TL-X 1d HDL Extension Syntax Specification

Table of Contents

1. Introduction

1.1. About TL-X

TL-X is a set of language constructs that can extend various Hardware Description Languages (HDLs), including Verilog/SystemVerilog (as “TL-Verilog”), VHDL (as “TL-VHDL”), and System-C (as “TL-C”). The “X” in TL-X is a wildcard, referring to the target HDL. Extension features provide higher-level context for digital hardware design than those available in Register Transfer Level (RTL) languages. This context enables a concise syntax that does not sacrifice RT-Level expressibility. TL-X constructs can be processed by a “TL-X processor” to generate native code in the target HDL (without parsing the constructs of the target HDL).

The TL-X definition is evolving through voluntary contributions by members of TL-X.org, who maintain this standard. When the time is right, other standards associations such as Accellera and IEEE will be considered.

1.2. About This Specification

This specification is not intended for those first learning TL-X. It is specifically a reference for the language extension syntax. It does not describe the complete semantics, or meaning of the features. Other resources are available at http://tl-x.org to introduce the main concepts of TL-X. Specifically, the reader should already have an understanding of “behavioral hierarchy” (`/hier`), TL-X pipelines (`|pipe`), and “pipestages” (`@1`).

This specification covers TL-X syntax, as well as syntax specific to TL-Verilog, TL-VHDL, and TL-C.

TL-X syntax is intentionally rigid. Improper syntax must be reported. The interpretation and handling of improperly-formatted files is up to the discretion of the TL-X processor, except in that the improper syntax must be reported in some manner.

1.3. TL-X Prior Versions Summary

Here we summarize the major contributions of each version of TL-X. For complete details about previous versions and the changes from one version to the next, please consult the specifications for prior versions, available at TL-X.org.

New features in:

- TL-X 1a:
  - Baseline language extensions, with a primary focus on pipelines.
- TL-X 1b:
  - State signals (`$StateSigName`).
- TL-X 1c:
  - Changed syntax for alignment from `#+3` (after pipesignal) to `%+3` (before pipesignal).
  - Relative pipestage scope (`@++` and `@+=`).

1.4. What's New in TL-X 1d

The new features in TL-X 1d are:

- Changed syntax for alignment from `%+3` to `>>3`.
- Changed syntax for behavioral hierarchy from `>` to `/`.

2. TL-X Source Code Structure

2.1. File Structure

A TL-X file begins with a file format line, similar to a "magic number" that identifies the TL-X language version, and target HDL used in the file.

The remainder of the file is organized in regions that are either *HDL regions* or *TL-X regions*. HDL regions contain native HDL code. TL-X regions enable the TL-X language extension. There are also *HDL_plus* regions that utilize the target HDL's syntax with the ability to reference TL-X signals. These exists to ease manual conversions of HDL code to TL-X, and to provide access to native HDL features that are not accessible in TL-X regions.

A TL-X file is expected to define a module in the target HDL. It should begin with an HDL region defining the module interface and end with an HDL region terminating the module definition.

The body of the module can be defined in a single TL-X code region. Lower-level module instantiations can be made from within the TL-X code. HDL and HDL_plus code regions might be used within the module body as well to:

- include leveraged HDL content that has not been translated to TL-X
- access native HDL features not accessible in TL-X context
- work around limitations/bugs

A TL-X region provides logic in *assignment statements* in the context of *hierarchy*, *pipelines*, and *pipestages*. This context is provided via *scope* lines, where each level of scope introduces indentation of three spaces.

2.2. Contexts

Example TL-Verilog code below is colored based on its parse context. Contexts listed here are defined in detail throughout this specification.

- File format line: Specifies the file format -- the version of the TLV language being used. (See 4. File Format Line.)
- Code Region Identifier: Begins a code region. (See 5. Code Regions.)
- HDL: HDL content (SystemVerilog in this case). (See 5. Code Regions.)
- Line type character: Highlights attributes of the line. (See 6. Line Type Character.)
- HDL comments: Comments using HDL comment syntax.
- Scope: Defines the scope of the contained statements. Indentation reflects levels of scope, three spaces per level. (See 9. Scope.)
- Assignment statement: The logic of the design. These use HDL logic syntax with signal references that use TL-X syntax. (See 15. Assignment Statements/Blocks.)
- Signal references: A reference to a TL-X "pipesignal" or HDL signal, contained within assignment statements. (See 19.4. Pipesignal References.)
- Range/index: Specifies integer ranges or indices, such as bit ranges. (See 8. Ranges and Indices.)
- Compile-time expressions: An expression that evaluates to a constant, either during TL-X processing or HDL processing, contained within ranges and indexes of scopes and pipesignals. (See 8. Ranges and Indices.)

2.3. Example Code (TL-Verilog)

```
\TLV_version 1d: tl-x.org
\SV
module example(
   input [4:0] opcode_u101H [3:0],
   input clk,
   output [3:0] arith_err_u104H
);

\TLV
   |pipe
      // Four instructions
      /inst[3:0]
         @0
            $valid = ...;
      ?$valid
         @1
!           $opcode[4:0] = *opcode_u101H[inst];
            $add = $opcode == 5'b00000;
      /* ... */
      /inst[*]
         ?$valid
            @3
               $write = $opcode == 5'b11000 /* precise decode */;
            @4
!              *arith_err_u104H[inst] = ($add && $overflow) || ($sub
&& $underflow);
\SV
endmodule;
```

3. Character Types/Classes

A TL-X source code file is a text file using UTF-8 character encoding. Characters fall into these mutually-exclusive types:

1. Word: [`a-zA-Z_`]
2. Numeric: [`0-9`]
3. Space: `[ ]`
4. Newline: This is the character sequence used to terminate the file format line. It can be:
   - LF (Line feed, '\n', 0x0A, 10 in decimal)
   - CR (Carriage return, '\r', 0x0D, 13 in decimal)
   - CR followed by LF (CR+LF, '\r\n',0x0D0A)
5. Non-conformant line terminator: This includes all Unicode line terminators other than *newline*:
   - LF (Line feed, U+000A)
   - VT (Vertical Tab, U+000B)
   - FF (Form Feed, U+000C)
   - CR (Carriage Return, U+000D)
   - CR+LF
   - NEL (Next Line, U+0085)
   - LS (Line Separator, U+2028)
   - PS (Paragraph Separator, U+2029)
6. Braces: `()[]{}`
7. Symbols: Everything else that can be produced on a standard keyboard. (`~`!@#$%^&*-+=\|:;"'<>,.?/`)
8. Unrecognized: All other UTF-8 characters.

Additionally, we define the following *character classes* as collections of character types:

1. Line terminator: {newline, non-conformant line terminator}
2. Whitespace: {space, newline, non-conformant line terminator}
3. Conformant whitespace: {space, newline}

4. Non-TL-X: {non-conformant line terminator, unrecognized}
5. TL-X: All characters that are not *non-TL-X*.

Non-TL-X characters are only permitted in *non-TL-X contexts* -- currently, HDL contexts. Their use is restricted only by external specifications or processors, such as HDL specifications and tools. Non-TL-X contexts include HDL regions, HDL_plus regions, and assignment statements outside of signal references.

Note that all TL-X characters, are in the range 0-127, and are therefore equivalent to ASCII text. Therefore, a TL-X file can be an ASCII text file, though, since Non-TL-X characters can be outside the range of ASCII characters, a legal TL-X file is not necessarily ASCII. (Future TL-X versions might allow non-ASCII TL-X characters.)

4. File Format Line

The first line of a TL-X file must identify the TL-X File Format Version and HDL Language, as well as a URL to the language specification. Currently, it must be exactly this:

For TL-Verilog: `\TLV_version 1a: tl-x.org`
For TL-VHDL: `\TLVHDL_version 1a: tl-x.org`
For TL-C: `\TLC_version 1a: tl-x.org`

The newline sequence used on this line defines the newline sequence for the remainder of the file.

5. Code Regions

There are three types of code regions, HDL, HDL_plus, and TL-X.

5.1. HDL Code Region

HDL regions contain native HDL code. No TL-X language extensions are accessible in these regions.

HDL regions begin with a line that begins exactly as follows, where the “`/`” is the line type character:

For TL-Verilog:
`\SV`
For TL-VHDL:
`\VHDL`
For TL-C
`\C`

There may be spaces and HDL-style comments following this region identifier.

The region ends where another code region is started, or with end-of-file.

It is expected that there will be an HDL region prior to any TL-X region that establishes the context for the TL-X code. This includes, primarily, the module interface definition. It might include other context as well. HDL regions and HDL_plus regions should not alter this context. Additionally, this region should provide the main clock. A final HDL region is expected to terminate the module definition.

5.2. HDL_plus Code Region

HDL_plus regions are parsed as a single large block of HDL code with TL-X signal references that are relative to the default top-level TL-X scope. (As described later, this is stage 0 (`@0`) of pipeline "none" (`|none`).)

HDL_plus regions' primarily uses are:

- as an incremental step when translating HDL code to TL-X; transitioning from HDL to HDL_plus can catch parsing issues where the original HDL code contains a syntax that is recognized as a TL-X construct; it then provides a context in which HDL signals can be translated to TL-X pipesignals without altering code structure
- to utilize a HDL constructs that do not have a suitable TL-X replacement

HDL_plus regions begin with a line that begins exactly as follows, where the "`/`" is the line type character:

For TL-Verilog:
`\SV_plus`
For TL-VHDL:
`\VHDL_plus`
For TL-C
`\C_plus`

The region ends when line indentation returns to the top level (to define a new code region).

5.3. TL-X Code Region

TL-X regions begin with a line that begins exactly as follows, where the "`/`" is the line type character:

For TL-Verilog:
`\TLV`
For TL-VHDL:
`\TLVHDL`
For TL-C

```
\C
```

The region ends when line indentation returns to the top level (to define a new code region).

6. Line Type Character

Lines of code in a TLV or SV_plus region begin with a character that defines the line type. Lines with a special line type are generally ones that deserve special attention. Line type chars are:

- <space>: Pure, safe TLV line (aka, nothing special).
- `!`: Impure line. This is an alert to the use of methods that require caution. It may be used on any lines at the designer's discretion, but is required on lines that contain:
    - an HDL signal reference (`*sig_name`) -- because changes to the stage of an assignment statement containing such a line will have an impact on functionality.
    - an HDL macro instantiation (excluding those contained within the left-hand side of an assignment statement) -- because text macros can be used to do all sorts of strange things.
    - a pragma or compiler directive -- because these can alter context should be consistent for all TL-X code in the file.

7. TL-X Identifiers

TL-X defines a rich set of identifier types, each with their own namespace. Though cryptic at first, once these fundamental language constructs become familiar they convey meaning very succinctly, without the need for naming conventions (that are never followed consistently).

Identifiers begin with a type prefix containing zero or more symbol characters. If the next character is numeric, the identifier is a *numeric identifier*. If the next character is alphabetic it starts the name of a *named identifier* -- either a *delimited identifier*, or a *mixed-case identifier*, determined by the prefix. (Name syntax is defined in 7.1. Delimited Identifier Names and 7.2. Mixed-Case Identifier Names). If the next character is a whitespace or brace (a non-TL-X character would be illegal), terminating the identifier, the identifier is an *operator*, though no operators are defined in TL-X 1a. The identifier's *type* is defined by its syntax, which includes the prefix and the delimitation style (for delimited identifiers) or the numeric style (for numeric identifiers). Some named identifiers types are reserved for *keywords*, defined by TL-X in 7.4. Identifier Types. This way, keywords do not interfere with the namespace of user-defined names. There is a single syntax-to-type translation that is consistent in all contexts where identifiers are processed, defined in 7.4. Identifier Types.

7.1. Delimited Identifier Names

A delimited identifier name is a string comprised of delimited alphanumeric tokens. A token is a caseless string of alphabetic characters optionally followed by any number of numeric characters. The first token must begin with a minimum of two alphabetic characters. There are four methods of delimitation, distinguished by case of these first two alphabetic characters. The four identifiers below have the same tokens but different delimitation style:

- delimited_identifier // Lower-case
- DelimitedIdentifier // Camel-case
- DELIMITED_IDENTIFIER // Upper-case
- dELIMITEDiDENTIFIER // Reverse camel-case (Not used)

Most names in TL-X are delimited identifiers, including those that translate to HDL. The restriction on name syntax allows names to carry over directly to HDL with additional available HDL namespace for other generated/derived names.

7.2. Mixed-Case Identifier Names

Some prefixes expect mixed-case identifiers. Mixed case identifier names can contain any alpha-numeric characters of either case plus '_'. They begin with an alphabetic character (of either case).

Mixed case identifiers generally do not transfer into HDL, as doing so would consume the entire HDL namespace.

7.3. Numeric Identifiers

Numeric identifiers are ones whose prefix character(s) (if any) are followed by a *numeric string* which begins with a decimal digit. However, if that decimal digit is preceded by “+” or “-”, the “+”/“-” is taken to be part of the numeric string. To avoid ambiguity, numeric identifier prefixes must not end with “+”/”-”. Currently the numeric string consists entirely of the optional “+”/”-” followed by decimal digits, to define a decimal number.

7.4. Identifier Types

TL-X identifier types are as follows. These are described subsequently.

Identifiers with the following prefixes utilize mixed case:

- `\`
- `*`
- `**`
- `^`

Keywords (which have syntaxes: `\keywords` (mixed case); and `$KEYWORDS` (upper case)):

- `\SV_plus`, `\VHDL_plus`, and `\C_plus` (in TL-Verilog, TL-VHDL, and TL-C, respectively) (mixed case) (See 5.2. HDL_plus Code Region.)
- `\source` (mixed case)
- `$RETAIN` (See 19.5. $RETAIN.)

Scope:

- `|pipeline`
- `/beh_hier`
- `?$when`

- `?$When`
- `@1` (non-negative pipe stage)
- `@-1` (negative pipe stage)
- `@++`
- `@+=1`
- `@+=-1`

Signals:

- `$pipesignal`
- `$$produced_signal`
- `$StateSignal`
- `*HDL_signal` (Mixed case.)
- `**HDL_type` (Mixed case.)

Other:

- `>>2` (Ahead reference. See 19.4. Pipesignal References.)
- `<<2` (Behind reference. See 19.4. Pipesignal References.)
- `<>0` (Naturally-aligned reference. See 19.4. Pipesignal References.)
- `^attribute` (mixed case)

8. Ranges and Indices

Behavioral scope indices and signal bit indices appear in range/index expressions in one of the following formats:
`[<expr>:<expr>]`: Range
`[{<expr>:<expr>}]`: Subset range
`[<expr>]`: Index
`[*]`: Complete range as defined elsewhere

`<expr>` use HDL syntax and may contain HDL constants. They may also contain TL-X signal references in situations where the HDL permits the use of signals.

The HDL-context escape character "`\`" can be used to avoid special parsing of a character. For example, "`\:`" or "`\]`".

TL-X processors may provide checking and optimization for the generated HDL if they can evaluate the expressions. For example, if only some bits of a pipesignal are required in a later pipestage, a TL-X processor could stage only those that are required.

9. Scope

9.1. Indentation

The use of whitespace is meaningful in TL-X. A single level of scope is conveyed by indentation of three spaces (or, in the case of the first level of indentation, the line type character followed by two spaces). Tabs are forbidden in TL-X regions. (One should beware of one's editor's auto-indentation mode.)

Using specific indentation to convey scope, rather than using begin/end delimiters enforces consistent coding style and avoids inconsistencies between indentation and scope.

9.2. Scope Lines

Every level of scope is introduced by a scope line containing an identifier, sometimes followed immediately (no whitespace) by a range and, optionally, (conformant) whitespace and an HDL-style single- or multi-line comment.

9.3. Reentrance

Scope in TL-X is termed *reentrant*, in the sense that the flow of the code can define logic in a particular scope, leave that scope, and reenter that scope to define more logic. This is unlike HDLs, where a module is defined contiguously, and repeating a similar loop results in different loop scopes. For scopes classes that are reentrant we say that multiple *lexical scopes* can define a single *logical scope*.

9.4. Behavioral Scope

Some scope identifiers are categorized as *behavioral scope* identifiers. Behavioral scope provides hierarchy for a TL-X model. It includes *behavioral hierarchy* and *pipelines*. The full path to a pipesignal includes all levels of behavioral scope and does not include other scope. (See 18. Pipesignal References.)

9.5. Scope Summary

This table summarizes scope classes.

Scope	**Range**	**Nesting Requirements**	**Behavioral Scope**	**Reentrant**	**Section**
Behavioral Hierarchy	none `[max:min]` `[{sub-max:sub-min}`	Any	Yes	Yes	10. Behavioral Hierarchy Scope

	`]` `[*]`				
Pipeline	none	Exactly one required	Yes	Yes	11. Pipeline Scope
Pipestage	none	Exactly one required, below pipeline	No	Yes	12. Pipestage Scope
When	none	Any	No	Yes	13. When Scope
Source	none	Any	No	N/A	14. Source Scope

10. Behavioral Hierarchy Scope

Behavioral hierarchy is a form of behavioral scope, and therefore provides a level of hierarchy with its own namespace. It also can enables replication of logic. Replication of the scope is specified as a range. The following forms are permitted:

1. `/my_hier` : Behavioral hierarchy namespace scope (no replication).
2. `/my_hier[<expr>:<expr>]` : Behavioral hierarchy replication scope.
3. `/my_hier[{<expr>:<expr>}]` : Behavioral hierarchy replication scope that is a subset of the full scope.
4. `/my_hier[*]` : Wildcarded behavioral hierarchy replication scope.

The lexical range expressions for a logical behavioral scope must define a single range for the logical scope. The lexical scopes must all specify a range, or none of them may. If ranges are provided, one must specify the complete scope (form 1). Others must be identical (by string compare), or be `[*]`, (form 4), or specify a subset range (form 3).

A behavioral hierarchy identifier must be distinct from identifiers of all parent levels of behavioral hierarchy.

11. Pipeline Scope

Pipeline scope (`|my_pipe`) is a form of behavioral scope, and therefore provides a level of hierarchy with its own namespace. Pipeline scope cannot be replicated, so there can be no range expression. All logic in TL-X is defined within a pipeline. Therefore, there must be exactly one pipeline scope for every assignment statement.

12. Pipestage Scope

All logic in TL-X is defined within the context of a pipeline and a pipestage within that pipeline. Therefore, there must be exactly one pipestage scope for every assignment statement, and that pipestage scope must be within a pipeline scope. The following forms of pipestage scope are permitted:

- `@<number>`: Positive pipeline stage. Const-expr is a decimal number.
- `@-<number>`: Negative pipeline stage. Const-expr is a decimal number.

13. When Scope

A “when statement” provides a condition under which the contained assignment statements produce valid results. When forms are:

- `?$valid_sig`: Conditioned on the pipesignal `$valid_sig`, which must be assigned within the scope of this when statement.
- `?*HDL_signal`: Conditioned on the HDL signal `HDL_signal`.

The condition signal must be a single-bit (boolean) signal in the same behavioral scope as the when statement, which, when asserted, indicates validity. When not asserted, the values that would be produced by the assignments under the when scope are “invalid” and may not be consumed *[the definition of consumed is under debate]*.

An assignment’s when scope is defined by all when scope levels within which the assignment is contained. If the condition is a pipesignal, that pipesignal’s when scope must be the same, or a subscope of that of the when statement (in addition to being within the same behavioral hierarchy level).

14. Source Scope

Source scope is provided as an aid to situations where TL-X code is generated by a preprocessor. TL-X.org is intentionally avoiding the temptation to aggressively specify new language features for every situation. Simplicity is golden, and legacy is an anchor. Many of the features that are not yet natively supported in TL-X, can be accomplished with standard macro preprocessors, such as M4. Even with a fully-featured language, it is impossible to plan for every situation, and there are always times when extending capabilities using a preprocessor is the best answer.

The primary difficulty presented by the use of a preprocessor (as with the use of a TL-X processor) is the fact that downstream tools are unaware of the source code and cannot relate error messages and the like back to the source code. To help, TL-X supports the \source construct:

```
\source <file> <line>
```

This is a scope line that indicates that this line and its contained TLV code came from the given source file and line number. To be precise, the line number identifies the source line that produced the "\source" line. These lines are not counted when reporting line numbers in error messages. It is expected that the line that ends this scope is the first that is no longer from the specified source file.

There can be multiple levels of `\source` scope, enabling multiple levels of preprocessor macro instantiation. Well-formed error messages will report all levels of instantiation.

15. Assignment Statements/Blocks

Assignment statements and *assignment blocks* (or just "assignments") are the leaves of the parse tree. They define logic. They utilize HDL syntax for logic expressions, but use TL-X signal references (`$sig`) in place of HDL signals. A TL-X processor need not parse the HDL expression syntax.

All assignment statements begin with a "`$`", "`%`", or "`*`" character. Assignments continue on subsequent lines with deeper indentation (any number of additional spaces of indentation; not necessarily three spaces). Assignment *blocks* begin on a line with a keyword that has a "`\`" prefix and continue on subsequent lines with a minimum of three spaces of additional indentation.

The following assignment forms are permitted:

1. Common form: "`$foo = $bar;`" (or "`$foo <= $bar;`" for TL-VHDL)
2. HDL_plus form:
   ```
   \HDL_plus
      <HDL code with pipesignal references>
   ```
3. Sequential form (TL-V only):
   ```
   \always_comb
      <SystemVerilog always_comb body with pipesignal references>
   ```

15.1. HDL Parsing

The HDL-context escape character "`\`" can be used within assignment statements to avoid special parsing of a character. For example, "`$display`" in SystemVerilog context might be undesirably interpreted as a pipesignal, so "`\$display`" can be used for the SystemVerilog display task and "`\%d`" for print formatting.

15.2. Common Assignment Form

The common assignment form (form 1.) has an explicit left-hand side. Signal references on the left-hand side are recognized as assigned without the need for "`$$`". The line begins with "`$`" or

"*". If it begins with "*", an HDL type declaration is expected as in 19.2. HDL Signal/Structure Type Support, followed by a single space, prior to the "$" of the assigned signal. "=" ("<=" for TL-VHDL) with conformant whitespace on both sides separates left-hand and right-hand sides. The left-hand and right-hand sides are interpreted as HDL syntax. "\" is an escape character that is dropped in translation and causes the next character to be interpreted as an HDL character, so it will not be interpreted as the start of a TL-X reference.

For TL-Verilog, "`assign`", "`always_comb`", "`begin`", and "`end`" are not required.

15.3. HDL_plus Blocks

The HDL_plus assignment form (form 2.) can be used for any HDL expressions. Lines of the body must be indented at least three additional spaces. The body is simply a block of HDL code that can contain TL-X signal references. It does not inherently distinguish produced signals from consumed signals, so produced signals must have a "$$" prefix.

15.4. Sequential Blocks (\always_comb)

Sequential, or always_comb blocks are supported in TL-Verilog only. They provide a code block with sequential execution semantics. They are a simple shorthand for a `\TLV_plus` block containing an `always_comb` block. The TL-Verilog code:

```
\always_comb
   expression1;
   expression2;
```

is equivalent to:

```
\TLV_plus
   always_comb begin
      expression1;
      expression2;
   end
```

16.1. References

A reference is a string of identifiers with no delimitation between identifiers. A reference refers to an element of the design -- most commonly, a pipesignal. It includes the scope of the design element, relative to the scope of the reference. Note that identifier types with no prefix characters (which don't currently exist) are not legal in references. The same is true of operators. Signal references are used within assignment statements, defined below.

16. Using Signals

16.1. Signal Identifier Types

These signal types are supported within logic expressions. (See 15. Assignment Statements/Blocks.)

- `$my_sig` : Pipesignal. Pipesignals can be thought of as fields of a transaction. The type of a pipesignal can be a single bit (with no range), a vector of bits (with a range), or an HDL type (see 19.2. HDL Signal/Structure Type Support). Pipesignals are invalid when any of their "when" scopes are not true. This means they cannot be consumed by a valid statement..
- `$MyState` : State signal. State signals hold values between transactions. These retain their state over invalid transactions, or specifically, when any containing "when" clauses are not all true. They should generally be assigned a value upon reset. State signals must be assigned with an explicit alignment or by `<=`. (See 15. Assignment Statements/Blocks.)
- `*MyHDL_sig` : An HDL signal. A line containing an HDL signal reference must be marked as impure, because it cannot be safely retimed (see 6. Line Type Character).
- `$$dest_sig` : A pipesignal that is explicitly an assigned signal. Use of these identifiers is not required for common form assignments but is required in other assignments. (See 19.3. Assigned and Used Signals.)

16.2. HDL Signal/Structure Type Support

Pipesignals can have datatypes from the targeted HDL, either user-defined structure types or built-in HDL types. The type of a pipesignal may be declared in a type declaration statement, or along with an assignment. The HDL type is prefixed with "`**`". A type declaration statement may be outside of stage scope, but must be within a pipeline.

Before declaring an instance with an HDL type within TL-X context, the type must be defined within HDL context. For example,

```
\SV
   typedef struct {
     node field1;
     node field2;
   } struct_name;
```

A type declaration statement begins with an HDL type identifier, then the pipesignal, followed by a semicolon, as follows:

```
\TLV
   |Pipeline
```

```
      **struct_name $object_name; // type declaration statement
```

The type can also be declared within a common form assignment (see 16. Common Assignment Form), as follows:

```
\TLV
   |Pipeline
      @2
         **struct_name $object_name.field1 = ...;  // assignment
statement
         $object_name.field2 = ...;  // assignment statement
```

Note: Text after `$object_name` above is interpreted as HDL-syntax. Since `$object_name[..]` would be a reference to a bit-range, `$object_name\[..\]` is required for some HDLs to index into an array type.

16.3. Assigned and Used Signals

A TL-X processor must be able to determine which pipesignals are produced (aka assigned) and which are consumed (aka used). (The distinction is not necessary for HDL signal references (`*HDL_sig`).) The signal identifier `$$sig`, explicitly references `$sig` as assigned. This syntax must be used for all assigned signals in all assignment forms except the common form (form 1.) (see 15. Assignment Statements/Blocks).

There must generally be exactly one assignment for every pipesignal. In situations where a pipesignal must be used in multiple places as an assignment, one of those places can be chosen to represent the assignment, and the others can safely be treated as uses. Within an assignment block multiple assignments to the same signal may be used, but each must specify the same signal type.

Generally, the single assignment (or in rare cases, multiple assignments) specifies the type of the signal. The exception to this rule is HDL types, which cannot be specified within assignment blocks, and can be specified separately. (See 19.2. HDL Signal/Structure Type Support.) The type can be specified as an HDL type within common form assignments, or in any context, a bit range may be provided immediately following the assigned signal identifier (with no intervening whitespace). If no HDL type or range is specified, the signal is a single bit.

16.4. Pipesignal References

Pipesignal references are used within assignment statements. A simple signal reference is:
`$sig`.
This refers to a pipesignal in the same scope.

A reference to a pipesignal produced in a different scope might look like:

`/scope|pipe/instr[2]<<2$addr[12:6]`

The path (`/scope|pipe/instr[2]`) identifies the behavioral scope instances of the reference. The first identifier of the path (if the path is not empty) refers to any parent hierarchy level of the assignment's scope (each of which is required to have a unique identifier) or to a child scope. Top level scope has an implicit identifier "`/top`". Subsequent path identifiers identify child scope instances. A replicated behavioral hierarchy level may be provided an index expression. If none is provided, the corresponding index of a same-named scope on the ancestry of the assignment's scope is assumed. In other words, if `/scope` is replicated, `/scope` refers to `/scope[#scope]`.

The right-most scope index(es) may be given as wildcards (`[*]`). This will reference the concatenation of all instances. This is most useful for bitwise reduction operators (`& /inst[*]$bit`) or for providing inputs to HDL modules that expect packed bit masks.

Assigned (left-hand, or `$$sig`) pipesignals have the scope of the assignment and must have an empty path.

`<<2`, `>>3`, and `<>0` are examples of pipeline alignments. They are used in references to specify the numeric pipestage of the reference relative to the assignment's stage scope. For references within a pipeline, `<<` is used to reference a pipesignal of a transaction that is *behind* the transaction of the assignment statement. Likewise, `>>` is used to reference *ahead*. A natural or zero alignment is assumed for references within a pipeline (into a pipeline by the same name) if no alignment is specified. For cross-pipeline references, the terms "ahead" and "behind" have less meaning, but the operators are used just the same to indicate a relative numeric stage offset, and an explicit alignment is required.

The waterfall diagram below, illustrates a reference, in stage 2, to `>>1`, the previous transaction, which is in stage 3.

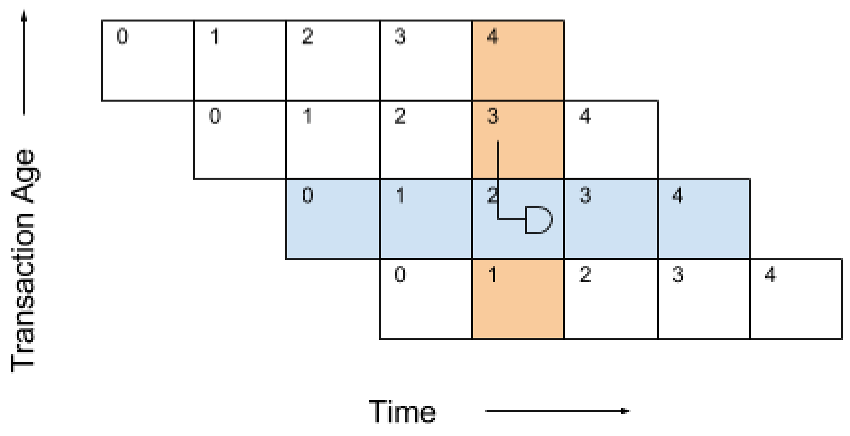

Figure 1: Waterfall Diagram

The bit range `[12:6]` may be interpreted by a TL-X processor to determine what bits must be made available in the consuming stage. The TL-X processor may provide more bits than necessary (though logic synthesis typically does an excellent job of pruning unused logic). This can be expected when non-numeric HDL expressions are used in the bit range.

16.5. $RETAIN

`$RETAIN` can be used as a used signal in an assignment. It translates to the single destination signal delayed by one cycle. It can be used to retain/recirculate the previous value of the signal.

17. What's Missing

- Parameterized/generic logic support
- Compile-time conditionals (Eg: this logic exists in instance 0 only)
- Special types for alignments and ranges

18. Comments

Comments use HDL comment syntax. They may appear within assignment statements and blocks or after scope identifiers (together with their range). Scope identifiers must begin their line, after indentation, so it is not legal to have a scope identifier after a multi-line-style comment that is terminated on the same line. Comments are otherwise ignored from the standpoint of indentation. Comments may begin anywhere in their line, including at the line-type character.

19. Acknowledgements

TL-X evolved from an open-source development effort from Intel Corporation, led by Steve Hoover, with significant contributions from Yura Pyatnychko.