Controlling the Run with SDL

Product Version WXE 23.03
February 2023

Contents

Overview of State Description Language (SDL)

SDL is a run-time debugging feature that enables, based on design signal values, to control several aspects of the run such as when to stop, or when to capture probed data (tracing). Certain run modes enable to schedule execution of Tcl or XEL commands during a run using the `EXEC` facility.

SDL is the trace control and break mechanism used by WXE. It can control data capture in In-Circuit Emulation (ICE) modes (both STB and LA modes) and Simulation Acceleration (SA) mode. It can stop the running design in Vector Debug (VD), STB and SA modes, and it can schedule execution of XEL commands during a run in STB and SA modes. SDL is specified as a program in human-readable text.

In VD, STB and SA modes, SDL will stop the run when executing a `TRIGGER` statement.

In Dynamic Target and LA modes, executing a TRIGGER statement typically stops only data tracing (probed data capturing). It is possible to override the default behavior by using XEL commands, and make the trigger stop the design clocks as well.

Additionally, if using a run mode that supports *Conditional Acquisition*, SDL can control when to capture or not capture probed data, using `ACQUIRE` or `NO_ACQUIRE` statements.

In most run modes, an `EXEC` statement can be used to execute XEL commands while the run is in progress.

Essentially, SDL controls the functions of the emulator:

- ❑ When running in STB or VD mode, a trigger condition stops the design clocks and data tracing.
- ❑ When running in LA or Dynamic Target mode under IXCOM, a trigger condition stops data tracing by default, The default behavior can be modified to stop both design clocks and data tracing.
- ❑ When running in SA mode with a design that does not have uncontrolled clocks, a trigger condition stops the design clocks and data tracing.
- ❑ When running in SA mode with a design that has uncontrolled clocks, a trigger condition might either stop the controlled clocks, uncontrolled clocks, data tracing,

or a combination of the above, depending on the settings specified with the `clockConfig` command.

- ❑ SDL can conditionally filter out traced data (Conditional Acquisition feature). The availability of this feature depends on the run mode.

SDL can print out formatted messages into a file and/or the console using the Using DISPLAY Action statement, or write transaction data into a SST2 database. Although `EXEC` can also print out messages by scheduling the Tcl command `puts`, the `DISPLAY` statement has advantages over using `puts` inside `EXEC`, which are discussed in Printing Messages from SDL with the DISPLAY Facility.

SDL provides a way to specify complex conditions, based on signal value patterns along the time axis. Unlike basic breakpoints (which are common in many simulators for stopping the run and are based on a single pattern of signal values) SDL can detect complex patterns along the time axis with relative ease. An example can be: trigger when signal X makes 1000 value transitions without staying at one value more than 2 consecutive cycles.

SDL programs can be quite complex by themselves. So, SDL comes with its own debugging facility called *SDL Trace Dump,* which enables to troubleshoot SDL programs using either the waveform display or text format. See Debugging with the SDL Trace Dump Facility for more details.

If you are running in SA mode (under IXCOM) and swapped the DUT from the software simulator into the Palladium hardware, breakpoints defined for the software simulator using the `stop` command will not function correctly after the swap if they use design objects from the DUT. With some limitations, you can import these breakpoints into an SDL program and use it on Palladium, provided that the breakpoints are of type `object` or `condition`, and use design objects only from the DUT. For more information, see Importing Breakpoints from the Xcelium Simulator into SDL section.

SDL can leverage the programming power of VHDL and/or Verilog languages by connecting to the dynamic RTL (DRTL) instances.

Related Sections:

- Preparing, Compiling and Executing SDL Programs
- Formatting and General Syntax Rules of SDL
- Overview of SDL Constructs
- Using SDL Actions
- Using TRIGGER Action
- Using EXEC Action

- Using NO_ACQUIRE Action

Overview of SDL Instances

SDL can independently track up to 64 different patterns along the time axis. Each pattern is tracked by a state machine (called an SDL instance), which has internal resources such as counters and expression evaluators. Each of the SDL instances operates independent of the others, and can independently stop emulation, control data capturing or invoke XEL commands through `EXEC` statements.

In simple cases, a single SDL instance is enough. A typical application of using multiple SDL instances is to trigger when any one of several possible sequences of events happen. Each SDL instance can be used to identify one of these sequences. Using this example (trigger when a signal makes 1000 value transitions without staying at one value more than 2 consecutive cycles), a set of three SDL instances can be used to detect this pattern independently on three different signals.

While each SDL instance can contain only one state machine (or none at all), it can also contain multiple number of global `EXEC`, `TRIGGER`, `DISPLAY`, `TX`, or `NO_ACQUIRE` statements in the Instance Definition Section (See Specifying SDL Statements).

The number of different SDL instances available during run time must be declared to the compiler using the `sdlInstances` compiler option. If not specified, the compiler will allocate the resources for a single SDL instance.

Enabling and Disabling Individual SDL Instances at Run Time

Each SDL instance can be enabled or disabled at run time (even while the emulator is running) independent of other SDL instances through the `sdl -enable` and `sdl -disable` XEL commands.

The software maintains two lists of SDL instance names, which are referred to as the ***Enabled Instances List*** and the ***Disabled Instances List***.

The Enabled Instances List is a list of all the SDL instances that are always enabled. An instance name can be appended to the list through the XEL command:

```
sdl -enable <SDL_instance_name>
```

The Disabled Instances List is a list of all the SDL instances that are always disabled. An instance name can be appended to it in a similar way using the `-disable` option.

We will refer to all the other SDL instances that do not appear in either the Enabled or Disabled Instances Lists as the ***Uncommitted Instances List***.

The Uncommitted Instances List, as well as all the `TRIGGER/DISPLAY/EXEC/TX/ACQUIRE/NO_ACQUIRE` constructs in Global Definition Section are enabled or disabled as a whole using one of the XEL commands:

```
sdl -enable
sdl -disable
```

For further details, refer to the `-enable` and `-disable` options of the `sdl` command in the *Run-Time Commands* chapter of the *WXE Command Reference Manual*.

When running under the xeDebug GUI, selecting *Enable* or *Disable* for the *SDL program* in the *Control* tab affects only the Uncommitted Instances List and the constructs in the Global Definition section, but does not affect instances in either the Enabled Instances List or the Disabled Instances List.

This mechanism was originally created to enable uninterrupted use of one SDL instance as a “system service” (for example, to unload a streaming memory at given intervals), while using another SDL instance for “regular debugging” (for example, to stop the emulator when a specific condition happens).

Using DRTL Instances in SDL Expressions

SDL can use the values of the output ports of DRTL instances in expressions, anywhere where a design signal can be used. The DRTL instances are defined from the Verilog or VHDL source files, and compiled and downloaded into the emulator at run time. The inputs to DRTL are the design signals. This enables SDL access to complex operations that are available in Verilog or VHDL.

DRTL also has some built in system tasks that mirror SDL actions such as `acquire`, `exec`, `trigger` and `display`.

Related Sections:

- Global and Instance Definition Sections
- Using SDL Actions
- Specifying SDL Statements

- For more information, refer to the Using DRTL Instances in SDL Programs section in *Controlling the Run with Dynamic RTL Guide*.

Enabling and Disabling Individual Trigger, Display, TX, or Exec Statements

In addition to enabling or disabling complete SDL Instances, it is also possible to disable individual `TRIGGER`, `DISPLAY`, `TX`, or `EXEC` statements in any part of the SDL program, provided that they have a label. The system only maintains a single list of disabled labels, and any statement with a label that exists in that list is disabled. Labels are added or deleted from the list through XEL.

Related Topic:

- For more details and other command-line options related to this feature, refer to the `-disable -label` and `-enable -label` options of the `sdl` command in the *WXE Command Reference Manual*.
- Using SDL Actions

Conditional Acquisition Using DCC

SDL syntax provides two methods to define conditional acquisition, one using `ACQUIRE` statements, the other using `NO_ACQUIRE` statements. The two methods cannot be mixed in the same SDL program.

In the first method, probed data is acquired into the trace buffer in every cycle in which any ACQUIRE statement is active (which can be either an ACQUIRE action or a Global ACQUIRE).

In the second method, probed data is acquired into the trace buffer in every cycle, except cycles in which any `NO_ACQUIRE` statement is active (which can be either a NO_ACQUIRE action or a Global NO_ACQUIRE).

When `visionMode` is set to DYNP, it is possible that one or two extra probe samples might be added to the end of each run, even if that time segment was not supposed to be captured due to conditional acquisition. When `visionMode` is set to `FV`, the extra number of captured samples at the end might be up to a full frame (32 samples).

If the SDL program does not specify any `ACQUIRE` or `NO_ACQUIRE` in any SDL instance, the probed data is acquired every cycle.

Some operation modes, such as Vector Debug and infiniTrace, do not support conditional acquisition.

If the design was compiled for FullVision mode, probed data is acquired in whole frames, where a *frame* is 32 consecutive samples or less. A whole frame is always acquired if the SDL program requests to acquire any sample within that frame. On the waveform display this might show up as extra samples that were originally requested to be filtered out.

Related Sections:

- Using ACQUIRE Action
- Using NO_ACQUIRE Action
- Defining Global ACQUIRE
- Defining Global NO_ACQUIRE

Conditional Acquisition for Streaming Probes

SDL provides conditional acquisition actions to probe with the SDL actions `ACQUIRE` and `NO_ACQUIRE`. You can also start or stop the acquisition of streaming probes independently of the DCC conditional acquisition using SDL.

To unlink SDL's acquisition of streaming probes from acquisition of DCCs, do the following:

In the SDL global definition section (above any `State` or `Instance` statements), add the following statements to control the streaming probes acquisition:

```
    if(<expression1>) acquire stream stop;
    if(<expression2>) acquire stream start;
    if(<expression3>) acquire stream;
```

<*expression1*> <*expression2*> <*expression3*> must be legal SDL expressions.

These statements must be specified only in the SDL global definition section. An error message will be displayed if these statements are used inside an instance or inside a state.

By default, streaming probes start acquiring data from the first cycle of each run. If SDL is enabled, streaming probes stop data acquisition when <*expression1*> becomes true. From that point, streaming probes acquisition remains stopped. The acquisition resumes the data capture if the run is restarted (with a new `run` command), or if <*expression2*> becomes true. It will also capture in specific cycles in which <*expression3*> is true.

The SDL program might contain any number of the above statements.

Note: The `acquire stream` or `acquire stream start` actions have priority over `acquire stream stop` action. For example, if <*expression1*> and <*expression3*> are executed in the same cycle, then acquisition will go into a stopped state, however, a probe sample will still be taken in that cycle.

Example:

Suppose that the `signal cnt[15:0]` represents the output of a binary up counter. The following is an example of an SDL program using that signal:

```
    if(cnt[15:0] == 100) acquire stream stop;
    if(cnt[15:0] == 200) acquire stream start;
    if(cnt[15:0] == 300) acquire stream stop;
    if(cnt[15:0] >= 1000 && cnt[15:0] <= 1100) acquire stream;
    if(cnt[15:0] >= 1300 && cnt[15:0] <= 1400) acquire stream;
```

In this example, assuming `cnt[15:0]` starts with value `0` at the beginning of the run, streaming probes will acquire data while the value of `cnt[15:0]` is within any of the following intervals:

```
    0 <= cnt[15:0] < 100
    200 <= cnt[15:0] < 300
    1000 <= cnt[15:0] <= 1100
    1300 <= cnt[15:0] <= 1400
```

You can also select whether to use the `acquire stream, acquire stream start,` and `acquire stream stop` actions, or to ignore them, and instead link the conditional acquisition of streaming probes with that of the DCC probes. To do this, use the following XEL command:

```
    xeset streamAcquireControl [linked | unlinked | auto]
```

For more information on the `xeset streamAcquireControl` command and its values, refer to the documentation of this command in the *Run-Time Commands* chapter of the *WXE Command Reference Manual*.

- `linked`: Streaming probes use the same conditional acquisition as DCC probes — that is, SDL actions `ACQUIRE` and `NO_ACQUIRE`, and the DRTL system tasks: `$qel`("`acquire`"), `$qel`("`acquire start`"), and `$qel`("`acquire stop`") are used. If the SDL program contains `acquire stream start` or `acquire stream stop` actions, then they are ignored.
- `unlinked`: Conditional acquisition for streaming probes are unlinked from DCC probing and only the following SDL actions are used to control the acquisition: `acquire stream`, `acquire stream start` and `acquire stream stop`.
- `auto`: The system automatically selects between `linked` and `unlinked` values depending on the SDL program. If the SDL program contains any of the actions: `acquire stream`, `acquire stream start` or `acquire stream stop`, then conditional acquisition behaves according to the `unlinked` value, otherwise, conditional acquisition behaves for the `linked` value. `Auto` is the default value.

If the `xeset streamAcquireControl` command is executed without any value, the command returns the current value, that is, `linked`, `unlinked`, or `auto`.

Example: Sample Script: Using linked, unlinked, and auto

Contents of run-time script:

```
    run -swap
    xeset streamAcquireControl unlinked
    sdl -setfile sdl.tdf
    sdl -enable
    sdl -verify
```

```
database -stream -open open_st
database -open open_dcc
…
getTime
xeset streamAcquireControl linked
run 50
xeset streamAcquireControl auto
run 100
```

In the above example, initially the streaming probes are unlinked from DCC. After the `xeset streamAcquireControl linked` command, the streaming probes are linked to DCC. When the `auto` option is used, the conditional acquisition is determined on the basis of the statements used in the SDL program — that is, if the SDL program contains any of the actions: `acquire stream` or `acquire stream start` or `acquire stream stop`, then conditional acquisition behaves according to the `unlinked` value, otherwise, conditional acquisition behaves for the `linked` value.

Specifying SDL Statements

TRIGGER, DISPLAY, EXEC, TX, ACQUIRE, and NO_ACQUIRE can be specified in one of two methods. The two methods can be used together in the same SDL program.

The first method, called **Global TRIGGER** (or **Global EXEC** or **Global ACQUIRE/Global NO_ACQUIRE**) uses a basic boolean expression based on design signal values (or assertions) evaluated repeatedly every cycle. Such global expressions can appear either at the very top of the SDL program (Global Definition Section), or at the top of each SDL instance (Instance Definition Section).

The second method uses a **state machine** that can track complex patterns along the time axis. Each SDL Instance can contain at most one state machine. Trigger or other actions are embedded inside the definition of the state machine. The state machine has two internal counters that can be used to count long sequences (up to 2^{40}-1). Other statements within the state machine can be used to control the state machine, for example, decide when to switch from one state to another, or when to load or decrement one of the two counters. Any Action can be executed based on boolean conditions made from design signal values or the outputs of the counters.

Related Sections:

- Specifying Global Definitions in SDL Programs
- Using SDL Actions

Using SDL General Purpose Counters

Each state machine has two 40-bit general purpose counters (named `COUNTER1` and `COUNTER2`), which can be set to auto-decrement every clock cycle, or can be incremented or decremented only in selected cycles. Each counter can be independently controlled and its output can be tested to compare the counter's value against any positive integer value between zero and (2^{40} -1).

The scope of each of these counters is the entire state machine. While they are shared among different states in a state machine, they are not shared between different SDL instances.

Each counter can be loaded with a new value, which can be different from one LOAD command to another.

Note: Values for SDL counters can also be loaded dynamically from the command line by using the following command:

```
sdl [-ldcnt1 | -ldcnt2] [-instance <instance>] <value>
```

For more information, refer to the `sdl` command description and syntax in the *Run-Time Commands* chapter of the *WXE Command Reference Manual*.

Each of the counters decrements or increments by 1 each time when DECREMENT or INCREMENT is executed. In addition, the counter can also be put in auto-decrement mode through the use of the START statement. When in auto-decrement mode, the counter will decrement by 1 each time when its Count Enable is high. The Count Enable is a boolean expression made of design signals.

Although the `DECREMENT` statement is honored also while in auto-decrement mode, the counter decrements at most by 1 each cycle.

Since the counters are state machine resources, they can only be used inside STATE Statements. This is true for both counter actions, such as DECREMENT, and Counter Conditions.

Related Sections:

- Using SDL Actions
- Using DECREMENT Action

- Using INCREMENT Action
- Defining STATE Statement
- Using Counter Conditions in SDL Expressions

Using the COUNT() Operator in SDL

SDL also supports an unlimited number of counters, each private to the expression that contains it. The output of these counters becomes true after counting a specified number of times in which a given expression is true. For example, the following will trigger after counting 35 times in which `net1` has the value `1`:

```
if(count(net1,35)) trigger;
```

The following will trigger after counting 50 clock cycles:

```
if(count(1,50)) trigger;
```

There is no limit to the number of these counters in an SDL program. The conditions under which these counters are reset are configurable.

Related Topic

Using Operators in SDL Programs

Statement Usage and Execution Semantics of an SDL State Machine

A basic state machine construct is the STATE Statement, which defines one state of the state machine. Within each state, input conditions (based on values of design signals and/or the internal SDL counters) can be tested by one or more IF - ELSE statement, and various actions can be taken, based on the results of these tests.

In each evaluation cycle, first all the conditions are evaluated, and then all the actions are executed concurrently. This means, for example, that the result of an action to decrement a counter in cycle N can be tested by a condition only in cycle N+1.

Some types of actions cannot be executed together. For example, two GOTO actions that both request to transfer control to two different states are conflicting, and cannot be executed together. Conflicts happen when any of the following types of actions are requested to be executed in the same cycle (within the same SDL instance):

- Two GOTO actions with different destinations.
- Any two different actions for the same counter, except the following two combinations. These two combinations are allowed and will be executed concurrently if encountered in the same cycle.
    - LOAD and START
    - LOAD and STOP

In case of a conflict between two actions, SDL picks the one that appears first in the SDL program (when scanning the text from top to bottom). The SDL compiler prints out warning messages about potential conflicts before execution begins.

If enabled, the SDL state machine continues to execute as long as the design in the emulator continues to run (or until the requested amount of trace buffer is filled, when running in LA mode).

Related Sections:

- Defining STATE Statement
- Defining IF - ELSE statement

- Using GOTO Action

Using SDL Clocking

When running in ICE mode, each SDL execution cycle is typically a single Fast Clock (FCLK) cycle. Design signals can toggle faster or slower than FCLK depending on the sampling ratio. This means that when running with sampling ratio of 1X or 0.5X, SDL is not guaranteed to detect all signal values. For more details, refer to .

When running in SA mode, SDL evaluation cycles run every FCLK cycle in which any design signal makes a transition. While this guarantees that SDL will detect all signal transitions, there is no guaranteed correlation between SDL evaluation cycles and simulation time. Also, SDL might or might not detect transitional values on combinatorial nets, depending on the way the design was compiled. Additional limitations exist when running in SA mode with 1X clocking.

See Limitations of SDL when Running with Sampling Ratio for more information.

When a design is compiled in SA mode using IXCOM with an uncontrolled clock generator (free running), you can select at run time the clocks (controlled or uncontrolled) that need to be stopped by an event, such as SDL trigger or simulator breakpoint.

Limitations of SDL when Running with Sampling Ratio

If the design was compiled to run with sampling ratio of 1x concerning `clockOption` technology CAKE), it is possible that a signal will make more than one transition within the same sampling cycle (one FCLK cycle).

If requested to test whether a signal is 0 or 1, SDL will pick the value at the end of the FCLK cycle. This means that SDL might miss every second value on the fastest changing signal.

If requested to test for a special value such as a positive edge, SDL will scan the entire FCLK cycle to determine whether the condition is true. For example, to qualify for a positive edge, it is enough that the signal will make a low to high transition anywhere in the last FCLK cycle. To qualify for a “stable low” value, the signal must be low throughout the entire FCLK cycle. For general information on the use of special values in expressions, refer to Using Special Non-Numeric Constants in SDL Expressions.

If the design was compiled to run with sampling ratio of 0.5x, special values such as edge detection should not be used in the SDL program because SDL will not handle them correctly.

Testing for regular values (0 or 1) will sample signals only once at the end of each sampling period.

Related Topic:

Running Designs with Uncontrolled Clocks Using xeDebug in SA Mode

Using File Inclusion in SDL

A single SDL program can span several different files through the use of File Inclusion.

SDL programs can include text from other files similar to the way this is done in Verilog through the use of an `include` directive.

The syntax of an include directive is as follows:

```
`include "<file-name>"
```

The line cannot contain any `/*…*/` style comments or substitution of TCL variables (with the syntax `$<var>` or `${<var>}`).

The first character is a backward single quote mark, while the filename must be enclosed in straight double quotes. The `include` directive is not a free format: It must be specified on a separate line, and the `include` keyword must be specified in lower case. On the other hand, the include directive can be inserted at any point in the SDL program.

Included files can include other files to any nesting level.

The string *`<file-name>`* can contain either an absolute directory path (starts with the character '/') or it can contain a relative directory path, or just the filename with no path at all.

If *`<file-name>`* does not contain an absolute directory path, then it is searched in the current directory. If not found in the current directory, then it is searched in a list of directories specified by the run time parameter `sdlIncludePath`. The list must be space separated. For example:

```
xeset sdlIncludePath {/home/george/myfiles/sdl /home/ben/projectX}
```

To make this search path visible to all your sessions with debug tools, whether running in script or GUI mode, you can place such definition inside the `.xerc` or `.rtxerc` files. The `.xerc` file is an XEL script that is always executed (if it exists) during the software initialization phase, while `.rtxerc` is executed on the initial call to the `configPM` command. It can be placed either inside the current directory from where you start the debug tool, or under the home directory.

Using Aliases in SDL

The Alias facility enables to assign any boolean expression to a single name, and then use that name in different places in the SDL program instead of writing the whole expression.

An alias defines a new name which can be used later instead of typing the whole expression. The expression used for defining an alias can itself contain names of other aliases already defined.

Syntax:

```
<name> = <Expression>;
```

`<name>` can contain only alphanumeric characters or an underscore (_), and cannot begin with a digit. Names of aliases are case sensitive.

An alias appearing in a Global Definition Section can be used anywhere below that definition, including all the SDL instances. An alias definition in an Instance Definition Section is private to that Instance.

The same alias name can be used to define a different alias value in different instances. However, if an alias was defined in the Global Definition Section, then it cannot be redefined in any Instance Definition Section.

An alias cannot be indexed and cannot be used as part of a hierarchical path. For example, if YYY is defined as an alias like `YYY = XXX`, then the following usage is not supported:

- `YYY[0]`cannot be used as `XXX[0]`
- `YYY.ABC` cannot be used as `XXX.ABC`

Related Sections:

Specifying Global Definitions in SDL Programs

Using Values of Tcl Variables Inside SDL

Values of global Tcl variables can be used directly inside the SDL program using either the *$<tcl-variable-name>* or *${<tcl-variable-name>}* syntax. The value of the variable is substituted once at the beginning of the run.

Any value can be substituted, whether numeric or a string. The second form with the `{}` is useful when the variable value has to be concatenated with another string without any intervening delimiter.

For example, suppose that a Tcl variable is assigned the value `top.alu`, which represents a hierarchy in the design, by executing the following Tcl statement (either from the XE prompt or from a script):

```
set path top.alu
```

Suppose that the SDL program contains the following construct:

```
state s1 {
      if(${path}.y[7:0] < ${path}.x[7:0]) trigger;
      ...
```

The above construct is equivalent to:

```
state s1 {
      if(top.alu.y[7:0] < top.alu.x[7:0]) trigger;
      ...
```

Note that `path` must be declared in the Tcl script at global call level. This means that if the variable `path` is assigned inside the definition of a Tcl `proc,` then it must be declared explicitly as global as shown in the following example:

```
proc foo {args} {
     global path
     set path top.alu
     ...
```

The Tcl variables referenced with the above syntax inside an SDL program are substituted with their values only when the SDL program is loaded into the emulator (for example, at the beginning of each run). This does not apply to Tcl variables that appear inside EXEC actions. The construct $var or ${var} inside EXEC actions are evaluated each time when the EXEC action is executed. See more details in the description of the EXEC action.

The substituted contents of TCL variables should not contain quoted strings, comments or other substitutions. This means that the following characters or sequences are not allowed in the substituted contents:

```
//
/*
`
$
"
```

Related Topic

Using EXEC Action

Using Auto-Probing Facility in SDL

The auto-probing facility enables to request from XEL to automatically probe every net or assertion that is actively used in the SDL program. Nets or assertions that appear in alias definitions that are not used actively in the SDL program are excluded.

Choosing Fast or Accurate Behavior

By default, the SDL instrumentation is sampling the design signal values with a delay of up to 2 fast clock (FCLK) cycles. In STB mode, the delay is almost always two FCLK cycles, except rare cases (for example, on the first cycle of the run), where the delay can be less than two. Where possible, the system tries to hide the effects of this delay from you. For example, the delay does not affect requests to filter out probe samples in specific cycles through Conditional Acquisition (ACQUIRE or NO_ACQUIRE), the special instrumentation re-synchronizes the timing of the probe samples with SDL. Also, the cycle number of the trigger point as reported on the GUI or by XEL, or in an SDL trace dump is not showing the delay.

However, this delay becomes visible when the emulator stops because of a trigger, or when emulation clock are suspended during execution of XEL commands through the `EXEC` statement. If you request from XEL at that point to view any SDL status information (such as state name or SDL counter values) you will see that the status information matches the cycle in which the trigger happened according to the SDL program, but is lagging after the design signals by up to 2 FCLK cycles.

When running under IXCOM (in co-simulation mode), the 2 FCLK cycles might fall within the same time slot in which the condition that caused the `TRIGGER` or `EXEC` happened, or it might fall in the next time slot.

Another possible anomaly that can be observed is when the run is stopped because of an event outside of SDL. For example, reaching the specified amount of time to simulate, or an assertion that stops the run. In that case, an `EXEC` or `TRIGGER` meant to happen on the same cycle (or time slot in IXCOM) in which the run stopped, might not be processed.

It is possible to override this behavior and eliminate the delay (switching to *accurate triggering*), but this comes with a speed penalty. In STB mode in such case, the emulator speed is only 1/3 of the regular speed. Switching between stopping the emulator with delay and without delay is performed by the XEL command:

```
xeset stopDelay 0|1
```

This command can only be executed while the emulator clocks are stopped (or suspended while executing an `EXEC` statement). A value of 0 means no delay between the trigger and the stop point (and running at 1/3 speed).

In SA mode, using `stopDelay 0` will typically have less impact on overall run speed, but this depends on the percentage of time used up by the software simulator. Also, it only guarantees that any `EXEC` or `TRIGGER` will cause the design to stop within the same time slot in which the condition that caused it happened, but it does not guarantee that the run will stop within the same FCLK cycle.

In some cases, one can avoid the delay without incurring the 1/3 speed penalty by switching from fast to accurate triggering only during strategic time segments in which triggering is expected. This is practically done by executing the `stopDelay` command itself from within SDL using the `EXEC` statements.

Related Sections:

Using SDL Actions

Delayed Execution of TRIGGER and EXEC under IXCOM

Under IXCOM, on top of the above delay of 2 FCLK cycles, `TRIGGER` and `EXEC` actions are executed only at the end of the simulation time slot after all the behavioral processes finished evaluation. This means that even if `stopDelay` was specified as `0`, the conditions that caused the `TRIGGER` or `EXEC` actions, might have already disappeared by the time the `TRIGGER` or `EXEC` are actually executed. This is true also for the SDL state itself. SDL will continue to process conditions, and possibly change states before the TRIGGER or EXEC request is finally executed.

Related Sections:

- Using TRIGGER Action
- Using EXEC Action
- Using SDL Actions

SDL Instrumentation Effect on Critical Path in SA Mode

If the critical path contains the `xc_top.sdlStop` signal, it indicates that the critical path goes through both the SDL instrumentation and the co-simulation instrumentation. If this behavior causes excessive increase in step count, you can break this critical path by inserting a flip-flop between the SDL instrumentation and the co-simulation instrumentation. To do this, use the following command:

```
precompileOption -add sdlTriggerPathBreak
```

The following command removes the flip-flop in case you want to override a previous request to add the flip-flop:

```
precompileOption -rm sdlTriggerPathBreak
```

Insertion of the flip-flop causes an addition of one FCLK delay in the path. This may cause in some cases an additional delay in SDL actions: `TRIGGER` and `EXEC`, or DRTL system tasks: `$qel("trigger")` and `$qel`("<*CMD*>"), where <*CMD*> is a TCL or XEL command.

The critical path is reported by default in the `tmp/et3compile.msg` file.

Related Sections

For information on how to control critical path reporting, refer to the description of the `compile` and `getCriticalPathDesignNetNames` commands in the *Compile-Time Commands in ICE Mode* chapter of the *WXE Command Reference Manual*.

Debugging with the SDL Trace Dump Facility

SDL comes with a debugging facility called *SDL Trace Dump,* which enables to troubleshoot the behavior of SDL programs.

During the run, key information on the state machine component of each SDL instance is recorded into dedicated memory, which is arranged as a circular buffer. The buffer's depth is 16k samples, and newer data samples overwrite the oldest. The data is recorded only in *interesting clock cycles*. By default, an interesting clock cycle is one in which any SDL action (except `ACQUIRE` or `NO_ACQUIRE`) is executed. You can change the default through XEL and designate what cycles to record based on the type of actions that SDL executes. For example, you can request to record information only in cycles in which SDL executes a GOTO action, or on the other extreme, record continuously every cycle.

The recorded data includes information such as the emulation time, value of the two SDL counters, SDL state name, and what actions were executed by the state machine in that cycle. The simulation time or cycle number that appears in the recorded data for TRIGGER and EXEC actions show the time when the expression that triggered them became true. The actions themselves are typically executed with a delay of 2 FCLK cycles.

The data is always collected during a run, but you can decide through XEL whether or not to dump it to a file. When the data is printed in textual form, the information is most detailed and can include (based on XEL switches) even the filename and line number in the SDL file for each action executed.

There is a limitation regarding SDL trace dump in text mode: if a label (for a `TRIGGER`, `DISPLAY`, or `EXEC`) is disabled or enabled in the middle of the run (for example, with `sdl -disable -label -now` ...), the resulting SDL trace might not correctly show the actions associated with that label.

A waveform format is also available, but it shows only SDL state name and values for the two counters.

Recording of SDL activity into the SDL trace memory is stopped when a trigger happens. You can request that the tracing stop immediately upon trigger, or only when regular probing is stopped (as specified by the post trigger amount), depending on settings in the `sdl -traceOn` XEL command.

The trace is dumped to a file by executing the `sdl -traceDump` XEL command. When the command is executed, it dumps to the specified file SDL activity traced in the most recent run. If no trigger happened in the most recent run, the trace dump includes whatever data was captured so far.

By default, at the beginning of each new run, the SDL trace memory is reset, which means that any traced data from the previous run is discarded. However, you can request through XEL to reset the SDL trace memory not every run, but only when dumping the data to the disk, or even request to never reset the SDL trace memory.

Note that if a trace dump is requested while the emulator is running, SDL tracing is suspended while the data is dumped into the file.

When running from GUI, you can request to automatically create an SDL trace dump every run by selecting the `Upload SDL Activity` in the *Control* tab or *Infinitrace* tab of the xeDebug GUI. The waveform will show SDL activity only for the last run, although you can override this behavior in the `Advanced SDL Control` window (which you launch from the *SDL* tab). This window enables you to control other advanced aspects of SDL and SDL trace dump in specific.

Related Sections:

Using SDL Actions

Saving and Restoring the Internal State of SDL

The internal state of the SDL program can be saved into a file and restored later, either in the same session, or in a new session. The saved file includes information for each SDL instance, such as the name of the SDL instance saved, the name of the current SDL state, current values of SDL counters, and whether they were auto-decrementing or not.

Saving (or restoring) SDL state is typically done together with saving (or restoring) the emulated design state, to enable to return to a previously saved checkpoint, including the SDL state.

The save/restore mechanism only requires matching of state and instance names, therefore it can be used to restore the state of one SDL program into another SDL program. The only requirement is that the two SDL programs agree with names of instances and states.

Detecting Events without Triggering

If you only want to detect whether specific events occurred during a run, and do not want to stop the emulator (or data tracing) each time a specified event happens, use the `-onlyReport` switch of the `sdl` command. Here, the term event refers to a specified expression becoming true. When the `-onlyReport` switch is turned on, trigger conditions defined in the Global Definition Section or Instance Definition Section are disabled from stopping design or capture clocks, but they are still reported by the `sdl -getTriggers` and `sdl -report` commands. Triggers coming from the state machine portion of SDL are not disabled, and will cause the design and/or capture clocks to stop.

Related Sections:

- For more details, refer to the description of the `sdl` command in the *Run-Time Commands* chapter of the *WXE Command Reference Manual*.
- Specifying Global Definitions in SDL Programs

Interaction between SDL and XEL

SDL and XEL can interact in several ways.

First, in emulation modes that support this feature, SDL can issue any Tcl or XEL commands during emulation using the `EXEC` facility (See descriptions of EXEC and Global EXEC). From a syntax point of view, any string that can be executed by the Tcl interpreter, including calls to Tcl procedures is accepted. There are some XEL commands that are not allowed in an `EXEC`, irrespective of whether these commands are called directly from the `EXEC` or indirectly through a Tcl procedure. In run modes that enable stopping the clocks, the design clocks are suspended during the execution of the XEL commands to guarantee time coherency. When running in LA or Dynamic Target mode, the clocks are free running, time coherency is not guaranteed, which means that `EXEC` requests are executed with significant delay.

Second, XEL can control SDL, or retrieve from it information through the `sdl` XEL command.

Here is a quick summary of the most important facilities provided by the `sdl` command. You will find more details as well as command syntax in the *Run-Time Commands* chapter of the *WXE Command Reference Manual*.

- Specify the name of the SDL file (the file that contains the SDL program).
- Verify the contents (for syntax errors) of an existing SDL file without actually requesting to run it.
- Enable or disable the SDL feature as a whole, as well as enable or disable individual SDL instances, or individual `TRIGGER`, `DISPLAY`, `TX`, or `EXEC` statements.
- Retrieve current status information such as:
  - Current state name and counter values for each of the SDL instances.
  - Did a trigger happen, and if it did, in what cycle/time, and which statement (or statements) caused it.
  - Check whether specific trigger(s) happened by providing the identifying label.
- Generate a log of recent SDL activity, called SDL Trace Dump (for example, state transitions, actions, counter values and so on) in either text or waveform format, and specify what information should be logged.
- Turn on/off automatic probing of all design signals used by SDL.

- Save and restore SDL internal state to/from a file.
- Control destination of `DISPLAY` and `TX` data.

Several other XEL commands, variables, and run-time parameters also affect the SDL execution as follows:

- Symbols (representing vectors), defined through the `symbol` XEL command, can be used in SDL expressions.
- The value of any Tcl variable declared at global call level (that is, by the `set` Tcl command outside any proc definition, or by the `global` command), can be used inside an SDL program with the `$var` or `${var}` syntax, where `var` is the name of the Tcl variable. Any numeric value or a character string can be passed that way from the Tcl script to the SDL program.
- Default values for post trigger samples or trigger position can be set by the `xeset` XEL command, and will be used by the SDL instrumentation if not set inside the SDL program.

Each time before starting SDL, the SDL program is recompiled if any of the following occurs:

- The main SDL file or any other file that it includes has changed.
- The value of any Symbol defined by XEL and used in the SDL program has changed.
- The value of any Tcl variable used by the SDL program has changed.
- Any one of the XEL commands that can affect the SDL run-time characteristics have been executed.

Typically, SDL is started by executing the `run` command. However, SDL will not be recompiled (regardless of any changes) if the run is started using the `run -continue` command.

These rules also apply to the independent trigger expression defined by the `sdl -expression` command (although the independent trigger expression is not considered part of the SDL program).

Related Sections:

- Using EXEC Action
- Using Global EXEC

DISPLAY and TX Facilities

The `DISPLAY` and `TX` facilities enable SDL to produce output during run time. Both facilities share the same hardware resources, which are independent of the hardware resources that are used for waveform capture. The data can be either streamed continuously while the run is ongoing, or it can be buffered and uploaded on demand similar to the way regular waveforms are handled. Data streaming is done in parallel to the run, and therefore can be used even with dynamic targets, provided that the average rate at which the data is generated is low enough to be processed without slowing down emulation.

Related Sections:

- Using DISPLAY Action
- Using TX Statement

Recording Transactions with the TX facility

The `TX` statement writes transaction type output into an SST2 database, which can be viewed later on SimVision's waveform browser or Transaction Stripe Chart Viewer.

The `TX` statement can define the beginning and end of a transaction. It can also define signal values that are associated with the transaction. The recording is done through Cadence's SDI interface.

Partial and Illegal Transactions

Transaction recording is not extended across consecutive `run` commands (unless the `-continue` option is used). This means that a transaction that has its beginning in one run, and its ending in a subsequent run will appear in the SST2 database as two partial transactions instead of one complete transaction (unless the second run was issued with the `-continue` option).

The existence of a partial transaction might indicate a protocol error in the design or an error in the SDL program, However, since it might also indicate a benign situation as explained, no warning is issued and each partial transaction is recorded in the SST2 database with a special attribute marking it as such.

Some transaction recording requests can be identified as definitely illegal. For example, two consecutive requests to record the beginning of a transaction using same transaction ID. This indicates either a protocol error in the design or an error in the SDL program. Illegal transactions are considered an error and will stop the run unless instructed otherwise from XEL.

Related Topic

Using TX Statement

Printing Messages from SDL with the DISPLAY Facility

The DISPLAY facility enables to print out messages from an SDL program based on specified conditions. The message might optionally contain signal values and simulation time at the exact emulation clock cycle from which the `DISPLAY` statement was called. The message is formatted using a syntax similar to the Verilog `$display` system task.

You can also use EXEC to print message, however, there are disadvantages to using EXEC and these might even render it useless for some situations. EXEC enables to execute any Tcl command; so, messages can be printed using the standard `puts` Tcl command. For example, printing the value of the signal `ABC`, and the current simulation time using `EXEC` might look like this:

```
if(some_condition)
    EXEC ”puts \”value of ABC at time [getTime]is [value ABC]\””;
```

Using a `DISPLAY` statement to print the same information might look like this:

```
if(some_condition)
    DISPLAY (”value of ABC at time %t is %d”, ABC);
```

Comparing DISPLAY and EXEC Methods

Use of `DISPLAY` statements has the following advantages over `EXEC`:

- Simplicity: The syntax of `DISPLAY` statement is simpler. Note that when using `EXEC`, you need to escape the inner set of double quotes with a backslash.
- Speed: Each `EXEC` statement causes the design clocks to pause while the Tcl interpreter executes the `puts` statement. This operation typically takes several hundred milliseconds, and might cause severe slow down when the number of `EXEC` requests is large.

  `DISPLAY` messages are buffered during the run in a special *display buffer* memory, and therefore do not affect execution speed as long as the workstation can read the contents of the display buffer fast enough. Reading the display buffer is done in large blocks while the design is running, so in most cases, the `DISPLAY` statements have no effect on the execution speed once the run starts.

- Use with Dynamic Target: If the design clocks are not allowed to stop, as might happen when running a design in Logic Analyzer, Dynamic Target mode, or with uncontrolled clocks under IXCOM, `EXEC` requests can easily get overwritten and lost because they are not buffered.

  Display requests are buffered. Therefore, `DISPLAY` can be used in these operation modes without losing any messages. The only requirement for not losing `DISPLAY` messages is that the rate at which the workstation reads the trace buffer contents must be faster than the average rate at which the `DISPLAY` messages are generated by SDL. And if this is not possible, you have an option to request a large display buffer depth at compile time.

- Time coherency: Values of arguments of the `DISPLAY` statements are captured in the exact cycle that invoked the `DISPLAY` statement. Therefore, the printed values will match the condition that invoked them.

  The `EXEC` statements are typically executed with some delay after the condition that invoked them. For example, in STB mode, the delay is `2` cycles or less (See Choosing Fast or Accurate Behavior). In LA mode, the delay can be thousands of cycles.

There is one advantage for printing messages with `EXEC` over `DISPLAY`:

- SDL program load time: Arguments of the `DISPLAY` statements must be routed to the display buffer when loading the SDL program into the emulator. This might add to the time it takes to start a `run` command with a new SDL program. Therefore, if your SDL program prints the message only a couple of times throughout the debug session, it might still be faster to use `EXEC` instead of `DISPLAY`.

Related Sections:

- Using DISPLAY Action
- Using EXEC Action

Forming SDL Conditions in IXCOM Mode

As an example, consider the following SDL statement:

```
if(dut.net1) display("net1=1");
```

When running in STB mode, the number of times that the `net1=1` string is printed out remains the same from one session to another (excluding asynchronous events coming from user interaction or external targets in non-blocking runs).

On the other hand, in IXCOM mode, the relationship between design-clock transitions and FCLK is not guaranteed, therefore when running under IXCOM mode, the number of SDL-evaluation cycles between design-state transitions is not guaranteed to be consistent from one release to another, or between different debug sessions.

The outcome is that the number of times that this string is printed, might vary between releases or even between debug sessions.

To get consistent SDL behavior in cases like regression tests, it is recommended to use edge detection. This can be done in one of the following ways depending on the objective:

- If the objective is only to find when the `dut.net1` signal changes state from `0` to `1`, use the following:

  ```
  if(dut.net1 == 'bP) display("net1=1");
  ```

  The expression `(dut.net1 == 'bP)` becomes true only when `dut.net1` make a positive transition.

- If the objective is to report every clock cycle in which `dut.net1` is high, select a design clock net and use it to qualify the condition. For example, suppose `dut.clk` is a design clock:

  ```
  if(dut.net1 && (dut.clk == 'bP)) display("net1=1");
  ```

  Here, one line is printed each time when `dut.clk` makes a positive transition, while `dut.net1` is high.

Expressions made from UPF, CPF, or assertion conditions should also use the above-mentioned guidelines.

For example, to trigger on an assertion entering the `finished` state, use the following:

```
if((dut.assert1 == "finished") == 'bP) trigger;
```

Emulation Resources Consumed by SDL

The typical overhead for each SDL instance requested during compile time is approximately `1MB` RAM and `3k` gates, irrespective of whether that instance is actually used during the run time or not.

The actual overhead can be larger when increasing the default values for some of the compiler options that affect the size of the display buffer or the SDL trace buffer.

The total size of the display buffer (in bytes) is determined by the following formula:

```
D * (G/8 + W/8 + 2*N)
```

Here,

- `D` is the depth as specified by compiler option `sdlDisplayDepth`.
- `G` is the value specified by compiler option `sdlGlobalDisplay`.
- `W` is the value specified by compiler option `sdlDisplayWidth`.
- `N` is the value specified by compiler option `sdlInstances`.

For example, the display buffer for a display depth of `4k`, `64` global `DISPLAY` statements, a display width of `1024`, and `3` SDL instances, will consume `640kB` of emulated memory:

```
4096 * (64/8 + 1024/8 + 2*3)  = 581,632 (equivalent to 568 kB)
```

The total size of the SDL trace dump buffer (in bytes) for all SDL instances is determined by the following formula:

```
T * N * 20
```

Here,

- `T` is the value specified by compiler option `sdlTraceDepth`.
- `N` is the value specified by compiler option `sdlInstances`.

For detailed information about the compiler options mentioned above, refer to the description of `compilerOption` command in the *User Data Commands* chapter of the *WXE Command Reference Manual*.

Summary of Basic SDL features

- Up to 64 SDL instances, each can be enabled or disabled individually.
- Each SDL instance can have a state machine with up to 64 states.
- If -else statements for testing conditions in any nested configuration.
- Each SDL instance has two auto decrementing general purpose 40 bits counters, each controlled independently.
- SDL may have any number of private expression counters.
- Ability to enable and suppress probed data acquisition in probing modes that support conditional acquisition.
- Ability to stop the run in some operation modes.
- Ability to execute Tcl or XEL commands during emulation in STB, LA or Dynamic Target modes (`EXEC` facility).
- Ability to generate an activity log (SDL trace dump) in either text or waveform format per SDL instance.
- Ability to generate formatted output in the form of plain text or transactions written into an SST2 database (`DISPLAY` and `TX` facilities).
- State-less `TRIGGER`, `DISPLAY`, `TX`, and `EXEC` statements.
- Ability to request auto probing of all nets and assertions referenced in the SDL program.
- Ability to save and restore internal SDL state.
- Ability to disable individual `TRIGGER`, `DISPLAY`, `TX`, and `EXEC` statements using their label.

Related Topic

Overview of State Description Language (SDL)

Preparing, Compiling and Executing SDL Programs

The following sections describe the way SDL programs are prepared, compiled, and executed.

Setup during Design Compilation

The maximal number of instances that can be used by an SDL program can be specified during design compilation by the compiler directive:

```
compilerOption -add {sdlInstances <num-instances>}
```

`<num-instances>` must be an integer number between 1 and 64. This number sets the upper limit for the number of SDL instances that can be used at run time. If unspecified, a value of 1 is assumed.

The maximal number of samples that an sdl trace dump can have can be specified during design compilation by the compiler directive:

```
compilerOption -add {sdlTraceDepth <depth>}
```

`<depth>` must be a positive integer number. This number can use a postfix of `k` or `M` (for example, 4k means 4x1024). If unspecified, a default value of 16k is used. Values above 16M or below 1k are converted to the maximal value of 16M or the minimal value of 1k.

Other than that, all other aspects of SDL are programmable at run time.

Preparing the SDL Program

During run time, SDL is prepared by a text editor and has to be stored in a file (or a set of files) before it can be executed. The SDL text is known as an *SDL Program*. The editor can be any text editor of your choice, but the recommended method is to use the *SDL Editor* group box under the *SDL* tab in the main xeDebug GUI window.

Using the *SDL* tab has the following advantages:

- It has a dedicated set of predefined buttons and menus for each of the SDL keywords.

- It highlights SDL keywords in a different color.
- It automatically verifies the SDL program for any errors and save it to a file before each run.
- It enables you to iterate through any errors found in the SDL program, while highlighting the location of the offending text on the screen.
- It enables you to drag and drop signal names from other browsers (for example, the Design Browser or the Schematic Browser).

When running in the xeDebug GUI, specify the filename in the *Edit File* field in the *State Editor* group box and click the ✓ button adjacent to it.

Before starting a run (either through the xeDebug GUI or by executing the `run` XEL command), the SDL program is compiled into a special binary image downloaded into the Palladium hardware. The SDL compiler, which performs that process, can produce error or warning messages. Any error conditions reported by the SDL compiler must be fixed before the run request can be executed.

SDL Enabling, Activation, and Execution

Once the file containing the SDL program is defined with the `sdl -setFile` command, and SDL (or any one of its components) is enabled by the `sdl -enable` command, SDL is ready for use (considered *enabled*). Assuming that the SDL program does not have any errors, SDL is compiled into a binary image and loaded into the emulator when any one of the following XEL commands is executed:

- `sdl -load`
- `run`
- `infiniTrace -goto trigger`

`sdl -load` only loads the SDL image into the emulator, which enables you to query about basic properties of the SDL program (for example, what are the names of instances being used). This is done whether SDL is enabled or not.

`run` and `infiniTrace -goto trigger` load the SDL image then start both SDL and design clocks and/or capture clocks (depending on operation mode).

Once SDL is loaded into the emulator, it is considered ***active***, whether it is enabled or not. Note that a successful `sdl -load` command will always load the SDL program into the emulator and make it active, while a successful `run` or `infiniTrace` command will load the SDL program and make it active only if it is enabled.

An active SDL program can be enabled during a run, even if the run started with the SDL program disabled, therefore if you want to start the run with an SDL program disabled, and then enable it while the run is in progress, you need to execute the `sdl -load` command before executing the `run` command. In the current release, this will work only if you do not make use of the independent trigger expression (with the `sdl -expression <`*`expressions`*`>` command).

You can query whether SDL is active or not through the `sdl -isActive` command. The return values of the `sdl -isActive` command only refer to the SDL program and do not consider the independent trigger expressions specified using the `sdl -expression <`*`expressions`*`>` command.

Formatting and General Syntax Rules of SDL

This section describe the way the SDL Preprocessor handles comments, file inclusion, text macros, and conditional compilation. In addition, these sections list the formatting and syntactical rules you need to follow in SDL programs.

- SDL Format
- Comments
- White Space
- Identifiers
- Case Sensitivity
- Using File Inclusion in SDL
- Text Macros
- Conditional Compilation
- Using Values of Tcl Variables Inside SDL

SDL Format

SDL is a free-format language. You can break statements across multiple lines. An end of line is considered a white space. You can also put multiple statements on the same line.

The syntax of the Using File Inclusion in SDL facility is an exception. It must be contained on exactly one line.

Statements are typically terminated by a semicolon. However, where several statements are grouped into a single block using braces, then the right brace is used as the terminating token.

Comments

Insert comments by using `/* */` or `//` notation. For example:

```
/*  This is a comment. This comment style
          can span multiple lines. */
// This is also a comment. This comment style
// restarts at each new line.
```

Comments cannot be inserted in the middle of tokens (for example, keywords, symbols, constants).

White Space

SDL ignores spaces, tabs, end of lines and empty lines (all considered as “white space”), except where used as separators between tokens. You must use white space between tokens in cases where it would otherwise be impossible to separate them.

Identifiers

Identifiers are user given names, used by various statements and constructs in SDL. A few examples: Labels, Aliases, and State Names.

Identifiers are case sensitive, might contain only letters, digits and underscores, and might not start with a digit.

Case Sensitivity

All SDL keywords are case insensitive; they are recognized in either upper, lower or mixed case.

For example, `Trigger`, `trigger`, `TRIGGER` are all recognized as the same keyword.

Names that represent design signals, or other user given names (aliases, instance names, labels, and state names) are case sensitive.

Text Macros

Define text macros that can be used to represent any string:

```
`define <name> <string>
```

Following this definition, the form `` `<name> `` or `` `{<name>} `` are replaced by <*string*>

For example:

```
'define INST top1.top2.top3
if('INST.net) trigger;
```

The above statement is evaluated as:

```
if(top1.top2.top3.net) trigger;
```

The following statement will produce the same result:

```
if('{INST}.net) trigger;
```

Text macros can also be undefined as follows:

```
'undef <name>
```

Conditional Compilation

The following directives can be used to conditionally include or exclude certain parts of the SDL program:

```
'if 'ifdef 'ifndef 'elseif 'else 'endif
```

Following are some typical ways of using these directives:

```
    'if <value>
    <SDL_statements>
    'elseif <value>
    <SDL_statements>
    'else
    <SDL_statements>
    'endif
```

Note that `'ifdef` and `'ifndef` can be used instead of `'if` to test whether a symbol was previously defined (or not defined) with a `'define` directive. It can also be used together with `'elseif` and `'else` directives.

For example:

```
    'ifdef <name>
    <SDL_statements>
    'elseif <value>
    <SDL_statements>
    'endif
```

Each of the above directives must be specified on a single line.

Here, <*value*> must be one of the following:

- A constant integer number: If the value is non-zero, all the following SDL statements up to the next `'elseif`, `'else` or `'endif` are compiled. If the value is `0`, these statements are skipped.

- The form `$<var>` or `${<var>}`, where `<var>` is the name of an existing global TCL variable. The value of `<var>` must be an integer constant. This value is used as explained in the above bullet. Also, `<var>` might be an indexed TCL array like `abc(def)`.
- The form `` `<name> `` or `` `{<name>} `` where `<name>` was previously defined in a `` `define `` directive. The value that was previously assigned to `<name>` must be an integer constant. This value is used as explained in first bullet in this list.
- If `<value>` is none of the above, an error is issued.

Using the `sdl -pp <output_file>` command, you can generate a text file that shows the effect of applying the conditional compilation directives. Refer to the *WXE Command Reference Manual* for details of the `sdl -pp <output_file>` command.

Related Sections:

- Overview of State Description Language (SDL)
- Preparing, Compiling and Executing SDL Programs
- Overview of SDL Constructs

Overview of SDL Constructs

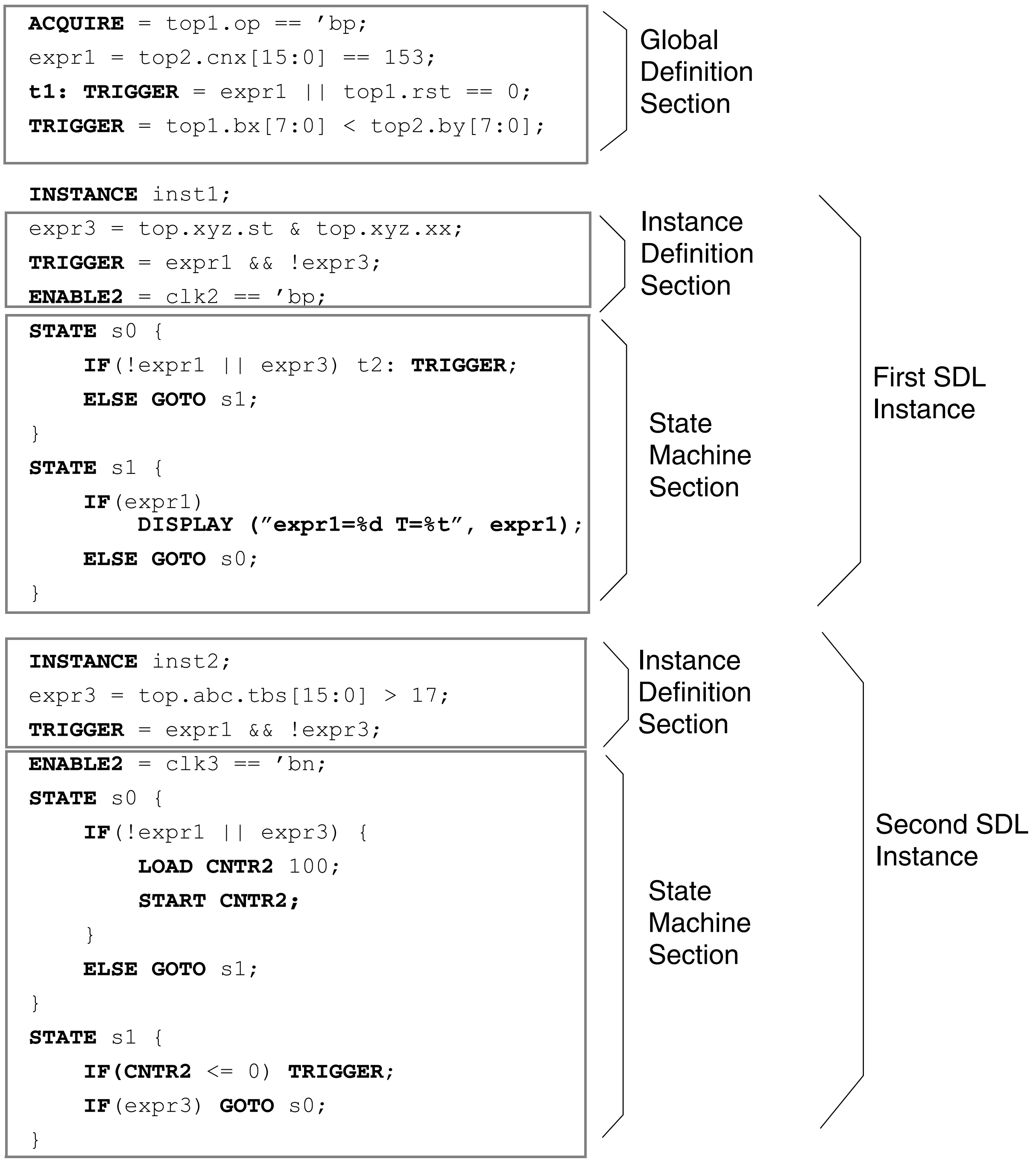
An SDL program can be as simple as a single statement of the form:

```
TRIGGER = <signal-name>;
```

Or it can be a complex program made of several SDL instances spanning on multiple files (through the use of the File Inclusion facility).

SDL Program Example depicts an SDL program with its various constructs. All SDL keywords appear in the example in uppercase bold letters. The SDL constructs explained below are optional.

Figure 30-1 SDL Program Example



```
ACQUIRE = top1.op == ’bp;
expr1 = top2.cnx[15:0] == 153;
t1: TRIGGER = expr1 || top1.rst == 0;
TRIGGER = top1.bx[7:0] < top2.by[7:0];

INSTANCE inst1;
expr3 = top.xyz.st & top.xyz.xx;
TRIGGER = expr1 && !expr3;
ENABLE2 = clk2 == ’bp;
STATE s0 {
    IF(!expr1 || expr3) t2: TRIGGER;
    ELSE GOTO s1;
}
STATE s1 {
    IF(expr1)
        DISPLAY (”expr1=%d T=%t”, expr1);
    ELSE GOTO s0;
}

INSTANCE inst2;
expr3 = top.abc.tbs[15:0] > 17;
TRIGGER = expr1 && !expr3;
ENABLE2 = clk3 == ’bn;
STATE s0 {
    IF(!expr1 || expr3) {
        LOAD CNTR2 100;
        START CNTR2;
    }
    ELSE GOTO s1;
}
STATE s1 {
    IF(CNTR2 <= 0) TRIGGER;
    IF(expr3) GOTO s0;
}
```

Global Definition Section
Instance Definition Section
State Machine Section
First SDL Instance
Instance Definition Section
State Machine Section
Second SDL Instance

An SDL program can have a Global Definition Section shared by all instances, followed by definitions of SDL Instances.

Each SDL instance starts with a Defining INSTANCE Statement, and extends until the next Instance statement (or the end of the file).

Each SDL instance can have its own Instance Definition section (whose definitions are limited only to that instance), followed by a State Machine section.

If the SDL program has only one instance, then the Instance statement can be omitted, and the software will assign to it a default instance name.

The State Machine section is made of a series of Defining STATE Statements.

SDL main constructs are grouped into the following categories:

- Using SDL Expressions
- Using Operators in SDL Programs
- Specifying General Definitions in SDL Programs
- Specifying Global Definitions in SDL Programs
- Defining INSTANCE Statement
- Defining STATE Statement
- Defining IF - ELSE statement
- Using SDL Actions

Related Sections:

- Using File Inclusion in SDL
- Using Predefined Keywords and Operators in SDL Expressions
- Using Predefined Keywords in Global or Instance Definition Sections in SDL Programs
- Using Predefined Keywords in State Machine Section

Using Predefined Keywords and Operators in SDL Expressions

Expressions typically use, in addition, names of design signals and assertions.

Table 31-1 Predefined Keywords and Operators for use in Expressions

Type	Keywords and Operators	Usage

Counter Names	`Counter1, Cntr1`
	`Counter2, Cntr2`
		Used in one of the following ways:
		`COUNTER1 <= 0`
		`COUNTER1 > 1000`
		`COUNTER2 <= 150`
		`COUNTER2 > 0`

Special Operators	`{}`	Concatenation operator. Creates a vector from a list of scalars.
	`[]`	Bit-select or part-select of vectored signals.
	`()`	Parenthesis for changing evaluation order.
Unary Operators	`transition()`	Detects any value transition
	`count()`	Counts events in an expression.
	`prev()`	Acts as a delay. The result is the expression delayed by one clock cycle.
	`isunknown ()`	Detects unknown values.
	`~`	Unary bitwise negation
	`!`	Unary logical negation

Table 31-1 Predefined Keywords and Operators for use in Expressions

Type	**Keywords and Operators**	**Usage**
Boolean Operators	`>`	greater than
	`<`	less than
	`>=`	greater than or equal to
	`<=`	less than or equal to
	`==`	equal to
	`!=`	not equal to
	`&`	unary reduction AND, bitwise AND
	`^`	bitwise XOR
	`|`	Unary reduction OR, bitwise OR
	`&&`	logical AND
	`||`	logical OR
Assertion state values	`"active"`	Used as in the following way: `<assertion-name> == "finished"`
	`"inactive"`	
	`"finished"`	
	`"failed"`	
	`"off"`	

Related Sections:

- Using Predefined Keywords in Global or Instance Definition Sections in SDL Programs
- Using Predefined Keywords in State Machine Section

Using Predefined Keywords in Global or Instance Definition Sections in SDL Programs

Predefined Keywords in a Global or Instance Definition Section lists the predefined keywords that can be used in the Global Definitions or the Instance Definition Section. The list does not include keywords that can be used in expressions, and are listed separately. The keywords listed as General Definitions (scope, enum, tdef) can also appear between state statements.

Table 32-1 Predefined Keywords in a Global or Instance Definition Section

Type	SDL Keyword	Usage
Definitions of Numeric Values	`counter1, cntr1`	Specify default load value for one of the two counters.
	`counter2, cntr2`	
	`triggerPos`	Specify trigger position in percent. Can appear only in Global Definition section.
	`postTriggerSamples`	Specify the number of FCLK cycles that will be captured into the trace buffer after trigger. Can appear only in Global Definition section.
Counter Decrement Enable	`enable1`	Specify Decrementing clock for each of the two counters.
	`enable2`	
Simple IF Block	`if`	Used for global trigger, display, tx, acquire, no_acquire and exec
Trigger	`trigger, trig`	Global Trigger Definition
Command Execution	`exec`	Execute a Tcl/XEL command
Message Printing	`display`	Generate a formatted message

Table 32-1 Predefined Keywords in a Global or Instance Definition Section

Type	SDL Keyword	Usage
Transaction Recording	`tdef`	Define list of transaction attributes
	`tx`	Define beginning, end, or attribute values of a transaction
Conditional Acquisition	`no_acquire, noAcquire, nacq`	Specify when to acquire or not acquire probed data
	`acquire, acq`	
General Definitions	scope	Scope resolution of design objects
	enum	Aids in formatting of arguments of display and tx statements
	tdef	Define list of transaction attributes

Related Sections:

- Using Predefined Keywords and Operators in SDL Expressions
- Using Predefined Keywords in State Machine Section
- Specifying General Definitions in SDL Programs
- Specifying Global Definitions in SDL Programs

Using Predefined Keywords in State Machine Section

Other Predefined Keywords lists the predefined keywords that can be used for all other SDL constructs. All the entries in this table, except INSTANCE are used for definitions in the State Machine section. The list does not include keywords that can be used in expressions, and are listed separately.

Table 33-1 Other Predefined Keywords

Type	SDL Keyword	Usage
Instance	`instance`	Beginning of a new SDL instance
Basic blocks of a State Machine	`state`	Definition of an SDL state
	`if`	Control execution based on values of expressions
	`else`	
Counter Names	`counter1`	Names of counters for use in counter actions
	`counter2`	

Table 33-1 Other Predefined Keywords

Type	SDL Keyword	Usage
Actions	`decrement, decr, count`	Decrement counter by 1
	`increment, inc`	Increment counter by 1
	`load`	Load counter with initial value
	`start`	Start counter auto-decrement mode (Start to count every cycle)
	`stop`	Stop counter auto-decrement mode
	`goto`	Transfer control to another state
	`trigger, trig`	Stop design clocks or tracing
	`no_acquire, noAcquire, nacq`	Do not acquire traced data in current cycle
	`acquire, acq`	Acquire traced data in current cycle
	`exec`	Execute a Tcl or XEL command
	`display`	Generate a formatted message
	`tx`	Define beginning, end, or attribute values of a transaction

Related Sections:

- Using Predefined Keywords in Global or Instance Definition Sections in SDL Programs
- Using Predefined Keywords and Operators in SDL Expressions

Using SDL Expressions

General Guidelines for SDL Expressions

Expressions are used in `IF` statements to define conditions, or in the Global Definition Section or Instance Definition Section for definitions of `TRIGGER`, `DISPLAY`, `EXEC`, `ACQUIRE`, `NO_ACQUIRE`, `ENABLE1`, and `ENABLE2`.

They can also be used as building blocks for other expressions through Using Aliases in SDL.

Expressions are made of Names, Numeric Constants, Operators, Assertion State Conditions, and Counter Conditions. An expression can also be a single name, a single assertion condition, or a single counter condition.

The use of boolean operators and parenthesis in expressions is very similar to their use in C or Verilog.

When evaluated during run, the result of an expression can be either a scalar boolean value (0 or 1), or a vector of boolean values. As in Verilog, a vector can be interpreted as an integer. This interpretation becomes significant when applying an operator between the vector and an integer constant. For example:

```
sig1[7:0] > 23
```

In this example, an 8 element vector representing an 8 bit integer is compared to the integer 23.

The same conventions used in C or Verilog are used when evaluating an expression as a condition: the result of a scalar expression is considered TRUE if it is 1, and FALSE if it is 0. The result of a vector expression is considered TRUE if *any* of its bits is 1, and FALSE if *all* its bits are 0.

Related Sections:

- Using Names in SDL Expressions
- Checking Assertion State Conditions in SDL Expressions
- Using Numeric Constants in SDL Expressions
- Using Operators in SDL Programs
- Using Counter Conditions in SDL Expressions
- Specifying Global Definitions in SDL Programs

Using Names in SDL Expressions

A name appearing in an expression can be one of the following:

- *Signal name* - a name of a design signal.
- *DRTL output port* - An output port of a DRTL instance.
- *Assertion name* - a name of an assertion from your design.
- Alias - a name representing a scalar or vector expression defined as an alias to an expression in the Global or Instance Definition Sections of the SDL program. Refer to Using Aliases in SDL for more information about Aliases.
- *User Symbol* - a name defined externally to the SDL program through the `symbol` XEL command.
- *CPF condition* - a name of a CPF object or event. For detailed information, refer to the Specifying CPF Conditions in SDL Expressions section of the *Working with CPF Low-Power Designs on Palladium*.
- Name of a UPF object (used in either a UPF condition, or as an argument of a `DISPLAY` statement).

A User Symbol cannot represent arbitrary expressions. It is used by xeDebug for grouping unrelated scalar signals into vectors to streamline and facilitate some operations (such as reading or forcing signals).

An assertion name must contain the full hierarchical path. It can be used in Checking Assertion State Conditions in SDL Expressions.

Signal Names must contain the full hierarchical path. If a signal name represents a vector, then it can be accompanied by an explicit index notation. If a name representing a vector appears without any index notation, then all its bits are used. An index notation can select a single bit of the vector (bit-select) as in:

```
sig1[3]
```

Or it can select a range of bits (part-select) in a vector as in:

```
sig2[7:2]
```

The order of bits is determined by the two indexes inside the `[]`. In the example above, bit 7 of sig2 is interpreted as the MSB, while bit 2 is interpreted as the LSB.

Aliases or User Symbols cannot be used with an explicit index notation.

If an Alias, User Symbol, or SDL keyword uses the same name as a design signal, then this does not cause an error condition, but the design signal becomes inaccessible to the SDL program. This is a potential issue only for top-level design signals specified without their hierarchy path.

A DRTL output port might be used in SDL like any design signal: for example, inside conditions or as arguments for DISPLAY or TX actions.

The DRTL output port can be referenced in the SDL file in the form:
`<`*`drtl_instance_name`*`>.<`*`drtl_output_port_name`*`>`

For example:

A DRTL instance is compiled from a verilog source using the following module definition:

```
module ftx (outp);
output[7:0] outp;
```

The verilog source is then compiled and a DRTL instance is defined using the following XEL commands:

```
drtl -definemodule mdx drtl.v -topmodule ftx
drtl -addinst mdx idrt()
```

The output port `outp[7:0]` of the DRTL instance will be referenced in SDL as `idrt.outp[7:0]`.

For example:

```
IF(idrt.outp[3:0] > 5 || idrt.outp[7:4] < 3) TRIGGER;
```

Related Sections:

- General Guidelines for SDL Expressions
- Using Counter Conditions in SDL Expressions
- Checking Assertion State Conditions in SDL Expressions
- Using IEEE 1801 Conditions in SDL Expressions
- Detecting any illegal UPF Events Using SDL Expressions
- Using Numeric Constants in SDL Expressions
- Using Special Non-Numeric Constants in SDL Expressions

Using Counter Conditions in SDL Expressions

Use the > and <= operators to compare the values of one of the two counters. Specify the counter name on the left side of the operator and the integer value on the right. The integer value must be between 0 and (2^{40} -1). Each SDL state can use only one unique integer value for comparing each counter.

The syntax can be any one of the following:

```
COUNTER1 <= <value>
COUNTER2 <= <value>
COUNTER1 > <value>
COUNTER2 > <value>
```

Here, `<value>` can be an integer between 0 and 2^{40}-1.

The first two cases that make use of the <= operator are evaluated as 1 (or TRUE) if the counter is less than or equal to the integer value specified by `<value>`.

The third and fourth cases are the inverse of the first two.

For example, `COUNTER1 <= 0 7` is equivalent to `!(COUNTER1 > 0 7).`

The keywords `CNTR1` and `CNTR2` can be used instead of `COUNTER1` and `COUNTER2`.

Related Sections:

- General Guidelines for SDL Expressions
- Using Names in SDL Expressions
- Checking Assertion State Conditions in SDL Expressions
- Using IEEE 1801 Conditions in SDL Expressions
- Detecting any illegal UPF Events Using SDL Expressions
- Using Numeric Constants in SDL Expressions
- Using Special Non-Numeric Constants in SDL Expressions

Checking Assertion State Conditions in SDL Expressions

Expressions in SDL program can check not only values of design nets, but also the state of assertions and CPF objects.

The state of an assertion can be tested by an expression using the following syntax:

```
<assertion-name> == "<assertion-state>"
```

<assertion-name> is the full hierarchical name of the assertion from the RTL description.

<assertion-state> is one of the following five predefined strings:

```
active inactive finished failed disabled
```

These strings can be used in either lower, upper or mixed case.

For example here is a statement that will print out a message when an assertion gets to a finished state:

```
IF (top.ctl.my_assertion == "finished")
    DISPLAY ("my_assertion arrived to a finished state");
```

It is also possible to check whether any assertion in the design failed, by testing the value of the net `ASSERTION.FAILURE.` This net is defined by the compiler only if the design has assertions in it.

For example:

```
IF (ASSERTION.FAILURE)
    DISPLAY ("some assertion failed");
```

Related Sections:

- General Guidelines for SDL Expressions
- Using Names in SDL Expressions
- Using IEEE 1801 Conditions in SDL Expressions
- Detecting any illegal UPF Events Using SDL Expressions
- Using Numeric Constants in SDL Expressions

- Using Special Non-Numeric Constants in SDL Expressions
- Compiling and Running Designs with Assertions

Using isunknown() Function Inside SDL Expressions

`isunknown()` is an SDL function dedicated to handling unknown values. The returned value of the function `isunknown(<val>)` is 0 if *`<val>`* represents a known value, and *`1`* if not.

`<val>` can be a scalar net, or a vector (bus). If *`<val>`* is a vector, the function returns *`0`* only if all the bits in the vector represent known values (0's or 1's). Otherwise it returns *`1`*.

`<val>` itself must be a design net or a vector of design nets (Or the result of a concatenation operator that represents such objects), as any other expression is flagged as an error.

Any SDL expression, regardless of where it is used can contain the function `isunknown()`. This function can be used multiple times within the same expression.

Related Sections:

- General Guidelines for SDL Expressions
- "4-State Support with SDL" section in *Working with 4-State Logic in Palladium* guide
- Using Counter Conditions in SDL Expressions
- Checking Assertion State Conditions in SDL Expressions
- Using IEEE 1801 Conditions in SDL Expressions
- Detecting any illegal UPF Events Using SDL Expressions
- Using Numeric Constants in SDL Expressions
- Using Special Non-Numeric Constants in SDL Expressions

Using IEEE 1801 Conditions in SDL Expressions

The state of an IEEE 1801 object can be tested by an expression using the following syntax:

```
<1801-object-name> == ”<1801-state>”
```

<1801-object-name> is the full hierarchical name of the IEEE 1801 object.

The `power::` prefix can be added to any of the objects. This is useful if there is a need to distinguish it from a design net with the same name.

<1801-state> is a mnemonic description of the state. It is case sensitive.

Note: Depending on the object type, it is possible that multiple state mnemonics match the state of an object at the same time.

Example:

```
IF (top.VDD1 == ”FULL_ON”) trigger;
```

Or,

```
IF (power::top.VDD1 == ”FULL_ON”) trigger;
```

Following is the list of possible mnemonics for each type of IEEE 1801 object:

- Supply Nets:
  - `OFF FULL_ON PARTIAL_ON UNDETERMINED`
- Supply Ports
  - User-defined states from the `add_port_state` command.
  - `OFF FULL_ON PARTIAL_ON UNDETERMINED`
- Supply Sets
  - User-defined state from the `add_power_state` command.
  - `NORMAL CORRUPT`
- Power Switches
  - User-defined state names from the `create_power_switch` command.

- Power State Table (pst)
    - User-defined state from the `add_pst_state` command.
- Power Domain
    - User-defined state from the `add_power_state` command.
    - `NORMAL CORRUPT`
- Isolation
    - `NORMAL CORRUPT`
    - `ISOLATED NOT_ISOLATED`

    Note: The `ISOLATED NOT_ISOLATED` mnemonics are specific for use in SDL.
- Retention
    - `NORMAL CORRUPT`
    - `RETAINED NOT_RETAINED`

    Note: The `RETAINED NOT_RETAINED` mnemonics are specific for use in SDL.

In addition, it is also possible to test for power states that are described by a single bit (either true or false), as if they are simple nets using the following syntax:

```
<1801-object-name>@<state>
```

Example:

```
IF (top.VDD1@corrupted) trigger;
```

Or,

```
IF (power::top.VDD1@corrupted) trigger;
```

You can get a list of all of the possible conditions in that format for the current design using the `lp -check sdl [<eventType>]` XEL command.

Related Sections:

- For more details about a formal definition for all possible objects and states that can be used with this syntax, refer to the Using IEEE 1801 Conditions in SDL Expressions section in *Working with IEEE 1801 Low-Power Designs on Palladium*.
- General Guidelines for SDL Expressions
- Using Names in SDL Expressions

- Using Counter Conditions in SDL Expressions
- Checking Assertion State Conditions in SDL Expressions
- Detecting any illegal UPF Events Using SDL Expressions
- Using Numeric Constants in SDL Expressions
- Using Special Non-Numeric Constants in SDL Expressions

Detecting any illegal UPF Events Using SDL Expressions

UPF instrumentation constantly monitors for power sequences that are illegal. You can use the `UPF_ILLEGAL_EVENT` signal in an SDL expression. This signal goes high whenever there is an occurrence of any illegal power sequence (Available only with UPF. Not supported with CPF).

Individual illegal events might be disabled by the `lp -illegal` command; disabled events will not contribute to the `UPF_ILLEGAL_EVENT` signal.

Related Sections:

- General Guidelines for SDL Expressions
- Using Names in SDL Expressions
- Using Counter Conditions in SDL Expressions
- Checking Assertion State Conditions in SDL Expressions
- Using IEEE 1801 Conditions in SDL Expressions
- Using Numeric Constants in SDL Expressions
- Using Special Non-Numeric Constants in SDL Expressions

Using Numeric Constants in SDL Expressions

Constants used in expressions can be specified with either binary, octal, decimal or hexadecimal radix. The radix notation for regular numerical constants is very similar to the one used by Verilog. For example, a 6-bit constant that represents the decimal value of 14 can be written as any one of the following:

binary	`6’b1110`
octal	`6’o16`
decimal	`6’d14`
hexadecimal	`6’hE`

Most of the Verilog conventions for writing integer constants are followed:

- The radix letter as well as hexadecimal letters are case insensitive.
- If the number of bits is not specified, then it is inferred from the value. For example, `’b1110` is a 4 bit constant representing the value 14.
- A number without an explicit radix notation is assumed to be decimal.
- Underscores can be inserted anywhere in the string (except left most character) to improve readability, but they are otherwise ignored. For example, `’b1110_1111` is equivalent to `’b11101111`.

Related Sections:

- General Guidelines for SDL Expressions
- Using Names in SDL Expressions
- Using Counter Conditions in SDL Expressions
- Checking Assertion State Conditions in SDL Expressions
- Using IEEE 1801 Conditions in SDL Expressions
- Detecting any illegal UPF Events Using SDL Expressions

- Using Special Non-Numeric Constants in SDL Expressions

Using Special Non-Numeric Constants in SDL Expressions

SDL also recognizes a set of *special values* that can be used only in conjunction with the comparison operator (`==`). These values are represented by the following letters, which can be in either upper- or lower-case:

`P` - Positive edge.
`N` - Negative edge.
`T` - Any transition (Positive or negative edge).
`L` - Value is stable at a low value (0 for two consecutive FCLK cycles).
`H` - Value is stable at a high value (1 for two consecutive FCLK cycles).
`S` - Value is stable at either low or high value for two consecutive FCLK cycles.
`X` - Don't care; any value is accepted as a match.

The only thing that can be done with these values is compare them to a signal or to an expression with the (`==`) operator. Expressions that are used in such comparisons cannot contain counter conditions.

Note: 4-state flow affects comparison operator (`==`)

Some of the special values in SDL comparison expressions have a new interpretation in the presence of unknown values in 4-state flow. For more information, see "New Interpretation of Non-Numeric Constants for Comparison with Special Values" section in *Working with 4-State Logic in Palladium* guide.

Unknown values are indicated through the character `X` in Verilog. While the value `X` in SDL comparison indicates `don't care` thereby has different semantics than unknown. Undriven values are indicated through the character `Z` in Verilog but `Z` is not recognized in SDL comparison statements.

With the exception of `X` (`don't care`), all other values represent a combination of two consecutive values from the current evaluated cycle and the previous cycle. For example:

```
sig1 == 'bP
```

is testing for a positive edge on the signal `sig1`. The result is 1 (`TRUE`) if the signal `sig1` had the value 0 in the previous cycle and the value 1 in the current cycle.

Consider another example:

```
(sig1 ^ sig2) == 'bT
```

Here, any transition occurring on the XOR result of two signals (the left operand) will make the result of the entire expression 1 (`TRUE`). If the two signals are transitioning only between 0 and 1 (no unknown values), then this expression is true if one of the signals sig1 or sig2 makes a transition, but not if both make a transition at the same time.

Constants using these values must be specified with an explicit radix notation of either binary, octal, or hexadecimal.

When specified as part of a multi-digit constant, each of the letters can represent either 1 bit, 3 bits or 4 bits, depending on the radix, and you can mix in the same constant both numeric digits and any of the special non-numeric constants. For example, the expression:

```
aa[11:0] == 12'h5PX
```

is equivalent to:

```
aa[11:0] == 12'b0101PPPPXXXX
```

Comparing anything to the value `X` always yields a result of 1. This is useful to mask out selected bits when comparing a vector to a constant.

The above expression can therefore be re-written also as:

```
(aa[11:8] == 5) && (aa[7:4] == 4'bPPPP)
```

Note that the least significant 4 binary digits are `X` (`don't care`), so the sub-expression (`aa[3:0] == 4'bXXXX)` became redundant and was omitted altogether.

A typical use of the S (stable) value is to test for a change (or transition) in the value of a vector (meaning change in any of its bits). This is done in practice by checking whether all of the bits are stable, and negating the result, as in the following example:

```
TRIGGER = !(myvec[16:0] == 'hSSSS); //trigger on any change
```

Note: When comparing the value of a design signal with any of the special values except X (that is, any one of: P,N,T,H,L,S), there are limitations that must be observed:

- If the design was compiled to run in 1x or 0.5x sampling mode, it is possible that a signal will make two or more transitions within the same FCLK cycle. For details about how SDL handles this, refer to Limitations of SDL when Running with Sampling Ratio section.
- Performing a deposit/force/release through XEL on a signal can cause SDL to evaluate incorrectly any condition with special values that depend on that signal or other signals that it drives through a combinatorial network.
- SDL assumes on the first cycle of each run (except a `run` command with the `-continue` option) that values of design signals did not change from the previous cycle, regardless of what the value in the previous cycle was. In other words, an expression such as

`(signal == 'bP)` is always evaluated on the first cycle of the run as false and (`signal == 'bS`) as true. This only affects the `ACQUIRE` or `NO_ACQUIRE` actions and does not affect other types of actions because all other types of actions start to evaluate only on the second cycle of the run.

- Creating an expression that tries to nest conditions with special values except `X` (that is, any one of: `P,N,T,H,L,S`), might not evaluate correctly on the first cycle of the run (except the `run -continue` command) because it requires knowledge of signal values for more than one cycle in the past. For example, an expression such as: `((signal == 'bs) == 'bp)` might not evaluate correctly on the first evaluation cycle of SDL. This limitation affects all types of SDL actions.

Related Sections:

- General Guidelines for SDL Expressions
- Using Names in SDL Expressions
- Using Counter Conditions in SDL Expressions
- Checking Assertion State Conditions in SDL Expressions
- Using IEEE 1801 Conditions in SDL Expressions
- Detecting any illegal UPF Events Using SDL Expressions
- Using Numeric Constants in SDL Expressions
- "4-State Support with SDL" section in *Working with 4-State Logic in Palladium* guide

Using Operators in SDL Programs

Using Vector Operands with Different Widths

SDL handles operations between two vectors that have different widths in a similar way to how Verilog handles unsigned operations, but with minor exceptions, as explained below.

Verilog semantics say that when evaluating a bit-wise operation (such as & | ^ == !=) between two vectors that are unsigned, if the two operands have different widths, then the operand with the smaller width is extended by adding most significant zero elements.

For example, the following expression:

```
A[5:0] == 4’b1111
```

is evaluated by SDL as follows:

```
A[5:0] == 6’b001111
```

SDL has two exceptions to this rule.

- If both operands are not constants, then SDL prints out a warning message if the sizes do not match.

  For example, the following expression is legal, and evaluated as explained above, but will produce a warning message anyway:

  ```
  A[5:0] == B[3:0]
  ```

- The second exception can happen with the == operator when the operand with the smaller width is a constant. If its most significant bit is one of the special non-numeric constants (P,N,T,L,H,S,X), SDL will flag the expression as an error.

  For example, the following will be flagged as an error:

  ```
  A[5:0] == 4’bxx11
  ```

Related Sections:

- Evaluation Order and Operator Precedence Rules
- SDL Operators Description

Evaluation Order and Operator Precedence Rules

When several different operators appear in the same expression, the evaluation order is done according to the same precedence rules used by C or Verilog. This is depicted by Operators Precedence. Operators in the same box have the same precedence. Parenthesis can be used to change evaluation order.

For example, the operator `==` has higher precedence than `&`, therefore:

```
A & B == C
```

Is equivalent to:

```
A & (B == C)
```

Related Sections:

- Using Vector Operands with Different Widths
- SDL Operators Description

Table 46-1 Operators Precedence

Priority	Operator	Description
Highest	■ ()	Parentheses
	■ transition()	Detect any transition
	■ count()	Count events
	■ prev()	Delay one cycle
	■ isunknown ()	Detects unknown values
	■ ~	Unary bitwise negation
	■ !	Unary logical negation
	■ &	Unary reduction AND
	■ |	Unary reduction OR
	■ >	greater than
	■ <	less than
	■ >=	greater than or equal to
	■ <=	less than or equal to
	■ ==	equal to
Lowest	■ !=	not equal to
	■ &	bitwise AND
	■ ^	bitwise XOR
	■ |	bitwise OR
	■ &&	logical AND
	■ ||	logical OR

SDL Operators Description

All the boolean operators can be used on any names (for example, signal names, aliases and so on), constants or expressions, with the only exception that the special non-numeric constants can only be used with the == operator and only against expressions that do not contain Counter Conditions.

Table 47-1 Operators Description

Operator	Description	Syntax
Unary Bitwise Negation (~)	When applied to a scalar expression, this operator performs a boolean NOT (inverse) operation. When applied to a vector expression, the result is also a vector with each bit inverted individually.	`~ <Expression>`
Unary Logical Negation (!)	When applied to a scalar expression performs a boolean NOT (inverse) operation. When applied to a vector expression, the result is a single bit with a value of 1 if all the bits in the vector are 0. If any of the bits in the vector are 1, the result is the single bit 0.	`! <Expression>`
Unary Reduction AND (&)	Performs a boolean AND operation between all the elements of a vector expression, producing a single bit.	& *<Expression>* For example, the expression: &A[3:0] is equivalent to: A[3] & A[2] & A[1] & A[0]

Table 47-1 Operators Description

Operator	Description	Syntax
Unary Reduction OR (|)	Performs a boolean OR operation between all the elements of a vector expression, producing a single bit.	| *<Expression>* For example, the expression: |A[3:0] is equivalent to: A[3] | A[2] | A[1] | A[0]

Magnitude Comparison Operators (> < >= <=)	These operators compare between two scalar or vector expressions, and produce a single bit as a result. The result is 0 if the comparison fails, and 1 if the comparison succeeds. Each of the two operands is considered as an unsigned integer.
	
	The number of bits in any one of the two operands is not limited by the SDL syntax, but keep in mind that magnitude comparison operators tax emulation resources more than basic equal/not equal operators, so a large number of bits can cause the SDL run time loader to fail.
		Greater Than:
		
		`<Expression> > <Expression>`
		
		Less Than:
		
		`<Expression> < <Expression>`
		
		Greater Than or Equal To:
		
		`<Expression> >= <Expression>`
		
		Less Than or Equal To:
		
		`<Expression> <= <Expression>`

Equal To (==)	Compares two scalar or vector expressions, and produces a single bit as a result. The result is 1 if the two expressions match, and 0 if not.	`<Expression> == <Expression>` The `==` operator can also be used to detect special time dependent conditions (for example, a positive edge). For details, see Using Special Non-Numeric Constants in SDL Expressions.

Table 47-1 Operators Description

Operator	Description	Syntax
Not Equal To (!=)	Compares two scalar or vector expressions, and produces a single bit as a result. The result is 0 if the two expressions match, and 1 if not.	*`<Expression>`* != *`<Expression>`*
Bitwise AND (&)	Performs a boolean AND operation between two expressions. If the operands are vectors, the result is also a vector with the same number of bits as the operands.	*`<Expression>`* & *`<Expression>`*
Bitwise XOR (^)	Performs a boolean Exclusive OR operation between two expressions. If the operands are vectors, the result is also a vector with the same number of bits as the operands.	*`<Expression>`* ^ *`<Expression>`*
Bitwise OR (|)	Performs a boolean OR operation between two expressions. If the operands are vectors, the result is also a vector with the same number of bits as the operands.	*`<Expression>`* | *`<Expression>`*
Logical AND (&&)	This operation produces a single bit as result whether the operands are vectors or scalars. If the operands are scalars, performs a boolean AND operation. If any (or both) of the operands is a vector, then it is first converted to a scalar by performing an OR operation between all its bits. The final result is therefore 0 only if all the bits are 0 in at least one of the operands.	*`<Expression>`* && *`<Expression>`*

Table 47-1 Operators Description

Operator	Description	Syntax
Logical OR (||)	This operation produces a single bit as result whether the operands are vectors or scalars. If the operands are scalars, performs a boolean OR operation. If any (or both) of the operands is a vector, then it is first converted to a scalar by performing an OR operation between all its bits. The final result is therefore the boolean OR operation between all the bits of both operands.	*`<Expression>`* || *`<Expression>`*

Table 47-1 Operators Description

Operator	Description	Syntax
Concatenation ({...})	A Verilog-style concatenation operator enables you to concatenate scalars and vectors to create new vectors for use in expressions.	{*`<name>`*,*`<name>`*,...} Example: {a,b,c,d,f[3:0]} The result is an 8 bit vector with signal `a` as the most significant bit, and `f[0]` as the least significant bit. It is equivalent to: {a,b,c,d,f[3],f[2],f[1],f[0]} Elements inside the concatenation can only be signal names and symbols defined by the `symbol` XEL command. Numeric constants, Aliases, or other expressions are not allowed as elements inside concatenation. **Note:** SDL propagates 4-state values through concatenation operator.

Table 47-1 Operators Description

Operator	Description	Syntax
transition()	The keyword transition is recognized in upper and lower case.	transition(<*expression*>) Here, <*expression*> is typically a vector, but can also be a scalar. The result of the transition operator is true if any of the bits in <*expression*> makes a low-to-high or high-to-low transition. If <*expression*> is a scalar, the `transition(<`*expression*`>)` is equivalent to `(<`*expression*`>== 'bT)`. A transition operator on a multi-bit expression can also be translated to an expression with the `==` operator. For example: transition(A[3:0]) is equivalent to the expression: !(A[3:0] == ’hS) Or, to the expression: A[3] == ’bT || A[2] == ’bT || A[1] == ’bT || A[0] == ’bT **Note:** 4-state flow affects transition operator `()`

Table 47-1 Operators Description

Operator	Description	Syntax
count()	The `count()` operator represents an actual counter that is private to the enclosing expression — that is, the counter is not shared with other expressions. This counter should not be confused with *counter expressions* that use the general purpose SDL counters and have a syntax like the following: *`(counter1 <= 8)`* The keyword `count` is recognized in either uppercase or lowercase.	count(<*`expression`*>, <*`num`*>) Or: count(<*`expression`*>, <*`num`*>, <*`mode`*>) Where, <*`expression`*> is any SDL expression. However, it should not contain a counter expression. <*`num`*> is a positive integer. <*`mode`*> is one of the following strings in either uppercase or lowercase: `P S PS SP`
	Without <mode>: The result of the operator becomes `1` after counting a total of <*`num`*> cycles (not necessarily consecutive) in which the SDL expression is high. The result remains 1 till the counter is reset at the beginning of the next run,	

Table 47-1 Operators Description

Operator	Description	Syntax

	With <mode>:
	
	The mode can be specified in either lower or uppercase. When the mode is specified, the counter operates as follows:
	
	■ If <*mode*> is `S`: The counter is reset each time when SDL enters the specific state in which the expression exists — that is, the `count()` operator becomes 1 only if the counter completes counting from 1 to <*num*> without SDL leaving the state.
	
	■ If <*mode*> is `P`: The counter counts the number of times <*expression*> changes from 0 to 1, instead of the number of times it stays at value 1.
	
	■ If <*mode*> is `PS` or `SP`: This mode is equivalent to the combination of modes `S` and `P`. The counter is initialized at the beginning of the state instead of beginning of the run, and the counter counts 0 to 1 transitions of the expression.
		The `S` mode is ignored when the expression is used in a condition for a global action (outside any state statement).
		
		The `P` mode serves as a shortcut for the syntax (<*expression == 'bp*>). For example,
		
		`count(abc[15:0] == 13, 50, P)`
		
		is equivalent to:
		
		`count((abc[15:0] == 13) == 'bp, 50)`
		
		For example:
		
		`if(count(net1,10,p) || count(net2,20)) trigger;`
		
		In the example above, trigger happens if either the value of `net1` undergoes 10 transitions from 0 to 1, or if `net2` stays high for a total of 20 FCLK cycles.

Table 47-1 Operators Description

Operator	Description	Syntax
isunknown ()	Detects unknown values. The returned value of the function is 0 if *`<val>`* represents a known value, and 1 if not.	`isunknown(`*`<val>`*`)` *`<val>`* can be a scalar net, or a vector (bus). If *`<val>`* is a vector, the function returns 0 only if all the bits in the vector represent known values (0's or 1's). Otherwise it returns 1. *`<val>`* itself must be a design net or a vector of design nets (Or the result of a concatenation operator that represents such objects), as any other expression is flagged as an error.
prev()	The `prev` operator acts as a delay — that is, the result of a `prev` operator is the expression delayed by one clock cycle. The keyword `prev` is recognized in either uppercase or lowercase.	`prev(<`*`expression`*`>)` <*`expression`*> is any SDL expression, scalar or vector. It should not contain a counter expression, however, <*`expression`*> can contain the `count` operator.

Related Sections:

- Using Vector Operands with Different Widths
- Evaluation Order and Operator Precedence Rules

Specifying General Definitions in SDL Programs

Any of the following definitions might appear anywhere in the SDL program, outside of `STATE` statements. The definition affects all subsequent statements until the end of the SDL program, the end of the SDL instance, or a new definition that overrides it (Details vary, depending on the type of statement, see the details mentioned below).

Table 48-1 General Definitions

Definition	Description	Syntax

Alias	An alias defines a new name which can be used later instead of typing the whole expression. The expression used for defining an alias can itself contain names of other aliases already defined.
	
	An alias appearing in a Global Definition Section can be used anywhere below that definition, including all the SDL instances. An alias definition in an Instance Definition Section is private to that Instance.
	
	The same alias name can be used to define a different alias value in different instances. However, if an alias was defined in the Global Definition Section, then it cannot be redefined in any Instance Definition Section.
	
	An alias cannot be indexed and cannot be used as part of a hierarchical path. For example, if YYY is defined as an alias like `YYY = XXX`, then the following usage is not supported:
	
	`YYY[0]` cannot be used as `XXX[0]`
	
	`YYY.ABC` cannot be used as `XXX.ABC`
		`<name>` = *`<Expression>`*;
		
		`<name>` can contain only alphanumeric characters or an underscore (_), and cannot begin with a digit. Names of aliases are case sensitive.

Table 48-1 General Definitions

Definition	Description	Syntax

enum	Provides an efficient way to chose a string out of a list of several strings, based on an integer value, when formatting integer values in `DISPLAY` or `TX` statements.	The syntax of the enumeration statement is:
		`enum <enum_type>`
		`     {`
		`        <string> [=<value>],`
		`        <string> [=<value>],`
		`        ...`
		`     };`
		For more details, refer to Using DISPLAY Action.

tdef	Defines a default set of attributes for `TX` statements.	The syntax of the `tdef` statement is as follows:
		`tdef <tdef_id>`
		`     {`
		`        <arg>,`
		`        <arg>,`
		`        ...`
		`     };`
		For more details, refer to Using TX Statement.

Table 48-1 General Definitions

Definition	Description	Syntax

scope	Defines a hierarchical scope for nets and other design objects that appear in expressions.
	
	The scope definition affects all the expressions below it up to the next scope statement or the end of the SDL program. Any design object below the scope definition is first searched in the design hierarchy as defined by the scope statement, and if not found there, it is searched in the top hierarchy.
	
	For example, consider the following:
	
	`scope top.h1.h2;`
	`if(a1.n1) trigger;`
	
	SDL will use for the trigger statement the signal `top.h1.h2.a1.n1` if it exists. If there is no such signal, it will use the signal `a1.n1`, and will issue an error message if this signal also does not exist.
		To set the scope:
		
		`scope <scope>;`
		
		To reset the scope:
		
		`scope;`
		
		In the first form, `<scope>` specifies the hierarchical scope. A period is used as a hierarchy separator.
		
		The second form in which `<scope>` is missing, resets the scope, which means objects are searched only under top hierarchy (The default behavior if the SDL program does not contain any `scope` statements).

Specifying Global Definitions in SDL Programs

Global definitions in an SDL program are statements that appear either in the Global Definition section, or in the Instance Definition section.

- **Global Definition Section**: Appears at the top of the SDL program above the first INSTANCE Statement. Its definitions apply to all the SDL instances.
- **Instance Definition Section**: Appears between the INSTANCE statement and the first STATE statement of that instance. Its definitions apply only to that instance. See SDL Program Example.

If the SDL program has only one instance with no INSTANCE statement, then the Global Definition section and the Instance Definition section are the same thing.

Except arguments of DISPLAY statements, expressions used in all other global definitions cannot rely on any state machine resources. This means they cannot contain counter values.

Global definitions use either the syntax:

```
<name> = <value>;
```

or the syntax:

```
if(<expression>) <action>;
```

<*name*> can be one of the keywords: `TRIGGER, ACQUIRE, NO_ACQUIRE, COUNTER1, COUNTER2, ENABLE1, ENABLE2, TRIGGERPOS, POSTTRIGGERSAMPLES`.

<*value*> is either a numeric value or an expression depending on what is being defined.

The second syntax is supported by the following statements types: `TRIGGER, DISPLAY, TX, ACQUIRE, NO_ACQUIRE,` and `EXEC`, which are similar (both syntax and functions) to the actions with same name inside the SDL state machine.

Various possible global definitions are discussed in the following sections:

- Specifying the Trigger Position
- Overriding Post Trigger Samples
- Specifying Count Enable Definition

Specifying the Trigger Position

SDL can override the value of the `triggerPos` run-time parameter by providing a different value in the Global Definition Section.

The syntax is:

```
triggerPos = <value>;
```

The keyword `triggerPos` can appear in lower, upper, or mixed case. *`<value>`* is an integer number between 0 and 100. This value is assigned to the `triggerPos` run-time parameter (overwriting any previous value) when the `run` command is executed, and it is used during the run.

A `triggerPos` definition can appear only in the Global Definition Section, but not in the Instance Definition Section. Both `triggerPos` and `postTriggerSamples` definitions cannot appear together in the same SDL program.

Related Sections:

- For more information on the semantics of the `triggerPos` run-time parameter, refer to the description of the `xeset` command in the *Run-Time Commands* chapter of the *WXE Command Reference Manual*.
- Specifying Global Definitions in SDL Programs
- Specifying General Definitions in SDL Programs

Overriding Post Trigger Samples

SDL can override the value of the `postTriggerSamples` run-time parameter by providing a different value in the Global Definition Section.

The syntax is:

```
postTriggerSamples = <value>;
```

The `postTriggerSamples` keyword can appear in lower, upper or mixed case. *`<value>`* is a non-negative integer number. This value is assigned to the `postTriggerSamples` run-time parameter (overwriting any previous value) when the `run` command is executed, and it is used during the run.

A `postTriggerSamples` definition can appear only in the Global Definition Section, but not in the Instance Definition Section. Both `triggerPos` and `postTriggerSamples` definitions cannot appear together in the same SDL program.

Related Sections:

- For more information on the semantics of the `postTriggerSamples` run-time parameter, refer to the `xeset` command in the *Run-Time Commands* chapter of the *WXE Command Reference Manual*.
- Specifying Global Definitions in SDL Programs
- Specifying General Definitions in SDL Programs

Specifying Count Enable Definition

The *Count Enable* definition defines a condition upon which one of the two counters will decrement when in auto-decrement mode (Using START and Using STOP Action actions). The Global Definition Section or Instance Definition Section can contain at most one count enable definition for each counter.

A count enable definition appearing in the Global Definition Section applies to all SDL instances. A count enable definition in an Instance Definition Section is private to that Instance.

Different enable definitions can be defined in different instances. However, if an enable definition appears in the Global Definition Section, then it cannot be redefined in any Instance Definition Section.

If no enable definition exists for a specific counter, then that counter will decrement once every FCLK cycle (when in auto-decrement mode).

Note: The VD and ICE run modes have clear semantics that define a linear relationship between FCLK cycles and simulation time.

This construct is therefore most useful for Simulation Acceleration mode which does not have such clear semantics.

Syntax:

```
ENABLE1 = <expression>;
```

or:

```
ENABLE2 = <expression>;
```

A typical use is decrementing on a specific clock edge from the design, as in the following example (count on positive edge of `mainclock`):

```
ENABLE1 = mainclock == 'bp;
```

Related Sections:

Specifying Global Definitions in SDL Programs

Specifying the Counter Load Value

The *Counter Load Value* defines the default numeric value for a `LOAD` action if none was specified with the `Load` action itself. The Global Definition Section or Instance Definition Section can contain at most one counter load value definition for each counter.

A load value definition appearing in the Global Definition Section applies to all SDL instances. A load value in an Instance Definition Section is private to that Instance.

Different load values can be defined in different instances. However, if a load value was defined in the Global Definition Section, then it cannot be redefined in any Instance Definition Section.

Syntax:

```
COUNTER1 = <value>;
```

or:

```
COUNTER2 = <value>;
```

The shorthand notations `CNTR1` and `CNTR2` are also recognized.

`<value>` is an unsigned integer value between 0 and 2^{40}-1. The value can be specified as a decimal number or in a different radix (as specified in Using Numeric Constants in SDL Expressions).

Related Sections:

- Specifying Global Definitions in SDL Programs
- Using LOAD Action

Defining Trigger Condition with Global TRIGGER

The `TRIGGER` statement defines a trigger condition. The trigger is executed whenever the specified expression becomes true regardless of any other trigger statements. Any number of global trigger statements can be declared in any of the Global Definition Section or Instance Definition Section, and each is evaluated independently every cycle. A true condition in any of them will cause a trigger. The condition can be constructed as explained in Using SDL Expressions, except that it cannot contain counter conditions (for example, `COUNTER1 <= 0`).

Syntax:

```
<Label>: TRIGGER = <Expression>;
```

The above construct has an equivalent form using an `IF()` clause as follows:

```
IF(<Expression>)
      <Label>: TRIGGER;
```

The shorthand notation `TRIG` is also accepted instead of `TRIGGER`.

`<Label>` is a string made of letters, digits, or underscores, and must start with a letter or underscore. It can be used by XEL for the purpose of identifying, enabling or disabling any `TRIGGER` statements that have a specific label. *`<Label>`* does not have to be unique within the SDL program. The construct *`<Label>`*`:` is optional. It can be omitted if labeling the statement is not needed.

Related Sections:

- For more details on this statement, see Using TRIGGER Action.
- Specifying Global Definitions in SDL Programs

Defining Global ACQUIRE

The global `ACQUIRE` statement defines a condition which when true in any cycle, will cause probed data capturing in that cycle. Any number of global `ACQUIRE` statements can be declared in any of the Global Definition Section or Instance Definition Section, and each is evaluated independently every cycle. A true condition in any one of them will cause data capturing in that cycle. The condition can be constructed as explained in Using SDL Expressions, except that it cannot contain counter conditions (for example, `COUNTER1 <= 0`).

`ACQUIRE` and `NO_ACQUIRE` statements (whether inside a STATE or a global/instance definition) cannot be used in the same SDL program.

Syntax:

```
ACQUIRE = <Expression>;
```

or an equivalent form:

```
IF(<Expression>)
    ACQUIRE;
```

The shorthand notation `ACQ` is also accepted instead of `ACQUIRE`.

Related Sections:

Specifying Global Definitions in SDL Programs

Defining Global NO_ACQUIRE

The `NO_ACQUIRE` statement defines a condition which when true in any cycle, will suspend probed data capturing in that cycle. Any number of global `NO_ACQUIRE` statements can be declared in any of the Global Definition Section or Instance Definition Section, and each is evaluated independently every cycle. A true condition in any one of them will suspend data capturing in that cycle. The condition can be constructed as explained in Using SDL Expressions, except that it cannot contain counter conditions (for example, `COUNTER1 <= 0`).

`ACQUIRE` and `NO_ACQUIRE` statements (whether inside a STATE or a global/instance definition) cannot be used in the same SDL program.

Syntax:

```
NO_ACQUIRE = <Expression>;
```

or an equivalent form:

```
IF(<Expression>)
    NO_ACQUIRE;
```

The shorthand notations `NOACQUIRE` and `NACQ` are also accepted instead of `NO_ACQUIRE`.

Related Sections:

Specifying Global Definitions in SDL Programs

Defining Global ACQUIRE for Streaming Probes

You can define conditions that control when to acquire streaming probes data, independent of regular probe data capturing.

Syntax:

```
if(<expression1>) acquire stream stop;
if(<expression2>) acquire stream start;
if(<expression3>) acquire stream;
```

Related Sections:

- Conditional Acquisition for Streaming Probes
- Specifying Global Definitions in SDL Programs

Using Global EXEC

The global `EXEC` statement defines a condition and an XEL command (or set of commands). The condition, when true, will initiate execution of the XEL command. In emulation modes that enable stopping the clocks, emulation clocks are suspended while the command executes. Refer to Using EXEC Action for more details on the `EXEC` facility.

The condition can be constructed as explained in Using SDL Expressions, except that it cannot contain counter conditions (for example, `COUNTER1 <= 0`).

Syntax:

```
IF(<Expression>)
   <Label>: EXEC "<command>";
```

`<Label>` is a string made of letters, digits, or underscores, and must start with a letter or underscore. It can be used by XEL for the purpose of identifying, enabling or disabling any `EXEC` statements that have a specific label. *`<Label>`* does not have to be unique within the SDL program. The construct *`<Label>`*`:` is optional. It can be omitted if labeling the statement is not needed.

Related Sections:

Specifying Global Definitions in SDL Programs

Using Global DISPLAY

The global `DISPLAY` defines a condition and a message to be printed. The condition, when true, generates a message printed to a file and/or the console.

Syntax:

```
IF(<Expression>)
   <Label>: DISPLAY (F=<ID>, L=<Level>, "<format-specifier>",
                    <Arg>, <Arg>, ...);
```

`<Expression>` is a condition. The `DISPLAY` message is printed whenever this *`<Expression>`* is true. It can be constructed as explained in Using SDL Expressions, except that it cannot contain counter conditions (such as, `COUNTER1 <= 0`).

By default, a maximum number of 256 `DISPLAY` and `TX` statements can be in the Global Definition Section and Instance Definition Sections (from all instances). If required, you can change this number by using the `sdlGlobalDisplay` option of the `compilerOption` command.

Related Sections:

- For more details on this statement, see Using DISPLAY Action.

Using Global TX

The global `TX` defines the beginning, end, or additional attributes of a transaction.

Syntax:

```
IF (<Expression>)
   <label>: TX(<Arguments>);
```

For more details on this statement, see Using TX Statement.

TX shares hardware resources with DISPLAY.

Related Sections:

For additional information on `<expression>` and maximal number of statements see the Using Global DISPLAY section.

Defining INSTANCE Statement

The `INSTANCE` statement defines the beginning of a new SDL instance, and assigns a name to the instance. The name is used for identification by XEL, but other than that is not used inside the SDL program itself.

Syntax:

```
instance <InstanceName>;
```

`<InstanceName>` is a string made of letters, digits or underscores and must start with a letter or underscore.

Related Sections:

See Defining INSTANCE Statement for information on how to use more than one instance.

Defining STATE Statement

The `STATE` statement declares the start of a state definition. Each state performs actions based on conditions and tests.

You can declare up to 64 states. The first state in the SDL program is the starting state. Only one state is active in any given cycle.

Syntax:

```
STATE <StateName>
{
    <StateBody>
}
```

where <StateName> is a string, and *<StateBody>* is a set of `IF-ELSE` statements or Action statements. The *<StateName>* is a string made of letters, digits or underscores and must start with a letter or underscore.

The following actions can be called from inside a STATE statement:

- Using GOTO Action
- Using LOAD Action
- Using DECREMENT Action
- Using INCREMENT Action
- Using START
- Using STOP Action
- Using TRIGGER Action
- Using ACQUIRE Action
- Using NO_ACQUIRE Action
- Using EXEC Action
- Using DISPLAY Action
- Using TX Statement
- Using TRACE_TSM Action

Related Sections:

Specifying Global Definitions in SDL Programs

Defining IF - ELSE statement

A basic `IF` statement evaluates an expression. If the expression is `TRUE`, then it executes a statement, or a set of statements. An `ELSE` statement can also be specified to execute a different set of statements if the expression evaluated to FALSE.

Syntax:

```
IF (<Expression>)
    <Statement>
```

or:

```
IF (<Expression>)
{
    <Statement>
    <Statement>
     . . .
}
```

An `ELSE` statement can immediately follow an IF statement.

Syntax:

```
ELSE
    <Statement>
```

or:

```
ELSE
{
    <Statement>
    <Statement>
     . . .
}
```

`<Expression>` can be scalar or vector. A scalar expression is considered `TRUE` if it has the value 1. A vector expression is considered `TRUE` if any of its bits is 1.

`<Statement>` can be another `IF-ELSE` statement or any `Action` statement (See SDL Program Examples). There are no restrictions on depth of nesting of `IF-ELSE` statements, although there are limitations on the maximal number of `IF` statements that can be used in each state.

When an `IF` statement is the only statement nested inside an `ELSE` block, then no braces are needed for the `ELSE` block, and the nesting takes the well known structure of `IF-ELSE-IF` chain. A nested IF chain can be written as follows:

```
IF (<expression1>)
    <action1>
ELSE IF (<expression2>)
    <action2>
ELSE IF (<expression3>)
    <action3>
ELSE
    <action4>
```

The expressions are evaluated from top to bottom, and the action attached to the first expression that is `true` is executed.

The following truth table shows how SDL interprets the above example:

<expression1>	<expression2>	<expression3>	Action Executed
true	don’t care	don’t care	<action1>
false	true	don’t care	<action2>
false	false	true	<action3>
false	false	false	<action4>

A state can contain more than one chain of `IF-ELSE-IF` constructs. Independent `IF` statements are executed concurrently. If there is a possibility for a conflict between actions from different `IF` statements within the same state (for example, `GOTO` actions to different destinations), then the SDL compiler will issue a warning message, and the first action from the top will be executed. Obvious cases, in which one of the actions has no chance of ever getting executed will be flagged as errors.

Limitations Related to Expressions and IF Statements

SDL has an inherent limit on the maximal number of IF statements in each state, which is mostly unrelated to how these IF statements are nested. The limit depends on the presence of counter conditions in the expressions inside the IF statements. Without counter conditions, the rule is simple: there can be no more than 8 expressions (or IF statements) in each state. There is no limit on the number of expressions constructed exclusively of counter conditions. If an expression contains both counter conditions and signal names, the SDL compiler tries to convert the expression into several sub-expressions that are either counter conditions, or expressions based on signal names. The resulting number of sub-expressions based only on signal names add up towards the maximal limit of 8. These restrictions do not apply to IF statements in the global or instance definition sections, and it also does not apply to IF statements that do not control directly any actions. Example:

```
    IF (exp1) {
        IF(exp2) { GOTO nxt; increment counter1;}
    }
```

In the above example, `IF(exp1)` does not control any action directly, therefore `exp1` is not counted towards the maximum of 8 expressions.

The SDL compiler might also fail to download complex expressions into the emulator due to lack of sufficient emulation resources to emulate these expressions. When this happens, an error message might provide some suggestions for working around the failure. One of the possible suggestions is to increase the step count per FCLK cycle. To change the step count, see description of the `extra_steps` option of the `compilerOption` XEL command in the *User Data Commands* chapter of the *WXE Command Reference Manual*.

Related Sections:

Specifying Global Definitions in SDL Programs

Using SDL Actions

The following actions can be called from inside a STATE statement:

Using GOTO Action

The `GOTO` action transfers execution from the current state to the specified state, in the next cycle.

Syntax:

```
GOTO <StateName>;
```

where *<StateName>* is the name of a state.

Related Topic:

Using SDL Actions

Using LOAD Action

The `LOAD` action loads one of the counters with an initial value. Each `LOAD` action can use a different load value. The `LOAD` action must be executed before executing a `START` or `DECREMENT` actions to guarantee that the counter starts from a known value.

Syntax:

```
LOAD COUNTER1 <value>;
```

or:

```
LOAD COUNTER2 <value>;
```

`<value>` is an integer value between 0 and (2^{40}-1). The value can be specified as a decimal number or in a different radix (see Using Numeric Constants in SDL Expressions).

If *`<value>`* is not specified, SDL uses a default value as explained in Specifying the Counter Load Value.

The counter is loaded with the new value on the cycle that immediately follows the LOAD action.

If SDL previously executed a `START` action, and did not execute a `STOP` action, or if a START action was executed concurrently with the Load action, then the counter is first loaded with the new value, and begins decrementing in the cycle after that.

For example, if at cycle 100 the following two actions were executed:

```
START COUNTER1
LOAD COUNTER1 20
```

then counter1 will show the value 20 in cycle 101, and the value 19 in cycle 102.

Related Topic:

Using SDL Actions

Using DECREMENT Action

The `DECREMENT` action decrements the specified counter by 1. This is done unconditionally regardless of the Specifying Count Enable Definition. This statement is meaningful only when the counter is in a stopped state. If a counter is auto-decrementing because a previous START statement defined it as free running and its enable control is true, the DECREMENT statement has no effect on the counter.

Syntax:

```
DECREMENT COUNTER1 ;
```

or:

```
DECREMENT COUNTER2 ;
```

When the counter reaches 0, decrementing stops.

The keywords `DECR` and `COUNT` are recognized as synonyms for `DECREMENT`.

Related Topic:

Using SDL Actions

Using INCREMENT Action

The `INCREMENT` action increments the specified counter by 1 unconditionally. If a counter is auto-decrementing because a previous START statement defined it as free running, the INCREMENT statement overrides the default auto-decrementing and causes the counter to increment instead of decrementing in that cycle.

Syntax:

```
INCREMENT COUNTER1 ;
```

or:

```
INCREMENT COUNTER2 ;
```

When the counter reaches its maximum value of (2^{40}-1), incrementing stops.

The keyword `INC` is recognized as a synonym for `INCREMENT`.

Related Topic:

Using SDL Actions

Using START

The `START` action puts the counter in auto-decrement mode. In this mode, the counter decrements by 1 each time when the expression represented by the Specifying Count Enable Definition is true. If no count enable is specified, then the counter is decremented once every cycle. When the counter reaches 0, decrementing stops, but only the Using STOP Action action takes the counter out of the auto-decrement mode.

Syntax:

```
START COUNTER1 ;
```

or:

```
START COUNTER2 ;
```

Related Topic:

Using SDL Actions

Using STOP Action

The `STOP` action stops the specified counter from auto-decrementing.

Syntax:

```
STOP COUNTER1 ;
```

or:

```
STOP COUNTER2 ;
```

Related Topic:

Using SDL Actions

Using TRIGGER Action

The effects of the `TRIGGER` action depends on the operation mode.

When running in LA mode, or in Dynamic Target mode the `TRIGGER` action causes probed data tracing to stop after the specified amount of post trigger samples are captured into the trace buffer. After trigger the SDL program continues to run, so the `NO_ACQUIRE` action can still control what data samples should be traced into memory (if the number of requested post-trigger samples is non zero).

When running in SA, Vector Debug, or STB modes, the trigger cause the design running on the emulator to stop (after the number of requested post-trigger samples is captured into the trace buffer).

The default behavior of `trigger` can be modified from XEL through the `sdl -haltOnTrigger` and `clockConfig -stopMode` commands. (For example, make Trigger stop the design clocks in LA mode).

Syntax:

```
<Label>: TRIGGER;
```

`<Label>` is a string made of letters, digits or underscores and must start with a letter or underscore. It can be used by XEL for the purpose of identifying, enabling or disabling any `TRIGGER` statements that have a specific label. *`<Label>`* does not have to be unique within the SDL program. The construct *`<Label>`*`:` is optional. It can be omitted if labeling the statement is not needed.

The effect of the `TRIGGER` action is independent of other TRIGGER statements. Each can cause a trigger independently.

When running in SA or STB mode, the SDL examines design signals with some delay. For precision debugging, you can eliminate the delay by running at a lower speed. See Choosing Fast or Accurate Behavior for more details.

Related Topic:

Using SDL Actions

Using ACQUIRE Action

The `ACTION` action enables probed data sampling in the current cycle. Use this action to more-selectively fill the trace buffer.

Syntax:

```
ACQUIRE ;
```

When using the `ACQUIRE` action, only cycles in which an `ACQUIRE` action (or a global acquire) is executed will capture a probe sample.

The use of the `ACQUIRE` action precludes the use of the `NO_ACQUIRE` action or the global `NO_ACQUIRE` definition.

Related Topic:

Using NO_ACQUIRE Action

Using NO_ACQUIRE Action

The `NO_ACQUIRE` action temporarily prevents probed data sampling in the current cycle. Use this command to more-selectively fill the trace buffer.

Syntax:

```
NO_ACQUIRE ;
```

The effect of this action is independent of the Defining Global NO_ACQUIRE. Each can prevent data sampling independently.

The use of `NO_ACQUIRE` precludes the use of the `ACQUIRE` action or the global `ACQUIRE` definition.

Related Topic:

Using SDL Actions

Using EXEC Action

The `EXEC` action causes execution of a Tcl or XEL command, as if it was executed from the `XE` command line prompt. In emulation modes that enable stopping the clocks, emulation clocks are suspended while the command executes. This feature is available in STB and LA modes and also under IXCOM with either static or dynamic target. It is not available when running in infiniTrace or VD modes.

Syntax:

```
<Label>: EXEC ”<command>”;
```

`<Label>` is a string made of letters, digits or underscores and must start with a letter or underscore. It can be used by XEL for the purpose of identifying, enabling or disabling any `EXEC` statements that have a specific label. *`<Label>`* does not have to be unique within the SDL program. The construct *`<Label>`*`:` is optional. It can be omitted if labeling the statement is not needed.

`<command>` represents almost any string that can be executed by the XEL command interpreter during a debug session, including calls to Tcl procedures defined earlier, or even a sequence of commands separated by semicolons.

For example, suppose that `my_proc` is a Tcl procedure that takes a single numeric argument and was already defined. The following prints a short message, then calls the procedure with an argument 11.

```
EXEC ”puts {executing my_proc} ; my_proc 11”;
```

The following restrictions apply:

The whole command string needs to be enclosed in double quotes. So, it cannot include any double quotes unless they are properly escaped with a ‘\’ (Following normal Tcl escape rules). For example:

```
EXEC ”puts \”time now is: [clock format [clock seconds] -format %c]\””;
```

Any Tcl variable reference that appears inside the `<command>` string (with the syntax `$var or ${var}`) is substituted with its value at the time when the `EXEC` action is executed. For example consider the following snippet from an SDL program:

```
IF (signalx == 1)
     EXEC ”puts \” xyz = $xyz\”; incr xyz”;
```

If the Tcl variable `xyz` was initialized before the `run` command to the value 5 and the `EXEC` action is executed 3 times during the run, then the following output is produced:

```
5
6
7
```

`EXEC` actions are always executed at global Tcl call level. In other words, variables local to a Tcl proc are not accessible to `EXEC`.

Some XEL commands normally allowed in a debug session cannot be executed by an `EXEC`, whether they are called directly from the `EXEC` statement, or indirectly from a Tcl procedure called from the `EXEC` statement. The list of XEL commands that are not allowed to be executed from an `EXEC` is:

- `configPM`
- `debug`
- `download`
- `host`
- All `infiniTrace` command options except:

  `-prepare -on -off -close`
- `restart`
- `reset`
- `run`
- `save`
- `vector`
- `waitWhileBusy`
- `xc`
- The following options of the `sdl` command:
  - `-setFile -traceOn -verify -expression -restoreState -load -onlyReport`

`EXEC` requests are serviced whether running in blocking or non-blocking mode.

When running in STB mode, SDL examines design signals with a delay of up to 2 Fast Clock (FCLK) cycles. This means that any `EXEC` based on values of design signals will be delayed by up to two FCLK cycles.

For precision debugging, you can eliminate the delay by running at 1/3 of the regular speed. See Choosing Fast or Accurate Behavior for more details.

When running in STB mode, each `EXEC` request cause FCLK to pause and the software service the request before any other `EXEC` requests are issued. For all other operation modes SDL maintains a small buffer for holding `EXEC` requests until the software can service them. The buffer can hold `EXEC` requests from up to `4` separate FCLK cycles.

When running in emulation modes that do not enable to stop the clocks, such as LA mode, Dynamic Target mode under IXCOM, or with uncontrolled clocks under IXCOM, clocks are not suspended while `EXEC` requests are executed, and the execution delay of the `EXEC` request can be thousands of cycles. Moreover, since the clocks do not stop, it is possible that SDL issues a new `EXEC` request before all the previous `EXEC` requests in the `EXEC` buffer are serviced, and in such case, the new `EXEC` request is lost. This is called an `EXEC` overflow condition.

The system buffers up to four EXEC requests at a time. To avoid EXEC overflow, ensure that the EXEC requests are adequately spaced in time and a new EXEC is not issued before the previous ones finish execution, otherwise, an error message will be generated.

SDL can determine if two or more pending EXEC requests are waiting in the buffer by checking the value of the `EXEC_BUSY` signal. A high value for this signal indicates that a new EXEC command will overflow the buffer. The following example shows the usage of the `EXEC_BUSY` signal in SDL expressions in order to avoid EXEC overflow:

```
if(cond && !EXEC_BUSY)
EXEC "my_tcl_command";
```

In other co-simulation modes under IXCOM, the design is paused to execute the `EXEC` action with possibly some delay. The amount of delay depends on the design itself and the selected run mode. If you executed the `xeset stopDelay 0` command earlier, then the `EXEC` command will be scheduled for execution at the end of the simulation time stamp from which it was invoked. If the `stopDelay` parameter was selected as `1`, then the `EXEC` will be scheduled for execution either at the end of the time stamp from which it was issued, or at the end of the following time stamp. If more than one `EXEC` actions are scheduled for execution in the same simulation time stamp, but issued from different FCLK cycles, then these `EXEC` requests might cause an `EXEC` overflow condition if they overflow the buffer size of `4` stages. One way to avoid an overflow is to make sure that SDL is evaluated no more than once per each simulation time stamp. This can be done with the following XEL command:

```
    sdl -simTime 1
```

By default, an `EXEC` overflow condition will stop the run with an error message; but this behavior can be modified from XEL (with the `sdl -execOvf` command) to either ignore the condition altogether or only produce a warning message without stopping the run.

Related Topic:

Using SDL Actions

Using DISPLAY Action

The `DISPLAY` action prints a message to the console and/or a file, using a syntax similar to the Verilog `$display` system task.

Syntax:

```
<Label>: DISPLAY (F=<ID>, L=<Level>, "<format-specifier>", <Arg>, <Arg>, ...);
```

Any of the components in the above syntax are optional, except the keyword `DISPLAY` and the parenthesis. If `<Label>:, F=<ID>, L=<Level>` are taken out, the above line takes the familiar form of the `$display()` statement from Verilog.

`<Arg>, <Arg>, ...` are optional arguments to be printed. Each is an expression as explained in Using SDL Expressions, except that they cannot contain SDL counter conditions. However, the counter name (counter1 or counter2) might be used as an argument by itself. In such case the integer value of the counter is printed, with a delay of up to 2 FCLK cycles.

Note: Values of arguments printed by `DISPLAY` statement are affected by 4-state flow.

`<Label>` is a string made of letters, digits or underscores and must start with a letter or underscore. It can be used by XEL for the purpose of identifying, enabling or disabling any , `DISPLAY` statements that have a specific label. `<Label>` does not have to be unique within the SDL program. It is also inserted as part of the output message if <*format-specifier*> is missing. The construct `<Label>:` is optional. It can be omitted if labeling the statement is not needed.

`F=<`*ID*`>` is an optional argument. <*ID*> is an identifier that enables XEL to route the message to a specific file associated with this identifier.

The file associated with <*ID*> is opened by the following XEL command:

```
sdl -display -id <ID> -file <name>
```

When the value of `<ID>` is the reserved string `MARKER`, the DISPLAY statement is used for creating time markers for viewing in the VERDI waveform browser or for use as bookmarks in InfiniTrace. For information on how to create time markers and bookmarks, refer to the Creating Time Markers and Bookmarks section.

`L=<`*Level*`>` is an optional argument. <*Level*> is an integer value that enables the messages to be enabled or disabled from XEL based on a single threshold value. The command `sdl -display -loglevel` <*loglevel*> specifies a single integer value

<*loglevel*>. The output from the `DISPLAY` statement is suppressed if <*level*> is less than <*loglevel*>. If <*level*> is not specified, then a default value is assumed. The default value is defined by the XEL command `sdl -display -deflevel` <*deflevel*>.

<*format-specifier*> is a string containing the main text of the message. It might possibly contain escape sequences, which are substituted with other strings according to Escape Sequences in the Format Specifier of a Display Statement. The sequences `%h %d %i %o %b %s %[<enum-type>]` and the arguments *<Arg>, <Arg>, ...* are matched in sequence. The <*format-specifier*> is an optional argument. If not specified, the output message takes a standard form as specified below.

The format specifier might also contain special characters using a leading backslash (or other combinations), as shown in the table. A backslash followed by any other character will be ignored and removed from the output string.

Tcl variable substitution using `$`*name* and `${`*name*`}` is also supported inside the format specifier. The Tcl variable value substitution is performed at the beginning of each run.

Table 75-1 Escape Sequences in the Format Specifier of a Display Statement

Escape Sequence	Explanation
`%h` or `%H`	Convert one *<arg>* using unsigned hexadecimal conversion.
`%d` or `%D`	Convert one *<arg>* using unsigned decimal conversion
`%i` or `%I`	Convert one *<arg>* using signed decimal conversion
`%o` or `%O`	Convert one *<arg>* using unsigned octal conversion
`%b` or `%B`	Convert one *<arg>* using unsigned binary conversion
`%c` or `%C`	Convert one *<arg>* using ASCII character conversion
`%s` or `%S`	*<arg>* is a design signal: Convert *<arg>* using ASCII conversion *<arg>* is a UPF or assertion object: Convert *<arg>* to mnemonic.
`%n` or `%N`	Name of SDL instance
`%t` or `%T`	Simulated time
$<var>	Insert the value of a global TCL variable

Escape Sequence	**Explanation**
`${<var>}`	Insert the value of a global TCL variable
`%[<enum_type>]`	Convert a numeric value to a string using `enum`
`%[...]`	Use a TCL command to perform complex conversions
`\n`	Newline
`\t`	Tab
`\\`	The character `\`
`\"`	The character `"`
`\$`	The character `$`
`%%`	The character `%`

Note: Printout of `DISPLAY` statement that are design nets have been enhanced to show the character `X` when a net value is unknown and `Z` when the net is undriven. This affects all display formats. See "Printout of SDL Using the DISPLAY Statement" section in *Working with 4-State Logic in Palladium guide* for usage details.

The format specifier character can be either lower or upper case. The only difference between lower case and upper case is when using a hexadecimal conversion. The resulting string from `%h` will be constructed with only lower case characters, while the resulting string from `%H` will be constructed with only upper case characters.

The field size (number of characters) allocated when substituting an argument is set to the maximal possible number of characters that the argument might require. For example, a `%b` conversion for an `8` bit argument will take `8` characters, while a `%d` conversion will take `3` characters (because the maximum decimal value of an `8` bits argument is `255`). A `%i` conversion will take `4` characters because it also needs to account for a sign character.

The value is right justified in the field. Binary, octal and hexadecimal conversions are padded with zeros on the left, while decimal values are padded with spaces. The padding can be eliminated and the result forced into the minimum required field by inserting the digit `0` immediately to the right of the `%`.

For example, suppose that the `8` bit argument `data[7:0]` represents the decimal value `12` (binary `1100`):

```
DISPLAY ("DATA=:%d:%0d::%b:%0b:", data[7:0],data[7:0],data[7:0],data[7:0]);
```

This produces the following output:

```
DATA=: 12:12::00001100:1100:
```

Note that the conversion rules are similar to those used by the Verilog `$display()` system task, except that `%t` is not matched with any argument, and Verilog does not support `%i` conversion.

`%c` prints out a single character representing the ASCII code of the low-order byte of the argument, regardless of the number of bits in the input argument.

`%s` can operate in one of two ways depending on the argument type:

If the corresponding argument is a multi-bit bus composed of regular design nets, then `%s` prints out one or more characters representing the ASCII code of the input argument. For example, if the argument is `ABC[31:0]`, the output string will have four characters with the first one representing the ASCII value of `ABC[31:24]` and the last one `ABC[7:0]`.

If the corresponding argument is the name of a UPF or assertion object, then `%s` prints the value of that object as a mnemonic string. (For example, `failed` is one of several possible mnemonic values for an assertion).

The list of possible mnemonics for UPF objects is specified at: Using IEEE 1801 Conditions in SDL Expressions. In addition, voltage of supply nets or supply ports can be printed with the `%s` specifier by appending `@voltage` to the name of an object. For example:

```
    display(”voltage of supply net xyz is %s”,xyz@voltage);
```

Depending on the object type, the printout from a single `%s` specifier might show more than one mnemonic. For example, both voltage and state are printed for supply nets with the `@voltage` specifier. For retention and isolation rules, the displayed string contains the state of the rule's supply and whether the rule is active.

`%n` prints the name of the SDL instance from which the `DISPLAY` statement is executed. It is not paired with an argument.

`%t` is simulated time, which is formatted using the default time units that were set by the `xeset timeUnit` XEL command. It is not paired with an argument.

The printed value of *`%t`* contains both a floating point number and time units.

For example: *`25.3 ns`*

The precision (number of digits in the fractional part of the printed value) can be specified by a decimal point followed by the number of digits. For example:

```
display(”%8.3t”);
```

This will print exactly 3 digits to the right of the decimal point, and the total number of characters printed will be no less than 8.

If a precision is not specified, the printed value will contain maximum accuracy and only trailing zeros are truncated.

Getting Live Feed Of Display Messages

The `DISPLAY` feature enables you to route different messages to different files based on the `F=<ID>` argument in the `display` statement. This enables to get a copy of a select set of display messages, without any other messages.

To see these messages scrolling on the screen at run time, open a separate shell window with the Unix command `xterm`, and in that window, execute the following Unix command:

```
tail -f <filename>
```

This will show you a live feed of the DISPLAY messages that are written to that file.

Formatting with Enum

The enumeration type conversion `%`**[**`<enum_type>`**]** provides an efficient way to chose a string out of a list of several strings, based on an integer value. The integer value must be unsigned with upto 64 bits.

A format conversion using a TCL procedure can also do the same job (for example, with the generic TCL statement `lindex`). However, it is expected to be slower because it has to be processed by the TCL interpreter. Therefore, if a conversion can be done using `%`**[**`<enum_type>`**]**, then it is preferable over using a TCL procedure conversion when processing time has a potential to slow down the run.

The syntax of the enumeration conversion is:

```
%[<enum_type>]
```

The square brackets above are part of the syntax. The `<enum_type>` is an identifier of the enumeration type. It must follow SDL lexical requirements for an identifier. It is defined by an enumeration definition as follows:

```
    enum <enum_type>
    {
       <string> [=<value>],
       <string> [=<value>],
       ...
    };
```

The <*string*> is either an identifier or an arbitrary string enclosed in double quotes.

The <*value*> must be an unsigned integer value, upto 64 bits.

A value that is not covered by the enumeration definition will be formatted as a decimal integer. The enumeration definition might appear anywhere inside the SDL program, except inside state statements.

The following example defines weekday as <*enum_type*>:

```
enum weekday {monday, tuesday, wednesday=3, thursday, friday };
```

In this example, the enumeration values of weekday are defined as:

```
monday=0
tuesday=1
wednesday=3
thursday=4
friday=5
```

Suppose, this enumeration is used in the following statement:

```
display ("%[weekday]", day[2:0]);
```

If `day[2:0]` has the value `3` then the attribute value will be formatted into the string "wednesday", but the value 2 will be formatted into the string "2".

Complex Formatting Using the TCL Programming Language

`%[...]` enables use of the full programming power of TCL scripting to further format the results, and even add information such as time of day. The string inside the square brackets is first formatted by performing the regular conversions on the specifiers: `%h`, `%d`, `%i`, `%o`, `%b`, `%c`, `%s`, and the resulting string is evaluated as a TCL command. The entire string starting from the `%[` and ending with the `]` is then substituted with the result returned by the TCL command. The command is typically a TCL proc that you have defined, but you can also use generic TCL commands.

The following example prints the arithmetic sum of `A[7:0]` and `B[7:0]`:

```
DISPLAY ("A+B=%[expr %d + %d]",A[7:0], B[7:0]);
```

With this example, if `A[7:0]` represents the value `12` and `B[7:0]` represents the value `7`, then the above statement will produce the following message:

```
A+B=19
```

XEL commands are not allowed to be called from a `%[]` specifier.

Controlling Field Size and Alignment

By default, the field size (number of characters) allocated when substituting an argument is set to the maximal possible number of characters that the argument might require. For example, a `%b` conversion for an 8 bit argument will take 8 characters, while a `%d` conversion will take 3 characters (because the maximum decimal value of an 8 bits argument is 255). A `%i` conversion will take 4 characters because it also needs to account for a sign character.

The value is right justified in the field. Binary, octal and hexadecimal conversions are padded with zeros on the left, while decimal values are padded with spaces. This padding can be eliminated and the result forced into the minimum required field by inserting a `0` after the `%`.

Additional padding with spaces (regardless of conversion size) can be added by specifying a minimum field size as a decimal number that follows the `%`. If preceded by a `-` character, the value will be left justified (padded with spaces on the right). Otherwise, it is right justified. This applies to all the escape sequences that start with a `%`.

The complete escape sequence from left to right is as follows:

1. The character `%`
2. Optional `-` (specifies that the result should be left justified)
3. Optional `0` (specifies that any leading zeros or spaces from a numeric value should be stripped off)
4. Optional minimum field size (decimal number)
5. The format specifier character or sequence from Escape Sequences in the Format Specifier of a Display Statement.

For example, suppose that the 8 bit argument `data[7:0]` represents the decimal value `12` (binary `1100`). This will produce the following output depending on the format specifier as follows:

- `%b` print the string `00001100` in an 8 character field
- `%0b` print the string `1100` in a 4 character field
- `%9b` print the string `00001100` right justified in a 9 character field
- `%-9b` print the string `00001100` left justified in a 9 character field
- `%09b` print the string `1100` right justified in a 9 character field
- `%-09b` print the string `1100` left justified in a 9 character field

The following example shows a complete display statement:

```
DISPLAY ("DATA=:%d:%0d::%9b:%-06b:",
        data[7:0],data[7:0],data[7:0],data[7:0]);
```

This will produce the following output:

```
DATA=: 12:12:: 00001100:1100 :
```

Compatibility of Format Specifier in SDL with Verilog Syntax

The conversion rules of the format specifier are similar to those used by the Verilog `$display()` system task, with the following exceptions:

- In SDL, `%t` is not matched with any argument. In Verilog it must be matched with the `$time` argument.
- Verilog does not support the `%i` specifier (signed integer).
- Verilog does not support the `%[<tcl_command>]` and `%[<enum_type>]` specifiers.
- Verilog does not support field size and field alignment modifiers, although it supports the '0' modifier (no padding).

Default Formatting when a <format_specifier> is Missing

If the <*format_specifier*> argument is an empty string or missing altogether, each generated message is written into a single line of text using a standard format that contains the following fields (space-separated):

<time_stamp> <level> <label> <arg1> <arg2> ...

<*time_stamp*> is written as a plain floating point number (without the time-unit string) representing the simulation time in nanoseconds.

<*level*> is written as an integer decimal value.

<*label*> is the label of the statement, if defined, or the single character * (asterisk) if a label was not defined.

<*arg1*> <*arg2*> ...are space-separated argument values formatted in hexadecimal.

Overflow Condition in SDL

Messages from `DISPLAY` statements are buffered in the memory of the emulator. While the emulator is running, the buffer contents are dumped into the workstation several times each second, formatted and sent to the console and/or a file. In emulation modes that enable stopping the clocks, the emulator clock will pause if the buffer becomes full before the workstation has a chance to read it. In LA or Dynamic Target mode, if the rate of writing into the buffer is higher than the rate at which the workstation reads it, then some messages are lost. This is called an overflow condition. You can increase the depth of the buffer during compilation using compiler option `sdlDisplayDepth` to address the overflow issue.

The total number of different arguments (*`<Arg>, <Arg>, ...`*) that can be used in all `DISPLAY` statements depends on the width of the display buffer in the hardware. If the same signal appears as an argument in more than one `DISPLAY` statement in the SDL program, that signal is counted only once for the purpose of calculating the required buffer width. The width of the display buffer can be increased at compile time using compiler option `sdlDisplayWidth`.

Several aspects of the `DISPLAY` facility can be controlled from XEL through the `sdl` command. For example, you can direct the messages to a file instead of (or in addition to) the console.

Using TX Statement

The `TX` statement is used for recording transactions into an SST2 database. The `TX` statement can define a transaction's beginning, ending, or additional attributes (such as, signal values associated with the transaction).

Table 76-1 TX Terminology

Term	Description
Stream and Stream Name	A *Stream* is a row on the waveform display, on which transactions are displayed. A single stream can show multiple transactions. A stream is identified by a name and a hierarchical scope.
Transaction	A *Transaction* typically has a beginning time and an ending time, which are paired by a unique identifier called a *Transaction ID*. Some *Transactions*, called *Event* or *Error* have zero duration, which means that their beginning and ending happen at the same time point. A transaction is considered as open only during the time period between its beginning and its ending.
	The *Transaction ID* is a user given name. It is used for matching a specific open transaction with different `TX` statements that specify various attributes of the same transaction, but triggered at different simulation time. The same *Transaction ID* can be used for several transactions, as long as they do not have overlapping time. A specific Transaction ID cannot belong to more than one stream or one scope. If the stream scope or stream name are not defined in the `TX` statement, they are inherited from another `TX` definition with same Transaction ID. If none exist then they get default stream and scope name.
Transaction Type	A *Transaction Type* is a user given name that identifies the general properties of the transaction.

Table 76-1 TX Terminology

Term	Description
Transaction's Attribute	Each *Transaction* is associated with several distinct pieces of information called *Attributes*. Each *Attribute* is made of two components: *Attribute Name* and *Attribute Value*, both of which are strings. There are several types of attributes: ■ *Attributes* that are pre-defined (for example, the transaction's beginning time). ■ *Named Attributes* are defined by the user as arguments inside the `TX` statement using the syntax: `<name> = <value>`. *Named Attributes* can only be defined at the beginning of the transaction. ■ *Signal Attributes* are also defined as arguments inside the `TX` statement, and show design signal values, optionally with additional formatting instructions. *Signal Attributes* can be defined at any time while the transaction is open.

Syntax of the TX Statement

The `TX` statement can be used anywhere inside the SDL program to record into the SST2 database the beginning or the ending of a transaction, or append additional data (Signal values) to an open transaction.

A complete transaction is defined by either a single `TX` statement with an *Event* or *Error* specifier, or by two or more `TX` statements, one with the `begin` specifier, followed by another with the `end` specifier with the same *Transaction ID*, and possibly additional `TX` statements in between with the `set` specifier.

Attributes can be defined at any point, either at the beginning, the ending or while the transaction is still open. However, *Named Attributes* can only be defined at the beginning of the transaction (only in the `TX` statements with the `begin`, `event`, or `error` specifier).

```
    [<label>:] tx (<specifier>, <tx_id> [, <arg>, <arg>, ...]);
```

The `<label>` is an optional SDL label that can be used for identifying or disabling the statement at the XEL command level. The same syntactical and semantical rules are followed as the labels used for the `trigger`, `exec`, or `display` statements.

- `<specifier>`: Specifies the type of action represented by this statement:
  - `begin`: Indicates the beginning of a transaction.
  - `end`: Indicates the ending of the transaction.
  - `event`: Indicates an *Event* transaction. The `TX` statement defines both the beginning and ending.
  - `error`: Indicates an *Error* transaction. The `TX` statement defines both the beginning and ending.
  - `set`: Sets additional attributes values for a transaction that is already open.
- `<tx_id>`: The *Transaction ID* identifies a specific transaction. This is a user given name that must follow SDL lexical requirements for an identifier.

The `<specifier>` and `<tx_id>` are always mandatory.

The arguments that follow (each of the `<arg>`) are optional. Each is a *Named Attribute*, *Signal Attribute*, or *Transaction Attributes Definition* argument. The *Named Attributes* that are not defined will assume default values. The default values are discussed in the following sections:

- Using Named Attributes in TX Statement
- Using Signal Attributes in TX Statement
- Using Transaction Attributes Definition in TX Statement

Using Named Attributes in TX Statement

The Named Attributes are as follows.

Table 77-1 Named Attributes

Attribute, Description, and Default Value
`type = <type>`
The *Transaction Type* is a user given name that follows SDL lexical requirements for an identifier. It is used for identifying the type of the transaction in messages and in the SimVision browsers. If not specified, it will be set to the value specified by `<tx_id>`.
`stream = "<stream_name>"`
The name of the stream as it appears on the waveform display. If not specified, it will be copied from another `TX` statement with same *Transaction ID*. If none exists, then it will assume the default string `SDL_DEFAULT_TX`.
`scope = "<stream_scope>"`
Design scope of the stream as it appears on the waveform display. If not specified, it will be copied from another `TX` statement with same *Transaction ID*. If none exists, then it will assume the default string `SDL_TX_SCOPE`.
`title = "<title>"`
In SDI terminology (and on the SimVision TX Explorer window) this is called *Label*. A different word is used here because the term label is already used for other purposes in SDL. The `<title>` is a very short description of the transaction. If not defined, the value will be set to `<type>`.
`description = "<description>"`
An arbitrarily long description for this transaction. If not defined, it will be set to an empty string.

Any of the keywords `type`, `stream`, `scope`, `title`, or `description` can be shortened to use just the first few characters. For example, `st` can be used instead of `stream`, and `ti` can be used instead of `title`.

The double quotes can be omitted for the values of `<stream_name>`, `<stream_scope>`, `<title>`, or `<description>` if they conform to the definition of a legal SDL identifier.

Named Attributes can only be defined with specifiers: `begin`, `event`, or `error`.

Related Sections:

- Using TX Statement
- Using Signal Attributes in TX Statement

Using Signal Attributes in TX Statement

The Signal Attributes are as follows.

Table 78-1 Signal Attributes

Attribute, Description, and Default Value
`<signal_name>` This is a transaction attribute that shows a signal value. The *`<signal_name>`* is the full hierarchical name of a design signal. The signal name serves as the attribute name. The signal name along with its value is displayed on the screen in hexadecimal format as an attribute of the transaction.
`{"`*`<format>`*`", `*`<signal_name>`*`}` This is a transaction attribute that shows a signal value. The *`<signal_name>`* is the full hierarchical name of a design signal. The signal name serves as the attribute name. The signal name along with its value is displayed on the screen as an attribute of the transaction. The *`<format>`* is a string that specifies how to format the value of the signal into a string.
`{"`*`<attribute_name>`*`", "`*`<format>`*`", `*`<expression>`*`, `*`<expression>`*`, ...}` This is a transaction attribute that shows a string value that was constructed from zero or more expressions. The *`<attribute_name>`* is the name of the attribute to be displayed in SimVision. The *`<expression>`*`, `*`<expression>`*`, ...` specifies zero or more legal SDL expressions. The *`<format>`* is a string that specifies how to format the expressions into a single string.

Note: 4-state flow affects signal attributes. It shows formatted printout of signal attributes in XR statement.

Related Sections:

- Using TX Statement
- Using Named Attributes in TX Statement

Using Transaction Attributes Definition in TX Statement

The Transaction Attributes Definition is as follows:

`tdef = <`*`tdef_id`*`>:` This argument enables to define a set of attributes through a separate statement and use it inside the `TX` statement. See *Defining Attributes with a tdef Statement* for more details.

Any number of *Attribute* arguments might appear in a single `TX` statement. However, the `type`, `stream`, `scope`, `title`, and `description` arguments might appear only once each, and only if the `<`*`specifier`*`>` is `begin`, `error`, or `event`.

Formatting of Attribute Values

Formatting in the `TX` statement (the `"<`*`format`*`>"` string) follows the same syntax and semantics as the format string in SDL `display()` statements. For example, `"%d"` will format the argument using unsigned decimal conversion.

Defining Attributes with a tdef Statement

Default values can be defined for any set of attributes in any `TX` statement using the `tdef` construct. Either *Named Attributes* or *Signal Attributes* can be defined with `tdef`. The `tdef` statement must appear above any `TX` statement where it is being used, and outside any `STATE` statement. If defined inside an `INSTANCE` statement, then its scope is limited to that instance.

The syntax of the `tdef` statement is as follows:

```
tdef <tdef_id>
{
   <arg>,
   <arg>,
   ...
};
```

The `<`*`tdef_id`*`>` is an identifier, which must follow SDL syntax requirements for an identifier.

The `<arg>` is same as defined for the `TX` statement, and might be any *Named Attribute* or *Signal Attribute*. Any argument defined here will be used in all `TX` actions that contain the `tdef = <tdef_id>` argument.

Related Sections:

Using TX Statement

Linking Transactions Using tlink Statement

The `tlink` statement specifies how transactions should be linked using a specified attribute from each transaction as the link tag.

The syntax of the tlink statement is as follows:

```
tlink <tx-id1> <attribute-name1> <tx-ID2> <attribute-name2>;
```

The *`<tx-id1>`* is the transaction ID of the first transaction to be linked.

The *`<attribute-name1>`* is the Attribute Name from the first transaction that is used as the link tag.

The *`<tx-ID2>`* is the transaction ID of the second transaction to be linked.

The *`<attribute-name2>`* is the Attribute Name from the second transaction that is used as the link tag.

For attributes that look like *`<signal-name>`* or `{"`*`<format>`*`", `*`<signal-name>`*`}`, the Attribute Name is the *`<signal-name>`* component.

For attributes that look like `{"`*`<attribute-name>`*`", "`*`<format>`*`", `*`<expression>`*`, ...}`, the Attribute Name is the *`<attribute-name>`* component.

During the run, every two transactions that have the specified transaction IDs are linked if the specified attributes have the same value, and if the begin time of transaction *`<tx-ID2>`* happens at or after the begin time of the transaction with *`<tx-ID1>`*.

The first transaction (with *`<tx-ID1>`*) is marked as predecessor, and the second transaction (with *`<tx-ID2>`*) is marked as successor.

The `tlink` statement must appear outside any `STATE` statement.

During run time, for each new transaction being recorded, with either `<tx-ID1>` or `<tx-ID2>`, the software will search for a match starting at the end time of the transaction and going backward in time until another matching transaction is found. At that point the matching transactions are linked, and the search stops.

Examples of Transaction Recording Statements

The following is a simple example that uses only the basic `TX` statement. It defines the beginning and the ending of a transaction using a transaction ID `tx01`. The transaction will appear in Simvision on a stream called `mem`, and will have a transaction type `tx01`.

```
if(top.mem.rd == 'bp)
    tx (begin, tx01, stream=mem,
        title="Memory Read Transaction",
        {"%d", top.mem.address[23:0]}
        ); //begin transaction
if(top.mem.data_ready == 'bp)
    tx(set, tx01, top.mem.data[15:0]); //record data being read as attribute
if(top.mem.rdack == 'bp)
    tx (end, tx01); //end transaction
```

The following example uses the `tdef` statements to simplify the `TX` statements by providing additional details, such as a transaction type, stream scope, a long description, and some *Signal Attributes*.

```
tdef Tw1 {
    type = Twrite,
    stream = "reg",
    scope = "top",
    title = "REG write Transaction",
    description = "Write Transaction into register file from REG",
    top.reg.data[63:0],
    {"%d",top.reg.address[3:0]}
};
tdef Tw2 {
    top.reg.data[63:0],
    {"%d",top.reg.control[3:0]}
};
if(top.reg.wr == 'bp)
    tx (begin, tx02, tdef=Tw1);
if(top.reg.wack == 'bp)
    tx (end, tx02, tdef=Tw2 );
```

The above example is equivalent to the following:

```
if(top.reg.wr == 'bp)
    tx (begin, tx02, type = Twrite,
        stream = "reg",
        scope = "top",
        title = "REG write Transaction",
        description = "Write Transaction into register file from REG",
       {"startData","%d",top.reg.data[63:0]},
        {"%d",top.reg.address[3:0]}
    );
if(top.reg.wack == 'bp)
     tx (end, tx02,{"endData","%d",top.reg.data[63:0]},
        {"%d", top.reg.control[3:0]}
        );
```

Example of using Transaction Linking

The following example shows two transactions with transaction ID `tx01` and `tx02`. Transaction `tx01` has an attribute named `A[7:0]`, and transaction `tx02` has an attribute named `B[7:0]`. During the run, a transaction with ID `tx01` is linked with a transaction with ID `tx02`, if the value of attribute `A[7:0]` in transaction `tx01` matches the value of attribute `B[7:0]` in transaction `tx02`.

```
tlink tx01 A[7:0] tx02 B[7:0];
if(cond1) tx(begin, tx01,A[7:0], S[15:0]);
if(cond2) tx(end, tx01);
if(cond3) tx(begin, tx02, S[15:0]);
if(cond4) tx(end, tx02, B[7:0]);
```

Related Section:

Using TX Statement

Using TRACE_TSM Action

This action will cause the current clock cycle to be unconditionally traced into the SDL trace memory (which can be later dumped to a file using the `sdl -traceDump` XEL command).

The `TRACE_TSM` action itself is not logged into the SDL trace dump file, and unlike other actions, it cannot be disabled from XEL. (The `sdl -traceOn` XEL command can configure which SDL actions will cause SDL tracing, but it has no effect on `TRACE_TSM`).

Related Topic:

Using SDL Actions

SDL Program Examples

This section covers some examples of SDL programs.

Example 1: A State-Less SDL Program

The following SDL program compares two 8 bit vectors `A[7:0]` and `B[7:0]` as unsigned integer numbers and triggers if the number represented by the vector `A[7:0]` is greater than the number represented by the vector `B[7:0]`.

In addition, the program filters out from captured data all samples in which the low order 4 bits of the vector `A` equal the number 13.

```
if(A[7:0] > B[7:0]) trigger;
if(A[3:0] == 13) no_acquire;
```

An equivalent form of the above, using a different variant of the syntax is:

```
trigger = A[7:0] > B[7:0];
no_acquire = A[3:0] == 13;
```

Example 2: A Single SDL Instance

The following SDL program has two states. It triggers if a specific signal becomes high within 100 cycles or less after `sig1` is high. The search for this pattern is restarted each time when sig1 remains low for 100 cycles or more.

Note: The name of the net tested in the first `IF` statement inside `state Two` is taken from the value of a Tcl variable (called `signame`). This enables to change the name of the net tested from XEL without modifying the SDL file.

```
/* Trigger on sig1 high followed by sig2 high, within 100 cycles.
    The number of cycles is counted by counter1
*/
State One       // wait until sig1 goes high
{
    if (sig1)
    {
        load counter1 99;
        start counter1;
        goto Two;
    }
}
```

```
State Two
{
    if ( $signame && Counter1 > 0 )
        trigger ; // trigger if $signame is high while counting
    if ( sig1 )
        load counter1 99; // restart the counter if sig1 high
    if ( Counter1 <= 0 )
        goto One ;      // start all over again
}
```

Example 3: SDL Program with Two Instances

The following SDL program demonstrates use of different SDL constructs. It has two SDL instances, and the program spans on two files.

The bodies of each of the two state machines are exactly the same text, derived from a single file called `instance.tdf`. However, the two state machines execute differently because they rely on the definition of the alias expr, which gets a different value in each of the two instances. Each of the two instances define also a global trigger and a global acquire.

```
// Global Definition Section
exp1 = counter1 > 0 || top.ctl.xyz; // an alias
exp2 = top.ctl.bus1[7:0] != 'b10011 || top.ctl.sig1 == 'bs; // another alias
address_plus_data = {address[15:0],data[15:0]}; // alias defined by concatenation
counter2 = 100; // define default load value for counter2
glob_trig1: trigger = top.cyx[7:0] == top.fsq[15:8]; // global trigger #1
// begin first instance definition
Instance INST_1;
expr = address_plus_data == 32'h874A_FFFF; // new alias using previous alias
`include "instance.tdf"
// begin second instance definition
Instance INST_2;
expr = address_plus_data == 0; // new alias using previous alias
`include "instance.tdf"
------- contents of file instance.tdf start below ------
glob_trig2: trigger = expr & !top.enbr; // global trigger (one for each instance)
acquire = !expr; // global acquire definition (one for each instance)
if(exp2)
    display("Address=%H Data=%H",address[15:0],data[15:0]);
// Definition of States
state first {
    load counter2 $cnx; //load counter from a value defined in a Tcl variable
    load counter1 511;
    if (exp1 && !(top.vrx[3:0] || top.vx[7:0] >= top.vy[7:0]))
        goto second;
    else if (expr) {
            if (top.bxx != 1)
                goto second;
            else
                acquire;
    }
    else
    {
        acquire;
        display (”uploading probes. time=%t”);
        exec ”database -upload”;
    }
}
state second {
    if( {ra,rb,ctr1,ctr2} == 5)
        tsm_trig: trigger;
    if(exp2 == 0 || exp1 && counter2 <= 0) {
        trigger;
        start counter2;
        decrement counter1;
    }
    if(!(top.ctl.bus2[15:8] == ’hps || top.xyz != top.vpprt))
        goto first;
}
```

Creating Time Markers and Bookmarks

SDL enables you to generate time markers based on various SDL conditions. The time markers can be used later as bookmarks in the infiniTrace sessions, and/or as visual markers on the Verdi waveform display. These time markers are shown as labeled time markers on the Verdi waveform display.

Note: Markers are supported only for the Verdi waveforms and are supported only in STB and LA mode.

The markers are written into one or more files for later use in the infiniTrace run and the waveform display. Each marker contains an identifier (string defined by you) and the time information. It is written into several files as follows:

1. If a regular FSDB waveform database is open, the marker is written into a *Marker* file with a name that is derived from the name of the waveform database.
2. If a FSDB streaming waveform database is open, the marker is written into a marker file with a name that is derived from the name of the streaming waveform database.
3. The marker is also written into a *Bookmark* file in a format that is compatible with the XEL command: `bm -import <`*`file`*`>`. The bookmark file enables the marker to be used later in infiniTrace run.

Generating Markers in Markers File

The marker file is generated for both regular and streaming probes. You must ensure that the FSDB waveform database is already open for write operation before executing the `run` command. When a run is started with an SDL program that contains a `DISPLAY` action with the `F=marker` argument, the marker file is opened for write operation for the waveform database. Subsequent runs will append to the existing marker file until the waveform database is closed.

When the FSDB waveform database is opened for write operation, if a prior marker file with the same name as that of the database exists, the marker file is deleted.

Syntax:

SDL generates a time marker each time when a DISPLAY action with *`F=marker`* parameter is executed. The syntax to specify the marker is as follows:

```
    display (F=marker, "<marker-ID>");
```

When the `DISPLAY` action is executed, a marker with the name <*`marker-ID`*> is written to the designated marker and bookmark files.

The keyword `marker` is case insensitive. No other arguments are allowed inside the parenthesis. <*`marker-ID`*> must follow regular SDL conventions for an identifier — it can contain only underscore or alphanumeric characters and should not start with a digit.

SDL `DISPLAY` actions can possibly try to generate multiple markers with conflicting data. The markers are processed according to certain rules. The rules are as follows:

- If you have multiple markers with same time and different identifiers, only one of the markers is written into the file, and the rest are discarded.
- If you have multiple markers with same identifier and different time, then the first marker with earliest time is written with its original identifier. The subsequent markers are written with identifiers that are composed from the original identifier plus a unique suffix. The suffix is an underscore followed by an integer. For example, if the marker identifier in the SDL program is `ABC`, then the first marker is written as `ABC`, the second as `ABC_1` the third as `ABC_2`, and so on.

Viewing Markers in Verdi Waveforms

When the Verdi waveform is brought up through the xeDebug GUI, the marker file for regular probes is read automatically and displayed on the first Verdi waveform window each time after new probed data is read from the DCC. The first Verdi waveform window is referred to as the one that was first opened by xeDebug. The markers are displayed only in the first window and do not affect other waveform windows that you might open later.

If you need to display the markers on other waveform windows, you must do that manually from the waveform window menu. When manually reading the marker file into the Verdi waveform, the markers will show the correct time only if the selected time unit in the waveform is ns. This is due to a limitation of the Verdi tool. When the marker file is read automatically (that is, controlled by xeDebug) into the first Verdi Waveform, xeDebug ensures that the correct time unit is used.

To ensure automatic loading of the marker file for regular probes in the Verdi viewer, you need to add probes with the `probe -waveform` command in the xeDebug GUI. The Verdi waveform does not display the markers when the signals are not selected from the FSDB file. This is a known limitation of Verdi. To see the markers, you must have some signals in the display. With the `probe -waveform` command, the signals and markers will be displayed in the waveform. For example:

```
xeDebug -gui -fsdb -input <qel_file>
database -open <database_name>
probe -waveform <object_name(s)>
```

The marker file is created from SDL DISPLAY action with `F=marker` parameter where the time unit is ns.

```
$ cat trace.rpt
bk2 105
bk1 120
bk1_1 125
bk1_2 130
bk1_3 135
bk1_4 140
```

Waveform markers file are not loaded automatically for streaming probes in the Verdi viewer. These files need to be loaded manually in the Verdi viewer, as follows:

1. From the Verdi menu bar, select the *waveform->marker* option that opens the marker dialog box.
2. In the marker dialog box, click *Restore* and select the marker file.

Generating Markers in Bookmark File

The bookmark file is opened for write the first time when xeDebug uses an SDL program that contains the `display` (`F=marker`, "<*marker-ID*>") action. The bookmark file remains open and is appended with new markers until xeDebug exits. The default name of the generated bookmark file is `sdl.bookmark`.

Example 1: Generating sdl.bookmark

The SDL file `x.tdf` contains the following:

```
    State s1 {
         display(F=marker, “bk1”);
         goto s2;
    }
    State s2 {
          display(F=marker, “bk2”);
          goto s3r;
    State s3 {
           display(F=marker, “bk2”);
           trigger;
    }
```

You execute the following XEL commands:

```
    debug .
    host .
    download
    database -open xyz
    sdl -enable
    sdl -setfile x.tdf
    run
```

A bookmark file with the name `sdl.bookmark` and a marker file with name `xyz.rpt` will be created. These files will have the following markers:

```
Identifier         Time

bk1          1
bk2          2
bk2_1        3
```

Example 2: Using sdl.bookmark in InfiniTrace Session

You can load the bookmark file later in the InfiniTrace session:

```
debug .
host .
download
configPM -stb
bm -import sdl.bookmark
infiniTrace -observe <prepare_session_name>
infiniTrace -goto bk2
......
infiniTrace -goto bk2_1
```

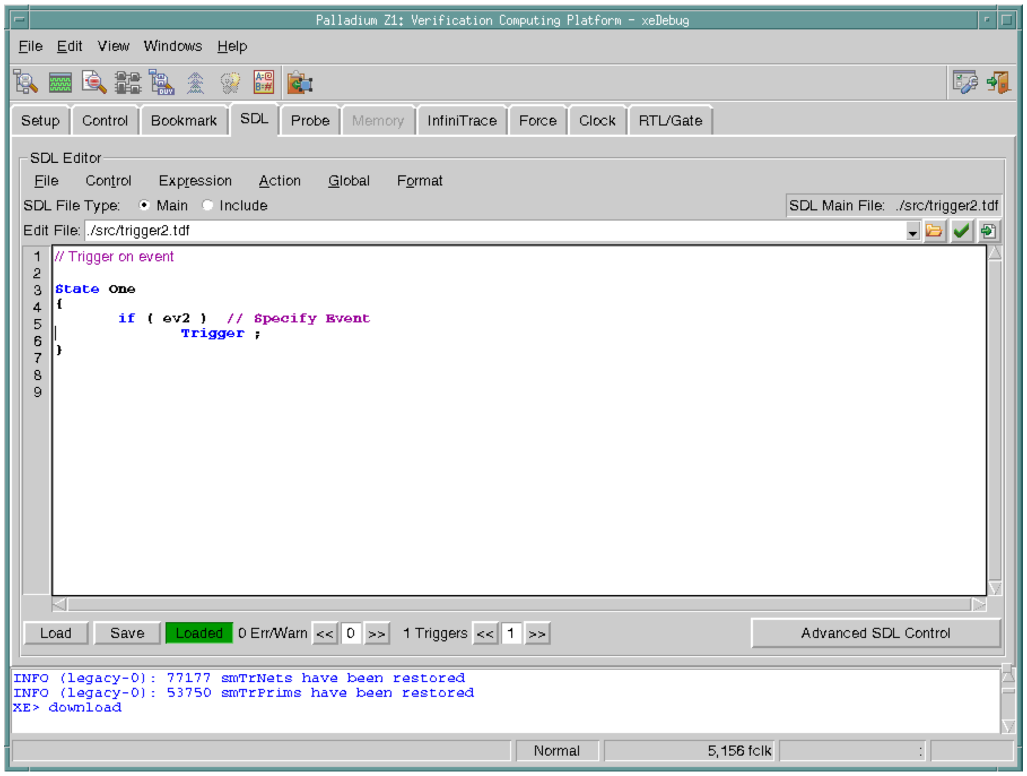
Creating SDL Programs Using xeDebug GUI

Use the SDL E*ditor* group box in the *SDL* tab of the xeDebug GUI to create and edit SDL (State Description Language) programs. The SDL programs can be saved to and restored from files by using the *Save* and *Load* buttons.

The *SDL Editor* group box provides drop-down menus, such as *File*, *Control*, and *Format*, that help you to build your SDL programs.

Note: You can use the right mouse button to drag and drop symbols or events from various forms, such as the *Symbol Editor*. Also, during the run, the current state of the loaded SDL file is highlighted.

Figure 84-1 SDL Tab


Palladium Z1: Verification Computing Platform - xeDebug
File Edit View Windows Help
Setup Control Bookmark SDL Probe Memory InfiniTrace Force Clock RTL/Gate
SDL Editor
File Control Expression Action Global Format

SDL File Type: [✓ Main] Include
SDL Main File: ./src/trigger2.tdf
Edit File: ./src/trigger2.tdf

```
// Trigger on event

State One
{
        if ( ev2 )  // Specify Event
                Trigger ;
}
```

Load
Save
Loaded
0 Err/Warn << 0 >>
1 Triggers << 1 >>
Advanced SDL Control

```
INFO (legacy-0): 77177 smTrNets have been restored
INFO (legacy-0): 53750 smTrPrims have been restored
XE> download
```

The following table lists the drop-down menus available within the *SDL Editor* group box:

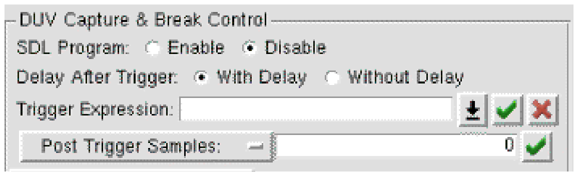
Menu Item	Function
File	Provides you the following sub-menu items: ■ *New*: To create a new SDL program ■ *Open*: To open an existing SDL program ■ *Undo*: To revert any edits done to the SDL program. ■ *Redo*: To re-apply the edits done to the SDL program. ■ *Save*: To save the new/edited SDL program ■ *Save as*: To save the SDL program as a file with extension `.tdf` or `.sdl`.
Control	Allows you to specify SDL controls by selecting the required control from the following sub-menu items: *Instance*, *Include*, *State*, *If*, *Else If*, and *Else*
Expression	Allows you to select various keywords and operators for creating boolean conditions, such as {}, >, <, and &, for use inside IF statements.
Action	Allows you to select an action from the following sub-menu items: Goto, Load, Increment, Decrement, Start, Stop, Trigger, No_Acquire, Acquire, DISPLAY, TX, and EXEC
Global	Provides options used in creation of global definitions for TRIGGER, DISPLAY, TX, EXEC, conditional acquisition, and initial load values or count enable expressions for the two 40 bit counters.
Format	Allows you to change the indentation of SDL statements.

Selecting Trigger Mode for SDL Instances from xeDebug GUI

Use the *DUV Capture & Break Control* group to select trigger modes for SDL instances. If no trigger mode is defined, the design runs without any trigger conditions set.

In SA mode, this group box is enabled only when the design is run on hardware. In this case, in addition to the fields explained below, an extra button, *Import Sim breakpoints*, is displayed (as shown in DUV Capture & Break Control Group Box in SA Mode (Hardware Run).) For corresponding details, refer to Importing Breakpoints from the Xcelium Simulator into SDL Using GUI.

Figure 85-1 DUV Capture & Break Control Group Box in SA Mode (Hardware Run)


DUV Capture & Break Control
SDL Program: Enable [✓ Disable]
Delay After Trigger: [✓ With Delay] Without Delay
Trigger Expression:
Post Trigger Samples: 0

1. Select either *Enable* or *Disable* radio button to control the *SDL Programs* that have not been specifically enabled or disabled using the *Advanced SDL Control* window.
2. Select either *With Delay* or *Without Delay* radio button to define whether delay needs to be added after a trigger occurs.
3. In the *Trigger Expression* field specify the event or logic expression for triggering by expression. The syntax and semantics of the expressions are the same as SDL expressions.
4. In STB mode, when this switch is selected, all the components of the SDL program defined in the SDL form are disabled without exception, even those that were separately enabled using the following XEL command:

   ```
   sdl -enable <instance_names_list>
   ```
5. In the *Trigger Position (%)* field, specify a percentage to calculate the number of samples to capture after a trigger occurs. By default, it is set to 100%.

   This is a toggle field that allows you to select either of the following options: *Trigger Position (%)*, *Post Trigger Percent (%)*, and *Post Trigger Samples*. Use *Post*

Trigger Samples to specify a value in terms of samples. Use either *Trigger Position (%)* or *Post Trigger Percent (%)* to specify a percentage of trace memory.

Related Sections:

- Creating SDL Programs Using xeDebug GUI
- Using Advanced SDL Controls in xeDebug GUI

Importing Breakpoints from the Xcelium Simulator into SDL

In SA mode, you can use the `stop` command to define on the simulator an arbitrary number of breakpoints independent of each other. These breakpoints do not function correctly after swapping the design into the Palladium emulator if they rely on design objects that reside in the DUT. It is possible to import some of these breakpoints into an SDL program that can run on the Palladium emulator. The only types of breakpoints that can be imported to SDL are `Object` and `Condition`. The following breakpoint options can be imported into SDL programs:

- `-if`
- `-execute`
- `-skip`
- `-delbreak`
- `-continue`

The breakpoints import operation is initiated either by the XEL command:

```
sdl -simBreakPoints -import
```

or automatically during swap-in by setting up the following run-time parameter:

```
xeset sdlAutoSimBreakpoints 1
```

Importing breakpoints involves more than just creating an SDL file. If the imported breakpoints contain `-skip` or `-delbreak` options, then the skip or delbreak counts are imported into the xeDebug run-time environment as values of Tcl variables. A successful import operation also loads the imported SDL file into the emulator, as if the `sdl -load` command was executed.

Disabled breakpoints are not imported.

Theory of Operation

The `Object` or `Condition` component of the imported breakpoint is combined with the expression of the `-if` component if one was specified, and the result is used to create the `if(<expression>)` of a global SDL `TRIGGER` and/or `EXEC` statements.

The `skip`, `delbreak` and `execute` features are handled by SDL `EXEC` statements, while keeping track of the `skip` and `delbreak` counts is done by elements of a Tcl array. The values in the array represent `skip` and `delbreak` counters and their values are initialized by the `sdl -simBreakpoints -import` command.

Expressions in Breakpoints

When breakpoints are imported to SDL, the following modifications are made to the expressions that come from the `if` component or the main condition of a `Condition` type breakpoint:

- References to signal values in the form `#<signal-name>` in expressions are translated to SDL by removing the `#`. The syntax `[value <signal-name>]` is not supported by the import operation.
- The operators = and === in the original simulator breakpoints are translated to ==.
- VHDL STD_LOGIC and integer constants are translated to SDL format (for example, `″10110″` is translated to `′b10110`).
- Not all operators allowed in simulator breakpoints can be imported. For example, XM-Sim enables you to use a multiplication operator inside a breakpoint condition, but this cannot be imported to SDL. See Table 44-4 for a list of operators allowed in SDL conditions.

Restrictions and Error Conditions for Importing Breakpoints

All breakpoints are not translated to SDL. The following rules decide which breakpoints are to be imported, and which cause the operation to fail:

- Breakpoints that rely only on design objects located in the TB (or that do not rely on any design object) remain functional on the simulator, and are not imported.
- Breakpoints of type `Condition` or `Object` are imported only if they use signals from the DUT, and expression syntax that can be processed by SDL.

- Breakpoints of the CPF object types are specific changes in isolation rules, retention rules, or power domain states. Only the following options are supported:
  - For the `iso_rule` breakpoint, the supported sub-options are: `-iso_disable` and `-iso_enable`.
  - For the `pdname` breakpoint, the supported sub-options are: `-isolation`, `-pd_off`, `-pd_on`, `-pd_trans`, and `-retention`.
  - For the `sr_rule` breakpoint, the supported sub-options are: `-sr_restore` and `-sr_save`.
- In all other cases, the software makes an effort to determine whether the breakpoint in question continues to operate correctly under the simulator after the swap-in.
  - If it is determined that the breakpoint does not operate correctly, the import operation fails.
  - If the software cannot determine for sure, then only a warning is printed.

Examples

- If a breakpoint contains an `if` component that uses a mix of TB and DUT objects, an error condition will be generated.
- If a breakpoint uses an expression that compares a DUT signal with a value of `unknown` (`X`), an error is generated because Palladium does not support unknowns. Note that, in SDL `'bX` means `don't care`, while `'bX` in a breakpoint means `unknown`.
- An isolation rule breakpoint in the DUT does not function correctly after swap-in, but only generates a warning because the current implementation cannot determine the location of the isolation rule.
- A delta breakpoint almost always behaves differently after swap-in, unless the entire DUT was input in the form of a gate-level netlist, therefore it generates a warning message.

Error Recovery Methods

By default, any error condition during import of any of the breakpoints causes the whole import operation to fail. If the import is done automatically during swap-in (by using `xeset sdlAutoSimBreakpoints 1`), then the swap-in fails and the system reverts to simulation mode.

You can choose to ignore breakpoints that generate error conditions during the import, and proceed with any breakpoints that can be imported successfully by using the following command:

```
sdl -simBreakpoints -errorok 1
```

This affects all future requests to import breakpoints to SDL. In such case, the same error messages are reported (if any error conditions are found), but they do not cause the operation to fail and an SDL file with any breakpoints that could be imported successfully is generated and loaded into the emulator. This also prevents a swap-in from failing due to errors in automatic import.

Limitations and Differences in Behavior

Even breakpoints imported successfully to SDL might produce a behavior that is not exactly the same as running under pure simulation. Following are the differences:

- The import operation ignores the options `-silent` and `-noexecout,` although all other options are honored.
- When running with Xcelium, a breakpoint is completely deleted when it reaches its `delbreak` count. Under SDL, the corresponding trigger is only disabled when it reaches the delbreak count.
- Tcl variables referenced as `$name` by an `execute` component in the breakpoint reference Tcl variables in the Xcelium Tcl name space, while the imported SDL program will reference Tcl variables in xeDebug's Tcl name space.
- Xcelium breakpoints cause the simulator to stop immediately at the exact simulation time in which the breakpoint's condition becomes true. When using SDL, such behavior is not guaranteed, and the run might stop with some delay (See Choosing Fast or Accurate Behavior in the Concepts of State Description Language (SDL) Guide for more details).
- For a Condition type breakpoint to stop the Xcelium simulator, the expression defined by the condition must be true, and an *event* must happen. The *event* is generated by a change in the value of any object used in the expression, and if the expression contains any bit select of a vector (for example *`name[17]`*), then any change in value of any bit in that vector will also generate an event, whether that bit is included in the expression or not.

  The translated SDL program uses, by default, simpler semantics to reduce the amount of emulation resources. With the simplified semantics SDL will stop the run each time when the result of the condition makes a transition from false to true.

  It is possible to request that the translated SDL use exactly the same event-driven behavior that Xcelium uses for breakpoints with the `sdl -simBreakpoints -exact 1` command.

- Using `uxc_hdl.var` file with xeDebug causes unpredictable behavior. Any breakpoints defined in the `uxc_hdl.var` file might not be imported correctly.

Importing Breakpoints from the Xcelium Simulator into SDL Using xeDebug GUI

In SA mode, to import the breakpoints from the Xcelium simulator into SDL, perform the following steps:

1. Access the *Setup* tab of the xeDebug GUI after starting a debug session.
2. Click the *--> Show Hot Swap Options* button from the *Configuration* group box to view the available options. You have the following two check boxes that support the corresponding run-time commands for importing the Xcelium breakpoints:
   - ❑ *Error Ok* (for `sdl -simBreakpoints errorok 0|1` command)
   - ❑ *Auto import breakpoints* (for `xeset sdlAutoSimBreakpoints 0|1` command)

   Note: These options are also available in the *Sim BreakPoints* group box of the *Advanced SDL Control* window accessible from the *SDL* tab.
3. When running the design on hardware, click the *Import Sim breakpoints* button from the *DUV Capture & Break Control* group box in the Control tab to execute the `sdl -simBreakpoints -import` command.

For detailed information on the importing of Xcelium breakpoints, refer to IImporting Breakpoints from the Xcelium Simulator into SDL section.

Importing Breakpoints from the Xcelium Simulator into SDL Using GUI

In SA mode, to import the breakpoints from the Xcelium simulator into SDL, perform the following steps:

1. Access the *Setup* tab of the xeDebug GUI after starting a debug session.
2. Click the *--> Show Hot Swap Options* button from the *Configuration* group box to view the available options. You have the following two check boxes that support the corresponding run-time commands for importing the Xcelium breakpoints:
    - ❑ *Error Ok* (for `sdl -simBreakpoints errorok 0|1` command)
    - ❑ *Auto import breakpoints* (for `xeset sdlAutoSimBreakpoints 0|1` command)

    Note: These options are also available in the *Sim BreakPoints* group box of the *Advanced SDL Control* window accessible from the *SDL* tab.
3. When running the design on hardware, click the *Import Sim breakpoints* button from the *DUV Capture & Break Control* group box in the Control tab to execute the `sdl -simBreakpoints -import` command.

Related Topics:

For detailed information on the importing of Xcelium breakpoints, refer to Importing Breakpoints from the Xcelium Simulator into SDL in Controlling the Run with SDL.

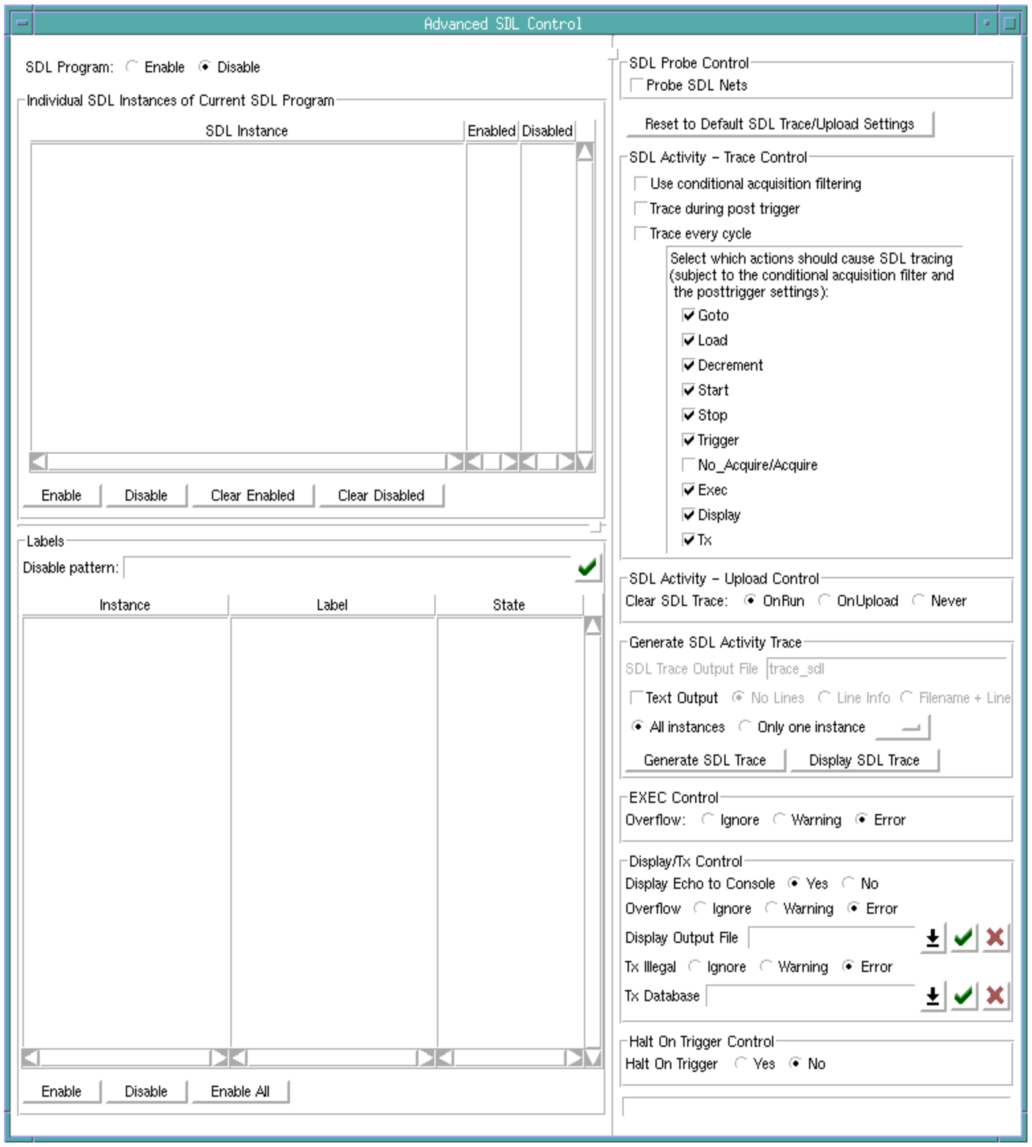
Using Advanced SDL Controls in xeDebug GUI

Clicking the Advan*ced SDL Control* button on the *SDL* tab displays the Advanced SDL Control window. This window enables you to control individual SDL instances and SDL activities like probes, trace, and upload through the GUI from the following group boxes:

- *Individual SDL Instances of Current SDL Program* to enable or disable specific SDL instances
- *SDL Trigger Labels* to enable or disable specific trigger labels
- *SDL Probe Control* for probing SDL nets
- *SDL Activity - Trace Control* to identify:
  - Conditions for trace, such as *Use conditional acquisition filtering*, *Trace during post triggers*, and *Trace every cycle*
  - Actions that cause SDL tracing, such as *Load*, *Trigger*, *Exec*, *Tx*, *Display*
- *SDL Activity - Upload Control* to identify when to *Clear SDL Trace*, such *OnRun* or *OnUpload*
- *Generate SDL Activity Trace*
- *Sim BreakPoints* to import Xcelium breakpoints to SDL during swap-in

  Note: This group box is displayed only when you are running the design in SA mode.
- *EXEC Control* to specify EXEC control conditions
- *DISPLAY Control* to specify DISPLAY and TX control conditions
- *Halt On Trigger Control* to specify trigger control conditions

Figure 89-1 Advanced SDL Control Window (ICE Mode)



Advanced SDL Control

SDL Program: Enable [✓ Disable]

Individual SDL Instances of Current SDL Program

SDL Instance	Enabled	Disabled

Labels

Disable pattern:

Instance	Label	State

SDL Probe Control
Probe SDL Nets

SDL Activity – Trace Control
Use conditional acquisition filtering
Trace during post trigger
Trace every cycle
Select which actions should cause SDL tracing (subject to the conditional acquisition filter and the posttrigger settings):
[✓ Goto]
[✓ Load]
[✓ Decrement]
[✓ Start]
[✓ Stop]
[✓ Trigger]
No_Acquire/Acquire
[✓ Exec]
[✓ Display]
[✓ Tx]

SDL Activity – Upload Control
Clear SDL Trace: [✓ OnRun] OnUpload Never

Generate SDL Activity Trace
SDL Trace Output File: trace_sdl
Text Output [✓ No Lines] Line Info Filename + Line
[✓ All instances] Only one instance

EXEC Control
Overflow: Ignore Warning [✓ Error]

Display/Tx Control
Display Echo to Console [✓ Yes] No
Overflow Ignore Warning [✓ Error]
Display Output File
Tx Illegal Ignore Warning [✓ Error]
Tx Database

Halt On Trigger Control
Halt On Trigger Yes [✓ No]

Related Sections:

- Creating SDL Programs Using xeDebug GUI
- Selecting Trigger Mode for SDL Instances from xeDebug GUI

Using a Helper HDL Module To Track Complex Design Behavior with SDL

Some hardware protocols might be too complex for SDL to handle directly.

A way to approach this is to provide a *helper* module, written in Verilog or VHDL, that can monitor the design, and generate signals that SDL can use in conditions. The helper module has to be compiled together with the design, so it becomes an integral part of the design database. Here is how you can do it without changing anything in the original design:

The helper module should be written in a separate file, and access signals in the original design by using their absolute path.

When using the IXCOM compiler, the file containing the helper module can simply be specified with the `ixcom` command. No other step is needed.

When using ICE flow, use the `import bind` command to add the file to the existing design, as follows:

1. Compile the file containing the helper module into a netlist using HDL-ICE Compiler.
2. Create a new file with the following single line:

   ```
   bind <topCellName> <moduleName> <instanceName>();
   ```

   <*topCellName*> is the name of the top cell in the original design.

   <*moduleName*> is the name of the helper module.

   The helper module will be instantiated with the instance name <*instanceName*> under <*topCellName*>.
3. Include the file containing the `bind` command as well as the netlist file with the helper module that was generated by HDL-ICE Compiler, and the original design netlist as inputs to the `import` command.

You can create any number of helper instances in this manner under the top cell (each will require its own `bind` command).

As an example, consider the following `bind` command:

```
bind mytop Hmod HLP();
```

`mytop` - is the top cell name of the design.

`Hmod` - module name of helper module.

The helper module will be instantiated as `mytop.HLP`.