Appendix E: Tcl Command Syntax

Tcl Command Syntax

Besides being able to access the menu options (e.g., gtkwave::/File/Quit), within Tcl scripts there are more commands available for manipulating the viewer.

addSignalsFromList: adds signals to the viewer

Syntax: `set num_found [ gtkwave::addSignalsFromList list ]`

Example:
```
set clk48 [list]
lappend clk48 "$facname1"
lappend clk48 "$facname2"
...
set num_added [ gtkwave::addSignalsFromList $clk48 ]
```
deleteSignalsFromList: deletes signals from the viewer. This deletes only the first instance found unless the signal is specified multiple times in the list.

Syntax: `set num_deleted [ gtkwave::deleteSignalsFromList list ]`

Example:
```
set clk48 [list]
lappend clk48 "$facname1"
lappend clk48 "$facname2"
...
set num_deleted [ gtkwave::deleteSignalsFromList $clk48 ]
```
deleteSignalsFromListIncludingDuplicates: deletes signals from the viewer. This deletes all the instances found so there is no need to specify the same signal multiple times in the list.

Syntax: `set num_deleted`
```
[ gtkwave::deleteSignalsFromListIncludingDuplicates list ]
```

Example:
```
set clk48 [list]
lappend clk48 "$facname1"
lappend clk48 "$facname2"
...

set num_deleted [ gtkwave::deleteSignalsFromListIncludingDuplicates
$clk48 ]
```
findNextEdge: advances the marker to the next edge for highlighted signals

Syntax: `set marker_time [ gtkwave::findNextEdge ]`

Example:
```
gtkwave::highlightSignalsFromList "top.clk"
set time_value [ gtkwave::findNextEdge ]
puts "time_value: $time_value"
```
findPrevEdge: moves the marker to the previous edge for highlighted signals

Syntax: `set marker_time [ gtkwave::findPrevEdge ]`

Example:
```
gtkwave::highlightSignalsFromList "top.clk"
set time_value [ gtkwave::findPrevEdge ]
puts "time_value: $time_value"
```
forceOpenTreeNode: forces open one tree node in the Signal Search Tree and closes the rest. If upper levels are not open, the tree will remain closed however once the upper levels are opened, the hierarchy specified will become open. If path is missing or is an empty string, the function returns the current hierarchy path selected by the SST or -1 in case of error.

Syntax: `gtkwave::forceOpenTreeNode hierarchy_path`

Returned value:
- 0 - success
- 1 - path not found in the tree
- -1 - SST tree does not exist

Example:
```
 set path tb.HDT.cpu
  switch -- [gtkwavetcl::forceOpenTreeNode $path] {
    -1 {puts "Error: SST is not supported here"}
    1 {puts "Error: '$path' was not recorder to dump file"}
    0 {}
  }
```

getArgv: returns a list of arguments which were used to start gtkwave from the command line

Syntax: `set argvlist [ gtkwave::getArgv ]`

Example:
```
set argvs [ gtkwave::getArgv ]
puts "$argvs"
```
getBaselineMarker: returns the numeric value of the baseline marker time

Syntax: `set baseline_time [ gtkwave::getBaselineMarker ]`

Example:
```
set baseline [ gtkwave::getBaselineMarker ]
puts "$baseline"
```
getDisplayedSignals: returns a list of all signals currently on display

Syntax: `set display_list [ gtkwave::getDisplayedSignals ]`

Example:
```
set display_list [ gtkwave::getDisplayedSignals ]
puts "$display_list"
```
getDumpFileName: returns the filename for the loaded dumpfile

Syntax: `set loaded_file_name [ gtkwave::getDumpFileName ]`

Example: `set nfacs [ gtkwave::getNumFacs ]`
```
set dumpname [ gtkwave::getDumpFileName ]
set dmt [ gtkwave::getDumpType ]
puts "number of signals in dumpfile '$dumpname' of type $dmt: $nfacs"
```
`getDumpType`: returns the dump type as a string (VCD, PVCD, LXT, LXT2, GHW, VZT)

Syntax: `set dump_type [ gtkwave::getDumpType ]`

Example:
```
set nfacs [ gtkwave::getNumFacs ]
set dumpname [ gtkwave::getDumpFileName ]
set dmt [ gtkwave::getDumpType ]
puts "number of signals in dumpfile '$dumpname' of type $dmt: $nfacs"
```
`getFacName`: returns a string for the facility name which corresponds to a given facility number

Syntax: `set fac_name [ gtkwave::getFacName fac_number ]`

Example:
```
set nfacs [ gtkwave::getNumFacs ]
for {set i 0} {$i < $nfacs } {incr i} {
    set facname [ gtkwave::getFacName $i ]
    puts "$i: $facname"
}
```
getFontHeight: returns the font height for signal names

Syntax: `set font_height [ gtkwave::getFontHeight ]`

Example:
```
set font_height [ gtkwave::getFontHeight ]
puts "$font_height"
```
getFromEntry: returns the time value string in the “From:” box.

Syntax: `set from_entry [ gtkwave::getFromEntry ]`

Example:
```
set from_entry [ gtkwave::getFromEntry ]
puts "$from_entry"
```
`getHierMaxLevel`: returns the max hier value which is set in the viewer

Syntax: `set hier_max_level [ gtkwave::getHierMaxLevel ]`

Example:
```
set max_level [ gtkwave::getHierMaxLevel ]
puts "$max_level"
```
getLeftJustifySigs: returns 1 if signals are left justified, else 0

Syntax: `set left_justify [ gtkwave::getLeftJustifySigs ]`

Example:
```
set justify [ gtkwave::getLeftJustifySigs ]
puts "$justify"
```
getLongestName: returns number of characters of the longest name in the dumpfile

Syntax: `set longestname_len [ gtkwave::getLongestName ]`

Example:
```
set longest [ gtkwave::getLongestName ]
puts "$longest"
```
getMarker: returns the numeric value of the primary marker position

Syntax: `set marker_time [ gtkwave::getMarker ]`

Example:
```
set marker_time [ gtkwave::getMarker ]
puts "$marker_time"
```
getMaxTime: returns the numeric value of the last time value in the dumpfile

Syntax: `set max_time [ gtkwave::getMaxTime ]`

Example:
```
set max_time [ gtkwave::getMaxTime ]
puts "$max_time"
```
getMinTime: returns the numeric value of the first time value in the dumpfile

Syntax: `set min_time [ gtkwave::getMinTime ]`

Example:
```
set min_time [ gtkwave::getMinTime ]
puts "$min_time"
```
getNamedMarker: returns the numeric value of the named marker position

Syntax: `set time_value [ gtkwave::getNamedMarker which ]`
```
such that which = A-Z or a-z
```

Example:
```
set marker_time [ gtkwave::getNamedMarker A ]
puts "$marker_time"
```
getNumFacs: returns the number of facilities encountered in the dumpfile

Syntax: `set numfacs [ gtkwave::getNumFacs ]`

Example:
```
set nfacs [ gtkwave::getNumFacs ]
set dumpname [ gtkwave::getDumpFileName ]
set dmt [ gtkwave::getDumpType ]
puts "number of signals in dumpfile '$dumpname' of type $dmt: $nfacs"
```
getNumTabs: returns the number of tabs shown on the viewer

Syntax: `set numtabs [ gtkwave::getNumTabs ]`

Example:
```
set ntabs [ gtkwave::getNumTabs ]
puts "number of tabs: $ntabs"
```
getPixelsUnitTime: returns the number of pixels per unit time

Syntax: `set pxut [ gtkwave::getPixelsUnitTime ]`

Example:
```
set pxut [ gtkwave::getPixelsUnitTime ]
puts "$pxut"
```
getSaveFileName: returns the save filename

Syntax: `set save_file_name [ gtkwave::getSaveFileName ]`

Example:
```
set savename [ gtkwave::getSaveFileName ]
puts "$savename"
```
getStemsFileName: returns the stems filename

Syntax: `set stems_file_name [ gtkwave::getStemsFileName ]`

Example:
```
set stemsname [ gtkwave::getStemsFileName ]
puts "$stemsname"
```
getTimeDimension: returns the first character of the time units that the trace was saved in (e.g., "u" for us, "n" for "ns", "s" for sec, etc.)

Syntax: `set dimension_first_char [ gtkwave::getTimeDimension ]`

Example:
```
set dimch [ gtkwave::getTimeDimension ]
puts "$dimch"
```
getTimeZero: returns the numeric value for what represents the time #0 in the dumpfile. This is only of interest if the $timezero directive is encountered in the dumpfile.

Syntax: `set zero_time [ gtkwave::getTimeZero ]`

Example:
```
set zero_time [ gtkwave::getTimeZero ]
puts "$zero_time"
```
getToEntry: returns the time value string in the “To:” box.

Syntax: `set to_entry [ gtkwave::getToEntry ]`

Example:
```
set to_entry [ gtkwave::getFromEntry ]
puts "$to_entry"
```
getTotalNumTraces: returns the total number of traces that are being displayed currently

Syntax: `set total_traces [ gtkwave::getTotalNumTraces ]`

Example:
```
set totnum [ gtkwave::getTotalNumTraces ]
puts "$totnum"
```
getTraceFlagsFromIndex: returns the decimal value of the sum of all flags for a given trace

Syntax: `set flags [ gtkwave::getTraceFlagsFromIndex trace_number ]`

Example:
```
set tflags [ gtkwave::getTraceFlagsFromIndex 0 ]
```

```
puts "$tflags"
```
getTraceFlagsFromName: returns the decimal value of the sum of all flags for a given trace

Syntax: `set flags [ gtkwave::getTraceFlagsFromName trace_name ]`

Example:
```
set tflags [ gtkwave::getTraceFlagsFromName {top.des.k1x[1:48]} ]
puts "$tflags"
```
getTraceNameFromIndex: returns the name of a trace when given the index value

Syntax: `set trace_name [ gtkwave::getTraceNameFromIndex trace_number ]`

Example:
```
set tname [ gtkwave::getTraceNameFromIndex 1 ]
puts "$tname"
```
getTraceScrollbarRowValue: returns the scrollbar value (which corresponds to the trace index for the topmost trace on screen)

Syntax: `set scroller_value [ gtkwave::getTraceScrollbarRowValue ]`

Example:
```
set scroller [ gtkwave::getTraceScrollbarRowValue ]
puts "$scroller"
```
getTraceValueAtMarkerFromIndex: returns the value under the marker for the trace numbered trace index

Syntax: `set ascii_value [ gtkwave::getTraceValueAtMarkerFromIndex trace_number ]`

Example:
```
set tvi [ gtkwave::getTraceValueAtMarkerFromIndex 2 ]
puts "$tvi"
```
getTraceValueAtMarkerFromName: returns the value under the primary marker for the given trace name

Syntax: `set ascii_value [ gtkwave::getTraceValueAtMarkerFromName fac_name ]`

Example:
```
set tvn [ gtkwave::getTraceValueAtMarkerFromName
{top.des.k2x[1:48]} ]
puts "$tvn"
```
getTraceValueAtNamedMarkerFromName: returns the value under the named marker for the given trace name

Syntax: `set ascii_value`
```
[ gtkwave::getTraceValueAtNamedMarkerFromName which fac_name ]
such that which = A-Z or a-z
```

Example:
```
set tvn [ gtkwave::getTraceValueAtNamedMarkerFromName A
{top.des.k2x[1:48]} ]
puts "$tvn"
```
getUnitTimePixels: returns the number of time units per pixel

Syntax: `set utpx [ gtkwave::getUnitTimePixels ]`

Example:
```
set utpx [ gtkwave::getUnitTimePixels ]
puts "$utpx"
```
getVisibleNumTraces: returns number of non-collapsed traces

Syntax: `set num_visible_traces [ gtkwave::getVisibleNumTraces ]`

Example:
```
set nvt [ gtkwave::getVisibleNumTraces ]
puts "$nvt"
```
getWaveHeight: returns the height of the wave window in pixels

Syntax: `set wave_height [ gtkwave::getWaveHeight ]`

Example:
```
set wht [ gtkwave::getWaveHeight ]
puts "$wht"
```
getWaveWidth: returns the width of the wave window in pixels

Syntax: `set wave_width [ gtkwave::getWaveWidth ]`

Example:
```
set wwt [ gtkwave::getWaveWidth ]
puts "$wwt"
```
getWindowEndTime: returns the end time of the wave window

Syntax: `set end_time_value [ gtkwave::getWindowEndTime ]`

Example:
```
set wet [ gtkwave::getWindowEndTime ]
puts "$wet"
```
getWindowStartTime: returns the start time of the wave window

Syntax: `set start_time_value [ gtkwave::getWindowStartTime ]`

Example:
```
set wst [ gtkwave::getWindowStartTime ]
puts "$wst"
```
getZoomFactor: returns the zoom factor of the wave window

Syntax: `set zoom_value [ gtkwave::getZoomFactor ]`

Example:
```
set zf [ gtkwave::getZoomFactor ]
puts "$zf"
```
highlightSignalsFromList: highlights the facilities contained in the list argument

Syntax: `set num_highlighted [ gtkwave::highlightSignalsFromList list ]`

Example:
```
set clk48 [list]
lappend clk48 "$facname1"
lappend clk48 "$facname2"
...
set num_highlighted [ gtkwave::highlightSignalsFromList $clk48 ]
```
installFileFilter: installs file filter number which across all highlighted traces. Using zero for which removes the filter.

Syntax: `set num_updated [ gtkwave::installFileFilter which ]`

Example:
```
set num_updated [ gtkwave::installFileFilter 0 ]
puts "$num_updated"
```
installProcFilter: installs process filter number which across all highlighted traces. Using zero for which removes the filter.

Syntax: `set num_updated [ gtkwave::installProcFilter which ]`

Example:
```
set num_updated [ gtkwave::installProcFilter 0 ]
puts "$num_updated"
```
installTransFilter: installs transaction process filter number which across all highlighted traces. Using zero for which removes the filter.

Syntax: `set num_updated [ gtkwave::installTransFilter which ]`

Example:
```
set num_updated [ gtkwave::installTransFilter 0 ]
puts "$num_updated"
```

loadFile: loads a new file

Syntax: `gtkwave::loadFile filename`

Example:
```
gtkwave::loadFile "$filename"
```
nop: calls the GTK main loop in order to update the gtkwave GUI

Syntax: `gtkwave::nop`

Example:
```
gtkwave::nop
```
presentWindow: raises the main window in the stacking order or deiconifies it

Syntax: `gtkwave::presentWindow`

Example:
```
gtkwave::presentWindow
```
reLoadFile: reloads the current active file

Syntax: `gtkwave::reLoadFile`

Example:
```
gtkwave::reLoadFile
```
setBaselineMarker: sets the time for the baseline marker (-1 removes it)

Syntax: `gtkwave::setBaselineMarker time_value`

Example:
```
gtkwave::setBaselineMarker 128
```
setCurrentTranslateEnums: sets the enum list to function as the current translate file and returns the corresponding which value to be used with gtkwave::installFileFilter. As a real file is not used, the results of this are not recreated when a save file is loaded or if the waveform is reloaded.

Syntax: `set which_f [ gtkwave::setCurrentTranslateEnums elist ]`

Example:
```
set enums [list]
lappend enums 0000000000000000        IDLE
lappend enums FFFFFFFFFFFFFFFF        BUSY
lappend enums 3000000000000000        OTHER
lappend enums 0123456789ABCDEF        HEXSTATE
lappend enums 1111111111111111        "All 1s"
set which_f [ gtkwave::setCurrentTranslateFile $enums ]
puts "$which_f"
```
setCurrentTranslateFile: sets the filename to the current translate file and returns the

corresponding which value to be used with gtkwave::installFileFilter.

Syntax: `set which_f [ gtkwave::setCurrentTranslateFile filename ]`

Example:
```
set which_f [ gtkwave::setCurrentTranslateFile ./zzz.txt ]
puts "$which_f"
```
setCurrentTranslateProc: sets the filename to the current translate process (executable) and returns the corresponding which value to be used with gtkwave::installProcFilter.

Syntax: `set which_f [ gtkwave::setCurrentTranslateProc filename ]`

Example:
```
set which_f [ gtkwave::setCurrentTranslateProc ./zzz.exe ]
puts "$which_f"
```
setCurrentTranslateTransProc: sets the filename to the current transaction translate process (executable) and returns the corresponding which value to be used with gtkwave::installTransFilter.

Syntax: `set which_f [ gtkwave::setCurrentTranslateTransProc filename ]`

Example:
```
set which_f [ gtkwave::setCurrentTranslateTransProc ./zzz.exe ]
puts "$which_f"
```
setFromEntry: sets the time in the “From:” box.

Syntax: `gtkwave::setFromEntry time_value`

Example:
```
gtkwave::setFromEntry 100
```
setLeftJustifySigs: turns left justification for signal names on or off

Syntax: `gtkwave::setLeftJustifySigs on_off_value`

Example:
```
gtkwave::setLeftJustifySigs on
gtkwave::setLeftJustifySigs off
```
setMarker: sets the time for the primary marker (-1 removes it)

Syntax: `gtkwave::setMarker time_value`

Example:
```
gtkwave::setMarker 128
```
setNamedMarker: sets named marker A-Z (a-z) to a given time value and optionally renames the marker text to a string (-1 removes the marker)

Syntax: `gtkwave::setNamedMarker which time_value [string]`

Example:
```
gtkwave::setNamedMarker A 400 "Example Named Marker"
gtkwave::setNamedMarker A 400
```
setTabActive: sets the active tab in the viewer (0..getNumTabs-1)

Syntax: `gtkwave::setTabActive which`

Example:
```
gtkwave::setTabActive 0
```
setToEntry: sets the time in the “To:” box.

Syntax: `gtkwave::setToEntry time_value`

Example:
```
gtkwave::setToEntry 600
```
setTraceHighlightFromIndex: highlights or unhighlights the specified trace

Syntax: `gtkwave::setTraceHighlightFromIndex trace_index on_off`

Example:
```
gtkwave::setTraceHighlightFromIndex 2 on
gtkwave::setTraceHighlightFromIndex 2 off
```
setTraceHighlightFromNameMatch: highlights or unhighlights the specified trace

Syntax: `gtkwave::setTraceHighlightFromNameMatch fac_name on_off`

Example:
```
gtkwave::setTraceHighlightFromNameMatch top.des.clk on
gtkwave::setTraceHighlightFromNameMatch top.clk off
```
setTraceScrollbarRowValue: sets the scrollbar for traces a number of traces down from the very top

Syntax: `gtkwave::setTraceScrollbarRowValue scroller_value`

Example:
```
gtkwave::setTraceScrollbarRowValue 10
```
setWindowStartTime: scrolls the traces such that the start time is at the left margin (as long as the zoom level permits this)

Syntax: `gtkwave::setWindowStartTime start_time`

Example:
```
gtkwave::setWindowStartTime 100
```

setZoomFactor: sets the zoom factor for the trace data (i.e., how compressed it is with respect to time)

Syntax: `gtkwave::setZoomFactor zoom_value`

Example:
```
gtkwave::setZoomFactor -3
```
setZoomRangeTimes: sets the visible time range for the trace data

Syntax: `gtkwave::setZoomRangeTimes time1 time2`

Example:
```
gtkwave::setZoomRangeTimes 100 217
```
showSignal: sets the scrollbar for traces a number of traces down from the very top (0), center (1), or bottom (2)

Syntax: `gtkwave::setTraceScrollbarRowValue scroller_value position`
Example:
```
gtkwave::setTraceScrollbarRowValue 10 0
```
signalChangeList: returns time and value changes for the signals indicated by the argument names

Syntax: `gtkwave::signalChangeList signal_name options`

Where `options` is are one or more of the following:

`-start_time` start-time (default 0)

`-end_time` end-time (default last sample in dump file)

`-max` maximum-number-of-samples (default 0x7fffffff)

`-dir` forward|backward (default forward)

The function returns a Tcl list of value changes for the `signal-name` starting at `start-time` and ending at `end-time` or an empty list in any other case.

Even members of the list hold the time of change and odd members hold the value that is associated with the time the precedes it. Values are given as strings in the base of the signal.

If `signal-name` is not present then the first highlighted signal is taken.

Length of the list is defined by both `end-time` and `max`, whichever comes first.

To specify backward search, `end-time` should be smaller than `start-time` or `dir` should have the value of `backward` and `end-time` is not defined.

A conflict between timing (start/end-time) and direction (forward/backward) returns an empty list.

Examples:

1. prints the first 100 changes of the signal tb.HDT.cpu.CS starting at time 10000

```
set signal tb.HDT.cpu.CS
set start_time 10000
foreach {time value} [gtkwave::signalChangeList $signal -start_time
$start_time -max 100] {
    puts "Time: $time value: $value"
}
```

2. retrieve the value of tb.HDT.cpu.CS at time 123456

```
   lassign [gtkwave::signalChangeList tb.HDT.cpu.CS -start_time
123456
-max 1] dont_care value
```

unhighlightSignalsFromList: unhighlights the facilities contained in the list argument

Syntax: `set num_unhighlighted [ gtkwave::unhighlightSignalsFromList list ]`

Example:

```
set clk48 [list]
lappend clk48 "$facname1"
lappend clk48 "$facname2"
...
set num_highlighted [ gtkwave::unhighlightSignalsFromList $clk48 ]
```

Tcl Callbacks

When gtkwave performs various functions, global callback variables prepended with gtkwave:: are modified within the Tcl interpreter. By using the trace write feature in Tcl, scripts can achieve a very tight integration with gtkwave. Global variables which may be used to register callback procedures are as follows:

gtkwave::cbCloseTabNumber contains the value returned is the number of the tab which is going to be closed, starting from zero. As this is set before the tab actually closes, scripts can interrogate for further information.

gtkwave::cbCloseTraceGroup contains the name of the expanded trace or trace group being closed.

gtkwave::cbCurrentActiveTab contains the number of the tab currently selected. Note that when new tabs are being created, this callback sometimes will oscillate between the old and new tab number, finally settling on the new tab being created.

gtkwave::cbError contains an error string such as “reload failed”, “gtkwave::loadFile prohibited in callback”, “gtkwave::reLoadFile prohibited in callback”, or “gtkwave::setTabActive prohibited in callback”.

gtkwave::cbFromEntryUpdated contains the value stored in the “From:” widget when it is updated.

gtkwave::cbOpenTraceGroup contains the name of a trace being expanded or trace group being opened.

gtkwave::cbQuitProgram contains the tab number which initiated a Quit operation. Tabs are numbered starting from zero.

gtkwave::cbReloadBegin contains the name of a trace being reloaded. This is called at the start of a reload sequence.

gtkwave::cbReloadEnd contains the name of a trace being reloaded. This is called at the end of a reload sequence.

gtkwave::cbStatusText contains the status text which goes to stderr.

gtkwave::cbTimerPeriod contains the timer period in milliseconds (default is 250), and this callback is invoked every timer period expiration. If Tcl code modifies this value, the timer period can be changed dynamically.

gtkwave::cbToEntryUpdated contains the value stored in the “To:” widget when it is updated.

 contains the total number of traces. This is called when traces are added, deleted, etc. from the viewer.

gtkwave::cbTreeCollapse contains the flattened hierarchical name of the SST tree node being collapsed.

gtkwave::cbTreeExpand contains the flattened hierarchical name of the SST tree node being expanded.

gtkwave::cbTreeSelect contains the flattened hierarchical name of the SST tree node being selected.

gtkwave::cbTreeSigDoubleClick contains the name of the signal being double-clicked in the signals section of the SST.

gtkwave::cbTreeSigSelect contains the name of the signal being selected in the signals section of the SST.

gtkwave::cbTreeSigUnselect contains the name of the signal being unselected in the signals section of the SST.

gtkwave::cbTreeUnselect contains the flattened hierarchical name of the SST tree node being unselected.

An example Tcl script follows to illustrate usage.

```
proc tracer {varname args} {
    upvar #0 $varname var
    puts "$varname was updated to be \"$var\""
}

proc tracer_error {varname args} {
    upvar #0 $varname var
    puts "*** ERROR: $varname was updated to be \"$var\""
}

set ie [ info exists tracer_defined ]
if { $ie == 0 } {
        set tracer_defined 1

        trace add variable gtkwave::cbTreeExpand write "tracer
gtkwave::cbTreeExpand"
        trace add variable gtkwave::cbTreeCollapse write "tracer
gtkwave::cbTreeCollapse"

        trace add variable gtkwave::cbTreeSelect write "tracer
gtkwave::cbTreeSelect"
        trace add variable gtkwave::cbTreeUnselect write "tracer
gtkwave::cbTreeUnselect"

        trace add variable gtkwave::cbTreeSigSelect write "tracer
gtkwave::cbTreeSigSelect"
        trace add variable gtkwave::cbTreeSigUnselect write "tracer
gtkwave::cbTreeSigUnselect"

        trace add variable gtkwave::cbTreeSigDoubleClick write
"tracer gtkwave::cbTreeSigDoubleClick"
        }
```

```
puts "Exiting script!"
```

Appendix F: Implementation of an Efficient Method for Digital Waveform Compression

Anthony Bybell
Advanced Micro Devices, Inc.
Austin, Texas
anthony.bybell@amd.com

***Abstract*—An efficient method in both speed and size for the reformatting, compression, and storage of digital waveform data as generated by digital system simulators is described.**

Keywords—Verilog; VCD; digital; waveform; compression

I. Introduction

Compression of analog waveform data has received much attention due to the shift from analog to digital media for the storage and delivery of entertainment content. A large number of lossy formats (e.g., MP3, ATRAC, RealAudio) and lossless formats (e.g., ALAC, MPEG-4 ALS, FLAC) exist ranging from proprietary to open source offerings.

Much less focus has been directed toward the efficient compression and retrieval of digital waveform data. One significant source of such data is the simulation of digital systems representing complex VLSI designs. IEEE 1800 [1] describes a format known as Value Change Dump (VCD), which although well-documented and supported, leaves much to be desired in file size and reader access speed: a flat text file is not a compressed, random access database. To this end, various commercial products [2][3][4][5] have surfaced to address size and performance issues, but the algorithms used by them to process digital waveform data have not been disclosed. Nevertheless, for [4] its writer API [6] which provides an interface to simulators, and its reader API [7] which provides an interface to other tools such as waveform viewers can yield substantial hints to the details of a commercial database's implementation. Of the sparse published information to be found regarding the topic of digital waveform compression, the approach described in [8] is an instructive starting point for study.

The approach described in this paper separates the waveform data generated by a simulator into a number of independent, temporal streams that are individually preprocessed, compressed, deduplicated, and finally emitted into a database either literally or as a reference to a previously encountered equivalent stream.

II. VCD file format

As described in [1], VCD is an ASCII-based file format for the storage of digital waveform data that is relatively easy to generate and to parse. VCD files contain three sections: header information, node information, and value changes.

A. Header Information

Header information is trivial. It contains information such as the simulator version, the timescale of the simulation, and optional comments.

B. Node Information

Node information contains a series of scope/upscope declarations and variable declarations.

Scope declarations contain a scope name and a scope type (such as "module") along with an appropriately paired "upscope" declaration.

A variable declaration contains variable type, size, and name fields, as well as an encoded version of an unsigned nonzero integer *identifier_code* value. *Identifier_code* values are encoded by most commercial simulators from an unsigned nonzero value *v* into bijective base-94 printable ASCII in character array *V* according to the following algorithm:

```
1: $i \leftarrow 0$
2: while $v \neq 0$ do
3:     $v \leftarrow v - 1$
4:     $V_i \leftarrow (v \bmod 94) + 33$
5:     $v \leftarrow v / 94$
6:     $i \leftarrow i + 1$
7:     end while
```

Fig. 1. Conversion of an unsigned nonzero integer value into a bijectively encoded character array

This bijective encoding is significant in that the values 1 to 94 are encoded, not 0 to 93. An interesting side effect of this is that it is impossible to construct two or more strings representing the same integer value as this number system lacks a zero symbol [9]. As each string in the set of all possible strings generates a unique integer value, it can be exploited for perfect hashing, thus eliminating the need during subsequent VCD file reading to process *identifier_codes* as strings.

Decoding of an encoded value from character array *V* into unsigned integer value *v* is similar:

```
1: $i \leftarrow V_{length} - 1$
2: $v \leftarrow 0$
3: while $i \geq 0$ do
4:     $v \leftarrow v * 94 + (V_i - 32)$
5:     $i \leftarrow i - 1$
6:     end while
```

Fig. 2. Decoding a bijectively encoded character array into an unsigned nonzero integer

The *identifier_code* is used to correlate a given variable declaration to its entries in the value changes section. As a space optimization, if a simulator identifies that two or more variables are functionally equivalent (e.g., as with a clock that propagates across a functional model), then it may reuse the same *identifier_code* for all of the *aliases* of the initial declaration.

To assist in parsing a VCD file so that associative arrays are not required to look up *identifier_codes*, the *identifier_code* starts at a value of one and increments by one from its previous maximum for each succeeding variable declaration that is not an *alias*. Thus, a simple array where the *identifier_code* functions as an array index suffices for lookups. VCD parsers taking advantage of this "parse by value" scheme must revert to using associative arrays ("parse by handle") of unsigned integers or some similar method to process *identifier_codes* when this property does not hold.

C. *Value Changes*

The value changes section is the final section of the VCD file and generally is the most substantial portion of the file with regard to the percentage of total file size. It contains digital waveform data stored as a series of *simulation_time* items and *value_change* items. A *value_change* item as specified by [1] is an encoded *identifier_code* paired with a value that is an integer, a double-precision floating-point number, or a multi-value 4-state "01XZ" (MVL-4) bit string.

A *simulation_time* item associates with all *value_change* items that follow until the next encountered *simulation_time* item. Thus, the value changes section encodes a series of {time, *identifier_code*, value} *transition triples* representative of all the traced variables in a simulation. As there is no limitation on where a *value_change* for a given *identifier_code* can be located, the determination of all of the *transition triples* for a given variable requires processing of the entire value changes section. This is clearly inefficient for interactive tools such as waveform viewers as much irrelevant data must be processed. To ameliorate this situation, most commercial tools such as [3][4][5] will convert a VCD file to a native database format rather than process the VCD file directly.

III. Limitations and acceptable shortcuts

Some simulation tools such as [5] provide precise reconstruction of the ordering for all the *value_change* items that share the same *simulation_time* value. Other commercial tools such as [4] by default do not preserve this ordering unless a sequence ordering option is specifically enabled. In [4], it is stated that enabling sequence ordering increases file size and simulation run time. The precise ordering offered by [5] is generally not enabled in [4] as it is not useful for functional debug of race-free zero-delay designs. It does have its place however, for debugging test bench code.

It is to be noted that for a variable that glitches (i.e., a single *identifier_code* contains multiple *value_change* items for a given *simulation_time* value), at least the final *value_change* for the variable must be stored. This is to ensure that the final state a variable settles at for a given *simulation_time* value is visible when the database is queried. A well-known glitch suppression method employed by Verilog simulators is that simulation data for all variables that change within a time step are written all at once during the REASON_ROSYNCH callback for the time step.

In order to minimize file size, sequence ordering is not preserved by the approach described in this paper. For the FST file format implementation in [10] that implements the algorithms described in this paper, glitch transitions are preserved as doing so simplifies file generation, though a compile-time option is available that permits elision of all glitch *value_changes* except the final one.

IV. Design goals

The features listed below are found in most commercial digital waveform database implementations and were treated as design goals for the implementation in [10].

A. *Fast generation*

Given that a common usage case is the interactive viewing of waveform data, there should be minimal overhead in the generation of a database file in order to make it available quickly for viewing.

B. *Small file size*

A small file size is highly desired; however this must be balanced with generation time.

C. *Fast reader initialization*

Opening up a database file for reading should proceed quickly in that large, irrelevant portions of the database do not need to be processed.

D. *Fast extraction of a handful of variables or all variables*

It is to be expected that extracting all of the data from a large database file will take some amount of time, but it is certainly not desired that extracting a handful of variables in order to perform debug will take much time at all, especially given that adding new signals into an interactive waveform viewer session is often an incremental and repetitive process.

E. *Allow reading of a database that is still writing*

It is helpful if a file can be accessed while it is being written as debugging can start with no need to wait for simulation to finish. An easy way to accomplish this is to segment the writing of the database into a series of independent blocks or sections such that a reader is permitted to access blocks whose contents have finalized.

V. Database writer API

As the internal format of any database is subject to change, a database writer API was created for [10] similar to that of [6] to shield users from the implementation. A reader API also was created for [10], but it is beyond the scope of this paper.

The database writer API was designed to map its function onto a superset of VCD constructs and also provide for future expansion. It achieves this by using tagged blocks in conjunction with a tagged binary format.

Similar to [6], the handle value returned by the API upon variable creation maintains the exact VCD "parse by value" property of the incrementing *identifier_code* values described earlier. Thus, for most simulators, there is no additional memory required to be allocated in a simulator for the storage of variable handles returned by the writer API. As such, much existing VCD emission code can be converted with minimal modification to use the database writer API instead.

In order to allow multiple, separate databases to be written simultaneously in a thread-safe lock-free manner, upon database creation the API generates an opaque *context* value representing the database. All subsequent API operations up to and including the closing of the database then refer to the *context*.

VI. Compression techniques and compressed data types used by the database writer API

The following three subsections are relevant to database construction so they will now be discussed.

A. *Iterative compression*

To facilitate late signal additions, node information is stored in a *hierarchy tree*, which is a separate file whose filename contains a ".hier" extension. When the writer API is directed to close the database, this file is compressed twice with byte-based compressor LZ4 [11] , it is appended as a block in the database, and then the ".hier" file is deleted. Experimentation has shown that [6] also double compresses data with its own proprietary byte-based LZ4-like compressors prior to writing into its database. Why was a compressor library such as zlib (used by gzip) not used instead?

One reason discovered from performance analysis is that zlib is slow when very many small, discontiguous regions of memory need to be compressed: zlib must reinitialize a non-trivial amount of its compressor state for every new invocation.

Another reason is that an iterative, multi-layered approach performs better for some data. The following table compares zlib against an iterative LZ4 compression strategy and also a combination of the two. In the final row of the table, LZ4 functions as a preprocessor for zlib, providing the smallest resulting file size at a total compression speed even faster than gzip executing at its weakest compression level.

TABLE I. COMPRESSION OF 111MB OF HIERARCHY TREE DATA (4.5 MILLION VARIABLE DECLARATIONS)*

Compressor	***Compressed size (bytes)***	***Compression time (seconds)***
gzip -1	15,213,337	1.15
gzip -4	13,345,371	1.53
gzip -9	12,236,236	13.39
lz4 (run once)	23,825,848	0.28
lz4 (run twice)	13,807,145	0.39
lz4 (run twice) then gzip -1	11,045,828	1.00

* All execution runs documented in this paper were run single-threaded on a Dell Optiplex 760 with a 3.0GHz Core2Duo processor and 8GB of RAM.

Processing any data through zlib destroys the alignment of data at byte boundaries due to the properties of Huffman encoding [12], so if zlib is used, it should be the last step for any form of byte-based iterative compression. It is to be noted that files created by [6] are often quite compressible by gzip and other compressors, so it may be deduced that to maintain high performance, [6] does not process much or any of its data using compression routines that employ statistical encoding or other bit reduction techniques.

B. Variable-length unsigned integer storage

In keeping with the previous observation concerning the usage of byte-based compressors, to save space prior to compression, various items of data are encoded throughout the database as variable-length unsigned integers using the formatting as documented in [13]. The algorithm to convert an unsigned integer *v* into a variable-length unsigned integer representation contained in array *V* is shown in Figure 3.

```
1: $i \leftarrow 0$
2: while ($k \leftarrow v >> 7) \neq 0$ do
3:     $V_i \leftarrow (v \& 0x7F) | 0x80$
4:     $i \leftarrow i + 1$
5:     $v \leftarrow k$
6: end while
7: $V_i \leftarrow v \& 0x7F$
```

Fig. 3. Conversion of an unsigned integer to an array of bytes

The following table illustrates a number of edge case values and the variable-length unsigned integer representations resulting from processing by this algorithm.

TABLE II. COMPARISON OF REPRESENTATIONS OF DECIMAL, HEXADECIMAL, AND VARIABLE-LENGTH UNSIGNED INTEGERS

Decimal	***Hexadecimal***	***Variable-Length Unsigned Integer***
0	00	00
1	01	01
127	7F	7F
128	80	80 01
130	82	82 01
16383	3FFF	FF 7F
16384	4000	80 80 01
65535	FFFF	FF FF 03
65536	10000	80 80 04

As shown in the table, this method achieves very good space savings for relatively small positive values. These small values occur quite often in the database.

It is sometimes useful for a payload of known bit width to "hitchhike" onto the low-order bits of another integer such that the shifted and combined result is encoded as a single variable-length integer. This feature is exploited in various places in the writer, most significantly for the encoding and compression of a stream of values for a single-bit MVL-4 variable. It is often the case that a *value_change* item in these streams can be transformed into a single byte. In a VCD file, such value changes occupy at least three bytes and do not compress well as they lack locality with related value changes.

C. Variable-length signed integer storage

As the sign of a number can function as a flag bit, variable-length signed integers as documented in [13] occasionally prove useful. The algorithm to convert a signed integer *v* into a variable-length signed integer representation contained in array *V* is shown in Figure 4.

```
1: $i \leftarrow 0$
2: $more \leftarrow$ true
3: do
4:     $b \leftarrow (v \& 0x7F) | 0x80$
```

```
5:      $v \leftarrow v >> 7$
6:      if (($v$ = 0) and ($b$ & 0x40 = 0)) or
7         (($v$ = -1) and ($b$ & 0x40 = 1))
8:          $more \leftarrow$ false
9:          $b \leftarrow b$ & 0x7F
10:         end if
11:     $V_i \leftarrow b$
12:     $i \leftarrow i + 1$
13:     while $more \neq$ false
```

Fig. 4. Conversion of a signed integer to an array of bytes

The following table illustrates a number of edge case values and the variable-length signed integer representations resulting from processing by this algorithm.

TABLE III. COMPARISON OF REPRESENTATIONS OF DECIMAL, HEXADECIMAL, AND VARIABLE-LENGTH SIGNED INTEGERS

Decimal	***Hexadecimal***	***Variable-Length Unsigned Integer***
0	0000	00
1	0001	01
-1	FFFF	7F
2	0002	02
-2	FFFE	7E
63	003F	3F
-63	FFC1	41
64	0040	C0 00
-64	FFC0	40
127	007F	FF 00
-127	FF81	81 7F
128	0080	80 01
-128	FF80	80 7F

As this encoding is not as space efficient as the encoding for variable-length unsigned integers, usage of signed variable-length integers in the writer in [10] was limited to a handful of areas where experimentation determined that is was beneficial.

VII. Processing of declarations and time/value change data into the database format

As shown in Figure 5, there are four basic phases that the database writer cycles through based on the API calls it receives.

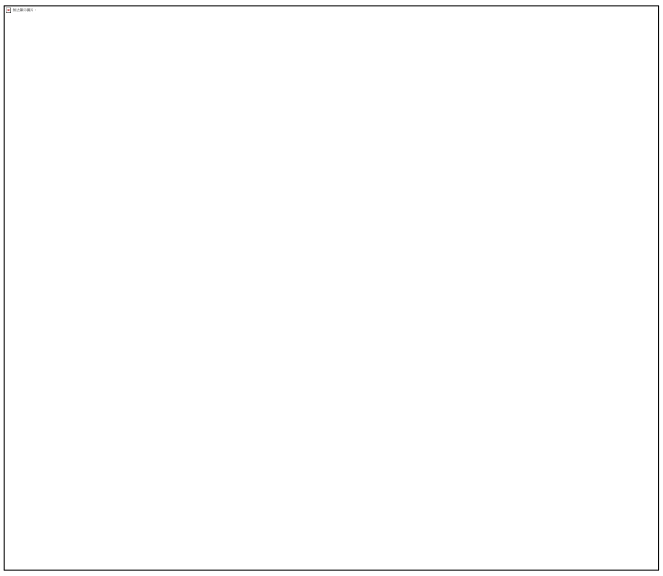

Fig. 5. Four basic phases of database writer execution

A. Scope and variable declaration processing

Minimal processing is performed upon node information. Mostly, this involves converting scope/upscope/variable declarations into an easily processed tagged binary format. Ordering of declarations as encountered by the writer API is strictly maintained as there is no compelling reason to modify the declared order prior to emission into the *hierarchy tree*. Some commercial tools such as [6][7] maintain separate data structures for scope declarations and for variable declarations. This allows for scope data to be retrieved faster by a reader during initialization and may aid in compression.

Once scope and upscope declarations are written into the *hierarchy tree*, they are no longer needed by the writer. These declarations are only useful for database reader code: the writer is more concerned with variable declarations.

During the writer API call for a variable declaration, various auxiliary structure arrays are updated by the writer. In [10], all of the following arrays exist as hidden temporary files and are subject to being mmap()'d in and munmap()'d out as necessary:

- Geometry: contains each variable's storage requirements, namely the length and number of bytes per unit of length. This is used by readers to speed up initialization for variables by not requiring a traversal of the *hierarchy tree*. Upon the closing of the database, *geometry* data are compressed and appended to their own block in the database.
- Bits Array: contains a full checkpoint of the database state across all variables. It exists to allow for reads to start at a time other than the initial simulation time, as well as to allow the opportunity for the future implementation of block splicing utilities.
- Time Chain: stores each *simulation_time* value encountered by the writer. Values in the *time chain* as emitted by simulators are monotonically increasing. When elements comprising the *time chain* are emitted to the database, a *simulation_time* value is compared to its preceding *simulation_time* value and the mathematical delta between the two is stored in the database. A 64-bit

time value thus can reduce to a much smaller variable-length unsigned integer value in the database. Additionally, the *time chain* aids in allowing for efficient value change compression as *value_changes* are transformed by the writer into a consecutive sequence of *time chain* index deltas paired with a "hitchhiker" value. The *time chain* is analogous to XTags in [6][7], however this implementation detail is hidden from the writer API.

- *Value/Position Structure Array*: contains a four element structure for each variable consisting of {position in *bits array*, variable-length, position in *value change preprocessing buffer*, *time chain* index}. Each time the writer processes a *value_change* for a variable, the final two fields of this structure are updated. This structure is used solely by the writer to provide bookkeeping and is not written into the database.

B. Time and value change processing

As writer API calls generate time changes, each change is added to the *time chain* and the current *time chain* index is incremented. As the APIs in [6][7] can be interfaced to various digital and analog simulators, integer or floating-point values are possible for their XTags. For Verilog simulation, all time values generated are 64-bit integers. The example that follows in the next subsection will use only integer time values for clarity. The implementation in [10] currently uses only 64-bit integer time values.

For value changes, the *value_change* items are reformatted and stored in the *value change preprocessing buffer*. This buffer is a large holding area that receives dynamically converted *value_change* items prepended onto a linked list of {previous list item pointer, *time_chain* index delta, value} triples. There is one linked list per variable, with each variable's corresponding element in the *value/position structure array* storing the head pointer for the variable's list of time reversed value changes.

To demonstrate value change processing, the following series of time and value changes are received by the writer API:

```
#0
A = '0'
B = '1'
#10
A = '1'
#15
B = '0'
#20
B = '1'
#30
A = '0'
```

After time value 30, the time index value would be 4 (starting the index count from zero) and the *time chain* would appear as follows in memory on a little-endian machine:

```
0000: 00 00 00 00 00 00 00 00
0008: 0A 00 00 00 00 00 00 00
0010: 0F 00 00 00 00 00 00 00
0018: 14 00 00 00 00 00 00 00
0020: 1E 00 00 00 00 00 00 00
```

In the *value change preprocessing buffer*, previous list item pointers are currently stored in [10] as 32-bit unsigned integers (limiting the size of the buffer to 4GB), the *time chain* index deltas are stored as variable-length unsigned integers, and finally the values themselves are stored as raw ASCII. This was a design choice dictated by performance in order to simplify this portion of value change processing.

For the example time and value changes, the final state of the *value change preprocessing buffer* would appear as follows:

```
0000: 21 00 00 00 00 00 30 00
0008: 00 00 00 00 31 01 00 00
0010: 00 01 31 07 00 00 00 02
0018: 30 13 00 00 00 01 31 0D
0020: 00 00 00 03 30 -- -- --
```

The first character (0x21 / "!") shown in the buffer at offset zero is nothing more than an unreachable placeholder character. During list traversal, a list item pointer containing a value of zero signifies the end of a list traversal.

To assist in demonstrating the traversal of the list for variable "A", only the bytes relevant to variable "A" will be shown such that the back pointers are shaded, the time change index values are in boldface italics, and the values are unformatted text:

```
0000: .. 00 00 00 00 00 30 ..
0008: .. .. .. .. .. 01 00 00
0010: 00 01 31 .. .. .. .. ..
0018: .. .. .. .. .. .. .. 0D
0020: 00 00 00 03 30 -- -- --
```

Now showing only the bytes relevant to variable "B" such that the back pointers are shaded, the time change index values are in boldface italics, and the values are unformatted text:

```
0000: .. .. .. .. .. .. .. 00
0008: 00 00 00 00 31 .. .. ..
0010: .. .. .. 07 00 00 00 02
0018: 30 13 00 00 00 01 31 ..
0020: .. .. .. .. .. -- -- --
```

The *value/position* structures {position in *bits array*, variable-length, position in *value change preprocessing buffer*, *time chain* index} for each variable would appear as follows at the end of time value 30:

```
A: {0, 1, 0x1F, 4}
B: {1, 1, 0x19, 3}
```

It is not necessary when processing value changes to update the *bits array*. As the *value/position* structure for each variable points to its final value change in the block, the *bits array* can easily be updated later to avoid unnecessary overwrites.

C. Context flush

When the *value change preprocessing buffer* is full or is about to become full, a *context flush* sequence will occur either when the next time change is encountered or when the database is closed. To prevent buffer overruns, the *value change preprocessing buffer* is dynamically enlarged as needed.

A *context flush* is analogous to the "flush session" documented in [6] and it performs the following actions which are summarized in Figure 6:

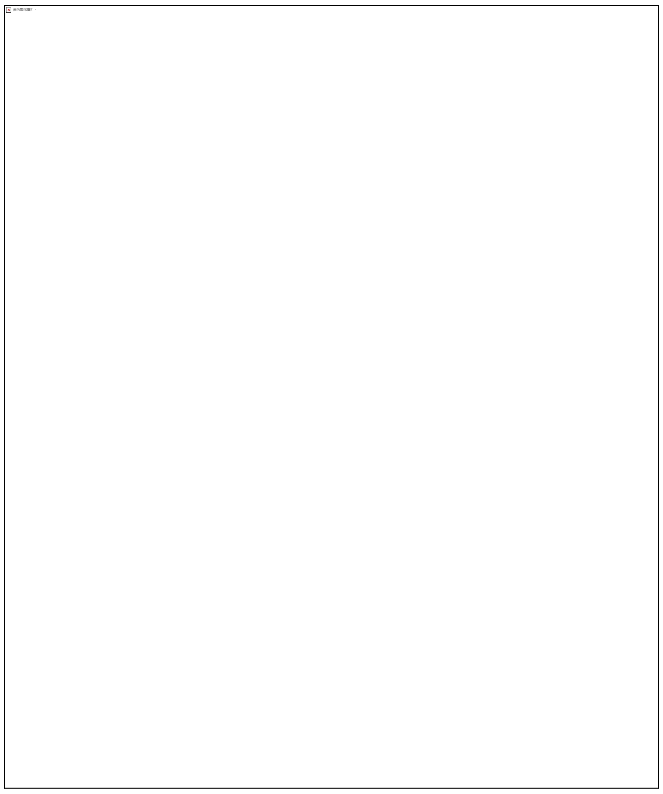

Fig. 6. Summary of the operations performed by a *context flush*

1) **Create a new, invalid block in the database. A major advantage of segmenting the data into independent blocks is that it gives reader code the ability to access all the currently valid blocks previously generated by a simulation while the simulation is still running.**

2) **Create new buffers if a producer-consumer model of parallel execution is optionally enabled. Upon creation of new buffers, a separate context flush thread is spawned to process the old buffers while the writer API returns control back to the calling process. By making time and value change processing as described earlier simple, more work can be offloaded to the context flush thread. This minimizes how long a simulator is blocked by a context flush. Parallel execution is more useful when zlib is enabled to compress transformed value_change data: LZ4 is so fast that the overhead of parallel execution can slow writing down.**

3) **Compress, write, and update the bits array. The bits array represents the checkpoint of simulation before any value changes in the block have been encountered. After the bits array has been emitted to the database, it may then be updated to reflect the final value change for each variable. As the value/position structure for each variable contains the position of its final value change in the block, the bits array can be quickly updated.**

4) **Segregate and reformat each variable's data into a contiguous stream of memory locations through a serialization process. For each variable, the linked list pointed to by its value/position structure is traversed and the time changes and values are reformatted and emitted into another buffer. As the list traversal proceeds in the reverse of simulation order, the reformatted data is built by [10] into descending memory locations in the destination buffer in order to reconstruct the original simulation order in a single pass.**

Recall from the earlier example how the relevant bytes of the *value change preprocessing buffer* for variable "A" would appear as follows at the end of time value 30:

```
0000: .. 00 00 00 00 00 30 ..
0008: .. .. .. .. .. 01 00 00
0010: 00 01 31 .. .. .. .. ..
0018: .. .. .. .. .. .. .. 0D
0020: 00 00 00 03 30 -- -- --
```

Again, recall the *value/position* structure {position in *bits array*, variable-length, position in *value change preprocessing buffer*, *time chain* index} for variable "A" as it would appear at the end of time value 30:

```
A: {0, 1, 0x1F, 4}
```

Starting at 0x1F, the following sets of value change data are encountered:

```
0x1F: 03 30 ('0')
0x0D: 01 31 ('1')
0x01: 00 30 ('0')
```

As this variable is a single-bit variable (shown by the length = 1 field for the *value/position* structure for "A"), time/value changes may be encoded using a variable-length unsigned integer with a fixed-width payload for the value specified in the low-order bits. For this example, assume an MVL-4 encoding where '0' equals 0, '1' equals 1, 'x' equals 2, and 'z' equals 3.

```
03 30 ('0') = (0x03 << 2)|0 = 0x0C
01 31 ('1') = (0x01 << 2)|1 = 0x05
00 30 ('0') = (0x00 << 2)|0 = 0x00
```

It follows that VHDL would use a larger payload to encode the MVL-9 values "01XZHUWL-".
Thus, the following sequence of bytes when correlated against the *time chain* and built in reverse in memory represent the time and value changes for variable "A":

```
00 05 0C
```

It should be obvious that for repetitive value changes such as those generated by clocks, a highly compressible, recurring stream of bytes such as the following will result:

```
04 05 04 05 04 05 04 05 04 05 …
```

Variables occupying more than one byte (bit vectors, integers, reals, strings, etc.) are handled differently in [10] in that the time delta value is stored as its own variable-length unsigned integer and the value is stored either in a packed binary representation or as raw ASCII. For reader code to determine which representation was used when an item was written into the database, a "hitchhiker" payload is contained in the low-order bits of the time delta value.
Unlike the approach taken in [8], there was no attempt made to predict values in a variable's value change stream. As a simple example of prediction, for a single-bit variable, if its value is '0', it can be predicted with a high degree of confidence that its next

value change is almost always a ‘1’, and vice-versa. Thus, to create a stream of bytes that is more compressible, the XOR of the actual value versus the predicted value would be stored instead of the actual value.

5) **Compress the reformatted data. After serialization, a variable’s data are compressed using LZ4. Unlike [6], in [10] this is only performed once. In lieu of double compression, multi-bit MVL-4/MVL-9 values are stored packed as eight bits per byte when a value is scanned and discovered to contain only ‘0’ and ‘1’ value bits. A modified form of Duff’s Device [14] is used to perform the packing operation.**

6) **Deduplicate the compressed data and generate an entry in the position table. Deduplication of the compressed, serialized data then occurs where the data are either compared against existing data stored in a Judy array [15] or a structure based on a move-to-front reference sorted Jenkins hash [16] array. The Jenkins hash deduplication performs slightly faster than the Judy array, however it may be subject to patent issues described in [17] so its selection is determined at compile time for [10] by a compile time option.**
If the compressed and *serialized* data are not already present in the deduplication structure, then the data are inserted into the deduplication structure, are emitted into the database, and the offset representing where the data are stored in the block is then stored in an element in the *position table* indexed by the *identifier_code* for the variable. Otherwise, a *dynamic alias* (the *identifier_code* for the matching data subtracted from zero, thus making it a negative value) is stored into the *position table* and no redundant data are added to the database. Variables which have no value changes are assigned the offset of zero, which never represents valid data and never represents a valid *identifier_code*. The *position table* as stored in the database is not compressed further in order to aid in reader access speed. This is to give reader code the opportunity to perform a partial decompression of this table if all variables do not need to be accessed. (e.g., if there are two million variables and the maximum *identifier_code* for a variable that needs to be read is 500000, then decompression to determine variable offsets can stop approximately one-quarter of the way through this table.) It is to be noted that as the transformed value change data for a variable have been *serialized* and are located in a contiguous range of locations in the database, reader code can immediately and directly seek to and process a variable’s data once the offset for the location of the data is known.

7) **Compress the position table. After all variables have been deduplicated, consecutive elements of the position table for non-dynamic aliases (positive values) are delta compressed and stored in the database as a positive variable-length signed integer. Dynamic aliases (negative values) are stored as a variable-length negative signed integer. To save additional space, a match of the current dynamic alias with the most recent previous one is represented as a value of zero. When one or more consecutive elements in the position table contain a value of zero (meaning each has no value changes), the zeros are run-length encoded then stored in the database as a variable-length unsigned integer. To differentiate in the reader between the two types of data (delta offsets or dynamic aliases versus counts of runs of zeros), the least significant bit of the variable-length integer is treated as a “hitchhiker” flag that differentiates between the two types of data, and either the signed or unsigned variable-length integer decoder are invoked appropriately.**

8) **Compress the time table. The final structure requiring emission into the database is a compressed version of the time table. It is first preprocessed by converting it to a series of variable-length unsigned integers representing deltas of consecutive time values. Recall the time table values encountered earlier:**

```
0000: 00 00 00 00 00 00 00 00
0008: 0A 00 00 00 00 00 00 00
0010: 0F 00 00 00 00 00 00 00
0018: 14 00 00 00 00 00 00 00
0020: 1E 00 00 00 00 00 00 00
```

The variable-length unsigned integers representing the delta compressed values would occupy this series of bytes:

```
00 0A 05 05 0A
```

This data exhibits the property that it is highly repetitive as time deltas in a simulation tend to occupy a small number of fixed “pound delay” values. As there is only one *time table* per block, zlib compression overhead with respect to processing of the full block is low, so the data are run through zlib at its highest compression level and then are emitted into the database. Floating-point time values would require different compression techniques such as that described in [18].

9) **Finalize the block. At this point, the context memory has been processed into a block in the database. The block is then marked as valid (in order to allow simultaneous reading of the database as it is generating), the context memory is recycled, and the writer API continues collecting more time changes and value changes until the database is closed.**

D. Close the database
Closing the database compresses and appends the *hierarchy tree* as a block and marks the entire database as finalized. Any externally visible temporary files that were created are deleted. In addition, the full database can optionally be recompressed using zlib and be emitted as a single special block. Compression as a single block overcomes the zlib performance issues discussed earlier.

VIII. Experimental results
Compression size and speed results for two non-trivial VCD files will be shown below. The tool vfast can be found in [4], the tool vcd2fst (results in italics) which implements the approach described in this paper can be found in [10], the tool vcd2vpd can be found in [2], and the tool vcd2wlf can be found in [5]. The utility gzip can be found in any Linux distribution. Results for [10] are shown in italics.

TABLE IV. COMPRESSION OF 1.5GB OF VCD (57312 TOTAL VARIABLE DECLARATIONS, 20826 ARE ALIASES)

Compressor	***Compressed size (bytes)***	***Compress time (seconds)***
gzip -1	483,489,275	28.88
gzip -4	453,921,410	39.74
gzip -9	425,526,503	338.32
vcd2fst (LZ4)	*19,964,916*	*18.34*
vcd2fst (LZ4) plus gzip -4	***11,528,306***	***19.39***
vcd2fst (zlib)	*12,144,313*	*23.89*
vcd2fst (zlib) plus gzip -4	*11,164,461*	*24.41*
vcd2fst (LZ4) no deduplication	*71,600,821*	*18.68*
vfast	Sanitized in public release due to anti-benchmarking clause in simulator EULAs.	
vfast plus gzip -4		
vfast -compact		
vfast –compact plus gzip -4		
vcd2vpd		
vcd2vpd plus gzip -4		
vcd2wlf		
vcd2wlf plus gzip -4		

TABLE V. COMPRESSION OF 5.0GB OF VCD (5.8 MILLION TOTAL VARIABLE DECLARATIONS, 0 ARE ALIASES)

Compressor	***Compressed size (bytes)***	***Compress time (seconds)***
gzip -1	1,331,508,502	87.44
gzip -4	1,262,303,074	124.12
gzip -9	1,258,996,174	1024.90
vcd2fst (LZ4)	*35,087,853*	*76.20*
vcd2fst (LZ4) plus gzip -4	***18,905,265***	***77.71***
vcd2fst (zlib)	*30,582,983*	*100.85*
vcd2fst (zlib) plus gzip -4	*16,852,741*	*101.76*
vcd2fst (LZ4) no deduplication	*88,209,725*	*75.76*
vfast	Sanitized in public release due to anti-benchmarking clause in simulator EULAs.	
vfast plus gzip -4		
vfast -compact		
vfast –compact plus gzip -4		
vcd2vpd		
vcd2vpd plus gzip -4		
vcd2wlf		
vcd2wlf plus gzip -4		

IX. Conclusion

Processing digital waveform data for efficient storage and retrieval does not require computationally expensive analysis heuristics. Instead, a fast method can be employed that separates value change data into individual streams that are reformatted and compressed independently. Streams identified as equal can be deduplicated dynamically, further reducing database size. The experimental results show that compression ratios and execution speeds achieved by this method can significantly exceed those of prior art.

X. Future work

Simulation data such as EVCD as described in [1] appear to compress much better with FastLZ [19] than LZ4 in [10], so it may be advantageous to employ multiple fast compression algorithms that are automatically selected based on the type of data being compressed.
Using [18] in conjunction with [11] could prove highly effective for reducing the storage requirements of IEEE-754 floating-point time change data in implementations that store such data.
Bijective encoding of MVL-4 and MVL-9 value change strings into variable-length perfect hash integers could merit further study as a space saving technique.

REFERENCES

[1] IEEE Computer Society, “IEEE Standard for System Verilog—Unified Hardware Design, Specification, and Verification Language,” 2009, pp. 572-592.

[2] Synopsys, Inc., “VirSim User Guide Version 4.4,” 2003, pp. 379-412.

[3] Cadence Design Systems, Inc., “SimVision User Guide Product Version 8.2,” 2009, pp. 109-124.

[4] Synopsys, Inc., “Verdi^3 and Siloti Command Reference,” 2013, pp. 1543-1559.

[5] Mentor Graphics Corporation, “Questa SIM User’s Manual Including Support for Questa SV/AFV Software Version 10.0d,” 2011, pp. 679-694.

[6] Synopsys, Inc., “Open FSDB Writer,” 2013, pp. 1-186.

[7] Synopsys, Inc., “Open FSDB Reader,” 2013, pp. 1-144.

[8] E. Naroska, et al., “A Novel Approach for Digital Waveform Compression,” ASP-DAC, 2003, pp. 712-715.

[9] A.R. Forslund, “A logical alternative to the existing positional number system,” Southwest Journal of Pure and Applied Mathematics, Volume 1, September 1995, pp. 27-29.

[10] A. Bybell, “GTKWave User Manual,” 2013, pp. 1-149.

[11] Y. Collet, “lz4: Extremely Fast Compression algorithm,” 2013, http://code.google.com/p/lz4/.

[12] D.A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” Proceedings of the I.R.E., September 1952, pp. 1098-1102.

[13] DWARF Debugging Information Format Committee, “DWARF Debugging Information Format Version 4,” 2010, pp. 161-163, 217-218.

[14] R. Holly, “A Reusable Duff Device,” Dr. Dobb’s Journal, August 2005, pp. 73-74.

[15] A. Silverstein, “Judy IV Shop Manual,” 2002, pp. 1-81.

[16] B. Jenkins, “Algorithm Alley: Hash Functions,” Dr. Dobb’s Journal, Sep. 1997, pp. 107-109, 115-116.

[17] C.A. Waldspurger, “Transparent sharing of memory pages using content comparison,” US 7620766 B1, 2009, pp. 1-22.

[18] M. Burtscher, P. Ratanaworabhan, “High Throughput Compression of Double-Precision Floating-Point Data,” 2007, pp.1-10.

[19] A. Hidayat, “FastLZ, free, open-source, portable real-time compression library,” 2013, http://fastlz.org.