TL-Verilog Macro-Preprocessor

User Guide (Draft)

RW-PP-UG (v0.0.0) November 8, 2022

This document applies to the following software versions: (Draft, not versioned)

Redwood EDA

Redwood EDA, LLC
36 Venus Drive
Shrewsbury, MA 01545
Website: https://www.redwoodeda.com

Table of Contents

Introduction

This guide describes the integration of the M5 macro preprocessor with TL-Verilog, supported by distributions of Redwood EDA, LLC's SandPiper™ TL-Verilog compiler. Advanced uses of TL-Verilog may warrant preprocessing with M5. M5 goes beyond simple text substitutions and is capable of parsing arbitrary inline syntax, providing customizable code construction backed by the powerful modeling constructs of TL-Verilog. While it is possible in TL-Verilog to utilize the Verilog macro preprocessor, the integration of M5 is far more capable. M5 is implemented using the Gnu M4 macro processor.

TL-Verilog's macro preprocessing provides:

- parameterization
- modularity and reuse
- code construction

Why M5?

This section discusses some of the motivating factors behind TL-Verilog's use of M5.

The Utility of Macro Preprocessing

Macro preprocessing provides an important safety net. When formalized language features lack the desired flexibility or syntax, macro preprocessing can come to the rescue. As we are still in the early stages of TL-Verilog's evolution, we currently lean heavily on macro preprocessing to fill in known gaps. Macro preprocessing helps the TL-Verilog community explore language features before they are formalized in the TL-Verilog language specification and tools where they become harder to change.

As some use models of macro preprocessing are highly exploratory, it should be understood that backward compatibility and robustness may take a back seat to experimentation, and support may be limited. Macro preprocessing features are made available on an as-is basis with no warranties.

The Role of Code Construction

TL-Verilog is a language to support the next generation of silicon with exponentially-increasing complexity. Modularity and reuse are essential to these goals. To improve modularity and reuse, reusable components must be highly configurable. Code construction, aka software that generates hardware models, is an important capability to enable this high degree of configurability.

Language Tradeoffs for Code Construction

Any language can be used to output hardware descriptions. Just about every popular programming language has been used to implement at least one domain-specific language (DSL) for hardware construction. These DSLs support hardware construction far better than traditional RTL languages. They are also very convenient for hardware designers with skills using the given programming language. It is a challenge, however, to get an entire team of logic designers to agree to any one particular DSL. Engineers unfamiliar with the underlying language would need to learn that language in addition to the hardware features of the DSL.

The fundamental contrast between a macro preprocessor and a programming language is the following. In a macro preprocessor, the default behavior is to echo the source code to the output. Procedural code for generating text output is atypical and more awkward. Programming languages are the reverse. Code is procedural, and explicit print statements echo text to the output.

Best of Both

M5 extends traditional macro preprocessing with features that look and act more like a simple, traditional programming language, so this use model is available in the rare cases where it is needed. Thus, as detailed below in Table 1, M5 provides the approachability of macro preprocessing for simple cases and the flexibility of DSLs for complex cases. It also provides the freedom to build support for customized syntax inline. On the downside, as with any new language, the ecosystem for M5, including editor modes and AI assistance, is, for now, limited.

	Traditional Macro Preprocessor	Programming Language	M5
Learning Curve	Easy	Hard	Moderate
Capability	Limited	Vast	Vast
Ease of Development for Simple Cases	Easy	Moderate	Easy
Ease of Development for Complex Cases	Hard	Easy	Easy
Maintainability	Awful	Okay	Okay

Ecosystem	Basic	Rich	Basic
Syntactic Flexibility	Moderate	Moderate	Good

Table 1: Language tradeoffs for code construction

Ease of Debug

Missing from Table 1 is a category for "Ease of Debugging". This is an important case that requires more discussion. Whether using a programming language or a macro preprocessor for code generation, the single most important factor for productivity is the ability to correlate failures reported against the generated code with the source code. The code generation process must result in metadata to assist in this correlation and tools must utilize this metadata to result in a seamless user experience.

The TL-Verilog preprocessor maintains a correlation at a granularity of code lines. A distinction is drawn between macros that expand within a line and those that can expand to multiple lines. Multiline expansion is either global or within `\TLV` context with appropriate indention applied automatically. Admittedly, this distinction does result in an additional burden to the developer. It does, however, result in a productive experience for code generation, which, in turn, leads to an ecosystem of flexible, reusable components.

Also important to consider is that, as engineers, we often debug or reverse engineer code we didn't write. Using M5 with TL-Verilog requires learning a moderate amount of new syntax, and beyond that, macro code can be understood by referencing macro documentation.

Structure of TL-Verilog Macro Preprocessing

File Structure Overview and Macro Categories

First, a simple TL-Verilog file *without* the use of M5 might look as follows:

```
\TLV_version 1d: tl-x.org
\SV
   module top(input wire clk, input wire reset, …)
\TLV
   $foo[7:0] = 5;
\SV
   endmodule
```

The `\TLV_version` line above specifies the file format and references documentation. `\SV` regions define the module interface using SystemVerilog syntax. And, the `\TLV` region defines the logic of the module using TL-Verilog constructs.

We introduce several aspects of TL-Verilog macro preprocessing in the file below, structured to compile in the Makerchip IDE.

```
\m5_TLV_version 1d: tl-x.org
\m5
   // This region contains M5 macro definitions. It will not appear
   // in the resulting TLV code.
   use(m5-0.1)    // Use M5 libraries
   var(five, 5)   // An example definition

\SV
   // Library inclusion (which may result in Verilog `includes).
   m4_include_url(['https://domain.com/lib.tlv'])

// A simple TLV macro block that assigns a pipesignal to a constant.
\TLV assign_const($_sig_arg, #_const_arg)
   $_sig_arg = #_const_arg;

// The main module (here as required for Makerchip).
\SV
   m5_makerchip_module   // Standard module interface for Makerchip.
\TLV
   // Logic. Here, a constant assignment as a macro call.
   m5+assign_const($foo[7:0], m5_five)
\SV
   endmodule
```

The file above illustrates the following parts:

- `\m5` regions: The `\m5` region defines macros used in the remainder of the file. In TLV library files, `\m5` regions can define macros that are exported to other files.
- File inclusion: Another TL-Verilog library file is included by URL.
- Within-line macros: Macros, like `five` above, are defined using M5 and expand within a line. They must be defined to produce no carriage returns.
- Multiline TL-Verilog macro blocks: The `\TLV` *`<name>`*`(`*`<params>`*`)` syntax defines a parameterized TL-Verilog macro block. These macros are instantiated (or called) using the `m5+<name>(<args>)` syntax, The body is produced with arguments substituted. Though not illustrated above, multiline macros may also be defined programmatically, without line tracking.

The code above results in the following TL-Verilog code after macro preprocessing:

```
\TLV_version 1d: tl-x.org
\source top.tlv 8
\SV
   // Library inclusion (which may result in Verilog `includes).
   // Included URL: "https://domain.com/lib.tlv"
\source top.tlv 16
// The main module (here as required for Makerchip).
\SV
   module top(input wire clk, input wire reset, …
\TLV
   // Logic. Here, a constant assignment as a macro call.
   \source top.tlv 14   // …
      $foo[7:0] = 5;
   \end_source
\SV
   endmodule
```

The `m5+` macro call is expanded to `$foo[7:0] = 5;`. The tags `\source` and `\end_source` are part of the TL-Verilog standard and enable tracking of the generated TL-Verilog code lines back to the source code. In Makerchip, NAV-TLV line numbers can be clicked to trace back to the source code. Generation of `\source` lines by Redwood EDA, LLC's SandPiper™ tool is controlled by the `--fmtNoSource` option.

The Sequential Nature of Macro Processing

While TL-Verilog is defined such that the order of statements does not affect their meaning, macro preprocessing is sequential. A macro call will appear within the definition of a macro within a macro library. It is important to understand whether this call will be made when the library is included, when the macro is defined, when the macro is instantiated, etc. As the text is processed, quoting controls whether the text is elaborated or taken as a literal string (which might be captured and later elaborated).

\m5 Regions

Similar to code regions defined by the TL-Verilog specification, the \m5 region identifier begins a line, and the region continues until the next line with no indentation. They are processed when encountered by the natural elaboration order of the macro preprocessor and are expected to define macros that can be used later. They result in no immediate content in the resulting TL-Verilog file. Like a function body, after elaboration, the only remaining/resulting text should be comments and whitespace, and it is discarded.

Within-line Macros

Macros that expand within a line would most often be defined within `\m5` regions, though they can be defined anywhere, including in the bodies of other macros, to be defined when those macros are called. Macros used to define other macros expand to no text.

Within-line macros must produce no carriage returns, as doing so would impact line tracking.

Within-line macros are often variables and simple substitutions.

Multiline TLV Macro Blocks: (`\TLV` and `m5+`)

TLV Macro Block Declarations

Multiline TLV macro blocks are declared as code regions (meaning they begin a line) as:

```
\TLV <name>(<params>)
   <body>
```

Its parts are:

- *<name>*: The name of the macro, comprised of word characters (`a-z, A-Z, 0-9, and _`)
- *<params>*: A comma-separated list of parameter names. Parameter names may begin with an optional string of symbol characters (`~, `, @, #, $, %, ^, &, *, +, =, |, \, :, ", ', ., ?, /, <, and >`) followed by a string of word characters.
- *<body>*: The macro body is captured as a literal string. Each call of the macro results in a copy of the body with any strings matching parameters substituted for the corresponding argument text and then elaborated. Matching strings cannot be preceded or postceded by a character that could be part of the name. `['']` can be used to create delimitations. As a special case, `m5+` may precede the name. [how does this work, exactly?]

TLV Macro Block Calls (`m5+`)

Calls of TLV macro blocks take the form:

```
   m5+<name>(<args>)
```

The parts of the call are:

- <*name*>: The name, matching the declaration.
- <*args*>: A comma-separated list of arguments. This list follows the rules of M5 argument lists.

The call may appear anywhere within a `\TLV` region that a TL-Verilog statement (or, generally, any node) could appear. Often a call will be on a line by itself. Multiline calls are supported, though with a restricted syntax. These are explained in the next two sections.

It is possible for a call to appear within a line if the surrounding text would be eliminated by macro preprocessing. For example:

```
m5_if(m5_do_it, ['m5+it()'])
```

The expansion of an `m5+` macro call is properly indented based on its calling context. Unless SandPiper is invoked with the `-fmtNoSource` option, the expansion is surrounded by `\source` and `\end_source` tags that map the generated lines back to their source code.

m5+ Calls That Span Multiple Lines

At times macro call argument lists can be lengthy. Calls may be split across multiple lines using very specific indentation. For example:

```
m5+my_code(arg1,
   arg2, arg3,
   arg4)
```

Optionally, arguments may appear on the first line. Any subsequent lines must be indented by three spaces. The closing parenthesis must follow the last argument immediately (as is a general requirement of M5). As with M5 in general, it is not possible to provide comments within the argument list (unless the macro specifically accepts comment arguments).

\TLV Block Arguments and Parameters

Special syntax is supported for passing `\TLV` code blocks as macro arguments. For example:

```
m5+simple_if(m5_condition, 1,
   \TLV
      $foo = $bar;
   ,
   \TLV
      $foo = 1'b0;
   )
```

The `simple_ifelse` macro call will result in one of the code blocks passed as arguments. The code blocks and their following commas or end parentheses must be indented exactly three spaces. The code blocks are passed in literally. In other words, they are implicitly quoted.

The `simple_ifelse` macro can be defined as follows:

```
\TLV simple_if(_cond, _yes_block, _no_block)
   m5_if(_cond, ['m5+_yes_block'], ['m5+_no_block'])
```

The `\TLV` block parameters can be called using the same syntax as for other macro blocks (with no arguments), using their parameter names.

Procedural TLV Code Generation: `m5_TLV_fn(...)`

Infrequently, it is convenient to generate multiline macros procedurally. This can be done to avoid the overhead of line tracking using `\source` and `\end_source`.

Such macros can be defined using the `m5_TLV_fn` macro, which follows the same protocol as `m5_fn`. The resulting output text is a block of TL-Verilog code. The output text should begin with a new line. Indentation will be added such that output lines with no indentation will align with the call.

Functions defined in this manner are called just like `\TLV`-style macros, using `m5+` notation. The arguments are passed into the function as for `m5_fn`.

TLV Macro Block Conventions

TLV macro blocks are simply blocks of text with text substitutions. They lack formality in that their parameters are not typed, the logic they define does not have explicit inputs and output, they have no scope, etc. These formalities will come in future features of TL-Verilog. Our use of TLV macros helps us to define these future features.

Certain conventions are used as a substitute for formalism. This example illustrates some of these conventions, discussed subsequently.

```
// wire(
//    /_top:  Top-level scope.
//    /_name: A unique level of hierarchy for the instance.
//    $_in:   An input pipesignal in /_top scope, optionally with a
//            bit range.
//    $_out:  An output signal in /_name with bit range (unless
//            boolean).
// Connects an input signal to an output signal.
// Should be called within a pipestage.
\TLV wire(/_top, /_name, $_in, $_out)
   /_name
      $_out = /_top$_in;

// A sample call in context.
\TLV
   |pipe
      @1
         $in_byte[7:0] = ...;
         m5+wire(|pipe, /connection1, $in_byte1, $out[7:0])
         $stuff = /connection1$out
```

Symbol characters are permitted in macro parameter names so that these names can match TL-Verilog identifier syntax. The following use of prefix characters is recommended:

- $: Pipesignal names and logic expressions. Comments should clarify whether bit ranges are to be included in the corresponding arguments and whether expressions are permitted.
- |: Pipeline identifiers or paths that include pipeline identifiers.
- @: Pipestages.
- /: Hierarchy identifiers or hierarchy paths not including pipelines.
- #: Elaboration-time constant values.
- (none): Other, including macro blocks.

By convention, the first word character of a parameter name is "_". Since TL-Verilog identifiers cannot begin with "_", this helps to distinguish parameters.

The use of `/_top` gives the macro the ability to reference signals in the scope of the instantiation.

The use of `/_name` provides a unique scope for any pipesignals (etc.) declared by the macro (though the example has none). It prevents declarations from multiple calls from the same TL-Verilog scope from conflicting.

The inputs and outputs of the logic defined by a TLV macro block can be parameters or hardcoded. It is generally best to parameterize inputs. Outputs may be parameterized or not. If parameterized, they may be assigned in the calling scope. If not parameterized, they should be assigned within the macro's scope, e.g. `/_name`, to avoid conflicts.

Macros may be written to be instantiated within a pipestage, within a pipeline, but not within a pipestage, or not within a pipeline. Header comments should specify the use model.

Library Inclusion

TL-Verilog macro preprocessing enables libraries to be included using URIs (uniform resource identifiers). URIs include local files and URLs (uniform resource locators). This applies to the inclusion of Verilog/SystemVerilog libraries and TL-Verilog libraries. Since these mechanisms require access to system commands and the internet, their availability depends upon the SandPiper installation and may be limited for security reasons.

To ensure reproducible results, it is important to only reference URLs that are trusted to contain immutable content. This might include URLs provided through content delivery networks (CDNs) or version control host links with specific commit IDs.

Verilog/SystemVerilog Library Inclusion

Some library inclusion mechanisms for M5 are under development. For now, some mechanisms utilize macros with `m4_` names.

In addition to local (System)Verilog header file inclusion using Verilog `` `include ``, header files may also be included by URL using `m4_sv_include_url(...)`. This should be done within `\SV` regions and the included content should be structured to define global content.

The macro call protocol is:

```
\SV
   m4_sv_include_url(<URL> [, <name>])
```

where *`<URL>`* is the complete URL to a raw immutable Verilog header file, and *`<name>`* is an optional name to give the local file (within `sv_url_inc/`) which defaults to the file name from the URL. The file is downloaded into a folder in the SandPiper object directory called `sv_url_inc/`, and it outputs a `` `include `` line that includes this file, assuming the Verilator include path includes this directory.

TL-Verilog Library Inclusion

All TL-Verilog files can be included as libraries. Library inclusion results in the inline elaboration of the included file. The resulting text from the elaboration is generally discarded, but definitions established in the elaboration persist. Primarily, the library provides macro definitions from `\m5` regions and `\TLV` macro blocks. Notably, the main `\TLV` region is discarded, as are `\SV` and `\SV_plus` regions. These regions however are elaborated, so they may define macros.

Text can be included in the resulting expansion by using `m5_show(<text>)`, though this is generally reserved for one specific scenario–the use of `m4_sv_include_url`. `m4_sv_include_url` outputs `` `include``s using `m5_show` such that the `` `include``s are passed along to the caller. Because of this, file inclusion should be done within an `\SV` code region. There may be multiple lines of output, so line tracking will not be properly preserved. The impact should be minimized by beginning a new code region immediately following library inclusion calls.

The macro call protocol is:

```
\SV
   m4_include_lib(<URL>)
```

where *<URL>* is the complete URL to a raw immutable `.tlv` library file. The file is downloaded into a file in the SandPiper object directory called `m4_include_url_file.#`, where `#` is replaced by an incrementing count, and the contents of this file are elaborated inline, possibly producing `` `include`` statements.

Issues

Errors or omissions with this guide as well as bugs or issues with Redwood EDA, LLC tools can be emailed to help@redwoodeda.com or filed publicly at https://gitlab.com/rweda/support/-/issues.