ZeBu® Limited Customer Availability (LCA) Features Guide

Version V-2024.03-1, July 2024

Feedback

Contents

About This Book 6

Contents of This Book 6

Related Documentation 7

Typographical Conventions 8

Synopsys Statement on Inclusivity and Diversity 8

1. Language Support 10

Verilog Switch-Level Modeling Support in Compilation 10

Enabling Switch-Level Modeling Support 10

Limitations 11

Initialization Assignments Check 11

2. Compilation Features 12

Capacity-Focused Compilation 12

Using Deferred DPI for Throughput Improvement 12

Challenges in Using Normal DPI Import Calls 13

Benefits of Using Deferred DPI 13

Enable Delayed DPI 14

Example 15

zPar Enhancements: Support for Single-Ended LVDS 16

Toggle Coverage 16

Enabling Toggle Coverage 17

Toggle Coverage Database 17

Limitations 18

3. ECO 19

ECO-Aware Compilation 20

Reserving Resource for ECO Flow 20

Example 20

Reserving Resource in a Specified Scope 20

Example 21

ECO Flow for Incremental Compilation with zCui . . . 21
zCui: Batch Mode . . . 21
zCui: GUI Mode . . . 22
ECO Flow for Runtime Triggers . . . 23
Specific Recommendations for ECO-Aware Compilation . . . 23
Launching the ECO flow for Runtime Triggers . . . 24
Example . . . 24
ECO Flow for Dynamic Triggers . . . 25
ECO-Aware Compilation . . . 25
Creating a New Trigger . . . 26
Modifying an Existing Trigger . . . 26
Checking the ECO Support . . . 26
Additional zECO Usage . . . 26
Checking ECO Support . . . 27
Example 1 . . . 27
Example 2 . . . 27
General Usage . . . 27
Limitations . . . 28

4. Runtime Features . . . 32
Dual zTime Support for Triggers . . . 32
Modes for Dual zTime Support . . . 32
Limitations . . . 33
Simulated Emulation . . . 33
License Requirements for SEM . . . 34
Design Compilation with SEM . . . 34
Runtime for SEM . . . 35
Platform Differences Between ZeBu and SEM . . . 35
Limitations of SEM . . . 35
Support for Multiuser on Single Host . . . 36
Benefits . . . 36
Disadvantages . . . 36
Limitations . . . 36
Use model . . . 36
Example (in `bash`) . . . 36

5. Performance Features 37

ZRM Performance Mode is Automatically Enabled 37

Support for SDC-MCP Relaxation on Verilog Force Release 38

Preface

This book has the following sections:

- About This Book
- Contents of This Book
- Related Documentation
- Typographical Conventions
- Synopsys Statement on Inclusivity and Diversity

About This Book

The *ZeBu® LCA Features Guide* describes the Limited Customer Availability (LCA) features and enhancements that are available in this release.

The LCA features are categorized as follows:

- Compilation stability improvements
- **zTime** frequency improvements
- Language support
- ECO
- DMTCP

Contents of This Book

The *ZeBu® LCA Features Guide* has the following sections:

Section	Describes...
Language Support	features to improve the Verilog standard support in the compilation flow.
Compilation Features	features to improve the compilation time, stability, and theoretical design performance.
ECO	usage of ECO feature.

Section	Describes...
Runtime Features	information about LCA features introduced to improve emulation runtime.

Related Documentation

Document Name	Description
ZeBu User Guide	Provides detailed information on using ZeBu.
ZeBu Debug Guide	Provides information on tools you can use for debugging.
ZeBu Debug Methodology Guide	Provides debug methodologies that you can use for debugging.
ZeBu Unified Command-Line User Guide	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
ZeBu UTF Reference Guide	Describes Unified Tcl Format (UTF) commands used with ZeBu.
ZeBu Power Aware Verification User Guide	Describes how to use Power Aware verification in ZeBu environment, from the source files to runtime.
ZeBu Functional Coverage User Guide	Describes collecting functional coverage in emulation.
Simulation Acceleration User Guide	Provides information on how to use Simulation Acceleration to enable cosimulating SystemVerilog testbenches with the DUT
ZeBu Verdi Integration Guide	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
ZeBu Runtime Performance Analysis With zTune User Guide	Provides information about runtime emulation performance analysis with zTune.
ZeBu Custom DPI Based Transactors User Guide	Describes ZEMI-3 that enables writing transactors for functional testing of a design.
ZeBu LCA Features Guide	Provides a list of Limited Customer Availability (LCA) features available with ZeBu.
ZeBu Synthesis Verification User Guide	Provides a description of zFmCheck.
ZeBu Transactors Compilation Application Note	Provides detailed steps to instantiate and compile a ZeBu transactor.
ZeBu zManualPartitioner Application Note	Describes the zManualPartitioner feature for ZeBu. It is a graphical interface to manually partition a design.

Document Name	Description
ZeBu Hybrid Emulation Application Note	Provides an overview of the hybrid emulation solution and its components.

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	`<sig-name>`
Message	Active low signal name '<sig-name>' must end with _X.
Message location	`OUT` <= IN;
Reworked example with message removed	`OUT_X` <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
... (Horizontal ellipsis)	Other options that you can specify

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

Feedback

1

Language Support

This section provides information about LCA features introduced to improve ZeBu usability.

See the following topics:

- Verilog Switch-Level Modeling Support in Compilation
- Initialization Assignments Check

Verilog Switch-Level Modeling Support in Compilation

The ZeBu compilation flow supports Verilog description in the switch-level modeling to describe how these switch-level networks can be handled and how it should behave when detecting an unsupported case.

SystemVerilog allows scalar net signal values to have a full range of unknown values and different levels of strength or combinations of levels of strength. This multiple-level logic strength modeling resolves the combinations of signals into known or unknown values to represent the behavior of hardware with improved accuracy. Therefore, SystemVerilog provides accurate modeling of signal contention, bidirectional pass gates, resistive MOS devices, dynamic MOS, charge sharing, and other technology-dependent network configurations.

This section describes the following subtopics.

- Enabling Switch-Level Modeling Support
- Limitations

Enabling Switch-Level Modeling Support

To enable or disable the switch-level modeling in SystemVerilog, use the following UTF command:

```
design -convert_switches true
```

Limitations

The switch-level modeling does not implement dynamic strength resolution because the additional logic to be inserted while manipulating all the strengths is large.

Initialization Assignments Check

Starting with the Q-2020.03-SP1-3 release, a synthesis option is added to enable the following checks for initialization syntax, which is not synthesizable. The following initialization assignments are checked:

- integer xxx = <constant> ;
- wire aaa = <constant> ;
- reg ccc = <constant> ;
- reg aaa ; initial aaa = 'b1 ;

The following message is reported:

```
Error-[UC_VERI_VAR] Non synthesizable code for variables
<file_name>.sv, <line_number>
This initialization syntax is not supported, and will not be synthesized
```

To enable the check, the following options are added to `design` UTF command:

```
design -decl_assignment_check <true|false>
-filter_context <ALL|DUT >
-exclude_modules {<modules>}
-severity <error|warning>
```

Where:

- `decl_assignment_check`: Default is false, which indicates the check is disabled. Set to true to enable the check.
- `filter_context`: Default is DUT, which indicates that transactors are excluded from the check. Set to ALL to include transactors in the check.
- `exclude_modules`: Specify list of modules to exclude from the check
- `severity`: Default is error. To lower severity to warning, set this option to warning.

2

Compilation Features

This section provides information about LCA features introduced to improve compilation time, stability, and theoretical design performance.

This section describes the following topics:

- Capacity-Focused Compilation
- Using Deferred DPI for Throughput Improvement
- Example
- zPar Enhancements: Support for Single-Ended LVDS
- Toggle Coverage

Capacity-Focused Compilation

Starting with the 2021.09 release, ZeBu enables you to prioritize capacity at the cost of compilation time and performance. To enable the capacity compilation objective, set the following UTF command:

```
compile -objective CAPACITY
```

Using Deferred DPI for Throughput Improvement

ZeBu provides a framework for DPI Import calls that does not require the design clocks to be stopped immediately and also lets multiple DPI calls be active simultaneously. This framework results in significant throughput improvement in using ZEMI-3 design clocks.

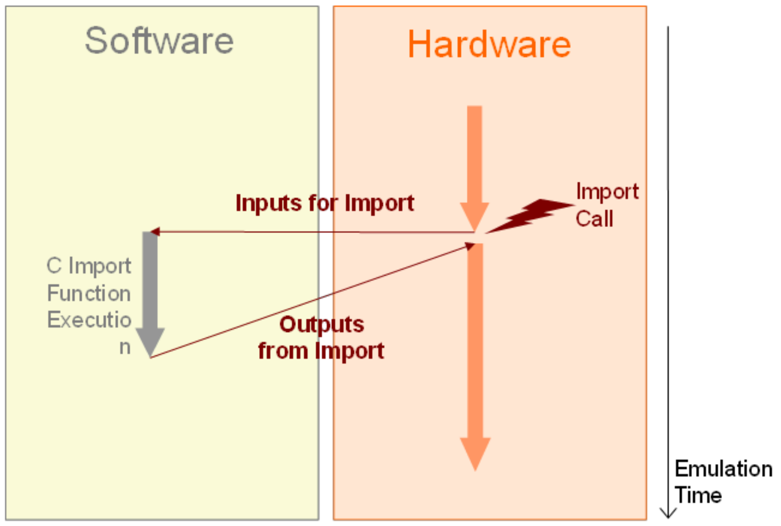
Challenges in Using Normal DPI Import Calls

Few challenges faced when using normal DPI Import calls are as follows:

- Normal Import calls stop the design clock until data is fetched from the software.
- Frequent Import calls imply frequent clock stoppage that results in decline of tickclk/ driverclk ratio. Ultimately, this results in performance decline.
- Stopping of design clock means stopping even those portions of the design that are not dependent on the import data.

The following figure describes the normal DPI import topology.

Figure 1 Normal DPI Import - Topology



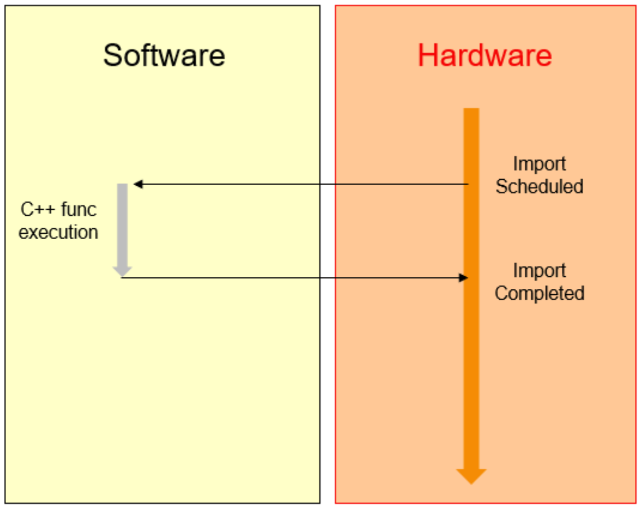
Benefits of Using Deferred DPI

The benefits derived when using Deferred DPI (DDPI) Import call are as follows:

- Import is launched but the design clock runs up to a `delay value`. The `delay value` is the number of design clocks you want to run before stopping the clock and waiting for import to return. The delay value can be changed during runtime but the logic written should ensure that no overlap takes place
- After this Delay value is reached, the clocks stop and wait for import to return.
- You can launch multiple consecutive DDPI calls. ZEMI-3 keeps a maximum buffer value for 64 calls.

Figure 2 displays the topology of the DDPI framework.

Figure 2 Deferred DPI - Topology



Enable Delayed DPI

To enable delayed DPI, perform the following steps:

1. Use the following UTF command:

   ```
   zemi3 -delayed_dpi true
   ```

2. Declare a pragma for the DPI that needs to be deferred:

   ```
   (* zemi3_delayed_dpi = N*)
   ```

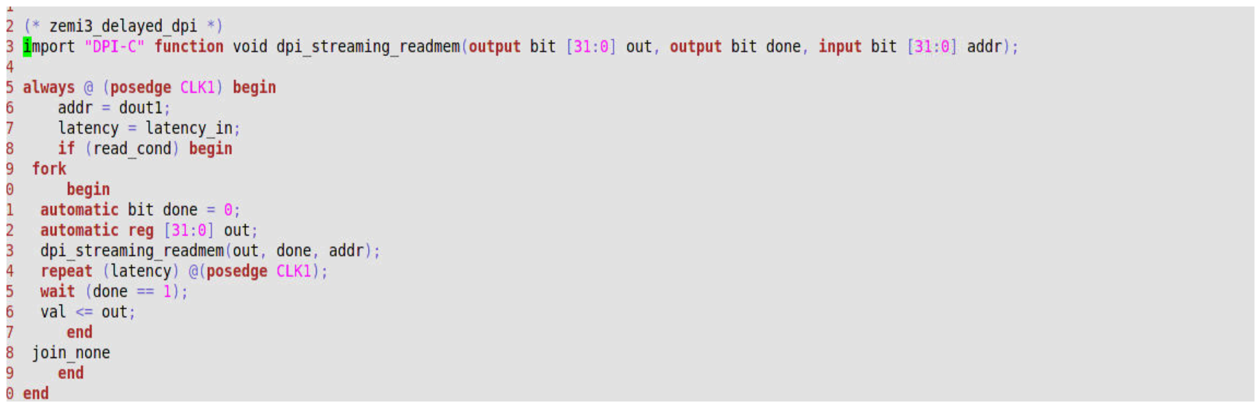
   Ensure it is included on top of DDPI declaration.

   where, `N` represents the maximum number of DPI calls that can be simultaneously active at any given emulation cycle. This implies that maximum `N` deferred DPI calls can be simultaneously active waiting for their latency to expire.

   Default is `N` is 16.

3. Define the "`delay value`", where "`delay value`" is defined as the number of design cycles by which the DPI call must return else design clocks would be stopped.

Consider the following example.



```
(* zemi3_delayed_dpi *)
import "DPI-C" function void dpi_streaming_readmem(output bit [31:0] out, output bit done, input bit [31:0] addr);

always @ (posedge CLK1) begin
    addr = dout1;
    latency = latency_in;
    if (read_cond) begin
 fork
     begin
  automatic bit done = 0;
  automatic reg [31:0] out;
  dpi_streaming_readmem(out, done, addr);
  repeat (latency) @(posedge CLK1);
  wait (done == 1);
  val <= out;
     end
 join_none
    end
end
```

In this example:

- If `N` is not specified, the default number of DPI calls is considered, that is, 16.
- `delay value` is the latency.

Here, the `design cycle` is defined by the event control in the repeat statement.

```
repeat (latency) @(posedge clk1);
```

Example

An example use case for implementing DDPI is as follows:

```
(* zemi3_delayed_dpi *)
import "DPI-C" function void dpi_call(output bit [31:0] out, output bit
 done, input bit [31:0] addr);
always @ (posedge CLK1) begin
    addr = dout1;
    latency = latency_in;
    if (read_cond) begin
        fork
            begin
                automatic bit done = 0;
                automatic reg [31:0] out;
                dpi_call(out, done, addr);
                repeat (latency) @(posedge CLK1);
                wait (done == 1);
                val <= out;
            end
        join_none
    end
end
```

zPar Enhancements: Support for Single-Ended LVDS

The following feature is supported only on EP1 in 21.09-2 and later versions.

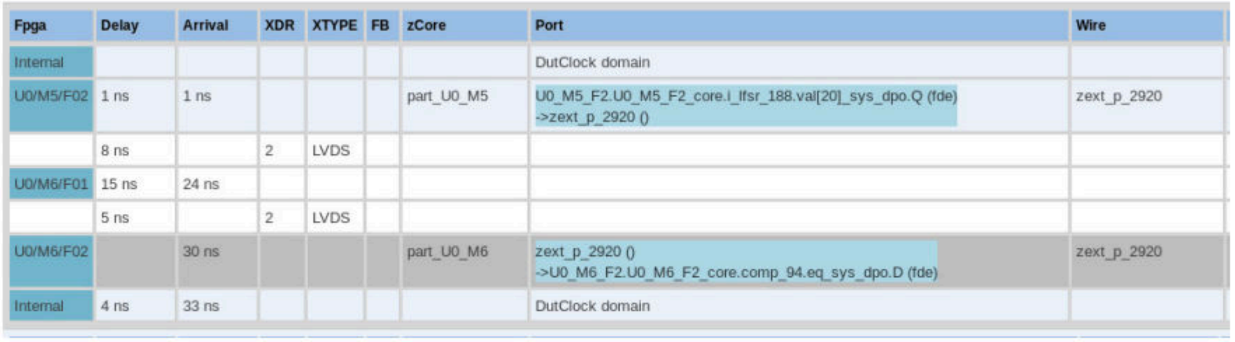
When a signal is directly routed, the legacy implementation of `zPar` uses one physical LVDS pair (2 pins) per signal. To directly route more critical signals, `zPar` now supports single-ended LVDS implementation. That is, `zPar` allows one signal per LVDS pin and therefore, `zPar` can direct two signals per LVDS pair. This helps to improve the performance for every designs.

To enable this feature, specify the following UTF command:

```
zpar -advanced_command {System enableSE true}
```

The XDR value for single-ended LVDS is displayed in the `zTime.html` report as “XDR=2.” The delay of single-ended LVDS is the same as regular xDR=1 delay.

Figure 3 zTime Report



Fpga	Delay	Arrival	XDR	XTYPE	FB	zCore	Port	Wire
Internal							DutClock domain	
U0/M5/F02	1 ns	1 ns				part_U0_M5	U0_M5_F2.U0_M5_F2_core.i_lfsr_188.val[20]_sys_dpo.Q (fde) ->zext_p_2920 ()	zext_p_2920
	8 ns		2	LVDS				
U0/M6/F01	15 ns	24 ns						
	5 ns		2	LVDS				
U0/M6/F02		30 ns				part_U0_M6	zext_p_2920 () ->U0_M6_F2.U0_M6_F2_core.comp_94.eq_sys_dpo.D (fde)	zext_p_2920
Internal	4 ns	33 ns					DutClock domain	

To restore the default behavior, use the following UTF command:

```
zpar -advanced_command {System enableSE false}FIXME-22:PDM
```

Toggle Coverage

Toggle Coverage is one of the effective methods for ensuring wide-spread coverage of the testbench and to achieve verification closure. This feature enables capturing Toggle Coverage information in VDB format. You can use Verdi or Unified Report Generator (URG) in VCS to view and analyze coverage reports.

Following features are supported with Toggle Coverage:

- Various VCS controls including `cm_hier` to enable or disable coverage in various modules/hierarchies
- Support for MDAs

- C++ control to enable/disable coverage collection at runtime
- Support for Simulation Acceleration flow
- Merging of coverage databases using Verdi or URG
- Coverage for VHDL modules in ZeBu flow

For more information, see the following topics:

- Enabling Toggle Coverage
- Toggle Coverage Database
- Limitations

Enabling Toggle Coverage

Perform the following steps to enable Toggle Coverage at compile time:

1. Specify the following UTF command:

   ```
   code_coverage -toggle true | false
   ```

2. Add the `-cm tgl` option while launching VCS as shown in the following example:

   ```
   vcs -full64 -sverilog ../common/hdl/<file>.sv -cm tgl
   ```

At runtime, ZTDB database is captured to collect coverage data by adding the following C ++ API to the testbench:

```
#include "Coverage.hh"
Coverage::SetCoverageTest("dump.ztdb");
Coverage::Start();
Coverage::stop();
Coverage::Close();
```

Toggle Coverage Database

zSimzilla has been enhanced to use the waveform computation for capturing toggle information based on the ZTDB generated during runtime.

To invoke zSimzilla, use one of the following options:

- `-cm_dir`: Specifies the path of the VDB directory captured during front-end compilation. The default is `simv.vdb` or `<zcui.work>/vcs_splitter/simv.vdb`.

  `-cm_name`: Specifies the name of the test, in which, the coverage data is captured. The default is the `zebuTest` directory.

Note:
You can also use both the commands simultaneously.

zSimzilla writes the toggle coverage database in the VDB directory. You can use Verdi or URG to generate various coverage reports.

Limitations

Following are the limitations on toggle coverage:

- Coverage for memories in ZeBu flow
- Support for toggle counters
- zRci Tcl commands for coverage on/off
- Support for toggle coverage and waveform generation in a single zSimzilla run

3 ECO

The newly developed Engineering Change Order (ECO) engine provides the capability to:

- Add debug signals through the incremental compilation with `zCui`:
    - New signals using FWC technology
    - New hierarchies using QiWC technology
    - Additional `zforce/zinject` commands
- Modify the CEL file describing the condition for Runtime Triggers with a specific `zECO` utility

  Note:

  Modifications of the design are not supported for ECO flow.

To achieve these capabilities, an initial ECO-Aware compilation is required with hardware resources allocated for the ECO support.

Any signal of the design, which is directly connected to an FPGA register of latch is eligible for ECO incremental compilation, and a check feature is available for this purpose.

This section describes the features in the following subtopics:

- ECO-Aware Compilation
- ECO Flow for Incremental Compilation with zCui
- ECO Flow for Runtime Triggers
- ECO Flow for Dynamic Triggers
- Additional zECO Usage
- General Usage
- Limitations

ECO-Aware Compilation

This section provides information on the following subtopics:

- Reserving Resource for ECO Flow
- Reserving Resource in a Specified Scope

Reserving Resource for ECO Flow

To prepare initial/reference compilation with default resource reservations for future ECO flow activation, use the following UTF command:

```
eco [-reserve_force <int>] [-reserve_fwc <int>] [-reserve_qiwc <int>]
```

The default values are as follows:

- FWC: 1152 (384 x 3)
- QiWC: 2048 (1024 * 2)
- zForce: 32

If the reserved bit number is not specified for each resource, the default number is taken.

The full command prototype is as follows:

```
eco
    [-recompile_on_error <NEVER|ON_OVERWRITE_ONLY>] # Specify the
 recompilation mode on error
    [-skip_checking <bool>] # Enable/Disable the skip checking design
 changes
    [-reserve_force <int>]  # Force bit number to reserve in each FPGAs
    [-reserve_fwc <int>]    # FWC bit number to reserve in each FPGAs
    [-reserve_qiwc <int>]   # QiWC bit number to reserve in each FPGAs
    [-reserve_scope <dut_top.module_0.instance_0> #Reserves a particular
 scope in a design for ECO support
```

Example

The following is an example usage if your are not using QiWC in the design.

```
eco -reserve_qiwc 0
```

Reserving Resource in a Specified Scope

By default, resources are reserved for ECO flow in all FPGAs of the design. To save resources in FPGAs, and disk space, declare a specific hierarchy in which the reservation for ECO resources using the `-reserve_scope` option.

To enable ECO-aware compilation to reserve resource in a specified scope, specify the scope as follows:

```
eco -reserve_scope <path_to_hierarchy>
```

When *-reserve_scope* is specified,

- Only signals in the specified scope are considered for ECO flow.
- Only FPGAs with signals in the specified scope are considered for ECO flow.
- For FPGAs that are not selected, no additional resource is added. Therefore, the disk space is saved.
- Multiple scopes are accepted.

  Note:

  If you specify signals, which not in the specified scope in ECO, an error is displayed.

Example

```
eco -reserve_scope ufe_top.dut.GPRs -reserve_fwc 1024 -reserve_force 64
eco -reserve_scope ufe_top.dut.InstDecode
```

ECO Flow for Incremental Compilation with zCui

The incremental compilation is applied with `zCui` in the following cases:

- New `$dumpvars`in RTL to add signals to be captured with FWC technology or to add hierarchies to be captured with QiWC technologies
- New signals declared with `zforce/zinject` UTF commands

zCui: Batch Mode

For information on prerequisites for ECO incremental compilation with **zCui**, see ECO-Aware Compilation.

There are two ECO compilation modes that exist from the `zCui` command line as follows:

- **Non-overwriting mode** (using the initial `zcui.work` as a reference and creating a new working `zcui.work` directory for incremental compilation)

  ```
  # Non overwriting mode usage
  %> zCui -u <utf_file> --base <reference zcui.work> --eco -w <new
   zcui.work>
  ```

In the non-overwriting mode, the ECO-aware compilation must be supplied with the `zCui`'s `--base` option as the reference `zcui.work` for the ECO-enabled incremental compilation.

- **Overwriting mode** (reusing the same *zcui.work* than the initial compilation):

  ```
  # Overwriting mode usage
  %> zCui -u <utf_file> --eco -w <zcui.work>
  ```

Note:

The *-w* option is not required if `zcui.work` is the default folder name. Otherwise, you should provide the existing `zcui.work` directory.

There are other `zCui` options available for the ECO flow as follows:

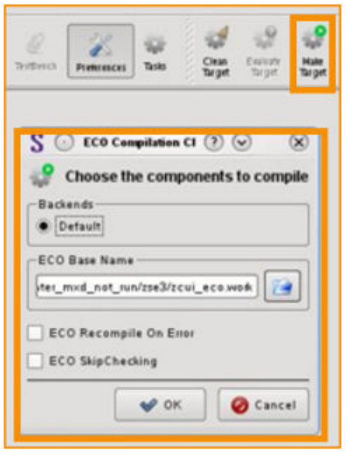
- `--ecoSkipChecking`: Skips design change and analyzes the module with `dumpvar` only.
- `--ecoRecompileOnError`: Re-compiles the design from scratch when ECO is not applicable.

zCui: GUI Mode

To run ECO incremental compilation from `zCui`, perform the following steps:

1. Click **Make Target** from the toolbar.

The **Choose the components to compile** dialog box opens. It lists the parameters specific to ECO.



Choose the components to compile

Backends
● Default

ECO Base Name
ter_mxd_not_run/zse3/zcui_eco.work

☐ ECO Recompile On Error
☐ ECO SkipChecking

1. Click **Default** in the **Backends** section.

   Note:

   Only default backend is supported.

2. Click the browse button to navigate to the base directory in **ECO Base Name** section.

   Note:

   If the base directory is left blank, the over-writing mode is used.

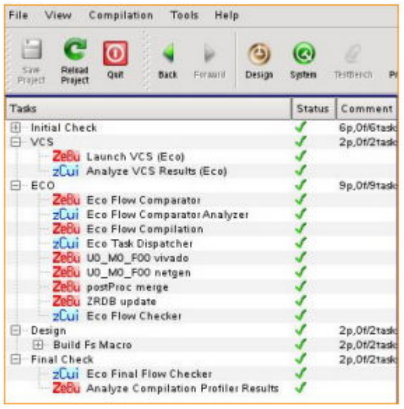
3. Select the **ECO SkipChecking** check box, if required.

   Note:

   The **ECO Recompile On Error** check box is not effective because this option is not supported.

The following figure displays the `zCui` with the ECO-related tasks.

Figure 4 ECO Incremental Compilation Tasks in zCui



Tasks	Status	Comment
⊞ Initial Check	✓	6p,0f/6task
⊟ VCS	✓	2p,0f/2task
ZeBu Launch VCS (Eco)	✓	
zCui Analyze VCS Results (Eco)	✓	
⊟ ECO	✓	9p,0f/9task
ZeBu Eco Flow Comparator	✓	
zCui Eco Flow Comparator Analyzer	✓	
ZeBu Eco Flow Compilation	✓	
zCui Eco Task Dispatcher	✓	
ZeBu U0_M0_F00 vivado	✓	
ZeBu U0_M0_F00 netgen	✓	
ZeBu postProc merge	✓	
ZeBu ZRDB update	✓	
zCui Eco Flow Checker	✓	
⊟ Design	✓	2p,0f/2task
⊞ Build Fs Macro	✓	2p,0f/2task
⊟ Final Check	✓	2p,0f/2task
zCui Eco Final Flow Checker	✓	
ZeBu Analyze Compilation Profiler Results	✓	

ECO Flow for Runtime Triggers

When using Runtime Triggers in your design, ECO Flow is supported after creating a new CEL file to describe the condition for the trigger, or after modifying and existing CEL file.

This section provides information on the following:

- Specific Recommendations for ECO-Aware Compilation
- Launching the ECO flow for Runtime Triggers

Specific Recommendations for ECO-Aware Compilation

To enable ECO for Runtime Triggers, reserve FWC in the ECO-Aware original compilation using the following UTF command:

```
eco -reserve_fwc 1024
```

Launching the ECO flow for Runtime Triggers

To launch *zECO* for runtime triggers, use the following command:

```
zECO -z zcui.work -do <incr_eco_for_cel.tcl> -dest new_zui.work
```

Where,

- `<incr_eco_for_cel.tcl>` contains the following:
    - `zeco_add_dbg_signal_file -ip fwc -cel_file <path of CEL file>`: Adds all the new signals that are declared directly under the CEL file and are inferred to FPGA FF/LD.
    - `zeco_config -no_dcp`: Disable writing the Vivado checkpoint.
    - `zeco_config -post_pnr_timing off`: Disables the post-pnr **zTime** analysis.
    - `zeco_config -pack_vector true`: Enables the vector packing with alignment with 32-bits word.
    - `zeco_config -grid "<qrsh/lsf>"`: Specifies the grid commands.

The signals on which `zECO` has been applied become available with the `ZECO_PRESERVE_FWC` Value Set.

The `ZECO_PRESERVE_FWC` Value Set (and potentially other Value Set targeting non ECOed signals) must be added to ZTDB attached to the Runtime Trigger.

Example

An example of a CEL fie is as follows:

```
module complex_cel;
clock posedge hw_top.async_fifo_a.wclk_i;
reset negedge hw_top.rstn_i;
var ff hw_top.ff_o;
var we hw_top.we_i;
var dut_counter [31:0]
hw_top.async_fifo_a.counters_st.cycle_counter_inc[31:0];
counter l_delay;
event e_ff := ff == 1'b1;
event e_we := we == 1'b1;
event e_00 := (dut_counter == 8'd00);
event e_01 := (dut_counter == 8'd01);
event e_02 := (dut_counter == 8'd02);
fsm my_FSM {
initial S00;
state S00 { if (e_00 occurs 1) then goto S01 }
state S01 { if (e_01 occurs 1) then goto S02 }
state S02 { if (e_02 occurs 1) then goto S03 }
state S03 { if (e_18 occurs 1) then goto S_Notify }
state S_Notify {
notify
}
}
endmodule
```

ECO Flow for Dynamic Triggers

When using Dynamic Triggers in your design, ECO Flow is supported to modify existing Dynamic Triggers or create a new Dynamic Trigger using the ECO transactor architecture. When you specify new input to Dynamic Triggers, the inputs are connected ECO transactors and Dynamic Triggers are calculated at workstation.

This section provides information on the following:

- ECO-Aware Compilation
- Creating a New Trigger
- Modifying an Existing Trigger
- Checking the ECO Support

ECO-Aware Compilation

To enable ECO for Dynamic Triggers, reserve transactor resources in the ECO-Aware original compilation using the following UTF command:

```
eco -reserve_trigger <bit numbers>
```

where,

- `<bit numbers>`: Denotes the supported bit numbers such as 128, 256, 512,1024, 2048, 3072, 4096
- `reserve_trigger`: Allows to reserve resources such as `reserve_xtor`, `reserve_force_assign`, and `reserve_design_change`.

  If more than one of these arguments are specified, the maximum value is considered.

To create a new trigger or modify an existing trigger, write a Tcl file with the following command:

```
zeco_add_trigger < trigger instance name> -input <signal list>
```

where,

- `< trigger instance name>`: Denotes the name of the trigger to add or modify.
- `-input <signal list>`: Specifies a list of signals which is inputs of the dynamic trigger.

Creating a New Trigger

To create a new trigger, use the following command:

```
zeco_add_trigger Counters_Dynamic_Trigger -input
{{hw_top.cycle_counter_in} {hw_top.cycle_counter_out}}
```

ECO adds a trigger named `Counters_Dynamic_Trigger` with inputs `hw_top..cycle_counter_in` and `hw_top..cycle_counter_out`.

Modifying an Existing Trigger

To modify the inputs of an existing trigger, use the following command:

```
zeco_add_trigger Counters_Dynamic_Trigger -input
{{hw_top..cycle_counter_in} {hw_top..cycle_counter_out}}
```

ECO replaces the trigger `Counters_Dynamic_Trigger`. So the inputs became `hw_top..cycle_counter_in` and `hw_top..cycle_counter_out`.

Checking the ECO Support

To check the ECO support, use the following command:

```
zECO -z zcui.work -checkECO -signal -check_type trigger_src
```

Where,

- `-checkECO`: Reports if all signal in `hw_top` can be an input of a Dynamic Trigger.
- `-check_type trigger_src` : Specifies the type for checking the ECO support. It is set to `debug`, if unspecified.

Additional zECO Usage

This section provides information on the following:

- Checking ECO Support
- General Usage

Checking ECO Support

For a better turn-around-time with ZeBu ECO flow, it is recommended to check the ECO support before modifying you source file for incremental compilation or modifying the CEL file for Runtime Trigger.

It is strongly recommended to use `-checkECO` before launching the ECO incremental flow, as follows:

```
zECO -checkECO -z <database> -signal <signal name or scope name>
 -signal_file <filename> -depth <depth> -check_type <force/debug>
```

Where,

- `-checkECO`: Specify to run **zECO** in check ECOable mode.
- `-signal`: Signal or instance to check.
- `-depth`: Depth to query for instances; if it's not specified, default value is 1.
- `-check_type`: Specify the type to check; if it's not specified, default value is debug.
- `-signal_file`: A text file; each line contain signal name and depth. If depth is not specified, the default value is 1. Both comma separated or space separated is supported.
- `-Output`: All queried signals with ECOable information, output to console/`zECO.log`.

Example 1

```
 zECO -z zcui.work -checkECO -signal hwtop.dut -depth 2 -check_type force
```

The tool reports all signals within depth 2 of *hwtop.dut* with force ECOable information.

Example 2

```
zECO -z zcui.work -checkECO -signal_file siglist -check_type force
```

General Usage

To view the usage of the `zECO` general options and environment options, use the following command:

```
zECO  [<General options>] [<Environment options>] -z <dirname> [-do
 <file>]
```

Where,

- `<General options>`
  - `-z | -zebu.work <dirname>`: Specifies the compiled ZeBu database; default is `./zebu.work`.
  - `-do < file>`: Specifies the ECO Tcl file for play.
  - `-h`: Shows this help message.
  - `-dryrun`: Reports the ECO summary and generates the related script without running.
  - `-drycont`: Continues ECO processes of the ZeBu database generated in dry-run mode.
  - `-dest <dirname>`: Specifies the destination directory.

    Note:

    If this option is specified, the ECO result is saved in the specified directory.
- `<Environment options>`
  - `-vivado <vivado executable>`: Specifies the Vivado executable. This can override the Vivado present in `ZEBU_ROOT` or specified by the environment variable, `VIVADO`.
  - `-grid <grid command>`: Specifies the grid commands.

    Note:

    This option can override the grid command specified by the environment variable, `REMOTECMD`.

Limitations

The following are the limitations of `zECO`:

- **zCui**: `RecompileOnError` in the non-overwriting mode is not allowed.
- **VCS**:
  - Only the following statements are parsed in the VCS stage for this flow:
    - `$dumpvar`
    - `$dumpports`

- **zECO**:
  - There are chances that the incremental Place and Route (PNR) might fail due to routing failures. If PNR runs more than 120 minutes, **zECO**treats it as a failure to prevent stops responding.
  - Consider the routability of the design. The ECO signals of each chip are suggested to be <1000. If it exceeds this number, **zECO** displays the warning messages.
  - New debug signals might not support ECO because of the following:
    - Wildcard in signal/instance name is not supported.
    - Signal is not in the design.
    - Signal is optimized.
    - Fwc/QiWc resource is not sufficient.
  - New forced signals might not support ECO because of the following:
    - Support is limited to dynamic force (*zforce*).
    - Signal is not in the design.
    - Signal is optimized.
    - Signal is a composite type.
    - Signal appears in multiple locations.
    - Force resources are not sufficient.

	Support Since 2018.09-SP1
`zforce`	Yes
`-fnmatch`	No
`-mode DYNAMIC`	Yes
`-mode STATIC`	No
`-module <string>`	No
`-object_not_found fatal`	Yes
`-object_not_found warning`	No
`-pin <list>`	Yes
`-pin_file <file>`	Yes

	Support Since 2018.09-SP1
`-pin_only`	No
`-rtlname <list>`	Yes
`-rtlname_file <file>`	Yes
`-signal <list>`	No
`-sync_enable`	Yes
`-wire <list>`	No
`-wire_file <file>`	No
`force_dyn`	No
`force_assign`	No

- For the `dumpvar` instance, only signals in the instance that is supported with ECO are added.
- The environment variable `ZEBU_DV_QA_ID_CHECK` is not supported.
- Only signal addition is supported and the removal of probed signal is not supported.
- Value-Set:

  Due to resource limitation, `dumpvar` signals are added to implicit Value-Sets (different from ones specified in the `dumpvar` statement). These implicit Value-Sets are automatically added at runtime when you add explicit Value-Set. You do not have to add implicit Value-Sets at runtime.
- **zECO** cannot add force to a constant signal in design.

  For example, adding *zforce* to "*wire en = 1*;" using *zECO* shows the following compile error:

  ```
  warning in zECO [ZECO0332W] : Could not find signal 'hw_top.en' in the
   database, the signal could be optimized or transformed.
  fatal error in zECO [ZECO0332F] : Could not find required signal(s) in
   the database.
  ```

  The same error appears for the reg type. That is, "`reg en; initial en = 1;`".

  A constant signal should be '`zforce`' in the first compile irrespective of the signal type (`wire`, `reg`, and so on).

- The newly added debug signals are supported only if they can be mapped from RTL to gate, are inferred to FPGA FF/LD, and can be located in `routed.dcp`.
- If the newly added debug signals are inferred to memory, they are not supported.

4
Runtime Features

This section provides information about LCA features introduced to improve emulation runtime.

This section describes the following topics:

- Dual zTime Support for Triggers
- Simulated Emulation
- Support for Multiuser on Single Host

Dual zTime Support for Triggers

To improve the emulation time, the dual `zTime` for triggers feature is introduced. This feature allows a higher `driverClk` frequency when no triggers are used. To enable this feature, specify the following UTF command:

```
ztopbuild -advanced_command {zcorebuild_command * {timing_model
 enable_switch=gear}}
```

Modes for Dual zTime Support

The modes are as follows:

- **Accurate:** In this mode, the triggers/monitors in the DUT are added to the `zTime` computational graph leading to a lower `driverClk` frequency.

  C++ Testbench

```
board.hh
/// @param[in] mode Which mode we are enabling. See enum definition.
    ///
    /// @sa ZEBU_TriggerTimingMode
    /// @note Needs to be enabled at compile time to be available.
    void setTriggerTimingMode(const ZEBU_TriggerTimingMode mode);

    /// Get which trigger timing mode ZeBu is currently running on.
    /// @return the mode, see enum definition.
    /// @sa Board::setTriggerTimingMode.
    ZEBU_TriggerTimingMode getTriggerTimingMode();
```

- **No trigger support:** In this mode, the triggers/monitors in the DUT are not taken into account when calculating the `driverClk` frequency.

  Enumeration Definition (in $ZEBU_ROOT/include/Types.h)

  ```
  typedef enum
  {
      ZEBU_TM_ACCURATE,
      ZEBU_TM_NO_TRIGGER_SUPPORT
  } ZEBU_TriggerTimingMode;
  ```

  These APIs must be added to the C++ and C testbenches (`Board.hh` and `Types.h`).

Limitations

This feature conflicts with multi-cycle path feature, which reduces the driver clock. An error is displayed when you try to enable both of them at the same time.

Simulated Emulation

Simulated Emulation (SEM) enables early RTL bring-up in emulation by using the VCS simulator.

SEM offers the following advantages:

- Uses the same inputs as ZeBu emulation, such as the RTL design, UTF file, and runtime environment.
- Reduces the turnaround time compared to ZeBu due to faster compilation.
- Allows VCS debug features for analyzing issues.
- Enables significant cost saving for early bring-up:
  - Requires only one host and a VCS license; does not require an emulator.

Although SEM provides a convenient stepping stone to ZeBu as it uses zCui to compile the design, due to lower runtime performance compared to emulation, SEM is not a replacement of emulation.

Note:
SEM is useful for few million cycles to debug basic RTL bugs such as reset and clock generation, connectivity and integration of design blocks, where test failures happen quickly and performance is not the highest priority. SEM enables users to quickly fix and validate the model, allowing quick iterations. When the model is validated in SEM, the user can smoothly transition to ZeBu using the same setup.

For more information, see the following subsections:

- License Requirements for SEM
- Design Compilation with SEM
- Runtime for SEM
- Platform Differences Between ZeBu and SEM
- Limitations of SEM

License Requirements for SEM

You need the following licenses for using SEM:

- `zs_SEM`
- `VCS-APEX-COMPILE`
- `VCS-APEX-RUNTIME`

Design Compilation with SEM

To enable compilation for SEM, use the following UTF command :

```
compile -sem true -sem_params {mode=behavior}
```

If you require additional VCS options for building the SEM model, use the `build_flags` option as follows:

```
compile -sem true -sem_params {mode=behavior,
 build_flags="-debug_access"}
```

ZeBu supports both zcei and RTL clocks. However, SEM supports only RTL clocking. Designs that use `zceiClockPort` for clocking must be modified to use the **clockDelayPort** macros or simulation-like delay control in Verlog.

zCui automatically processes the UTF commands in the project file as applicable in SEM. Some commands such as `synthesis` and backend-related commands are ignored in SEM. Others such as `load_edif` are not supported due to platform-level differences.

Runtime for SEM

SEM supports runtime control through the following:

- **C++ testbench**: For C/C++ testbench, a custom translation layer is provided.
- **ZeBu Runtime Control Interface (zRci)**: **zRci** provides commands to control the emulation runtime in ZeBu. Even though SEM does not require an emulation host to run and can be executed on any host, SEM supports a subset of these **zRci** commands to enable transition between SEM and ZeBu and it is compatible with Tcl 8.6.

For details on using zRci, see the *ZeBu Unified Command-Line User Guide*.

SEM does not require an emulation host to run and can be executed on any host.

There are Zebu specific technologies around runtime (for example, power-aware emulation) and debug (for example, FWC/readback, coverage, runtime performance) that need to be handled differently because SEM is running a VCS simulation. Usability of these features may be extended for SEM to provide simulation-like flexibility in a **zRci** based testbench environment.

For details on VCS debug capabilities, see the *VCS documentation*.

Platform Differences Between ZeBu and SEM

Due to fundamental differences between a simulator and an emulator platform, cycle-by-cycle match between the two is not guaranteed. Few ZeBu centric features, such as zcei transactors, run manager, and designFeatures file, are not supported architecturally.

Limitations of SEM

VCS as a simulation engine already has assertion, coverage, and low power functionality support. The emulation controls for SVA, coverage, and UPF are specific to ZeBu and are currently not supported in SEM.

Other limitations are:

- Support for `zceiClockPort` and `zceiMessagePorts`
- Support for clock tolerance
- Support for ZeBu public C++ testbench APIs
- Support for save/restore or checkpointing feature in VCS

Support for Multiuser on Single Host

A single host PC can be shared to drive up to 4 emulations on a single ZeBu hardware system. There is a minimum version of the ZeBu driver to be used for this feature.

Benefits

- Reduce infrastructure cost
- Optimize workstation usage

Disadvantages

- Testbenches can interfere because they compete for host resources (RAM, Disk, CPU).
- The host PC must have a bigger form factor to be able to contain all the PCIe boards.

Limitations

- This feature is only supported on ZeBu Server 4, ZeBu Server 5, and ZeBu EP1.
- Up to 4 users per host are allowed, one per PCIe board.
- All PCIe boards must be able to reach all FPGAs of the ZeBu system, which has the following implications:
  - The ZeBu system must include one or two System Hubs.
  - All PCIe boards should be connected to all System Hubs by using their 2 connectors.
  - A maximum of 8 units on ZeBu Server 4 and ZeBu Server 5 are supported.

Use model

To use this feature, set the `ZEBU_PCIE_LOCATION` to the number of the PCIe board that you wish to use.

Example (in bash)

```
export ZEBU_PCIE_LOCATION=1
```

5

Performance Features

This section provides information about LCA features introduced to improve performance.

It covers the following topics:

- ZRM Performance Mode is Automatically Enabled
- Support for SDC-MCP Relaxation on Verilog Force Release

ZRM Performance Mode is Automatically Enabled

The latency optimization of ZRM memories (named `zrm_performance_mode`) is automatically enabled. When enabled, the `zrm_performance_mode` reduces the latency of the ZRM memories by 140ns. The enhanced runtime impacts runtime throughput by 8%.

`zrm_performance_mode` is automatically disabled when the DUT does not contain any ZRM memory. The zTopBuild log file displays the following log:

```
#  step zTopBuild : Enabling 'zrm_performance_mode' (default when design
 contain ZRM)
#  step zTopBuild : *** Switch zrm_transactional_mode    = disable
#  step zTopBuild : *** Switch zrm_performance_mode      = enable
#  step zTopBuild : *** Switch manipulate_bram_for_sw_rw = disable
```

However, you can override the `zrm_performance_mode` setting by specifying the mode explicitly using the following UTF command:

```
set_app_var zrm_performance_mode <true|false>
```

For more details about the UTF command, see the *ZeBu UTF Reference Guide*.

Table 1 Example of Memory Latencies When ZRM Performance Mode is Enabled Vs Disabled

Number of Ports	**zrm_performance_mode is Disabled**	**zrm_performance_mode is Enabled**
1	306	163
2	378	235

Table 1 Example of Memory Latencies When ZRM Performance Mode is Enabled Vs Disabled (Continued)

Number of Ports	`zrm_performance_mode is Disabled`	`zrm_performance_mode is Enabled`
3	449	306
4	520	378

Support for SDC-MCP Relaxation on Verilog Force Release

The instrumentation of Verilog force consists of registers clocked by `driverClk`. When Verilog force statements are used on clock (or async set/reset) signals, then it causes `driverClk` constraints to propagate downstream in the clock cone (or reset cone). In some cases, it is possible to replace `driverClk` with a slower clock in the Verilog-Force instrumentation while maintaining functional equivalence. This clock replacement avoids propagation of `driverClk` constraints in the downstream clock cone.

Requirements

This optimization feature requires:

- Native Verilog force support (not enabled by default)
- SDC-MCP libraries of clock-domain analysis (enabled by default since V-2024.03)
- clock-edge alignment mode 2 (also called `tolerance mode 2`) enabled using the following `ztopbuild` command:

  ```
  ztopbuild -advanced_command {clock_delay -sdc_mcp_mode 2}
  ```

Enabling the Feature

To enable SDC-MCP relaxation, perform the following steps:

1. Use the following UTF command to enable native Verilog force:

   ```
   verilog_force_release -enable_wls true
   ```

2. Set the following environment variable to enable SDC-MCP relaxation:

   ```
   setenv ZEBU_SDC_MCP_OPTIONS=enableMcpFriendlyVerilogForce
   ```

After it is enabled, `zTopBuild.log` displays the following messages:

```
#   step MCPFriendlyVerilogForceTransformation : Starting
#   step MCPFriendlyVerilogForceTransformation : Done in …
```