Visual Debug

User Guide (Draft)

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

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

LLC SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF REDWOOD EDA, LLC HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Redwood EDA, LLC or other parties. No one is permitted to use these Marks without the prior written consent of Redwood EDA, LLC or the owner of the Mark, as applicable. The use herein of a third party Mark is not an attempt to indicate Redwood EDA, LLC as a source of a product, but is intended to indicate a product from, or associated with, a particular third party.

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

Table of Contents

Introduction

Overview

Redwood EDA's tool suite supports Visual Debug (or VIZ) capabilities that put engineers in control of their debug experience. Compatible with any major hardware description language (HDL) and design environment, Visual Debug enables logic designers, verification engineers, and other users to create custom visualization that best represents the state and simulation behavior of a given model. This is useful for engineers as well as their various customers.

Compatibility

Visual Debug works with any HDL and tool suite capable of producing industry-standard Value Change Dump (`.vcd`) trace files. Trace files must contain a single global clock signal, called `clk` and a single global reset signal called `reset` that provide a notion of time for the visualization.

Versioning

This specification is in Draft form. It is made available to support collaboration and research. It is currently maintained along with software updates without versioning. API changes can be expected that are not backward compatible. Versioning and support for commercial users is currently on a case-by-case basis. You may contribute to this specification as described in the Issues section.

Availability

Visual Debug is available commercially via redwoodeda.com/software.

For open-source Verilog/SystemVerilog/TL-Verilog development, Visual Debug is freely available online, on an as-is basis, at makerchip.com.

Document Conventions

This document includes features that are not yet implemented. These are described with gray highlighting.

Section references are not currently maintained. They appear as XXX.

Use Models

It is generally recommended to develop visualization in tandem with the development of logic to streamline development. The utility of Visual Debug continues throughout the design process as a useful form of documentation and as an aid in daily collaboration and design handoff.

Visualizations can be encapsulated with soft IP. They can be used to illustrate the operation of the IP to potential customers and consumers, and they can be delivered together with the IP to ease the handoff. IP visualization is easily embedded within the visualization of surrounding logic in a hierarchy of visual components.

VIZ can be used with any HDL, and it has additional benefits when used with TL-Verilog. These two use models are described in separate sections, XXX and XXX. Each section stands alone to provide complete information in the absence of the other.

Visual Debug for Any HDL Model

This section explains the use of Visual Debug with any HDL. While VIZ utilizes some syntax of TL-Verilog, this section is written for users who may be unfamiliar with TL-Verilog syntax. TL-Verilog developers can instead refer to section XXX.

Tool Flow

Visualization code is encapsulated in an extension of TL-Verilog file format and thus uses a `.tlv` file extension. This extended TL-Verilog file format is processed by Redwood EDA, LLC tools.

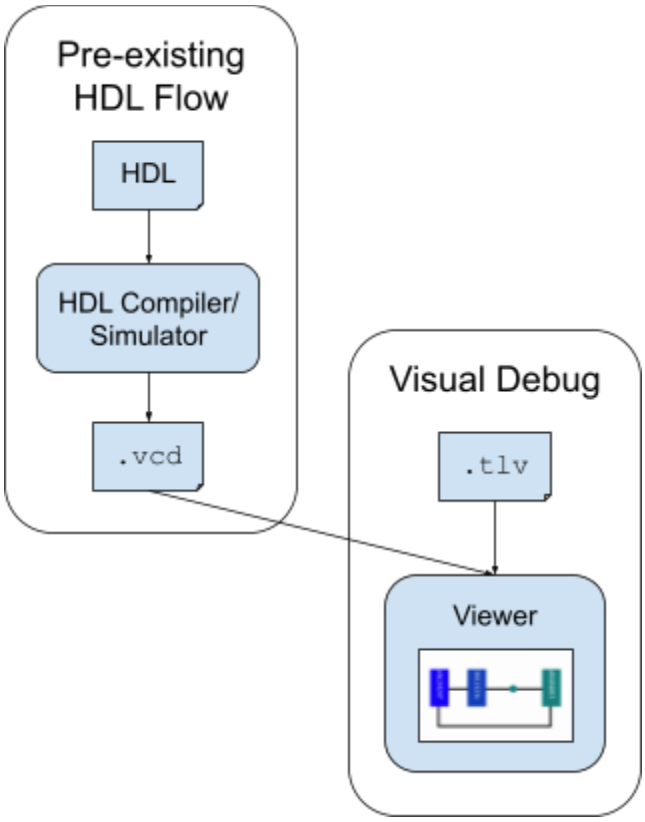

Figure XXX. VIZ flow

Figure XXX shows a pre-existing HDL tool flow augmented with VIZ. Here, the `.tlv` source file is assumed to contain VIZ code only with no TL-Verilog hardware logic. The Viewer (of SandStorm™ or Makerchip) represents data from the `.vcd` trace file according to the VIZ code. Users navigate the visualization, generally by zooming, panning, and stepping through simulation time.

Simple VIZ `.tlv` File Example

In its simplest usage model, the `.tlv` file contains a single *visual description*, expressed as JavaScript code. The following example conveys a single byte value from an on-chip thermal sensor to represent the heat measurement as a circle ranging from black (zero) to red (255).

```
\m5_TLV_version 1d: tl-x.org
\SV
\TLV
   \viz_js
      template: {dot: ["Circle", {radius: 2, fill: "black"}]},
      render() {
```

```
      let heat = this.sigVal("sensor1.heat").asInt()
      let dot = this.getObjects().dot
      dot.set("fill", `#${heat.toString(16).padStart(2, "0")}0000`)
   }
```

This visual description is embedded as a `\viz_js` block within a TL-Verilog file, and as such the file contains a bit of TL-Verilog structure, as described below.

`\m5_TLV_version 1d: tl-x.org`:
This first line of the above code is a "magic number" defining the file format. It is required to include a link to the specification of the file format (`tl-x.org`). While you can follow this link for full details on the TL-Verilog file format for logic modeling, the portions of that syntax that are relevant to VIZ are described here. VIZ language features are a language extension and are not included in the open TL-Verilog standard.

`\SV`:
This second line enters SystemVerilog context. No content is required within this context, but this line is (currently) required (for reasons related to Verilog translation, even though no Verilog is produced in this use model).

`\TLV`:
This line enters TL-Verilog context (where visualization can be defined).

`\viz_js`:
This line begins the `\viz_js` block containing a visual description expressed essentially in JavaScript. This code references signal values from the simulation trace.

In this example, the `template` property provides a circle element on the canvas, and each time the user changes the current cycle of the view, `render()` colors this circle based on the value of a signal `heat` in the top-level module `sensor1`. This signal value is accessed by `this.sigVal("sensor1.heat").asInt()`.

The structure of the JavaScript code is described in XXX ("API").

Hierarchy

Integrated circuits contain a significant amount of machine state and often a deep hierarchy of components. Visualization can correspondingly contain a significant number of visual elements. It is helpful to organize visualization into a hierarchy of visual descriptions, each defining *visual components* that correspond to components of the HDL model.

The following example provides a 20x20 grid of cells that are colored according to the one-bit signal `grid_y[<Y-index>].alive(<X-index>)`.

```
\TLV
   /yy[3:0]
      \viz_js
         where: {layout: "vertical"}
      /xx[3:0]
         \viz_js
            where: {layout: "horizontal"},
            renderFill() {
               let alive_sig_name =
                   `grid_y[this.getIndex("yy")]` +
                   `.alive(this.getIndex())`
               let alive =
                   this.sigVal(alive_sig_name).asBool()
               return alive ? "blue" : "gray";
            }
```

Figure XXX. Example VIZ code

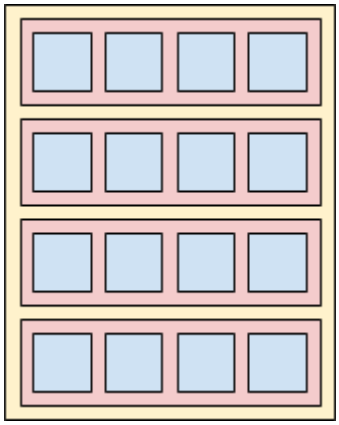

Figure XXX. Example VIZ layout

In Figure XXX, `/yy` and `/xx` are hierarchy levels that create the 4x4 grid of cells depicted in Fig. XXX (with no gap between cells). While JavaScript for loops could have been used for this purpose, using this syntax provides the following benefits:

- simpler (declarative) syntax
- clear organization of the hierarchy
- localized coordinate systems for each visual component
- semi-automated layout
- enabling of rendering optimizations provided by the VIZ framework

Note that this form of hierarchy is actually TL-Verilog design hierarchy. Visual descriptions can be integrated with TL-Verilog logic, providing further benefits for TL-Verilog designs, as covered in XXX.

Each level of indentation up to the `\viz_js` *identifier* must be three spaces (no tabs), and the visual description code must be indented any number of spaces beneath the `\viz_js` identifier. Hierarchy levels without replication ranges (e.g. `[3:0]`) are also permitted and can be useful to partition visual descriptions.

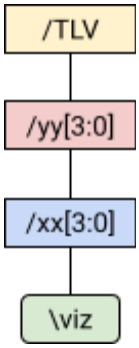

Figure XXX. Definition hierarchy

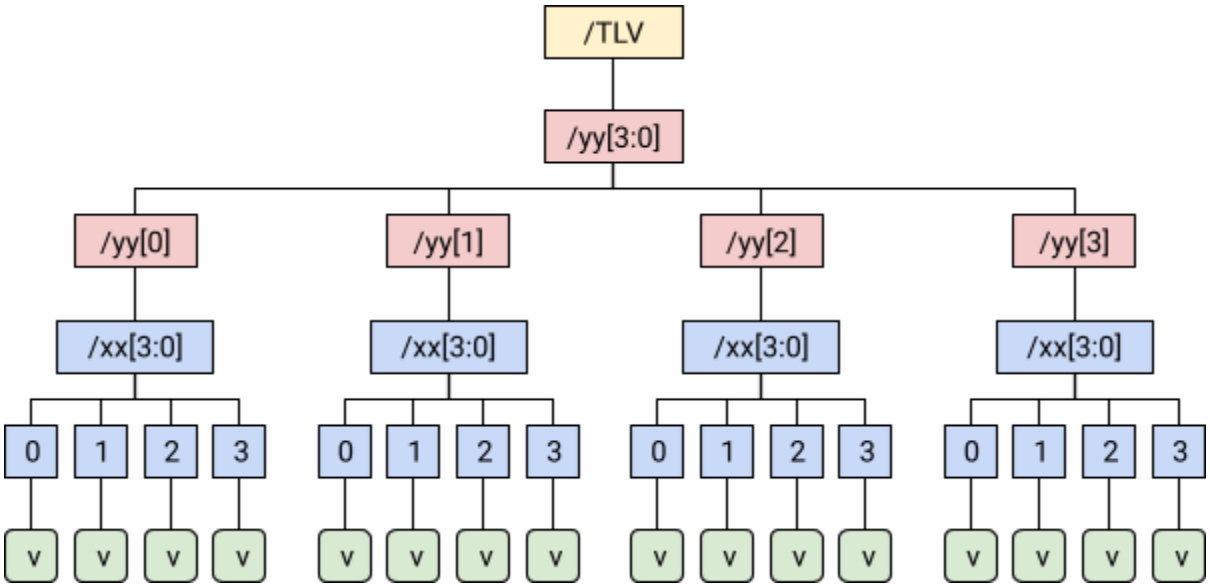

Figure XXX. Instance hierarchy

The hierarchy of the source code, depicted in Fig. XXX, is referred to as the *definition hierarchy*. Similar to the elaboration of HDL for simulation or synthesis, the visual descriptions are also elaborated into a hierarchy of instances. This is depicted in Fig. XXX and can also be viewed as Fig. XXX ("Example VIZ layout"). This hierarchy is referred to as the *elaborated hierarchy* or *instance hierarchy*.

Modularity and Reuse

Visualization can be encapsulated within reusable components, such as:

```
\TLV my_component(/_name, _where, _base_name)
   /_name
      \viz_js
         where: { _where },
         render() {
            ...this.sigVal(_base_name + "sig_name")...
          }
```

This component can then be instantiated multiple times in various contexts, such as:

```
\TLV
   m4+my_component(/foo, ['top: 20, left: 20'], `top.`)
   /child[1:0]
      m4+my_component(/bar, ,
                      `top.child[${this.getScope("child").index}`)
```

Parameters:

The `/_name` parameter is used to provide a named context for the visualization.

The `_where` parameter is used to specify the placement of `my_component`'s `\viz_js` within its parent `\viz_js`, as described in XXX.

The `_base_name` parameter is used to pass the SystemVerilog scope of `my_component` as a signal prefix string.

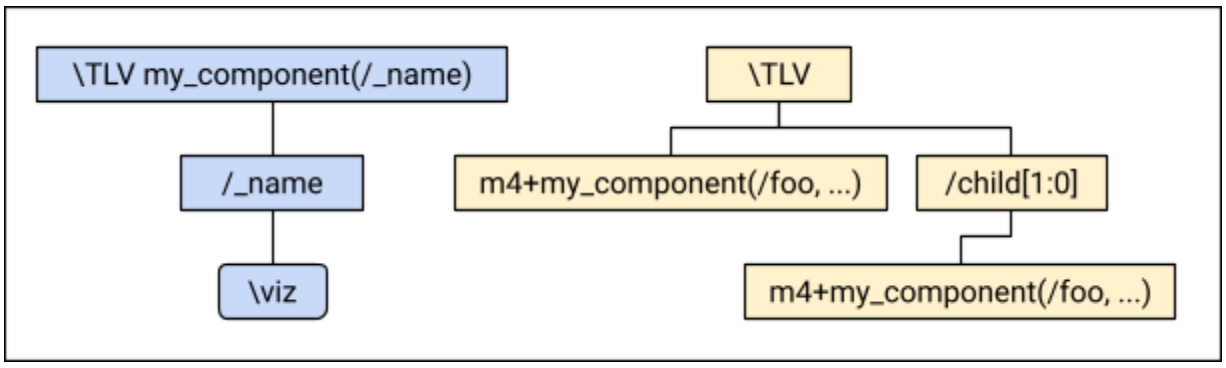

Figure XXX. Definition hierarchy for `my_component` example

The example code above is depicted graphically in Figure XXX.

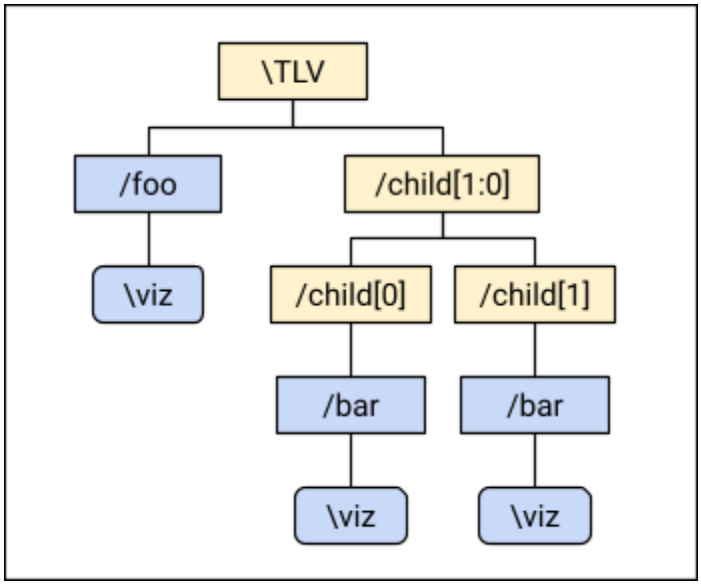

Figure XXX. Instance hierarchy for `my_component` example

The elaboration of this example can be viewed as in Fig. XXX.

This example component is a text macro. M4 macro preprocessing is performed before any other processing as the first step to elaboration. This macro syntax, too, comes from TL-Verilog (extended with M4 macro preprocessing).

A few general notes about M4 use:

- Both `\TLV` blocks are global and are unindented.
- `['']` are M4 quotes. Macro arguments can always be quoted. The quoted string is passed literally to the instantiation. Often the quotes are unnecessary. They are used above to ensure that `['top: 20, left: 20']` is treated as a single argument, rather than as two arguments.
- The M4 use in this example is for multi-line macros. Macros that expand within a line are prefixed with `m4_`. For example, M4 can be useful for defining global constants. `m4_def(three, 3)` defines `m4_three` to substitute as `3`.

The full specification of M4 use for TL-Verilog can be found in XXX.

Soft IP modules are best bundled with VIZ support by providing visualization as a `\TLV` macro definition that can be instantiated within an appropriate context.

Visual Debug for TL-Verilog

Visual Debug is tightly integrated with Redwood EDA's TL-Verilog tools to provide improved abstraction and integration benefits. This section describes the use of Visual Debug with TL-Verilog logic for user's who are already familiar with TL-Verilog, as described in:

- the TL-Verilog specifications at http://tl-x.org
- TL-Verilog M4 use, at XXX

Tool Flow

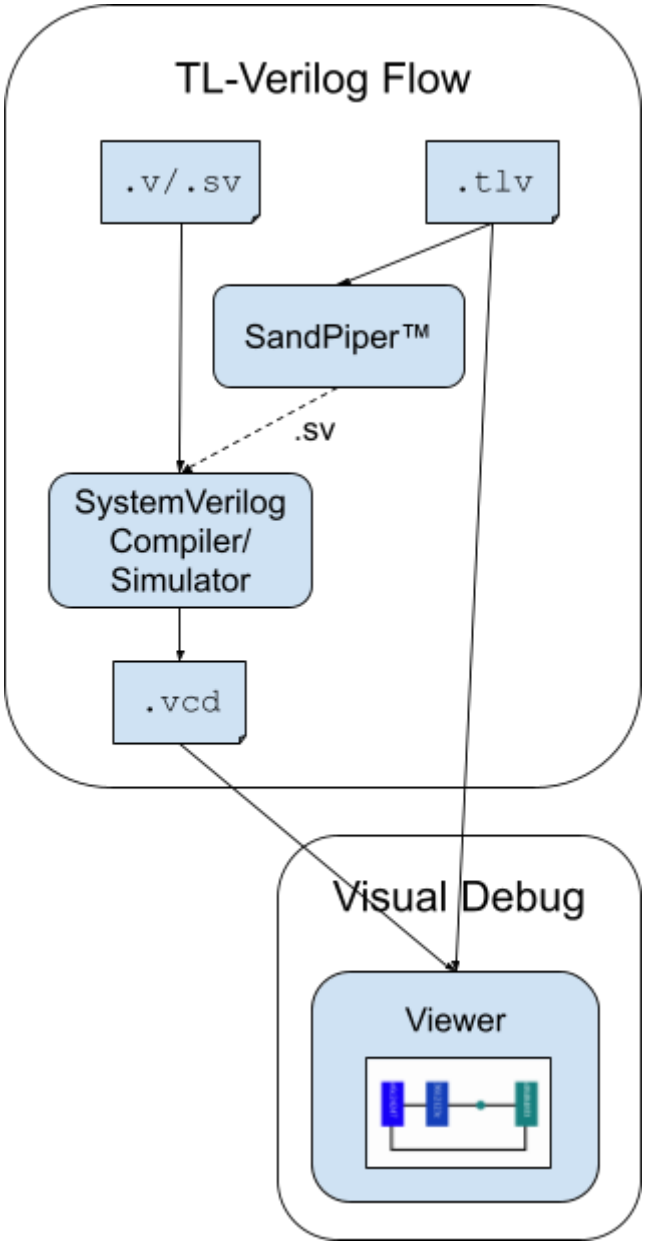

Figure XXX. TL-Verilog Visual Debug Workflow

A TL-Verilog workflow is depicted in Figure XXX. This flow may be entirely encapsulated within Redwood EDA, LLC's SandStorm or Makerchip, or compilation and simulation may be performed externally with Redwood EDA, LLC tools used for debugging.

\viz_js Blocks

Visual Debug is supported within TL-Verilog code through the use of \viz_js blocks. As with other TL-Verilog blocks, \viz_js blocks end with the line before indentation is returned to the level of the \viz_js keyword (or a lesser level).

The structure of the JavaScript code is described in XXX.

Within the JavaScript code of \viz_js blocks, TL-Verilog syntax can be used to reference pipesignals.

TL-Verilog VIZ By Example

```
\TLV
   /yy[3:0]
      \viz_js
         where: {layout: "vertical"}
      /xx[3:0]
         $alive = ...;
         \viz_js
            where: {layout: "horizontal"},
            renderFill() {
               return '$alive'.asBool() ? "blue" : "gray";
            }
```

Figure XXX. Example TL-Verilog VIZ code

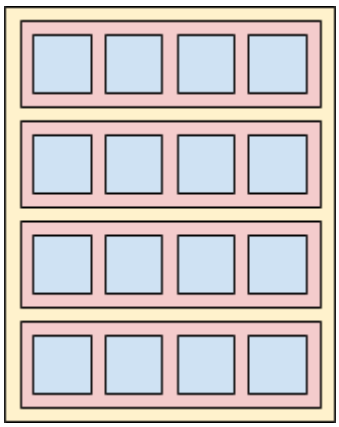

Figure XXX. Example TL-Verilog VIZ layout

Figure XXX shows an example of a design containing a two-dimensional array of cells that compute a one-bit pipesignal `$alive` (for which the expression is not included). The value of alive is represented as either a blue (alive) or gray (not alive) cell. Figure XXX depicts the hierarchy of visual components. This hierarchy corresponds to `/xx` and `/yy` design hierarchy, thus cells are the leaves of the hierarchy. JavaScript attributes define the layout of these cells in a grid (that would not actually have any space between cells).

Within the `\viz_js` block, the value of the `$alive` pipesignal at the current time of the view is referenced as `'$alive'`. Its one-bit value is interpreted as a boolean value by `.asBool()`. Single quotes are used to delimit the TL-Verilog pipesignal value reference.

Since single quotes are used also in m4 quotes (`['` and `']`). care should be taken when M4 processing is (or may at a later time be) enabled. Attempts to use signal values as array indices, like this `['$foo'.asInt()]` would be interpreted by M4 to begin with an open quote. It is good practice to precede pipesignal references with whitespace in general, e.g. `[ '$foo'.asInt() ]`.

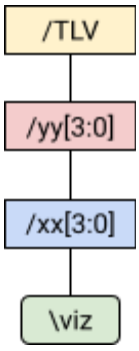

Figure XXX. TL-Verilog VIZ definition hierarchy

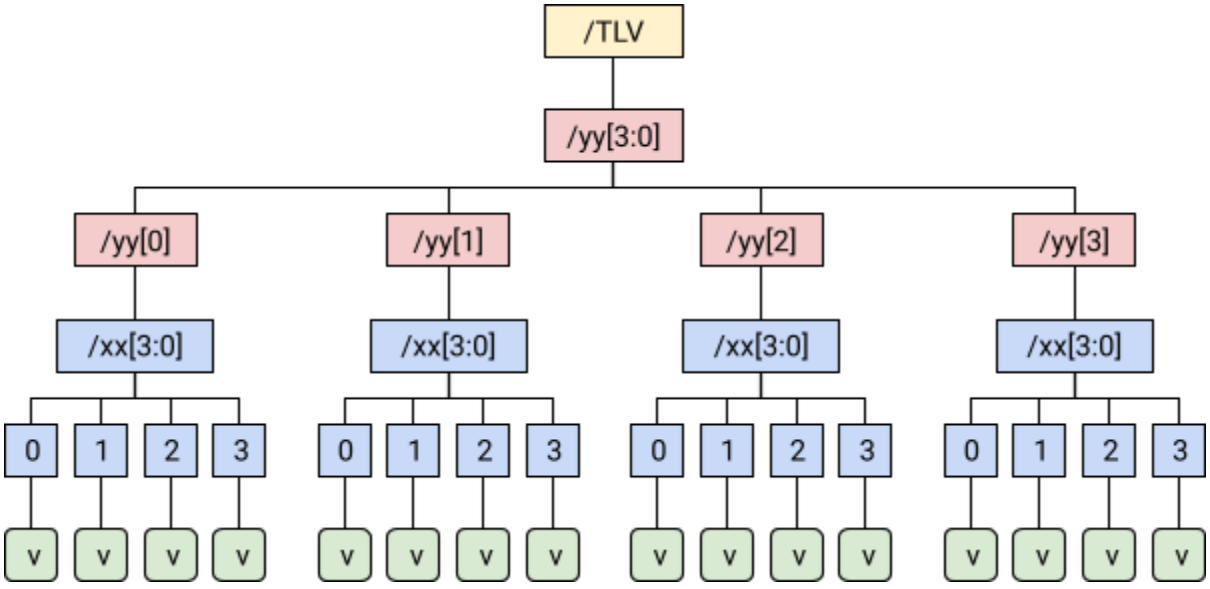

Figure XXX. TL-Verilog VIZ instance hierarchy

The hierarchy of the hardware model provides a default hierarchy for the visual components. The source code hierarchy, depicted in Fig. XXX, is referred to as the *definition hierarchy*. Similar to the elaboration of the TL-Verilog logic for simulation or synthesis, the visual descriptions are also elaborated into a hierarchy of instances. This is depicted in Fig. XXX and can also be viewed as Fig. XXX ("Example TL-Verilog VIZ layout"). This hierarchy is referred to as the *elaborated hierarchy* or *instance hierarchy*.

Signal Value References

Pipesignal value references within the single quotes are much like pipesignal references in logic expressions. In most respects, `\viz_js` blocks are interpreted by tools as logic expressions with no outputs. Hierarchical references may be provided with explicit scope (relative to the scope of the `\viz_js` block) and alignment operators. The differences between references in `\viz_js` blocks versus in logic expressions are:

- Explicit delimitation is required using single quotes.
- It is legal to reference pipesignals in stages that precede their assignments.
- Indexing expressions, such as `x` in `'/foo[x]$sig'` are interpreted as JavaScript expressions.

Note that the use of pipesignal references is only appropriate within `render` and `renderFill` functions.

Integration Benefits

The tight integration and tool support of visualization with TL-Verilog provides a number of benefits. The fact that `\viz_js` blocks are interpreted as a part of the logic network contributes significantly to these benefits, which include:

- Visualization is naturally encapsulated in design blocks for reuse.
- The hardware model hierarchy provides a default hierarchy for visualization.
- TL-Verilog compilation detects and reports uses of unassigned signals in visualization code.
- IDE support for navigating the logic network also provides navigation of `\viz_js` blocks.
- IDE features can associate visualization with corresponding signals and model hierarchy.
- Tools can automatically identify the minimal set of signals needed in trace files for visualization.

API

This section describes the JavaScript API available within `\viz_js` blocks in compatible software versions.

The body of a `\viz_js` block is JavaScript code with the one exception that single quotes have special meaning. As such, only double quotes must be used for literal strings. When copying code from external sources, it is important to replace all single quotes with double quotes.

The visual description JavaScript code is interpreted by SandStorm or Makerchip. An open-source library called Fabric.js provides the drawing canvas and methods to create objects on that canvas.

Though Redwood EDA, LLC makes no warranty of any kind with regard to compatibility between versions and shall not be liable for any damages whatsoever, Redwood EDA, LLC generally aims to maintain support for the API as specified herein between minor versions. Redwood EDA, LLC explicitly does not make any attempt to support properties and methods that are not documented herein. As JavaScript does not distinguish public and private interfaces, other properties and methods may be available beyond this specification that are easy to discover using a web browser debugger. It is important to avoid the use of such properties and functions to reduce the likelihood of forward compatibility issues.

JavaScript Code Structure

This section presents the high-level structure of visual description code. Often visual representations can be created by following other examples. For expert development and when detailed reference documentation is needed, it can be found elsewhere.

- The detailed JavaScript API is documented at XXX.
- Fabric.js documentation can be found at http://fabricjs.com/.
- JavaScript resources are easy to come by.

`\viz_js` Blocks

Every "parse hierarchy", in other words, a `/hier` or `|pipeline` scope, can have at most one `\viz_js` block.

```
\TLV
   /scope[2:0]
      \viz_js
         ...
```

```
/scope[*]
   \viz_js
      ...
```

Figure XXX. Multiple \viz_js blocks for a single logical TL-Verilog scope.

It is however possible to have multiple `\viz_js` blocks for the same "logical hierarchy". Figure XXX shows that this can be achieved by using multiple parse scopes representing the same logical hierarchy. Note that, although moving logic between parse scopes for the same logical hierarchy has no impact on the hardware, moving `\viz_js` blocks can impact the layout of the visualization.

The body of a `\viz_js` block provides the contents of a JavaScript object (as would appear between `{}` in the JavaScript object definition). This JavaScript object may define the properties specified in this section, all of which are optional.

Hierarchy levels may be replicated. A `\viz_js` block for a replicated hierarchy (or one that is written to support a replicated hierarchy) defines two levels in the visual hierarchy, one for the scope and one below that for instances within that scope. The `\viz_js` block defines these one or two components, and their embedding in their parent component. The block, therefore, may contain properties related to up to three coordinate systems--those of the parent, the scope, and the instances. All properties related to the parent are contained within a `where` property. The scope level exists only for replicated blocks, and generally only the layout of its elements is important. Properties related to visualizing the scope, are contained within the `all`, `where0`, and `layout` properties. (`layout` is generally sufficient.) Other properties relate exclusively to the instance in its coordinate system.

A `box` property defines a coordinate system for the component as a bounding box. It may also provide a background `fill` and `stroke` colors. (Some properties of the `box` property can be defined as top-level properties for convenience.) A `template` property defines the visual objects representing the component (within the bounds of the `box`), such as a rectangle containing text. A `render()` function reads simulation state from the trace file and based on this updates properties of the template objects and/or creates additional objects. A `where` property, including `where{top, left}`, specifies the placement within the parent. A `layout` property defines the layout of replicated instances within the scope instance. A `where0` property is like `where`, but for instances.

The API is built to balance simplicity for quick debug and generality for advanced visualization and optimal performance for packaged IP. A simple minimal subset is sufficient for quick debug, and this is presented first, followed by the full API specification.

Minimal API Subset:

- `width, height`: [default: bounds of child components, or `1,1` if none] Height and width of a bounding box defining the coordinate system of an instance, with its upper-left corner at 0,0. Note that for replicated scopes, instances will be laid out vertically if height < width, or horizontally otherwise.
- `fill`, `stroke`: [default: transparent] Colors (e.g. "#808080" for gray) for the background (`fill`) and border (`stroke`) of the `box`.
- `render()`: Called for each instance when rendered for each new cycle. Returns an array of new Fabric.js objects representing this component at this cycle of simulation (ordered bottom to top).
- `where: {`
    - `top`, `left`: [default `0`, `0`] position within parent of this component (including its instances).

  `}`

Full API (alphabetically):

- `all: {...}`: A level in the hierarchy of visual components representing the collection of all instances of this hierarchy level. This is meaningful for replicated hierarchies, but may be present regardless. The contained properties and functions match those of the `\viz_js` body, with the exception that `all, where, and layout` properties are not permitted. Note that `where` specifies the placement of this (perhaps implicit) collection of all instances, whereas `where0` specifies the placement of instance zero.
- `base`: [not yet supported] An object of type `VizComponent`, to which the other properties and functions of the \viz_js block are applied as a mixin. The resulting object is the one defined by this \viz_js block. All functions are invoked in a context in which `super` references this base object. This property enables inheritance.
- `box: {`

  Properties that are approximately those of a bounding background `fabric.Rect`. Notably, these include `width`, `height`, `left`, and `top`, which define the bounds of this component in its own coordinate system. `originX/Y`, `scaleX/Y` and `angle` properties are not permitted. (`where` can be used to position this box with angle and scale.) A corresponding `fabric.Rect` is added as the lowest-layered object of the component. Note that several properties of the box can also be defined as top-level properties (not contained within `box`) as a shorthand, though definitions within `box` are preferred.

  The bounds of this component are defined by this `box`, if provided, or by the tightest containing bounds of `template` and `init` objects and `box`, `template` and `init` objects of all child components.

    - `left`, `top`, `width`, `height`: [default: containing bounds of template objects, init objects, and child components' `where` areas, with a minimum `width` and `height` of `1,1`] Height and width of a bounding box defining the coordinate system of an instance. Note that `width` and `height` determine the default `layout`. Also note that the `width` and `height` of the corresponding `fabric.Rect` may vary from the box `width` and `height` in order to contain the stroke within these bounds. Providing explicit bounds can improve rendering performance. Objects and child components must fit within these bounds to avoid unpredictable rendering behavior and object placement.
    - `fill`, `stroke`, `strokeWidth`: [defaults: `"transparent"`, `"#808080"` (translucent gray), (see description)] Colors (e.g. `"#FF0000"`, "`rgb(255, 0, 0)`", "`red`") for the background (`fill`) and border (`stroke`) of the box. `strokeWidth` is limited to 1/3 the box `height` and `width`. `strokeWidth` defaults to `1` with two exceptions. If `fill` is given and `stroke` is not `strokeWidth` defaults to `0`. Also, levels of the hierarchy with no explicit `\viz_js` or `all` definition will have a box with a `strokeWidth` of `0` (and will thus be fully transparent).

  Some subtle adjustments are made to these `box` properties versus the ones that are actually provided to the corresponding `fabric.Rect`.

  - The stroke (border line) of a `fabric.Rect` affects its width and height. The stroke is drawn above, along, and centered to the border of the rectangle defined by the properties provided to the constructor. The `getWidth()` and `getHeight()` of the `fabric.Rect`, as well as its `left`, `top` properties, include this extra half `strokeWidth` on all sides. The box's `fabric.Rect` is constructed such that the stroke is fully contained within the given `width` and `height`.
  - The default `fill` and `strokeWidth` may differ.
  - Properties can be provided as functions with no arguments that return the property value. These functions can access properties of this scope and its descendants via `this.getVizBlock()`(?).

  Non-positional box properties can be modified in the `render()` function based on the trace data by accessing the `fabric.Rect` using `this.getObjects().box` or `this.getBox()`. Properties `left`, `top`, `width`, `height`, `strokeWidth`, and `fill`, however, should not be modified in `render()`. Properties of `fabric.Rect` that can be set in `render()` include: `stroke`, various properties of the stroke, not including `strokeWidth`), `rx` and `ry` (rounded corners), etc. These are documented here.

  `}`

- `dynamicSigs() {...}`: Used to identify pipesignals that might be used by this component dynamically. In other words, signals that might be used, but for which there is

no single-quoted static reference. Such a static reference can be used in this function. This function will never be called, but the static reference here conveys the fact that this visual component requires it.

- `fill`: an alternative for `box.fill`.
- `init()`: Called once for each instance, with no access to trace data. Returns a JavaScript object of Fabric.js objects to add to this component. These objects are made available to other functions via `this.getObjects()`.
- `height`: an alternative for `box.height`.
- `layout: [`default: `"vertical"` if `height < width`, `"horizontal"` if `height > width`, unpredictable if `height == width]` For replicated instances, this defines how the instances are laid out. `"horizontal"`/`"vertical"` or
    - `{left, top, angle}`: [defaults `{left: 0, top: 0, angle: 0}`] Each an offset value relative to the previous element, or a function of `(index)` providing the offset relative to index zero (which is not required to exist). `left` and `top` should both be given or neither.

  Note that:
  `"horizontal"` corresponds to `{left: box.width}`.
  `"vertical"` corresponds to `{top: box.height}`.

- `left`: an alternative for `box.left`.
- `onTraceData()`: Called for each instance once new trace data is available, after `init()` and by the time the component is in view. This function may be used to preprocess the trace and to cache the processed data to avoid recomputation for each individual `render()`. This function is responsible for cleaning up from previous `onTraceData` calls from previous trace data. It optionally returns the following object, where properties are optional:
    - `minCyc`, `maxCyc`: [default: all of time] a range of cycle times for which processing has been completed (which must include all active view times). If the current view time leaves this window, this function will be called again and return a new window.
    - `objects`: a JavaScript object of `Fabric.Object`s to add to this component. These objects are made available to other functions via `this.getObjects()`.
- `overlay: {...}`: If present, this provides an overlay, layered above the children. Properties of this object are a subset of those of the `\viz_js` block, including:
    - `template`
    - `init()`
    - `render()`

  `where` and `box` properties of the `\viz_js` block apply to the overlay as well. The components of the overlay do not impact the bounds, and are expected to be contained

within the bounds of the `\viz_js` box. Other properties/functions of the `\viz_js` block are not applicable to the overlay.

- `render()`: Called for each instance when rendered for each new cycle. Returns an array of Fabric.js objects that represent this component in the current cycle (`this.getCycle()`) (ordered bottom to top, added above other objects and children). Properties of `init()` and `template` objects from `this.getObjects()` can be modified. (These properties will not be automatically restored to render other cycles, so they must always be assigned or they must be restored by `unrender()`.)
- `renderFill()`: Called prior to `render()` to return a background color for the bounding box based on simulation state. Note that `render()` will not necessarily be called if the component is below a certain size on the screen, so the use of this function is recommended when the fill color reflects the simulation.
- `sigs() {...}`: Returns a flat object (aka a dictionary) of signal value references. This function is purely for optimizing performance. These references are maintained by the framework to avoid allocating signal value references with each `render()`. `render()` and `renderFill()` are invoked in a context in which `sigs` contains the return value of this function, adjusted to the proper view cycle. These references may be stepped within render functions, and their time will be restored for each adjustment of the current view cycle.
- `stroke`: an alternative for `box.stroke`.
- `strokeWidth`: an alternative for `box.strokeWidth`.
- `template`: A JavaScript object whose properties define Fabric.js objects, or a function that returns the same, called once for the `\viz` block (not for each instance). These Fabric.js objects are created and added to each instance's `fabric.Group` prior to `init()` and made available to other functions via `this.getObjects()`. This is mainly a performance optimization vs. `init()` that avoids re-execution for every instance. The format for each property is: `<name>: [<class-name>, <constructor-arg>, …]`. For example: `dot: ["Circle", {left: 10, top: 10, radius: 2}]`. `template()` is called with `this = {}`. Properties added to `this` are shallow-copied into `this` for other (per-instance) functions.
- `top`: an alternative for `box.top`.
- `unrender()`: Called once for every `render()` call, when the rendering is no longer needed. If `render()` created any data that must be explicitly deleted, that can be done here.
- `where: {` or `where: [{` and `where0: {`

  An object or array of objects, each providing properties for placing a component within a parent. Properties are with respect to the coordinate system(s) of the parent(s).

  `where` specifies the embedding of the scope (and its instances) in the parent of this scope.

`where0` can be used in conjunction with `all` to specify the embedding of the instance with index zero within the `all` component, which is the parent of the instances. (Note that there might not be an instance with index zero, but this provides a point of reference for other instances regardless.) The positioning of other instances is determined by `layout`.

Note that positioning properties are collected under `where` so that an instantiation of a generic component (IP) can provide all properties related to its visual instantiation in a single argument. For definitions and instantiations of replicated components, `layout` and/or `where0` might require separate parameters.

- `angle`: [default `0`] clockwise angle of the component in degrees.
- `left, top`: [default: left, top of this component's box] position in the parent's coordinate system of the top-left corner of the containing rectangle of this instance within the parent. (Note that these will have no visible impact if the parent has no `box` to define its coordinate system and no other children relative to which to place this component.)
- `name`: A name for this instance to distinguish it from others in the same scope.
- `parent:` The `\viz_js` block (or `\viz_js` block `all`) within which to embed this one. Functions for identifying `\viz_js` blocks are TBD.
- `justifyX/Y`: [default: `"left"/"top"`] The positioning heuristic in case the `width`/`height` does not fill the corresponding `where` bound. Legal values are identical to those of `originX`/`Y`. For example, `"bottom"` aligns the box bottom with the where bottom bound, and `"center"` leaves equal padding on either side.
- `scale`: [default: `1.0`, if no `width`/`height` given] An (X and Y) scaling factor (or maximum scaling factor, if `width`/`height` given) relative to the scale of the parent.
- `scaleX`, `scaleY`: Override scale with independent X/Y scaling factors.
- `width, height`: [default: use `scale` (and/or `scaleX/Y`) to determine] bounds within which to contain the component. (Determines maximum scale).
- `visible`: [default: `false`] Set to true to make the where area visible as a `fabric.Rect`, in which case `fabric.Rect` properties may also be provided to stylize this `Rect`.

Note that the top-level where/where0 has no effect.

`}` or `, ...]}`

- `width`: an alternative for `box.width`.

Processing `\viz_js` Bodies

Components in the visual hierarchy (visual descriptions (and their `all`'s)) are processed in the following order:

- When the viewer processes the visual description for a model, it, per description, leaf-first:
  - evaluates properties (including all)
- Prior to the availability of simulation data, the viewer creates visual component instances. This may be deferred until the instance is in view and/or big enough to see. Per instance, the viewer:
  - recurses into (visible) children
  - applies `template`
  - calls `init()`
  - applies `overlay.template`
  - calls `overlay.init()`
- After new trace data is loaded (and after recursively calling `init()`) and by the time the component is in view, the viewer:
  - calls `onTraceDataTopDown()` top-down
  - recurses into (all) children
  - calls `onTraceData()` leaf-first
- Each time the current view time changes or new simulation data is loaded, the viewer:
  - recurses into (visible) children
  - calls `renderFill()` and (if visible) `render()`

Any calls that were pruned for components that were not in view, will be made, in the proper sequence, once the component is in view.

Execution Context

Each component instance has a `VizJSContext` object. In all functions, `this` is a reference to this object. Properties of `this` can be assigned to pass information between functions according to their execution sequence and even from one render to the next (though this is generally discouraged). `VizJSContext` provides the following API.

- `getCycle()`: Returns the active cycle number of the view (for render functions).
- `getIndex()`: Returns the index of this instance within its own elaborated scope.
- `getIndex(name)`: Returns the index of the given elaborated ancestor scope of this instance, identified by its TL-Verilog scope identifier (though currently, mis-implemented to exclude the prefix character).
- `steppedBy()`: Returns the cycle delta from the previous `render()` or `0` for the first render. This can be useful in `render()` for animating backward stepping.
- `getObjects()`: Returns the objects returned by `template`, `init()`, and `render()`, as well as the Fabric.js `Rect` of the `box` (as `this.getObjects().box`)
- `getScope()`: Returns an Object representing the definition scope of this instance (the structure of which is TBD).

- `getScope(name)`: Returns an Object representing the definition scope of the given ancestor instance (the structure of which is TBD), identified by its TL-Verilog scope identifier (though currently, mis-implemented to exclude the prefix character), or `null` if no such scope.
- `sigVal(sig, cyc = 0)`: See XXX ("Signal Value References" below).
- `stillRendered()`: Returns true if this component is still rendered, false otherwise. Asynchronous functions, like animation callbacks, should confirm `stillRendered()` before acting on previously rendered objects.
- ~~`thenRender(fn[, elseFn])`: Especially when animating, asynchronous callbacks might be used in `render()`. In place of a callback function, `fn`, using `this.thenRender(fn)` will prevent `fn` from being called if there is an intervening `unrender()`. (Using `thenRender(..)` is a bit easier than canceling timeouts in `unrender()`.) `elseFn`, if provided, is called if `fn` is not. `this` of `fn` is as in the caller, and the argument list is passed through. `thenRender` also registers the need to rerender with Fabric.js.~~

VIZ code runs in a sandboxed context where `window` contains a safe subset of the normal contents of `window`. Details can be determined through experimentation. Added to the standard `window` contents are the following:

- `wait(ms)`: An asynchronous function that waits the given delay in milliseconds. This is useful for animations.

Signal Value References

Signal value references

`SignalValue` objects are returned by `this.sigVal(...)` or, for TL-Verilog pipesignals, `'$my_sig'`. This object is a reference to a signal in the simulation at a particular cycle. The reference cycle can be adjusted to access any value of this signal.

`this.sigVal(sig_name, cyc_offset = 0)`:

- `sig_name`: (string) The full signal name and path, as given in the trace file, using "." to delimit scopes and signal name.
- `cyc_offset`: (int) The cycle offset relative to the current view cycle at which to reference the signal. This offset is clock-phase-granular, meaning it should be a multiple of `0.5`.

Returns a `SignalValue` representing the value of the given signal at the requested time. If the signal is not found, a message may be reported to the console and the return value is currently undefined, but attempts to call the `mustExist()` method will throw an exception (currently because `null` is returned; in the future, because the `SignalValue` is "non-existent").

`'$my_sig'`: May use any syntax permitted for TL-Verilog pipesignal references, including alignment (which is analogous to `cyc_offset` for `this.sigVal`).

`SignalValue` Class

The following is the commonly-used subset of the SignalValueReference class.

Signal Value Access

`as*(default = undefined)`, where * is one of: `Bool`, `Int`, `String`, `Real`, and other types/formats:

- `default`: (any) The value returned if the signal or the referenced time is not available in the trace.
- Returns: The value of the signal at the currently-referenced cycle time. `null` if the signal is dont-care or invalid based on the TL-Verilog when condition (e.g. `?$valid`), or if the signal is incompatible with the requested type.

Adjusting Referenced Cycle

`step(cycle_delta = 1)`:

- `cycle_delta`: (int) The number of cycles by which to adjust the reference time (positive, negative, or zero).

The full API of `SignalValue` can be found in XXX (code docs not yet available).

Setting and Animating Object Properties

Managing Properties Through Cycle Changes

Fabric.js Objects can be created by `\viz_js` `template`, `init()`, `onTraceData()`, and `render()`. `render()` and `unrender()` may change Object properties.

Users may navigate to different simulation cycles by single-stepping, jumping to a specific cycle, and using playback to step through cycles sequentially. Especially during playback and single-stepping, animating the transition between cycles can help a user to more easily follow the flow of data.

`render()` should produce the same visualization for a given cycle (after animation), regardless of which cycle was previously rendered and whether that cycle finished its animation. For each property of an Object, this can be ensured one of three ways:

- Objects that are created and returned by `render()` are created and destroyed (automatically) for each cycle, and are thus independent of the previously rendered cycle.
- Properties modified by `render()` can be restored to their initial value by `unrender()`.
- Properties can always be set by `render()`, overriding their initial value.

A few approaches are reasonable for animation:

- Set the properties in `render()` to deterministic values (independent of the previously rendered cycle), then animate. This tends to result in jerky motion when stepping backward as the animation typically illustrates a forward motion of data at each cycle.
- Use `steppedBy()` to determine whether to render forward or backward motion, animating from the +1 or -1 cycle state to the `getCycle()` state.
- Animate from whatever the state is initially. This is a little odd since it ignores intermediate states between discontiguous cycles, but it avoids discontiguous jumping, which can impact the user's ability to keep track of data.

Setting Properties

Properties of Objects are set using `Object.set(...)`. For example:

```
…
render() {
    …
    objs = this.getObjects()
    # Color dot red.
    objs.dot.set("fill", "red")
    # Set frame to green with blue border.
    objs.frame.set({stroke: "blue", fill: "green"})
    …
}
…
```

Setting properties, as above, simply uses the features of Fabric.js.

Animation

Animation primarily leverages Fabric.js's facilities for animation, but VIZ makes a few improvements. As long as you follow recommended coding patterns, you won't have to worry about:

- Explicitly rendering the `Canvas`.
- Stopping the animation if the user changes the active cycle.

- Chaining animation steps with callbacks.
- Speeding up or slowing down the animation for playback, as set by the user.

Objects support a modified `animate` method that works similarly to `set`, but it also takes a `duration`. The Object properties are transitioned over this duration from their current value to the new value.

```
…
render() {
   …
   objs = this.getObjects()
   # Turn the arrow to 90 degrees.
   objs.arrow.animate("angle", 90, {duration: 200})
   # Move marker and grow.
   objs.marker.animate({left: 100, width: 20, height: 20}, {duration: 200})
   …
}
…
```

VIZ adds a cleaner mechanism to chain animations using new `thenAnimate`, `wait`, and `thenWait`.

```
…
render() {
   …
   objs.arrow.animate("angle", 90)
   # Move and spin marker, wait 100ms, and snap back.
   objs.marker
      .animate({left: 100, angle: 90}, {duration: 200})
      .thenAnimate({left: 100, angle: 90}, {duration: 200})
      .thenWait(100)
      .thenAnimate({angle: 0}, {duration: 20})
   # Independently, grow marker.
   objs.marker
      .wait(100)
      .thenAnimate({width: 40, height: 40}, {duration: 600})
   …
}
…
```

The above pattern is generally sufficient, but here are the details for advanced users:

- VIZ makes `Object`s "thenable", supporting `await` semantics (though `render()` itself is synchronous).
- `animate`, `thenAnimate`, `wait`, and `thenWait` return the `Object` itself.
- A `getPromise()` method returns the `Promise` of the most recent `animate`/`thenAnimate`/`wait`/`thenWait`. (This is the `Promise` used by `.then(...)`.)
- `then*` methods chain to this `getPromise() Promise`.

In playback, the period of a cycle defaults to 1.0 seconds, and your animation should assume this period. Time is accellerated or decellerated based on the playback cycle period by internally adjusting the given ms values. When using other functions, such as `setTimeout(...)`, use `this.ms(ms)` to adjust a time value, e.g. `setTimeout(fn, this.ms(100))`.

Debugging VIZ Code

Since Visual Debug utilizes JavaScript running in a web browser, it benefits from standard browser debug capabilities. In Chrome and Firefox, the Developer Tools can be opened using <ctrl-shift>-I. Note that the visual elements are rendered on a canvas, so they are not exposed in the HTML DOM (Document Object Model).

Debug scenarios include:

- parse errors
- runtime errors

Parse Errors

Parse errors will be reported in the browser's Console.

Runtime Errors

Runtime errors will be reported in the browser's Console. Runtime errors can be debugged by stepping through the execution of `\viz_js` functions in the browser's debugger. To pause execution within a \viz_js function, it is generally only necessary to trigger execution of the failing function with the Developer Tools open. When the error is caught, program execution will pause on a `debugger` statement, and the failing function will be executed a second time for debug purposes. Except in unusual coding patterns, the second invocation should execute the same as the first, and this execution can be debugged. From the `debugger` statement, be sure to step into the second invocation.

It can also be useful to explicitly use `debugger` statements in your `\viz_js` functions to halt execution and step through your VIZ code.

Common Error Signatures

The VIZ canvas stops rendering and/or renders with objects out of place or duplicated. This may indicate a malformed hierarchy of Fabric Objects on the canvas. The same Fabric Object may have been added to the canvas multiple times. This shouldn't happen in typical usage, but could happen when Objects are created in one `\viz_js` block and accessed and returned for rendering by another.

Further References

While the information in this manual is sufficient for common use of Visual Debug, sophisticated use models may require a complete understanding of:

- the TL-Verilog specification, available at http://tl-x.org.
- M4 use with TL-Verilog, available at XXX. This builds upon M4, documented at https://www.gnu.org/software/m4/manual/
- The VIZ API Reference at XXX.
- The fabric.js documentation available at http://fabricjs.com.

Issues

Errors or omissions with this guide can be addressed by commenting on the working version of this document. or by emailing help@redwoodeda.com or filing an issue publicly at https://gitlab.com/rweda/support/-/issues. Your contributions are appreciated.

Glossary

Box: In the context of Visual Debug, the box is the boundary of a visual component and the Fabric.js Rect object confined within that boundary, providing the background of the component.

Definition hierarchy: The hierarchy of the hardware and/or visualization source code.

Elaborated hierarchy: The hierarchy of the actual hardware after macro expansion, instantiation and elaboration and/or the hierarchy of instantiated visual components.

Fabric.js: The underlying graphics library for representing visual components on a canvas in Visual Debug.

HDL: Hardware description language, such as Verilog, VHDL, TL-Verilog, etc.

IDE: Integrated development environment, such as SandStorm or Makerchip.

Identifier: As defined in the TL-X specification at https://tl-x.org.

Instance hierarchy: See "elaborated hierarchy".

M4: A macro preprocessor tool and language commonly used in conjunction with TL-Verilog.

Makerchip: A free online IDE for open-source hardware development from Redwood EDA, LLC that incorporates Visual Debug.

RTL: Register transfer logic. This is the level of hardware abstraction embodied in HDL languages such as Verilog and VHDL.

SandStorm: A commercial IDE tool from Redwood EDA that includes Visual Debug capabilities.

Scope: TL-Verilog scope, as defined in this TL-X specification at https://tl-x.org.

Signal: An RTL construct representing a wire or collection of wires.

Stroke: The border line for a Fabric.js object.

Visual component: The entity defined by a `\viz_js` code block.

Visual Debug: The feature defined in this document.

Visual description: A `\viz_js` code block or a component therein.

VIZ: A short name for Visual Debug.

TL-Verilog: Transaction-Level Verilog. Further information can be found at https://redwoodeda.com/tl-verilog