, enclose the expression in between parenthesis as follow:

```
`(expression)
```

Example:

```
local lang = "verilog"
string my_`(lang)_variable;
```

In this example, the expanded text will be:

```
string my_verilog_variable;
```

7 FUNCTIONS

A function associates to a set of values an output value with a sequence of statements. It can have zero or more input parameters but exactly one output.

7.1 Functions definition

They are defined as follow:

```
`[global|local] expression([identifier {, identifier}]) begin \n
  statements
`end \n
```

or:

```
`[global|local] expression([identifier {, identifier}]) = expression \n
```

with the left expression being declarable, that is, either an identifier or a dynamic name. The parameters of a function definition are variables definition, with visibility restricted to the function scope.

Functions have visibility on the caller scope. This removes the need for two-steps string interpolation, when some variables are defined in the caller's scope, some in the callee's one.

Example:

```
`global say_hello(name) begin
Hello `name
`end
```

Called with "world" as parameter, this function will print "Hello world" in the output file.

7.2 Functions call

Functions are called as follow:

```
expression([expression {, expression}])
```

with the left expression being addressable, that is, either an identifier, a dynamic name, an array element, or a dictionary entry. When a function is invoked, the parameters are initialized from the arguments and the statements in the function body are executed.

Functions reproduce the plain text that they enclose, executing the statements inside, and performing the evaluations. Alternatively, one can explicitly return from a function:

```
`return expression \n
```

This allows the caller to catch a result. In the absence of a return statement, the result is void and must not be used in further computation. Note that the plain text up to the return statement is copied to the output file.

Example:

```
`global say_hello(name) begin
`return $"Hello `name"
`end
```

This function will return the string to the caller that must catch it in a variable, if not in an evaluation.

7.3 Built-in functions

The preprocessor features some built-in functions. The associated keywords are reserved.

Syntax	Description
`log2(expression)`	This function returns the logarithm in base 2 of the expression, which must be an integer.
`size(expression)`	This function returns the size of the array passed as argument.

8 FILES

To be processed, files must have one of the two following extensions:

- ".pp" for source files. They are compiled and a file is outputted for each one, in the same directory, and with the name but the extension removed. Preprocessor source files are usually Verilog/SystemVerilog source and header files.
- ".pph" for header files. They are compiled, but no file is outputted. Those are preprocessor exclusive files, such as libraries.

File inclusion can be used to insert the entire content of a source file in another file during generation. The result is as though the contents of the included source file appear in place of the statement. The syntax is as follow:

```
`include expression \n
```

with the expression being a string, representing the path from the working directory.

9 COMPILE AND RUN

The compilation of the preprocessor requires only a C++ compiler. The preprocessor has been tested in Debian 10 with G++ 8.3.0 with the following command:

```
g++ -o a.out -O3 *.cpp -pthread
```

To run the preprocessor, pass the files to process as follow:

```
./a.out *.pp *.pph
```

Presently, there is no option to set a destination folder. The output files are written in the same directory as their source file.

A EXAMPLE

The following example is derived from Western Digital SweRV-EH1 core.

```
/// design/lsu/lsu_bus_intf.sv.pp (original lines 305-365)

`include "libhdl.pph"

`for (i : [0..3])
`local bus = $"`((8*i)+7):`(8*i)"
`for (a : ["hi", "lo"])
`for (b : ["hi", "lo"])
`for (j : [3..5])
assign ld_byte_dc`(j)hit_`(b)_`(a)[`i] = ld_addr_dc`(j)hit_`(b)_`(a) &
  ldst_byteen_`(b)_dc`(j)[`i] & ldst_byteen_`(a)_dc2[`i];
`endfor
`endfor

assign ld_byte_hit_`(a)[`i] =
  `or_reduction("ld_byte_dc`(j)hit_lo_`(a)[`i]", "j", [3..5]) |
  `or_reduction("ld_byte_dc`(j)hit_hi_`(a)[`i]", "j", [3..5]) |
  ld_byte_hit_buf_`(a)[`i];

`for (j : [3..5])
assign ld_byte_dc`(j)hit_`(a)[`i] = ld_byte_dc`(j)hit_lo_`(a)[`i] |
  ld_byte_dc`(j)hit_hi_`(a)[`i];
assign ld_fwddata_dc`(j)pipe_`(a)[`bus] =
  ({8{ld_byte_dc`(j)hit_lo_`(a)[`i]}} & store_data_lo_dc`(j)[`bus]) |
  ({8{ld_byte_dc`(j)hit_hi_`(a)[`i]}} & store_data_hi_dc`(j)[`bus]);

`endfor
assign ld_fwddata_`(a)[`bus] = `decreasing_prio("ld_byte_dc`(j)hit_`(a)",
  "ld_fwddata_dc`(j)pipe_`(a)[`bus]", "ld_fwddata_buf_`(a)", "j", [3..5]);

`endfor
`endfor
`end

/// libhdl.pph

`global or_reduction(data, iter, range) begin
`for (idx, @iter : range)
`if (idx == 0)
( `$data
`else
| `$data
`endif
`endfor
)
`end

`global decreasing_prio(select, data, data0, iter, range) begin
```

```
(
`for (@iter : range)
   `$select ? `$data :
`endfor
   `$data0 )
`end

`end
```

Note that the only portion of code reproduced here has been modified.