ZeBu™ Fault EmulationApplication Note

Version V-2024.03-1, July 2024

1 Introduction

Fault injection mechanism is used to find potential failures that may exist within the design of a product or process. Failure mode and effect analysis can be used to identify, prioritize and limit the failures.

Fault simulation with parallel fault and concurrent fault technology can achieve significant acceleration gain compared to simulation.

- Fault simulation at subsystem level might not be the optimum outcome
- Emulation compile-time and rerun overhead should not be competitive for short tests

Fault emulation is the primary area of interest in software context that cannot be effectively covered by simulation. The solution focuses on:

- Unifiedfault database integration
    - Integration with the unified Functional Safety (FuSa) platform with interoperability of other tools for fault campaign management, including fault pruning with static/formal tools
- Performance optimization
    - Effective fault emulation cycle reduction with **zPostRunDebug** (optimization with respect to baseline emulation characteristics)
    - Real-time fault detection monitor
    - Parallel emulation in multisystem and localized replication

This chapter describes the debug features in the following subtopics:

- Prerequisites
- Supported Fault Models

Prerequisites

The prerequisites for fault emulation are as follows:

- **SFF**: You can use Z01X Standard Fault Format (SFF) file for fault generation, detection,and coverage calculations. For details on creating SFF file, see *Z01X™ Simulator Safety Verification User Guide*.
- **FDB Server**: ZeBu fault emulation flow requires a fault database input data to compile and run. You must create an FDB server before starting ZeBu compilation using **zCui**.

The FDB server can be shared between ZeBu and Z01X. FDB should contain the following fault campaigns:

- Fault list
- Observation point list
- Detection point list
- Failure mode list
- Safety mechanism list
- Test case list
- Fault test result list

Note:

Check the following while handling fault mechanisms:

- CPU time impact on compilation
- LUT/REG resources impact: It can affect default or user-defined partitioning. You must verify filing rates.

Supported Fault Models

The following table lists the supported fault models.

Fault Models	Supported	Comments
Stuck-at (SA)	Yes	Supported signal type: • Flop • Port • Array • Variable • Wire • Assign
Transient	Yes	Supported signal type: • Flop • Port • Array • Variable

Where,

- **Stuck-At Faults** refer to stuck at 0 and stuck at 1 faults.
- **Transient Faults** refer to faults that occur once and then disappear.

Note:

If a fault model is not supported, the fault model is skipped and a warning message is reported by the tool.

2

ZeBu Compilation

This section includes the following subtopics:

- Environment Settings
- Enabling Fault Emulation

Environment Settings

You must set the ZeBu environment variables defined in ZeBu documentation.

Enabling Fault Emulation

To enable fault emulation, use the following UTF command:

```
fault_emulation  -fdb_server <FDB server> -fdb_name <FDB name>
-campaign <campaign name> [-dut_path <scope name> ]
```

Options	Description
`-fdb_server`	Specifies the connection information for FDB server. Format is `<server>:<port>`.
`-fdb_name`	Specifies the name of the FDB database.
`-campaign`	Specifies the name of the fault campaign.
`-dut_path`	Specifies path (full hierarchical name) to DUT in ZeBu. Original DUT path of FDB can be checked from SFF file or use FDB tool to list campaign information.
`-object_not_found <string>`	Specifies the message severity for the objects that are not found. Default is *fatal*.

If you are using a compute farm, verify that FDB server is reachable from the remote host.

Note:

Do not use '`localhost`' and replace with `hostname` or IP address.

Debug Configuration

You must enable sniffer in UTF using the following command:

```
debug -use_offline_debug true
```

Feedback

3 zFaultEmu

ZeBu runtime utility **zFaultEmu** supports the following fault emulation features:

- **UnifiedFDB** flow
  - Input data from FDB to extract DP/OP lists and tests
  - Update the fault test result from waveform comparison and write back to FDB
- **At speed zPRD-**based fault emulation optimization
  - Specify replay cycle/time without any fault to generate GM (golden machine) waveform
  - Specify replay cycle/time with fault from FDB to generate FM (fault machine) waveform
  - Capture FM waveforms from hardware to compare with GM waveforms by high performance fault detection monitor
- Performance optimization
  - High performance fault detection monitor with capability to stop early and reduce emulation cycles
  - Fault activation detection to reduce emulation cycles
  - Support parallel capabilities to handle large number of tests and multiple ZeBu hosts
- **Testbench Support**: Following testbench technologies are supported:
  - zRci
  - C/C++
  - SimXL

This section includes the following subtopics:

- zFaultEmu Flow
- Invoking zFaultEmu

Feedback

- zFaultEmu Outputs
- Using Static/Dynamic Trigger with zFaultEmu
- FDB Status

zFaultEmu Flow

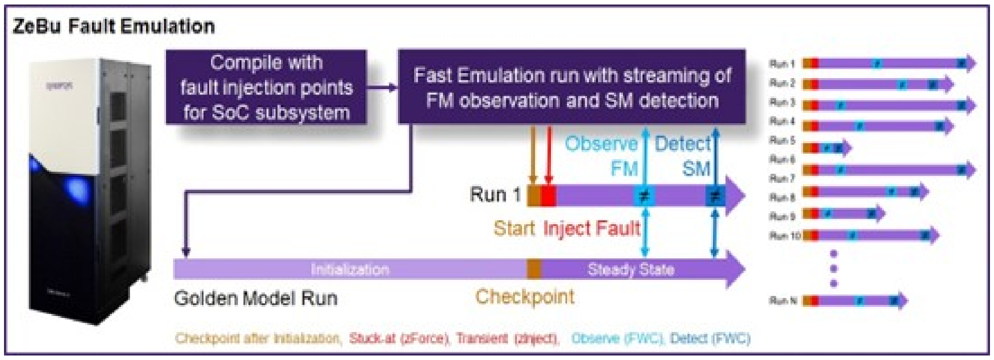
The **zFaultEmu** flow runs all faults as follows:

1. **Play initial run**:Run the testbench to generate testbench stimulus data in sniffer frame.
2. **Play golden run**:Convert sniffer frame to `prd.work` and replay it to generate golden machine waveform.
3. **Play fault run**: Replay `prd.work` by injecting a fault and catch the waveform in memory to compare with golden machine waveform to see if the fault can be detected.

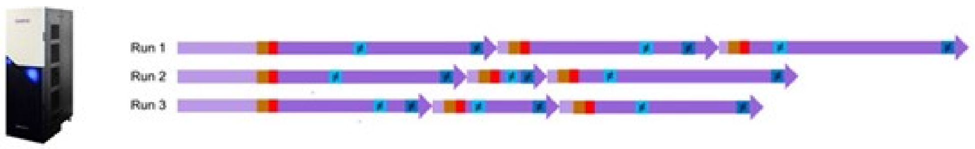
   Each fault submits a signal play run to play.

   After all faults are tested, summarize all the test results and update to FDB.

Figure 1 Fault Emulation Flow


ZeBu Fault Emulation
Compile with fault injection points for SoC subsystem
Fast Emulation run with streaming of FM observation and SM detection
Observe FM
Detect SM
Run 1
Start
Inject Fault
Initialization
Steady State
Golden Model Run
Checkpoint
Checkpoint after Initialization, Stuck-at (zForce), Transient (zInject), Observe (FWC), Detect (FWC)
Run 1
Run 2
Run 3
Run 4
Run 5
Run 6
Run 7
Run 8
Run 9
Run 10
Run N

You can define the number of faults handled per run.



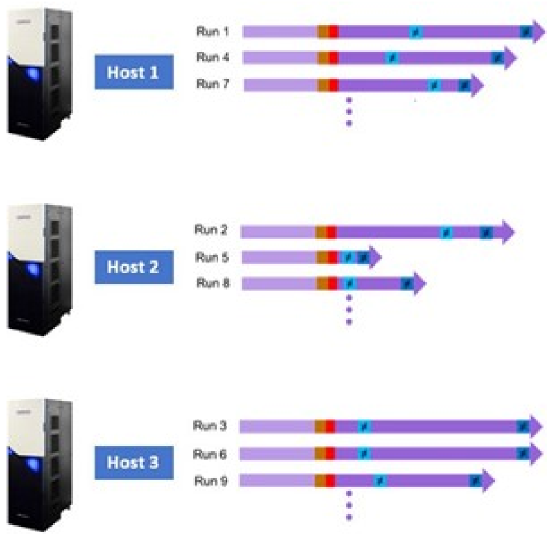
Multiple ZeBu Hosts

ZeBu can handle multiple remote hosts to distribute all fault run tasks on every host in parallel to reduce total wall time (not for initial run and golden run as they are prerequisites for the fault run).

You can set maximum faults on each host and timeout value to avoid timeout issues caused by a single host for too long.

Total wall time is reduced as per the corresponding number of hosts.

Figure 2 Multiple Hosts



Directory Structure

FuSa directory is the root fault emulation directory.

```
FuSa
├──FaultEmu_zRci_db
│  ├──golden.ztdb # Golden run waveform
│  └──prd.work
│
├──Sniffer_dir
│  └──frame_0 # Sniffer frame for replay
│
├──FaultRun_<number> # Fault Run working directories
│  ├──FaultEmu_zRci_db
│  └──prd.work
│
├──ActivationStatus
└──zFaultEmuSummary.rpt # Summary log file
```

Invoking zFaultEmu

Before invoking `zFaultEmu`, ensure the following:

- FDB server connection is ready
- **zCui** compilation is done

  To invoke `zFaultEmu`, use the following command:

  ```
  zFaultEmu [General Options] [FDB Options] [Emulation Options]
  ```

  Where,

  - `[General Options]`supports the following options:
    - `-z, -p, --zebu.work <zebu.work>`: Specifies the path to your `zebu.work`directory. Default is `<PWD>/zebu.work`, if this argument is skipped.
    - `-l,--log <log file>`: Alternative log file. Default is `zFaultEmu.log`.
  - `[FDB Options]`supports the following options:
    - `--fdb_import<file>`: Runs **zFaultEmuin** offline mode by importing all object lists from a file instead of fault database. In this mode, `--fdb_server,--fdb_name, --fdb_timeout,and --dut_path` are ignored.
    - `--fdb_export<file>`: Exports all object lists to a file before **zFaultEmu** ends program. The outputted file can be used to imported by `--fdb_import`again.
    - `--fdb_server<host:port>`: Specifies the host where fault server is running.
  - `[Emulation Options]` supports the following options:
    - `--output_dir<path>`: Specifies the working path to run fault emulation flow. Default path is `./Fusa/`.
    - `--inject_when_trigger <trigger name> [<dynamic trigger expression>]`: Specifies static/dynamic trigger to determine when to inject.
    - `--testbench<file>`: Specifies the testbench to run and create sniffer frame.
    - `--start_cycle<cycle>`: Specifies the snapshot start cycle to sniffer frame and replay.
    - `--end_cycle<cycle>`: Specifies the snapshot end cycle to sniffer frame and replay.
    - `--inject_cycle<cycle>`: Specifies delay time when faults are injected to the circuit. By default, faults are injected at cycle 0.

- `--timeout_cycle<cycle>`:Specifies timeout cycle in replay flow.
- `--start_time <time> <unit>`: Specifies the snapshot start time to sniffer frame and replay. Allowable unit specifiers include: `s/ms/us/ns/ps/ fs`. Only available with RTL clock database.
- `--end_time <time> <unit>`: *(Optional)* Specifies the snapshot end time to sniffer frame and replay. Allowable unit specifiers include: `s/ms/us/ns/ps/fs`.

  Note:

  If the test time duration = 1000ps and `zFaultEmu` is run with the following options:

  - `{*}Case 1{*}: zFaultEmu --start_time 500ps --inject_time 1500ps --end_time 2000ps ......`

    A message should be displayed to the user highlighting that the inject time (1500ps) is **higher than** the test time duration (1000 ps).

  - `{*}Case 2{*}: zFaultEmu --start_time 1100ps --end_time 2000ps ......`

    A message should be displayed to the user highlighting that the start time (1100ps) is **higher than** the test time duration (1000 ps).

- `--inject_time <time> <unit>`: Specifies delay time when faults are injected to the circuit. By default, faults are injected at time 0. Allowable unit specifiers include: `s/ms/us/ns/ps/fs`.
- `--timeout<time> <unit>`: Specifies timeout in replay flow. Allowable unit specifiers include: `s/ms/us/ns/ps/fs`.
- `--sniffer_frame<num>`: Splits `start_cycle`to`end_cycle`into number of frame. The default value is 1.
- `--clock<clock>`: Specifies primary clock to run.
- `--fwcSamplingClock<clock>`:Specifies sampling clock for FWC.
- `--disable_early_abort`: Disables early-abort when any fault is detected. By default, the early abort is turned on.
- `--keep_waveform`: Retains the waveform file generated in last fault runs. By default, old waveform file are not overwritten.

    - `--debug_fwc_valueset <valueset1> [<valueset2> <valueset3>...]`: Specifies a list of FWC value set. These value sets are automatically enabled in emulation flow to dump corresponding waveform files.
    - `--debug_qiwc_valueset <valueset1> [<valueset2><valueset3>...]`: Specifies a list of QiWC value set. These value sets are automatically enabled in emulation flow to dump corresponding waveform files.
  - `[Flow Control Options]` supports the following options:
    - `--zs_cmd<cmd>`: Specifies the command to access ZeBu. If this command was specified multiple times, `zFaultEmu`submits the task by using the command cyclically. The maximum number of parallel tasks are equal to the command number.
    - `--zs_grid_cmd<cmd>`: Specifies the grid command to access the ZeBu. With this command, parallel tasks are supported.
    - `--parallel_num <number>`: Specifies the parallel task number. This option is effective if `--zs_grid_cmd` is specified. Fault lists are partitioned into multiple jobs according to parallel number or maximum fault number
    - `--max_fault<number>`:Sets maximum fault number to run in each fault run. `zFaultEmu` queues up the extra faults and run them after a fault run is finished. Default is 0 which means each fault run has no limited number.
    - `--timeout_run <minutes>`: Sets timeout in minutes of all fault runs. `zFaultEmu` stops the external process if it reaches timeout minute number. By default, the value is 0 which means no timeout checking.
    - `--sniffer_dir <path>`: Specifies the path of sniffer database as stimulus data. `zFaultEmu` starts from golden run if this option is specified.
    - `--skip_fault all|<faultId1> [<faultId2> <faultId3> ...]`: Specifies fault list by `faultId`.These faults are not injected in the fault run.
    - `--rerun_fault all|<faultId1> [<faultId2> <faultId3>...]`: Specifies fault list by `faultId`(all indicates all faults). These faults are injected again in the fault run even though they were already run or is set to skip (higher priority than `--skip_fault`).
    - `--summary <directory>`: Specifies a fault emulation directory, collects all FaultStatus under specified `<directory>/FaultRun_*/` into `zFaultEmuSummary.rpt`.

zFaultEmu Outputs

This section provides the information on the following:

FDB Update

zFaultEmu updates FDB faults statuses with following values:

- `OD`:The fault is Observed and Detected
- `ND`:The fault is Not Observed but Detected
- `ON`:The fault is Observed but Not Detected
- `NN`:The fault is Not Observed and Not Detected

Faults are skipped in the following cases:

- Fault signal is not activated and has the same value as forced value
- Collapsing fault
- Unsupported fault types

zFaultEmu also updates the FDB faults **Start Time**, **End Time** and **Diagnostic Coverage**.

These FDB fields can be checked using the **Verdi Fault Analysis Add-on** (`set VERDI_FUSA_DEBUG =1`).

Fault Emulation Summary Log File

zFaultEmu report summary is written in the `zFaultEmuSummary.rpt` file. The report contains the following data:

- Information of FDB / Campaign / FailureMode / Testcase
- Fault emulation runtime summary statistics
    - Wall time/Elapsed time/State of initial run, golden run, and fault run
- Correct fault list and result
    - Fault ID, signal name, detection result, runtime breakdown
- Fault coverage number and rate

Examples

Example of fault emulation summary is as follows:

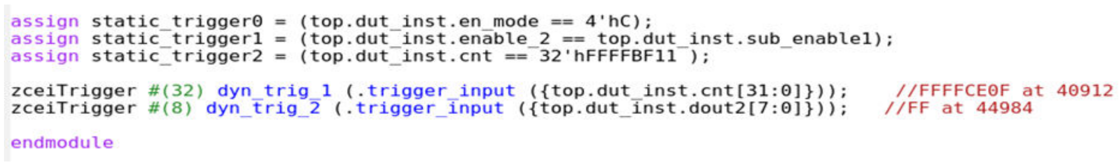
```
FaultId=67 SigPath="emu_top.dut.Multiplier.Yin[9]" Value=ONE
 TaskRun=OK FaultRunTask=0 Detected=1 Observed=1 ReplayCycle=0~9999
 ReplayExec(WaitHW/ InitHW/Frame)=0:00:00/0:00:04/0:00:01 Result=TestPass
 State=OD
………..
# Fault run tasks (Parallel/Total)=2/2
………
# Initial run elapsed=0:01:16
# Golden run elapsed=0:01:14
# Fault run elapsed=0:05:27
# Total emulation elapsed=0:07:57
#

# Total Faults: 67
# Faults injected this time: 65
# Faults skipped(already tested): 0
# Faults skipped(manually): 0
# Faults skipped(non-forcible): 0
# Faults skipped(inactivated):  2
# Not Observed Not Diagnosed NN  5 7.46%
# Not Observed Diagnosed ND  0 0%
# Observed Not Diagnosed ON  1 1.49%
# Observed Diagnosed OD  61 91.04%
```

Using Static/Dynamic Trigger with zFaultEmu

Perform the following steps to use static/dynamic trigger with **zFaultEmu**:

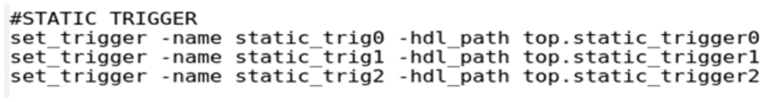
1. Define in your top-level SystemVerilog file one or more static/dynamic triggers.

   Example: There are 3 static triggers and 2 dynamic triggers:



```
assign static_trigger0 = (top.dut_inst.en_mode == 4'hC);
assign static_trigger1 = (top.dut_inst.enable_2 == top.dut_inst.sub_enable1);
assign static_trigger2 = (top.dut_inst.cnt == 32'hFFFFBF11 );

zceiTrigger #(32) dyn_trig_1 (.trigger_input ({top.dut_inst.cnt[31:0]}));    //FFFFCE0F at 40912
zceiTrigger #(8) dyn_trig_2 (.trigger_input ({top.dut_inst.dout2[7:0]}));   //FF at 44984

endmodule
```

2. In the UTF file, add the following lines for Static triggers only:



```
#STATIC TRIGGER
set_trigger -name static_trig0 -hdl_path top.static_trigger0
set_trigger -name static_trig1 -hdl_path top.static_trigger1
set_trigger -name static_trig2 -hdl_path top.static_trigger2
```

3. Compile the design and testbench environment.

4. Use the `--inject_when_trigger` option with **zFaultEmu**:

   Example for static trigger to run `static_trig0` only:

   ```
   zFaultEmu --fdb_import ../run_zebu/test_fusa_all.txt --campaign
    fc_test --fm FM1 --testcase FM1_Testcase1 --testbench
    'zRci -testbench "zEmiRun -z ../zcuiUC.work/zebu.work
    -l ./tb.so -x ./xtor.so:xtor"  -do "../run_zebu/zrci.tcl"'  -p
    $(PWD)/$(ZCUIWORK)/$(ZEBUWORK) --clock tickClk --fwcSamplingClock
    tickClk --inject_when_trigger  static_trigger0  --end_time 21000 ns
    --output_dir Fusa_offline
   ```

   Example for dynamic trigger to run `dyn_trig1` only:

   ```
   zFaultEmu --fdb_import ../run_zebu/test_fusa_all.txt --campaign
    fc_test --fm FM1 --testcase FM1_Testcase1 --testbench
    'zRci -testbench "zEmiRun -z ../zcuiUC.work/zebu.work
    -l ./tb.so -x ./xtor.so:xtor"  -do "../run_zebu/zrci.tcl"'  -p
    $(PWD)/$(ZCUIWORK)/$(ZEBUWORK) --clock tickClk --fwcSamplingClock
    tickClk --inject_when_trigger top.dyn_trig_1 "top.dut_inst.cnt[31:0]
    == 32'hFFFFF63D" --end_time 21000 ns --output_dir Fusa_offline
   ```

5. Run the fault emulation.

Note:

- At fault emulation, only one trigger condition is used for all faults.
- For static trigger, recompile the design each time the trigger signal value changes.
- For dynamic trigger, no need to recompile the design. Update the **zFaultEmu** command line only with the new signal value to trigger.
- Example error message when trigger is not fired:

  ```
   /
  remote/vgpln1/users/millerwu/PP/ZEBU.2020.03/regression.B4/uni
  t_ZEBU/qa_zebu/FuSa_test_16CLK_zEmiRun_UTF/ZEMI3/run_gold/femu
  _static2.log
  ```

  `-- ZeBu : zRci : ERROR : ZPRIV3111E`: Some triggers `(00000000000000000000000000000010)` cannot get fired during initial run.

- Check the related tcl/log for initial run stage.

FDB Status

For the online mode, user can run multiple testcases with different `start times` and `end times` for the same design using the same faults list. All results are collected and merged automatically in the FDB. This helps increase the fault coverage.

Consider the following example. For the same fault, if the result of the first testcase is `NN`, and the result of the second testcase is `OD`, then the result reflected in the FDB is `OD`.

User can check the faults status with more details using the FCM shell. The FCM shell can be called using the command line "`vc_fcm`":

- To check the status of all faults with promotion: `%>show_fault_results`.
- To check the status of all faults without any promotion: `%>show_fault_results -raw`.
- To check the merge for one fault or a list of faults: `%>show_fault_results -fid {list of fault ID} -raw`.

The FDB provides also a command line to generate a text report in SFF format. By default, the report name is `<campaign name>_report.sff`.

```
vc_fdb_report  -campaign ${SNPS_FDB_FAULT_CAMPAIGN} -showfaultid
```

4 Debug

This section provides information on the following:

- Generating ZeBu Waveform
- Searching Fault Details

Generating ZeBu Waveform

By default, ZTDB is generated in `Fusa/FaultEmu_zRci_db/golden.ztdb`.

Fault Run

ZTDB is not generated by default because ZeBu captures waveform only in memory. You must specify `--keep_waveform`to dump ZeBu waveforms in the file directory. Waveform of each fault is generated in the following location:

`Fusa/FaultRun_[TaskRunId]/FaultEmu_zRci_db/[FaultId].ztdb`

`FaultId` and its corresponding `TaskRunId` is located in `zFaultEmuSummary.rpt`.

ZTDB is not generated by default because ZeBu captures waveform only in memory. You must specify `--keep_waveformtodump` ZeBu waveforms in the file directory. Waveform of each fault is generated in the following location: `Fusa/FaultRun_[TaskRunId]/FaultEmu_zRci_db/[FaultId].ztdbFaultId` and its corresponding `TaskRunIdislocated` in `zFaultEmuSummary.rpt`.

Searching Fault Details

When a fault is triggered, you can find the cycle and its corresponding status. Triggered point and DP/OP signal name of the fault is available in the following file: `Fusa/FaultRun_[TaskRunId]/FaultEmuMonitor.log`.

Example 1   FaultEmuMonitor.log

```
FaultEmuMonitor start for F3103! FaultEmuMonitor enable correctly
Found mismatched signal @62075 Cycle in DP
 "emu_top.ddr_chip.u_DWC_ddr.reg_par_err_intr"
```

```
Found mismatched signal @62063 Cycle in OP
 "emu_top.ddr_chip.u_DWC_ddr.U_apb_slv.reg_par_err_pulse_cclk"
 FaultEmuMonitor disable correctly
```