



# Application Note

**Arm® CoreSight™ ELA-600**

Version 1.0

**Non-Confidential - Published**

#### Release Information

The following changes have been made to this Application Note.

Document History			
Date	Issue	Confidentiality	Change
[Publish Date]	A	Non-Confidential - Published	First release

#### Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT.

For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at <http://www.arm.com/company/policies/trademarks>.

Copyright © 2018 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.  
110 Fulbourn Road, Cambridge, England CB1 9NJ.  
LES-PRE-20348

**Confidentiality Status**

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

**Product Status**

The information in this document is Final, that is for a developed product.

**Web Address**

<http://www.arm.com>

# Contents

## Arm® CoreSight™ ELA-600

<b>1</b>	<b>Conventions and Feedback .....</b>	<b>5</b>
<b>2</b>	<b>Preface .....</b>	<b>7</b>
2.1	References .....	8
2.2	Terms and abbreviations .....	9
<b>3</b>	<b>Introduction .....</b>	<b>10</b>
<b>4</b>	<b>Problem Description .....</b>	<b>11</b>
<b>5</b>	<b>Solution .....</b>	<b>12</b>
5.1	CoreSight ELA-600.....	13
5.2	The System .....	14
5.3	DS-5 Flow .....	15
<b>6</b>	<b>Appendix: Text Object Examples .....</b>	<b>24</b>
6.1	Test Code .....	25

# 1 Conventions and Feedback

The following describes the typographical conventions and how to give feedback:

## Typographical conventions

The following typographical conventions are used:

<code>monospace</code>	denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
<u><code>monospace</code></u>	denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
<i>monospace italic</i>	denotes arguments to commands and functions where the argument is to be replaced by a specific value.
<b>monospace bold</b>	denotes language keywords when used outside example code.
<i>italic</i>	highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
<b>bold</b>	highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for Arm® processor signal names.

## Feedback on this product

If you have any comments and suggestions about this product, contact your supplier and give:

- Your name and company.
- The serial number of the product.
- Details of the release you are using.
- Details of the platform you are using, such as the hardware platform, operating system type and version.
- A small standalone sample of code that reproduces the problem.
- A clear explanation of what you expected to happen, and what actually happened.
- The commands you used, including any command-line options.
- Sample output illustrating the problem.
- The version string of the tools, including the version number and build numbers.

## Feedback on documentation

If you have comments on the documentation, e-mail [errata@arm.com](mailto:errata@arm.com). Give:

- The title Arm® CoreSight™ *ELA-600 Embedded Logic Analyzer Application Note*
- The number, ARM-ECM-0442477, A.
- If viewing online, the topic names to which your comments apply.
- If viewing a PDF version of a document, the page numbers to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Arm periodically provides updates and corrections to its documentation on the Arm Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

#### **Other information**

- Arm Information Center, <http://infocenter.arm.com/help/index.jsp>.
- Arm Technical Support Knowledge Articles, <http://infocenter.arm.com/help/topic/com.arm.doc.faqs/index.html>.
- Arm Support and Maintenance, <http://www.arm.com/support/services/support-maintenance.php>.
- Arm Glossary, <http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html>.

## 2 Preface

This preface introduces the Arm® CoreSight™ ELA-600 Embedded Logic Analyzer Application Note.

It contains the following:

- References on page 8.
- Terms and abbreviations on page 9.

## 2.1 References

- Arm® CoreSight™ ELA-600 Embedded Logic Analyzer Technical Reference Manual (ARM 101088).
- Arm® CoreSight™ LAK-500I Logic Analyzer Kit Integration and Implementation Manual (ARM 100128).
- Arm® AMBA® AXI and ACE Protocol Specification (ARM IHI 0022).
- Arm® AMBA® APB Protocol Specification (ARM IHI 0024).
- Arm® CoreSight™ Architecture Specification (ARM IHI 0029).
- Arm® CoreSight™ SoC-400 User Guide (ARM 100479).
- Arm® CoreSight™ SoC-600 User Guide (ARM 101128).
- Arm® Socrates™ System Builder User Guide (ARM 100328).
- Arm® Socrates™ System Builder Installation Guide (ARM 100329).



## 2.2 Terms and abbreviations

### Signal Group

A group of input signals from the Observation interface. The ELA-600 supports up to 12 *Signal Groups*, each of which is 64 bits, 128 bits, or 256 bits wide, determined by the GRP\_WIDTH parameter.

### Trigger State

One of the eight states that the ELA-600 trigger logic can be in. The *Trigger State* controls which *Signal Group* signals are routed to the comparison logic, target comparison values, comparison and counter control, and output actions. The ELA-600 advances to the next *Trigger State* when its *Trigger Condition* is met.

### Trigger Signal Comparison

The comparison of the *External Trigger Input Signals* and selected *Signal Group* with a target value and mask that is determined by the current *Trigger State*.

### Trigger Condition

When the *Trigger Condition* is met, the ELA-600 generates an *Output Action* and transitions to the next *Trigger State*. If *Trigger Counter Comparison* is enabled, the *Trigger Condition* is met when the *Trigger Counter Comparison* is true. If *Trigger Counter Comparison* is disabled, the *Trigger Condition* is met when the *Trigger Signal Comparison* is met.

### Output Action

The ELA-600 generates an *Output Action* when the *Trigger Condition* is met. The *Output Action* can:

- Drive the STOPCLOCK output for scan-dump analysis.
- Drive a CoreSight Embedded Cross Trigger through CTTRIGOUT[1:0] to a CoreSight *Cross Trigger Interface* (CTI).
- Drive other logic through ELAOUTPUT[3:0].

## 3 Introduction

The Arm CoreSight ELA-600 Embedded Logic Analyzer provides low-level signal visibility into Arm IP and 3rd party IP. When connected to a processor or interconnect bus, it provides visibility of loads, stores, Speculative fetches, cache activity and transaction life cycle, none of which are available through instruction tracing.

CoreSight ELA-600 enables swift hardware assisted debug of otherwise hard-to-trace issues, including data corruption and dead/live locks. As well as accelerating debug cycles during complex IP bring up, it provides extra assistance for post deployment debug.

CoreSight ELA-600 offers on-chip visibility of both Arm and proprietary IP blocks. Trigger conditions can be programmed over standard debug interfaces either directly by an on-chip processor or an external debugger.

This guide is intended to demonstrate how the CoreSight ELA-600 can be used with Arm DS-5 Development Studio to debug a real-world deadlock scenario on a Cortex-A55 + CoreSight ELA-600 + CCI-500 based system, caused by a bus transaction hang emulated by a simulated lock-up using the HLT halt instruction on the core.

## 4 Problem Description

One of the most common deadlock scenarios can be caused when a processor initiates memory transactions to a location in the system in which no bus slave exists or the bus slave has limitations such as not being able to handle burst or wrapped burst transactions. This type of incomplete transaction can ultimately lead to the processor locking-up (deadlock).

In a perfect world, systems should be designed in such a way that all the entire physical memory map is fully populated. Meaning that all memory transactions, to all addresses, will correctly respond with either a valid transaction result or a bus fault. This said, for certain designs this may not always be the case. The aggressive speculation and prefetching performed by Arm processors mean that these memory map “holes” are more likely to be exposed by incorrect software, even if these memory “holes” are not explicitly referenced by software.

Software can prevent this by correctly configuring the MMU translation tables to accurately describe the physical memory map. Software should configure any memory map “holes” as being Invalid. Configuring the MMU this way will prevent the processor from making any physical bus transactions to that location, and ultimately preventing this type of deadlock scenario.

Additionally, data corruption can occur when contents of a memory location are modified outside of expected program/language behavior. This can include unexpected software behavior, incorrect memory management, or hardware malfunctions. Data corruption can occur randomly or systematically, and subsequent use of the corrupted data may have unintended consequences for a system - from minor loss of data to a system crash.

Debugging these types of deadlock scenarios pose an issue when debugging using traditional methods, such as external debug, and instruction / data trace. A processor core which has locked-up due to an incomplete transaction, will likely not be able to enter halt mode debug. Effectively, the external debugger is unable to break the processor and inspect its internal state. Trace capture may still be available but will not provide any record of the speculative or prefetched transaction which may be responsible for deadlock.

## 5 Solution

CoreSight ELA-600 can be used effectively in this scenario to trace the external bus transactions made by the processor (both explicitly and speculatively). This guide intends to showcase the use case scripting capabilities of DS-5 to solve such kind of issues.

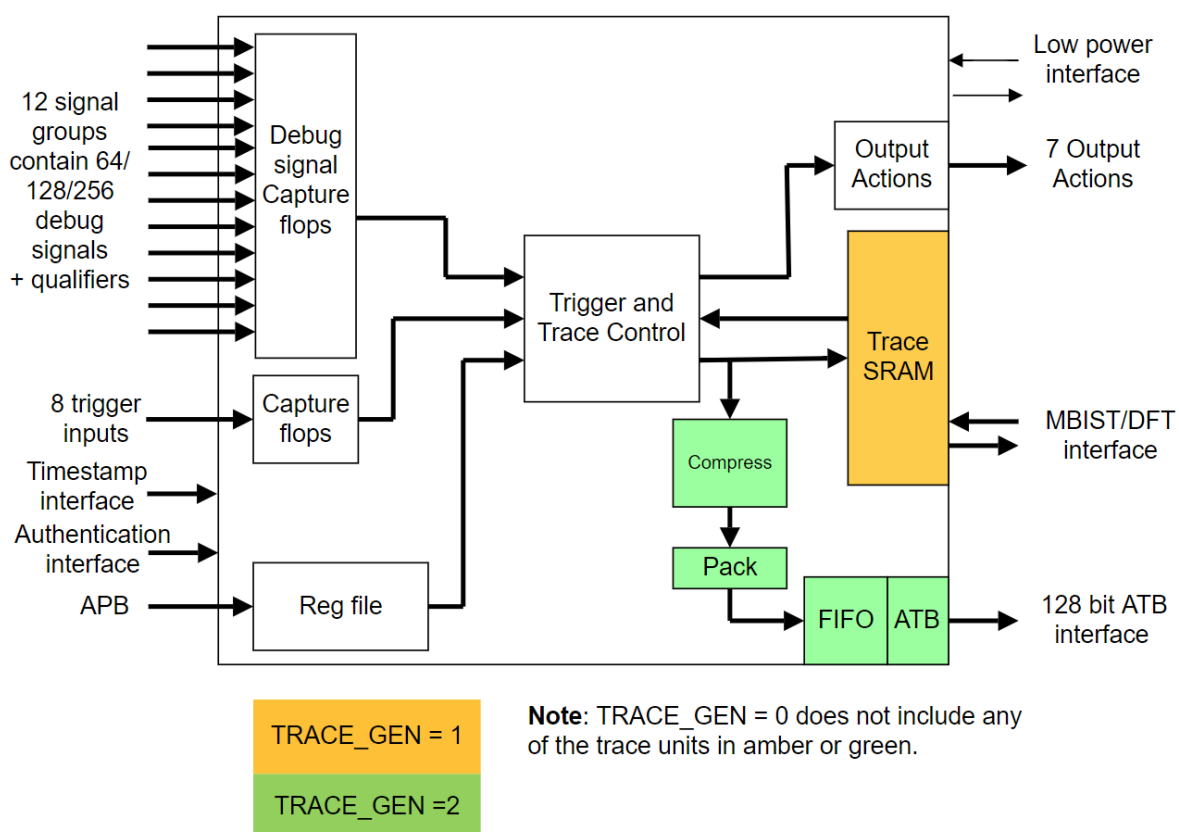
## 5.1 CoreSight ELA-600

The ELA-600 can be implemented with up to 12 Signal Groups, each containing 64, 128, or 256 signals. Which signals are connected to each of the signals in the signal groups will be dependent on the system and the IP that it is connected to. The specific signal interfaces will be documented in the relevant documentation (low-level signal description documents like this are typically not publicly available and are made available only to licensees of the Arm IP). Arm IP connected to an ELA will be supplied with a signal mapping file which documents and annotates the signal group connections for that particular IP, in a machine-readable JSON format. The JSON file can be interpreted by DS-5 to allow seamless debugging of a piece of IP using DS-5 and the ELA. The JSON mapping file specification is available as part to the ELA-600 license bundle. This allows third-party or in-house IP-providers to create custom mapping files for the specific IP to which the ELA is connected.

Signals typically consist of debug signals (status or output), and qualifiers (trigger). Qualifier signals may be required to determine that the debug signal is valid. Debug signals are valid when the qualifier signal(s) are asserted.

### 5.1.1 Differences to ELA-500

The ELA-600 is a superset of the ELA-500 features (SRAM support is the same, without compression) containing improved trace capture and logic analysis, with minimal area (same area as 5 trigger-state, 128-bit ELA-500 with 8.5k SRAM) growth.



The main improvements are the following:

- Supports a configuration that can trace over a 128 bit-ATB interface to allow tracing a greater amount of data without needing an integrated SRAM.
- ATB trace supports data compression, byte packing, and selective trace of bytes in a SIGNALGRP to reduce the amount of trace data.
- ATB configuration supports predicted delta basis change by detecting repeated data patterns to improve compression and reduce payload size.

- Each trigger state comparator can be segmented into multiple 32-bit comparators to allow greater comparison capabilities for logic analysis and tracing capability.
- Supports 5 or 8 trigger states for triggering and trace filtering.
- Supports SIGQUAL qualifier signals for each SIGNALGRP for signals that can be used to qualify valid data without needing to be traced.
- Supports trace of trigger state counters to support measurement of events for support of performance measurements such as latency.
- Supports simultaneous trace of two SIGNALGRPs on the same clock cycle with configurable FIFOs.

Feature	ELA-500	ELA-600
Trigger states	5	8
Embedded RAM config	✓	✓
Data compression		✓
ATB interface		✓
Simultaneous trace of 2 SIGNALGRPs on same clock cycle		✓
Trigger state counters tracing		✓
32-bit segmented trigger state comparators		✓

## Programming

- For SRAM trace, ELA-500 programming will work unmodified with ELA-600
- For ATB trace, any ELA-500 programming sequence will need to be updated to add configuration writes to:
  - Write 1s to the TWBSELn registers
  - Set ATBCTRL[ATID] to a nonzero value
- Trace data will need to be fetched from the connected ATB sink (ETF, ETR/DRAM, DSTREAM...) using DS-5 rather than with the ELA's RRAR/RRDR registers

## 5.2 The System

For the purposes of this demonstration, an FPGA kit has been used containing a Cortex-A55 + ELA-600 + CCI-550 based system. The kit exposes a number of pre-defined debug observation ports to the CCI-550 (Signal Groups) and provides the corresponding JSON signal mapping file.

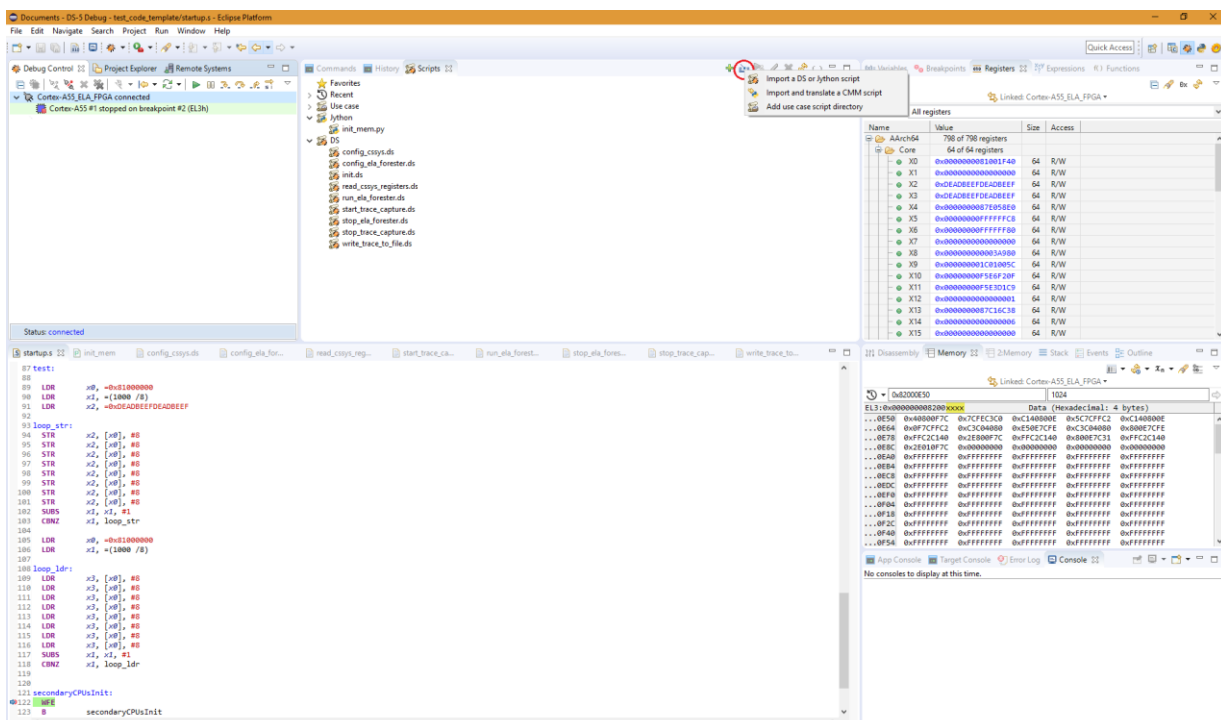
The debug observation ports to the CCI-550 expose, among others, transaction addresses (Address\_Px) and transaction types (Type\_Px) on each of the CCI-550 partitions and qualifiers indicating whether the transactions are valid or not (Valid\_Px). Post-analysis of these captured transactions will help identify which one of them may have caused the lock-up or data corruption.

### 5.3 DS-5 Flow

The following flow was executed in DS-5 using Jython and DS-5 scripts to demonstrate debugging a simulated lock-up or data corruption scenario happening on the Cortex-A55:

1. Connect to the FPGA image on which the discussed Cortex-A55 + CCI-550 + ELA-600 based system was set up
2. After the CPU program execution stopped at the entry point, set up the system for tracing the lock-up or data corruption scenario
  - [JYTHON] Initialize the system memory area are in which the trace data will be captured
  - [DS-5] Configure the CoreSight Subsystem as per section 5.3.2
  - [DS-5] Configure the ELA-600 as per section 5.3.3
  - [DS-5] Enable trace capture and run the ELA-600
3. Resume the code execution on the Cortex-A55 to complete the programmed loads and stores to system memory
  - For the simulated lock-up scenario, lock-up will occur after loads and stores
  - For the data corruption scenario, an AXI write to the target memory location must occur for the core to halt
4. [DS-5] Stop trace capture and disable the ELA-600
5. [DS-5] Dump the captured trace data from the system memory to a binary file and convert it to hex
6. [DS-5] Use the trace decompression script that is shipped with the ELA-600 product bundle and the CCI-550 observation signal mapping JSON file to decode the captured transactions from the trace file and find the last transaction that has been initiated by the Cortex-A55 before the hang

In DS-5 you can write or import your own Jython and DS-5 scripts that you can run from within the tool. To import a script, click on the highlighted icon and select “Import a DS or Jython script”. You can run them anytime by double-clicking on the appropriate script or right-clicking and selecting “Run”. Any logs will be displayed under the Commands tab.



All the source code used in the test scripts are provided under Chapter 8.

### 5.3.1 CPU Test Code

#### Deadlock Scenario

The test code for Cortex-A55 sets up non-cacheable page table entries and runs loads and stores to memory through the CCI-550. After the transactions complete, the HLT (V8 compiled) halt instruction is executed to stop the processor in DS-5 view:

- Set up X0, X1, X2 core registers to be used by the Store instructions:
  - Load the target memory address 0x81000000 into X0 register
  - Load 125 as the loop execution number into X1 register
  - Load the 64-bit data value 0xDEADBEEF into X2 register
- Run a loop of 125 below:
  - 8 times 64-bit Store to non-cacheable memory addresses starting from X0
- Set up X0 and X1 core registers to be used by the Load instructions
  - Reset the memory start address into X0 register
  - Reset the loop execution number into X1 register
- Run a loop of 125 below:
  - 8 times 64-bit Load to X3 core register from non-cacheable memory addresses starting from X0
- Execute the HLT halt instruction to cause a simulated lock-up in the Cortex-A55

#### Data Corruption Scenario

The test code for the Cortex-A55 sets up cacheable page table entries and runs. After completing the transactions, the core idles at a branch-loop, waiting for a 'data corruption' access via AXI. The code below executes two accesses to the target memory location, an exclusive read and a write back/write clean from a cache maintenance instruction. After completing loops of stores and load, the code idles until a third access, an unexpected AXI write, causes a data corruption at the target address. The ELA-600 is configured to count and trace the accesses to the target memory location and triggers a core halt upon the third (corruption) access.

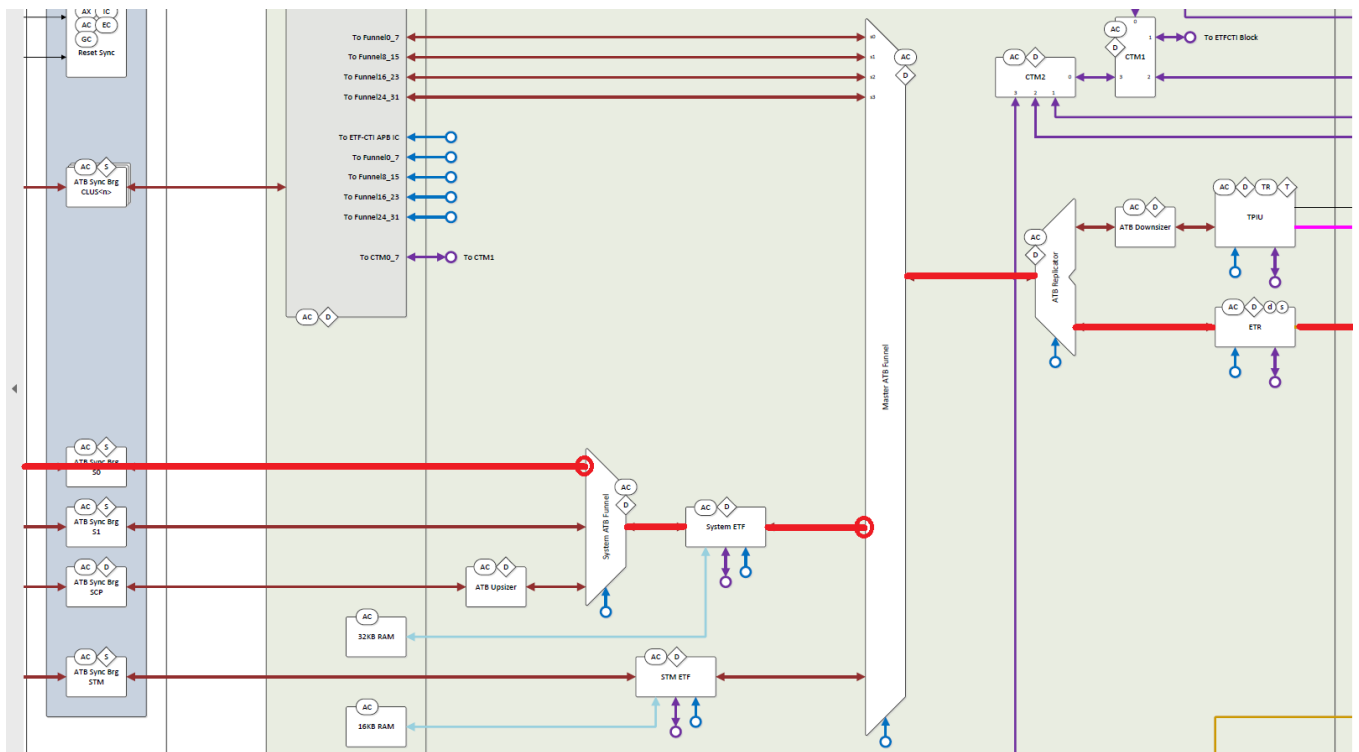
- X0, X1, X2 set up to be used by the load, store, and CIVAC instructions
  - Load the target memory address 0xB1000000 into X0
  - Load X1 with number of loop iterations (125)
  - Load X2 with the initial value (0xDEADBEEF) to be stored at the address in X0
- Single instruction executions
  - Load Exclusive to target address in X0
  - Store Exclusive to target address in X0
  - Cache Clean and Invalidate by Virtual Address to the Point of Coherency (CIVAC)
    - Causes Write Back/Write Clean to the address in X0
- Run a loop of 125
  - 8 times 64-bit store to cacheable memory addresses starting from X0



- Reset memory start address into X0 register
  - Reset loop count into X1 register
- Run a loop of 125
  - 8 times 64-bit load from cacheable memory addresses starting from X0
- Branch-loop indefinitely until 'data corruption' occurs
  - Data corruption will cause ELA to send out a signal halting the core

### 5.3.2 Configuring the CoreSight Subsystem

The trace captured by the ELA-600 is routed to the system memory through the following trace infrastructure:



To ensure that the trace data makes its way properly to the system memory, the following CoreSight components need to be programmed accordingly:

- System ATB Funnel
  - Enable Slave port 0 in Ctrl\_Reg (Funnel Control Register) to route the trace data coming from the ELA-600 to the System ETF
- System ETF
  - Select Hardware FIFO mode in the MODE register to automatically drain the trace data captured in the ETF trace SRAM from the ATB Slave interface through the ATB Master interface
  - Initialize the RAM read and write pointers to 0x0 in the RRP (RAM Read Pointer) and RWP (RAM Write Pointer) registers
- Master ATB Funnel
  - Enable Slave port 4 in Ctrl\_Reg (Funnel Control Register) to route the trace data flushed from the System ETF to the Replicator

- CTI
  - Enable the correct CTIs (CTI3 and CTI4) to allow the propagation of the ELA 'halt' signal to the core, and the core's response signal
- ETR
  - Set the 40-bit target system memory address in the DBAHI (Data Buffer Address High) and DBALO (Data Buffer Address Low) registers
  - Set the size of the RAM area allocated for the ETR in the system memory in RSZ (RAM Size) register
  - Set the write burst length and protection control settings in AXICTL (AXI Control) register for performing AXI transfers
  - Select Circular Buffer mode in the MODE register to automatically route the incoming trace data into the specified system memory location through the AXI master interface

### 5.3.3 Configuring the ELA-600

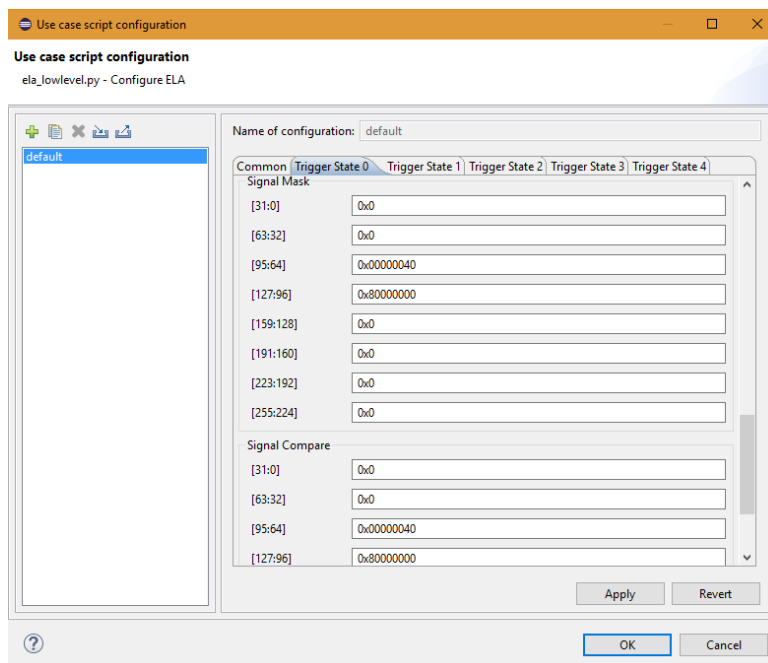
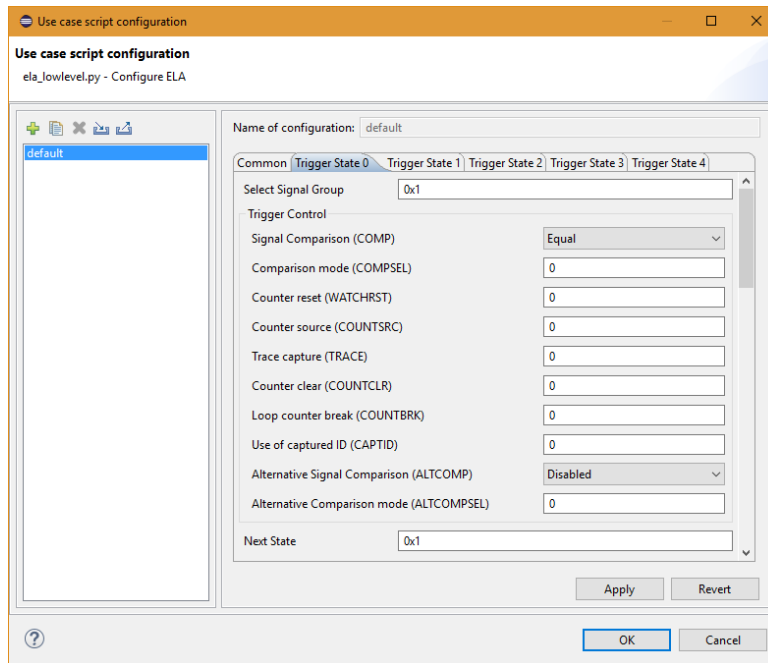
#### Deadlock Scenario

In this particular example, the ELA-600 Trigger State 0 is programmed to always trace based on a signal comparison, whenever there is a valid transaction on CCI-550 Partitions P0 or P1:

- Set up Trigger State 0
  - Select Signal Group 0 for comparisons in SIGSEL0 (Signal select register for Trigger State 0) as the VALID\_P0 and VALID\_P1 transaction qualifiers are exposed on the CCI-550 as part of Signal Group 0
  - Set up SIGMASK0 (Signal Mask register for Trigger State 0) to enable signal comparison only for VALID\_P0 and VALID\_P1 that are at bit positions 69 and 127 respectively in Signal Group 0
    - The certain bit positions can be obtained from the corresponding CCI-550 JSON file or documentation
  - Set the compare value as "1" in SIGCOMP0 (Signal Compare register for Trigger State 0) for VALID\_P0 and VALID\_P1 in Signal Group 0
  - Enable trace in PTACTION (Pre-Trigger Action register) and ACTION0 (Action register for Trigger State 0) in which you can program the initial output action and the output actions for each trigger state
  - Set the next trigger state in NEXTSTATE0 (Next State register for Trigger State 0) to remain in Trigger State 0 after a signal comparison match
  - Set up TRIGCTRL0 (Trigger Control register for Trigger State 0) to capture trace when the selected trigger signals on Signal Group 0 match the configured compare values
  - Set "1" in TWBSEL0 (Trace Write Byte Select register) for each SIGNALGRP0 byte that will be traced on Signal Group 0
- Set the desired trace ID value in ATBCTRL (ATB Control register) to be driven on ATID[6:0]

There will shortly be an ELA-600 GUI configuration utility added into DS-5 as well, where users will be enabled to program different application-specific debug recipes easily on a high-level without the necessity to know in advance which exact registers and bitfield values they need to set in order to achieve the desired functionality.

The use-case scripts will be accessible from DS-5 under Scripts window → Use case.



## Data Corruption Scenario

Trigger State 0 is programmed to count the number of accesses to a specified memory location (0xB1000000) and will move on to the next state once that count is reached. All the transactions of interest are captured on CCI-500 Partition P1. Trigger State 1 will be set up to wait for a response signal from the core once it is stopped. In Trigger State 0 the ELA will trace a count of the cycles between sending the core 'halt' signal in TS0 to receiving the core's response.

- Set up Trigger State 0:
  - Set SIGSEL0 (Signal Select Register) to select Signal Group 0, as the VALID\_P1 and Address\_P1 transaction fields are part of Signal Group 0.

- Set SIGMASK0 (Signal Mask Register) to mask VALID\_P1, Address\_P1, and Type\_P1 for Comparison. VALID\_P1 is at bit position 127 and Address\_P1 is at bit positions 114 to 75. Bit 73 is should be set for Type\_P1.
  - Type\_P1 bit 73 isolates the Exclusive Read and Writes to our targeted Address.
  - Bit positions are specified within the corresponding CCI-550 JSON file or documentation
- Set SIGCOMP0 (Signal Comparison Register) compare values as '1' for VALID\_P1, 'B1000000' for Address\_P1, and '1' for Type\_P1 Bit 73 in Signal Group 0.
- Set COUNTCOMP0 (Counter Comparison Register) value as '3'. When the COUNTCOMP value is matched, the current state will transition to the next state.
  - The third access to the target address is the 'memory corruption' and will cause the next state transition.
- Enable trace in PTACTION (Pre-Trigger Action Register) and ACTION0 (Action Register) in which you can program the initial output action and output action for each trigger state.
  - Also set bit 0 in the ACTION register to drive CTTRIGOUT[0] upon moving to the next Trigger State
  - CTTRIGOUT[0] corresponds to a 'halt' signal on the core
- Set the next trigger state in NEXTSTATE0 to move to Trigger State 1 upon Counter comparison match.
- Set TRIGCTRL (Trigger Control Register) bit [5] COUNTSRC to '1' to have the counter increment on every Trigger Signal Comparison Match. Set bit [3] COMPSEL to '1' to Enable Counters and Select Counter Comparison mode. Set bits [2:0] COMP to '1' to select Signal Comparison type to equal (==).
- Set TWBSEL0 (Trace Write Byte Select register) to '1' for each Signal Group 0 byte that will be traced on Signal Group 0.
- Set up Trigger State 1
  - Set SIGSEL1, SIGMASK1, SIGCOMP1 values all to 0. No comparisons will be made with any Signal Group <n> signals.
  - Set EXTMASK1 (External Mask Register) and EXTCOMP1 (External Comparison Register) to '1' to set up for comparison with the external signal associated with the CTIIN[0] bit.
  - When the 'memory corruption' occurs, the core will be stopped and will send a 'response' signal via the CTTIN[0] bit.
  - Set COUNTCOMP1 (Counter Comparison Register) to a non-zero value to ensure TS1 will count
  - If left as zero, TS1 will not increment the counter
  - Advised to use the maximum count value (0xFFFFFFFF) if a max count is unknown for your trace
  - Set the ACTION1 register to enable trace for Trigger State 1
  - Set the NEXTSTATE1 register to '0' to enable TS1 as the Final State
  - Set TRIGCTRL1 to increment the counter every clock cycle, and capture a counter trace upon a Trigger Signal Comparison match for the selected external signal
  - Set the CNTSEL (Counter Select Register) to capture the counter value for Trigger State 1
  - Set the desired trace ID value in ATBCTRL (ATB Control Register) to be driven on ATID[6:0]

### 5.3.4 Capturing Trace Data

If the system has been configured as it has been described previously, the trace data produced by the ELA-600 from the transactions going through the CCI-550, will be saved accordingly by the CoreSight trace infrastructure at the configured system memory address. The data can then be dumped to a file in multiple formats (bin, ihex, vhx etc.) using the following DS-5 command:

- `dump [<format>] memory <filename> <start_address> <end_address>`

As the ELA-600 trace decompression script that is shipped with the product bundle expects hex characters, the dumped binary trace memory content needs to be converted to hex for example, by using the following command:

- `hexdump -v -e '1/4 "%08x"' -e ""\n" ela600_trace_capture_etr.bin > ela600_trace_capture_etr.hex`

### 5.3.5 Decoding and Analyzing the ELA Trace Capture

#### Deadlock Scenario

The captured ELA trace needs to be decompressed to obtain the ATB trace packets. To achieve this, use the trace decompression script that is shipped with the ELA-600 product bundle:

- `TM310-BU-50000-r0p0-00eac0\cse600\logical\testbench\shared\tools\bin\decompress_atb_cse600.py`

The produced output file will look like the following:

```
Header[7:0]          Data[127:0]          Compression OFF
...
c1      80000104004b46420c00408007480573
c1      80000104004b86420c00408007500966
c1      80000104004bc6420c004080075c015a
c1      80000104004e06400000410013719598
...
```

After obtaining the decompressed ATB trace file, use the trace decoding script that is also delivered as part of the ELA-600 product bundle and the CCI-550 observation signal mapping JSON file to decode the captured transactions and find the last one that has been initiated by the Cortex-A55 before the hang. The trace decoding script can be found in the ELA-600 product bundle on the similar path as the trace de-compression script:

- `TM310-BU-50000-r0p0-00eac0\cse600\logical\testbench\shared\tools\bin\decode_trace_cse600.py`

Running the script will produce a text file of the decoded transactions on the CCI-550 that will show that the last address explicitly written by the core before our simulated lock-up was 0x81001F38.

```
Header => Raw: 8'hcl, Trace type: Data Trace, Trace Stream: 0, Overrun: 0; Data =>
88300102003e72420c0040800f7c975d
P1_VALID      : 1'h1
P1_AXID       : 12'h106
P1_addr       : 42'h81001F38
P1non-secure  : 1'h0 => secure
Type_P1       : 4'h9 => Write No Snoop
P0_VALID      : 1'h0
P0_AXID       : 12'h106
P0_addr       : 42'h81001EF8
P0non-secure  : 1'h0 => secure
Type_P0       : 4'h9 => Write No Snoop
TTID_P1       : 6'h1D
TTID_P0       : 6'h1D
```

In case of having a real-life lock-up, for example, due to the core prefetcher trying to prefetch data from memory locations exceeding the internal SRAM area that should have been configured as invalid in the translation tables,

the ELA-600 would be able to trace the prefetched addresses causing the lock-up the same way as described previously in this document.

### Data Corruption Scenario

The ELA is configured to trace and count the accesses to a target memory address where a data corruption will occur. The decoded ATB file will contain transactions on the CCI-500 that show three accesses to the target address: An Exclusive Read, a Write Back/Write Clean, and a Write No Snoop. The third access is a result of an indirect AXI write to our target address, which causes the data corruption. Additionally, the ATB file will contain a Counter Trace with the cycle count between sending the 'halt' to the core and receiving a core response. An example trace is provided below:

```
Header => Raw: 8'hcl, Trace type: Data Trace, Trace Stream: 0, Overrun: 0 ;
Data => 80300162000003481c0040000060249d
```

```
P1_VALID      : 1'h1
P1_AXID       : 12'h6
P1_addr       : 42'hB1000000
P1non-secure  : 1'h0 => secure
Type_P1       : 4'hD => Exclusive Read
P0_VALID      : 1'h0
P0_AXID       : 12'h40E
P0_addr       : 42'h800000C0
P0non-secure  : 1'h0 => secure
Type_P0       : 4'h2 => Read Shared, Read Clean, Read No Snoop Dirty
TTID_P1       : 6'h12
TTID_P0       : 6'h1D
```

```
Header => Raw: 8'hcl, Trace type: Data Trace, Trace Stream: 0, Overrun: 0 ;
Data => a03001620000002c81c0040000060289d
```

```
P1_VALID      : 1'h1
P1_AXID       : 12'h406
P1_addr       : 42'hB1000000
P1non-secure  : 1'h0 => secure
Type_P1       : 4'hB => Write Back, Writes Clean
P0_VALID      : 1'h0
P0_AXID       : 12'h40E
P0_addr       : 42'h800000C0
P0non-secure  : 1'h0 => secure
Type_P0       : 4'h2 => Read Shared, Read Clean, Read No Snoop Dirty
TTID_P1       : 6'h22
TTID_P0       : 6'h1D
```

```
Header => Raw: 8'hcl, Trace type: Data Trace, Trace Stream: 0, Overrun: 0 ;
Data => 80400162000000650fc005881f9802cf3
```

```
P1_VALID      : 1'h1
P1_AXID       : 12'h8
P1_addr       : 42'hB1000000
P1non-secure  : 1'h1 => non-secure
Type_P1       : 4'h9 => Write No Snoop
P0_VALID      : 1'h0
P0_AXID       : 12'h87E
P0_addr       : 42'hB103F300
P0non-secure  : 1'h0 => secure
Type_P0       : 4'h2 => Read Shared, Read Clean, Read No Snoop Dirty
TTID_P1       : 6'h33
TTID_P0       : 6'h33
```

```
Header => Raw: 8'hc4, Trace type: Counter Trace, Trace Stream: 1, Overrun:
0 ; Data => 0000000300000003000000030000000c
```

```
CNTSEL[0] : 32'hc
CNTSEL[1] : 32'h3
```

```
CNTSEL[2] : 32'h3  
CNTSEL[3] : 32'h3
```

Once the ELA sees there has been a third access to our target location, it will move onto the next trigger state and send a 'halt' signal to the core. The next trigger state will record the time between sending the 'halt' signal and the ELA receiving a response from the core. In the example trace above, CNTSEL[0] recorded 12 cycles passed between the core stop and core response signals.

By utilizing the ELA-600, valuable information about the data corruption at our target address is captured and can be used to aid the debugging process.

## 6 Appendix: Text Object Examples

This chapter demonstrates common text objects, in the following topics:

- *Test Code* on page 25.



## 6.1 Test Code

### 6.1.1 CPU test code

#### Deadlock Scenario

test:

```
LDR    x0, =0x81000000
LDR    x1, =(1000 /8)
LDR    x2, =0xDEADBEEFDEADBEEF
```

loop\_str:

```
STR    x2, [x0], #8
STR    x2, [x0], #8
STR    x2, [x0], #8
STR    x2, [x0], #8
STR    x2, [x0], #8
STR    x2, [x0], #8
STR    x2, [x0], #8
STR    x2, [x0], #8
SUBS   x1, x1, #1
CBNZ   x1, loop_str
```

```
LDR    x0, =0x81000000
LDR    x1, =(1000 /8)
```

loop\_ldr:

```
LDR    x3, [x0], #8
LDR    x3, [x0], #8
LDR    x3, [x0], #8
LDR    x3, [x0], #8
LDR    x3, [x0], #8
LDR    x3, [x0], #8
LDR    x3, [x0], #8
LDR    x3, [x0], #8
SUBS   x1, x1, #1
CBNZ   x1, loop_ldr
```

HLT

#### Data Corruption Scenario

test:

```
LDR    x0, =0xB1000000
LDR    x1, =(1000 /8)
LDR    x2, =0xDEADBEEFDEADBEEF
```

```
LDXR   x5, [x0]
STXR   w4, x2, [x0]
```

```
DSB SY
DC CIVAC, x0
DSB SY
```

loop\_str:

```
STR    x2, [x0], #128
STR    x2, [x0], #128
```

```

STR      x2, [x0], #128
STR      x2, [x0], #128
STR      x2, [x0], #128
STR      x2, [x0], #128
STR      x2, [x0], #128
STR      x2, [x0], #128
SUBS     x1, x1, #1
CBNZ     x1, loop_str

LDR      x1, =(1000 /8)
LDR      x0, =0xB1000000

loop_ldr:
LDR      x3, [x0], #128
LDR      x3, [x0], #128
LDR      x3, [x0], #128
LDR      x3, [x0], #128
LDR      x3, [x0], #128
LDR      x3, [x0], #128
LDR      x3, [x0], #128
LDR      x3, [x0], #128
SUBS     x1, x1, #1
CBNZ     x1, loop_ldr

idle:
B         idle

```

### 6.1.2 Memory initialization test code

```

from arm_ds.debugger_v1 import Debugger
from arm_ds.debugger_v1 import DebugException

if __name__ == "__main__":
    debugger = Debugger()
    ec = debugger.getExecutionContext(0)
    mem = ec.getMemoryService()

    for i in range(0x82000000, 0x82002FFC, 4):
        mem.write(i, "\xFF\xFF\xFF\xFF")

    print "Memory initialization completed!"

```

### 6.1.3 CoreSight Subsystem configuration test code

```

echo System ATB Funnel
echo Set Ctrl_Reg (Funnel Control register) @ DP_1 AP_0 (APB):0x80090000 => 0x00000301
echo ***** [11:8]:HT = 0x3 -> Hold Time: 4 transactions hold time
echo ***** [0]:EnS0 = 0x1 -> Enable slave port 0\n
memory set APB_0:0x80090000 0 0x00000301

echo System ETF
echo Set FFCR (Formatter and Flush Control register) @ DP_1 AP_0 (APB):0x80020304 => 0x00000000
memory set APB_0:0x80020304 0 0x00000000
echo Set MODE (Mode register) @ DP_1 AP_0 (APB):0x80020028 => 0x00000000
echo ***** [1:0]:MODE = 0x2 -> Select the operating mode: Hardware FIFO mode
memory set APB_0:0x80020028 0 0x00000002
echo Set RRP (RAM Read Pointer register) @ DP_1 AP_0 (APB):0x80020014 => 0x00000000
echo ***** [31:0]:RRP = 0x00000000 -> Location in trace memory that is accessed on a subsequent RRD read
memory set APB_0:0x80020014 0 0x00000000
echo Set RWP (RAM Write Pointer register) @ DP_1 AP_0 (APB):0x80020018 => 0x00000000
echo ***** [31:0]:RWP = 0x00000000 -> Location in trace memory that are accessed on a subsequent write

```

```

memory set APB_0:0x80020018 0 0x00000000
echo Set CTL (Control register) @ DP_1 AP_0 (APB):0x80020020 => 0x00000001
echo ***** [0]:TraceCaptEn = 0x1 -> Enable trace capture\n
memory set APB_0:0x80020020 0 0x00000001

echo Master ATB Funnel
echo Set Ctrl_Reg (Funnel Control register) @ DP_1 AP_0 (APB):0x800A0000 => 0x00000310
echo ***** [11:8]:HT = 0x3 -> Hold Time: 4 transactions hold time
echo ***** [4]:EnS4 = 0x1 -> Enable slave port 4\n
memory set APB_0:0x800A0000 0 0x00000310

echo ETR
echo Set DBAHI (Data Buffer Address High register) @ DP_1 AP_0 (APB):0x8012011C => 0x00000000
echo ***** [7:0]:BUFADDRHI = 0x0 -> Upper 8 bits of the 40-bit trace buffer address in system memory
memory set APB_0:0x8012011C 0 0x00000000
echo Set DBALO (Data Buffer Address Low register) @ DP_1 AP_0 (APB):0x80120118 => 0x82000000
echo ***** [31:0]:BUFADDRLO = 0x82000000 -> Lower 32 bits of the 40-bit trace buffer address in system memory
memory set APB_0:0x80120118 0 0x82000000
echo Set RSZ (RAM Size register) @ DP_1 AP_0 (APB):0x80120004 => 0x20000000
echo ***** [30:0]:RSZ = 0x20000000 -> Size of the RAM in 32-bit words
memory set APB_0:0x80120004 0 0x20000000
echo Set AXICTL (AXI Control register) @ DP_1 AP_0 (APB):0x80120110 => 0x00000303
echo ***** [11:8]:WrBurstLen = 0x3 -> Maximum of 4 data transfers per burst
echo ***** [1]:ProtCtrlBit1 = 0x1 -> Controls the value driven on ARPROTM[1] or AWPOTM[1]: Non-secure access
echo ***** [0]:ProtCtrlBit0 = 0x1 -> controls the value driven on ARPROTM[0] or AWPOTM[0]: Privileged access
memory set APB_0:0x80120110 0 0x00000303
echo Set CTL (Control register) @ DP_1 AP_0 (APB):0x80120020 => 0x00000000
echo ***** [0]:TraceCaptEn = 0x0 -> Disable trace capture
memory set APB_0:0x80120020 0 0x00000000
echo Set FFCR (Formatter and Flush Control register) @ DP_1 AP_0 (APB):0x80120304 => 0x00000000
memory set APB_0:0x80120304 0 0x00000000
echo Set MODE (Mode register) @ DP_1 AP_0 (APB):0x80120028 => 0x00000000
echo ***** [1:0]:MODE = 0x0 -> Select the operating mode: Circular Buffer mode
memory set APB_0:0x80120028 0 0x00000000
echo Set RRP (RAM Read Pointer register) @ DP_1 AP_0 (APB):0x80120014 => 0x00000000
echo ***** [31:0]:RRP = 0x00000000 -> Location in trace memory that is accessed on a subsequent RRD read
memory set APB_0:0x80120014 0 0x00000000
echo Set RWP (RAM Write Pointer register) @ DP_1 AP_0 (APB):0x80120018 => 0x00000000
echo ***** [31:0]:RWP = 0x00000000 -> Location in trace memory that are accessed on a subsequent write
memory set APB_0:0x80120018 0 0x00000000
echo Set CTL (Control register) @ DP_1 AP_0 (APB):0x80120020 => 0x00000000
echo ***** [0]:TraceCaptEn = 0x1 -> Enable trace capture\n
memory set APB_0:0x80120020 0 0x00000001

```

#### 6.1.4 ELA-600 configuration test code

##### Deadlock Scenario

```

echo Set SIGSEL0 (Signal Select register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910100 => 0x00000001
echo ***** [11:0]:SIGSEL = 0x1 -> Select Signal Group 0\n
memory set APB_0:0x80910100 0 0x00000001

echo Set SIGMASK0[31:0] (Signal Mask register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910140 => 0x00000000
echo ***** [31:0]:SIGMASK = 0x00000000 -> Mask bits from SIGCOMP0[31:0]
memory set APB_0:0x80910140 0 0x00000000
echo Set SIGMASK0[63:32] (Signal Mask register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910144 => 0x00000000
echo ***** [63:32]:SIGMASK = 0x00000000 -> Mask bits from SIGCOMP0[63:32]
memory set APB_0:0x80910144 0 0x00000000
echo Set SIGMASK0[95:64] (Signal Mask register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910148 => 0x00000020
echo ***** [95:64]:SIGMASK = 0x00000020 -> Mask bits from SIGCOMP0[95:64] except [69]:VALID_P0 on CCI-550
memory set APB_0:0x80910148 0 0x00000020
echo Set SIGMASK0[127:96] (Signal Mask register for Trigger State 0) @ DP_1 AP_0 (APB):0x8091014C => 0x80000000
echo ***** [127:96]:SIGMASK = 0x80000000 -> Mask bits from SIGCOMP0[127:96] except [127]:VALID_P1 on CCI-550\n
memory set APB_0:0x8091014C 0 0x80000000

echo Set SIGCOMP0[31:0] (Signal Compare register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910180 => 0x00000000
echo ***** [31:0]:SIGCOMP = 0x00000000 -> Compare value for Signal Group 0 signals [31:0]
memory set APB_0:0x80910180 0 0x00000000
echo Set SIGCOMP0[63:32] (Signal Compare register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910184 => 0x00000000
echo ***** [63:32]:SIGCOMP = 0x00000000 -> Compare value for SG 0 signals [63:32]
memory set APB_0:0x80910184 0 0x00000000
echo Set SIGCOMP0[95:64] (Signal Compare register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910188 => 0x00000020
echo ***** [95:64]:SIGCOMP = 0x00000020 -> Compare value for SG 0 signals [95:64], 0x1 for [69]:VALID_P0\n

```

```

memory set APB_0:0x80910188 0 0x00000020
echo Set SIGCOMP0[127:96] (Signal Compare register for Trigger State 0) @ DP_1 AP_0 (APB):0x8091018C => 0x80000000
echo ***** [127:96]:SIGCOMP = 0x80000000 -> Compare value for SG 0 signals [127:96], 0x0 for [127]:VALID_P1\n
memory set APB_0:0x8091018C 0 0x80000000

echo Set ACTION0 (Action register for Trigger State 0) @ DP_1 AP_0 (APB):0x8091010C => 0x00000008
echo ***** [3]:TRACE = 0x1 -> Trace is active\n
memory set APB_0:0x8091010C 0 0x00000008

echo Set PTACTION (Pre-Trigger Action register) @ DP_1 AP_0 (APB):0x80910010 => 0x00000008
echo ***** [3]:TRACE = 0x1 -> Enable trace\n
memory set APB_0:0x80910010 0 0x00000008

echo Set NEXTSTATE0 (Next State register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910108 => 0x00000001
echo ***** [7:0]:NEXTSTATE = 0x1 -> Select Trigger State 0\n
memory set APB_0:0x80910108 0 0x00000001

echo Set TRIGCTRL0 (Trigger Control register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910104 => 0x00000001
echo ***** [7:6]:TRACE = 0x0 -> Trace is captured when Trigger Signal Comparison succeeds
echo ***** [3]:COMPSEL = 0x0 -> Disable counters and select Trigger Signal Comparison mode
echo ***** [2:0]:COMP = 0x1 -> Trigger Signal Comparison type select: equal (==)\n
memory set APB_0:0x80910104 0 0x00000001

echo Set TWBSEL0 (Trace Write Byte Select register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910128 => 0x000FFFFF
echo ***** [15]:TRACE_BYTE15 = 0x0 -> Trace write byte from SIGNALGRP0[127:120]
echo ***** ...
echo ***** [5]:TRACE_BYTE5 = 0x1 -> Trace write byte from SIGNALGRP0[47:40]
echo ***** [4]:TRACE_BYTE4 = 0x1 -> Trace write byte from SIGNALGRP0[39:32]
echo ***** [3]:TRACE_BYTE3 = 0x1 -> Trace write byte from SIGNALGRP0[31:24]
echo ***** [2]:TRACE_BYTE2 = 0x1 -> Trace write byte from SIGNALGRP0[23:16]
echo ***** [1]:TRACE_BYTE1 = 0x0 -> Trace write byte from SIGNALGRP0[15:8]
echo ***** [0]:TRACE_BYTE0 = 0x0 -> Trace write byte from SIGNALGRP0[7:0]\n
memory set APB_0:0x80910128 0 0x000FFFFF

echo Set ATBCTRL (ATB Control register) @ DP_1 AP_0 (APB):0x8091000C => 0x00001800
echo ***** [14:8]:ATID_VALUE = 0x18 -> Value to be driven on ATID[6:0]

memory set APB_0:0x8091000C 0 0x00001800

```

## Data Corruption Scenario

```

echo Set SIGSEL0 (Signal Select register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910100 => 0x00000001
echo ***** [11:0]:SIGSEL = 0x1 -> Select Signal Group 0\n
memory set APB_0:0x80910100 0 0x00000001

echo Set SIGMASK0[31:0] (Signal Mask register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910140 => 0x00000000
echo ***** [31:0]:SIGMASK = 0x00000000 -> Mask bits from SIGCOMP0[31:0]
memory set APB_0:0x80910140 0 0x00000000
echo Set SIGMASK0[63:32] (Signal Mask register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910144 => 0x00000000
echo ***** [63:32]:SIGMASK = 0x00000000 -> Mask bits from SIGCOMP0[63:32]
memory set APB_0:0x80910144 0 0x00000000
echo Set SIGMASK0[95:64] (Signal Mask register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910148 => 0x00000200
echo ***** [95:64]:SIGMASK = 0x00000200 -> Mask bits from SIGCOMP0[95:64]
memory set APB_0:0x80910148 0 0x00000200
echo Set SIGMASK0[127:96] (Signal Mask register for Trigger State 0) @ DP_1 AP_0 (APB):0x8091014C => 0x80000162
echo ***** [127:96]:SIGMASK = 0x80000162 -> Mask bits from SIGCOMP0[127:96]
memory set APB_0:0x8091014C 0 0x80000162

echo Set SIGCOMP0[31:0] (Signal Compare register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910180 => 0x00000000
echo ***** [31:0]:SIGCOMP = 0x00000000 -> Compare value for Signal Group 0 signals [31:0]
memory set APB_0:0x80910180 0 0x00000000
echo Set SIGCOMP0[63:32] (Signal Compare register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910184 => 0x00000000
echo ***** [63:32]:SIGCOMP = 0x00000000 -> Compare value for Signal Group 0 signals [63:32]
memory set APB_0:0x80910184 0 0x00000000
echo Set SIGCOMP0[95:64] (Signal Compare register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910188 => 0x00000000
echo ***** [95:64]:SIGCOMP = 0x00000000 -> Compare value for Signal Group 0 signals [95:64]
memory set APB_0:0x80910188 0 0x00000000
echo Set SIGCOMP0[127:96] (Signal Compare register for Trigger State 0) @ DP_1 AP_0 (APB):0x8091018C => 0x80000000
echo ***** [127:96]:SIGCOMP = 0x00000000 -> Compare value for Signal Group 0 signals [127:96]
memory set APB_0:0x8091018C 0 0x80000000

echo Set COUNTCOMP[1:0] (Counter Compare Register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910120 => 0x00000003
memory set APB_0:0x80910120 0 0x00000003

```

```

echo Set ACTION0 (Action register for Trigger State 0) @ DP_1 AP_0 (APB):0x8091010C => 0x00000008
echo ***** [3]:TRACE = 0x0 -> Trace is active\n
memory set APB_0:0x8091010C 0 0x00000009

echo Set PTACTION (Pre-Trigger Action register) @ DP_1 AP_0 (APB):0x80910010 => 0x00000008
echo ***** [3]:TRACE = 0x1 -> Enable trace\n
memory set APB_0:0x80910010 0 0x00000008

echo Set NEXTSTATE0 (Next State register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910108 => 0x00000002
echo ***** [7:0]:NEXTSTATE = 0x2 -> FINAL STATE \n
memory set APB_0:0x80910108 0 0x00000002

echo Set TRIGCTRL0 (Trigger Control register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910104 => 0x000000C1
echo ***** [7:6]:TRACE = 0x0 -> trigger state counter enabled with countsrc=0 and trace on signal comparison
echo ***** [5]:COUNTSRC = 0x1 -> Counter incremented every Trigger Signal Comparison match
echo ***** [3]:COMPSEL = 0x1 -> Disable counters and select Trigger Signal Comparison mode
echo ***** [2:0]:COMP = 0x1 -> Trigger Signal Comparison type select: equal (==)\n
memory set APB_0:0x80910104 0 0x00000029

echo Set TWBSEL0 (Trace Write Byte Select register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910228 => 0x000FFFFF
echo ***** [15]:TRACE_BYTE15 = 0x0 -> Trace write byte from SIGNALGRP0[127:120]
echo ***** ...
echo ***** [5]:TRACE_BYTE5 = 0x1 -> Trace write byte from SIGNALGRP0[47:40]
echo ***** [4]:TRACE_BYTE4 = 0x1 -> Trace write byte from SIGNALGRP0[39:32]
echo ***** [3]:TRACE_BYTE3 = 0x1 -> Trace write byte from SIGNALGRP0[31:24]
echo ***** [2]:TRACE_BYTE2 = 0x1 -> Trace write byte from SIGNALGRP0[23:16]
echo ***** [1]:TRACE_BYTE1 = 0x0 -> Trace write byte from SIGNALGRP0[15:8]
echo ***** [0]:TRACE_BYTE0 = 0x0 -> Trace write byte from SIGNALGRP0[7:0]\n
memory set APB_0:0x80910128 0 0x0000FFFF

echo Set SIGSEL1 (Signal Select register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910200 => 0x00000001
echo ***** [11:0]:SIGSEL = 0x1 -> Select Signal Group 1\n
memory set APB_0:0x80910200 0 0x00000000

echo Set SIGMASK1[31:0] (Signal Mask register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910240 => 0x00000000
echo ***** [31:0]:SIGMASK = 0x00000000 -> Mask bits from SIGCOMP1[31:0]
memory set APB_0:0x80910240 0 0x00000000
echo Set SIGMASK1[63:32] (Signal Mask register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910244 => 0x00000000
echo ***** [63:32]:SIGMASK = 0x00000000 -> Mask bits from SIGCOMP1[63:32]
memory set APB_0:0x80910244 0 0x00000000
echo Set SIGMASK1[95:64] (Signal Mask register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910248 => 0x00000000
echo ***** [95:64]:SIGMASK = 0x00000000 -> Mask bits from SIGCOMP1[95:64]
memory set APB_0:0x80910248 0 0x00000000
echo Set SIGMASK1[127:96] (Signal Mask register for Trigger State 1) @ DP_1 AP_0 (APB):0x8091024C => 0x80000000
echo ***** [127:96]:SIGMASK = 0x00000000 -> Mask bits from SIGCOMP0[127:96]
memory set APB_0:0x8091024C 0 0x00000000

echo Set SIGCOMP1[31:0] (Signal Compare register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910280 => 0x00000000
echo ***** [31:0]:SIGCOMP = 0x00000000 -> Compare value for Signal Group 1 signals [31:0]
memory set APB_0:0x80910280 0 0x00000000
echo Set SIGCOMP1[63:32] (Signal Compare register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910284 => 0x00000000
echo ***** [63:32]:SIGCOMP = 0x00000000 -> Compare value for Signal Group 0 signals [63:32]
memory set APB_0:0x80910284 0 0x00000000
echo Set SIGCOMP1[95:64] (Signal Compare register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910288 => 0x00000000
echo ***** [95:64]:SIGCOMP = 0x=00000000 -> Compare value for Signal Group 0 signals [95:64]
memory set APB_0:0x80910288 0 0x00000000
echo Set SIGCOMP1[127:96] (Signal Compare register for Trigger State 1) @ DP_1 AP_0 (APB):0x8091028C => 0x80000000
echo ***** [127:96]:SIGCOMP = 0x00000000 -> Compare value for Signal Group 1 signals [127:96]
memory set APB_0:0x8091028C 0 0x00000000

echo Set EXTMASK1 (External Mask register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910230 => 0x00000001
echo ***** [0] CTIIN[0] = 0x1 -> mask CTIIN signal bit [0] \n
memory set APB_0:0x80910230 0 0x00000001

echo Set EXTCOMP1 (External Compare register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910234 => 0x00000001
echo ***** [3]:TRACE = 0x0 -> Trace is active\n
memory set APB_0:0x80910234 0 0x00000001

echo Set COUNTCOMP[1:0] (Counter Compare Register for Trigger State 0) @ DP_1 AP_0 (APB):0x80910220 => 0x00000003
memory set APB_0:0x80910220 0 0xFFFFFFFF

echo Set ACTION1 (Action register for Trigger State 1) @ DP_1 AP_0 (APB):0x8091020C => 0x00000008
echo ***** [3]:TRACE = 0x0 -> Trace is active\n
memory set APB_0:0x8091020C 0 0x00000008

```

```

echo Set NEXTSTATE1 (Next State register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910208 => 0x00000002
echo ***** [7:0]:NEXTSTATE = 0x0 -> FINAL STATE \n
memory set APB_0:0x80910208 0 0x00000000

echo Set TRIGCTRL1 (Trigger Control register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910204 => 0x000000C1
echo ***** [7:6]:TRACE = 0x11 -> Trigger state counter enabled with countsrc=0, and trace on signal comparison
echo ***** [5]:COUNTSRC = 0x0 -> Counter incremented every ELA clock cycle
echo ***** [3]:COMPSEL = 0x0 -> Disable counters and select Trigger Signal Comparison mode
echo ***** [2:0]:COMP = 0x1 -> Trigger Signal Comparison type select: equal (==)\n
memory set APB_0:0x80910204 0 0x000000C1

echo Set TWBSEL1 (Trace Write Byte Select register for Trigger State 1) @ DP_1 AP_0 (APB):0x80910228 => 0x000FFFFF
echo ***** [15]:TRACE_BYTE15 = 0x0 -> Trace write byte from SIGNALGRP0[127:120]
echo ***** ...
echo ***** [5]:TRACE_BYTE5 = 0x1 -> Trace write byte from SIGNALGRP1[47:40]
echo ***** [4]:TRACE_BYTE4 = 0x1 -> Trace write byte from SIGNALGRP1[39:32]
echo ***** [3]:TRACE_BYTE3 = 0x1 -> Trace write byte from SIGNALGRP1[31:24]
echo ***** [2]:TRACE_BYTE2 = 0x1 -> Trace write byte from SIGNALGRP1[23:16]
echo ***** [1]:TRACE_BYTE1 = 0x1 -> Trace write byte from SIGNALGRP1[15:8]
echo ***** [0]:TRACE_BYTE0 = 0x1 -> Trace write byte from SIGNALGRP1[7:0]\n
memory set APB_0:0x80910228 0 0x000FFFFF

echo Set CNTSEL (Counter Select register) @ DP_1 AP_0 (APB):0x80910018 => 0x00000012
memory set APB_0:0x80910018 0 0x00000012

echo Set ATBCTRL (ATB Control register) @ DP_1 AP_0 (APB):0x8091000C => 0x00001800
echo ***** [14:8]:ATID_VALUE = 0x18 -> Value to be driven on ATID[6:0]
memory set APB_0:0x8091000C 0 0x00001800

echo ***** Setting up CTI4 CORE SIDE *****
echo ***** Set CTI4 CTIOUTEN0 -> Map channels in CT system to trigger outputs*****
echo Set CTIOUTEN (CTI Channel to Trigger 0 Enable register) @ DP_1 AP_0 (APB):0x820200A0 => 0x0000000F
memory set_typed APB_0:0x820200A0 (unsigned int) (0x0000000F)

echo ***** Set CTI4 CTIINEN0 -> Map Trigger Inputs to Channels in CT System *****
echo Set CTIINEN0 (CTI Trigger 0 to Channel Enable register) @ DP_1 AP_0 (APB):0x82020020 => 0x0000000F
memory set_typed APB_0:0x82020020 (unsigned int) (0x0000000F)

echo ***** Enabling CTI4 CTRL Register ****
echo Set CTICTRL (CTI CTRL register) @ DP_1 AP_0 (APB):0x82020000 => 0x00000001
memory set_typed APB_0:0x82020000 (unsigned int) (0x00000001)

echo ***** Setting up CT3 ELA SIDE ****
echo ***** Enabling CTI3 CTRL Register ****
echo Set CTICTRL (CTI CTRL register) @ DP_1 AP_0 (APB):0x80920000 => 0x00000001
memory set_typed APB_0:0x80920000 (unsigned int) (0x00000001)

echo ***** Enabling CTI3 CTIINEN0 -> Map Trigger Inputs to Channels in CT System ****
echo Set CTIINEN0 (CTI Trigger 0 to Channel Enable register) @ DP_1 AP_0 (APB):0x80920020 => 0x0000000F
memory set_typed APB_0:0x80920020 (unsigned int) (0x0000000F)

echo ***** Set CTI3 CTIOUTEN0 -> Map channels in CT system to trigger outputs *****
echo Set CTIOUTEN (CTI Channel to Trigger 0 Enable register) @ DP_1 AP_0 (APB):0x820200A0 => 0x0000000F
memory set_typed APB_0:0x820200A0 (unsigned int) (0x0000000F)

```

### 6.1.5 Start trace capture test code

```

echo System ETF
echo Set CTL (Control register) @ DP_1 AP_0 (APB):0x80020020 => 0x00000001
echo ***** [0]:TraceCaptEn = 0x1 -> Enable trace capture\n
memory set APB_0:0x80020020 0 0x00000001

echo ETR
echo Set CTL (Control register) @ DP_1 AP_0 (APB):0x80120020 => 0x00000001
echo ***** [0]:TraceCaptEn = 0x1 -> Enable trace capture\n
memory set APB_0:0x80120020 0 0x00000001

```

### 6.1.6 Run ELA-600 test code

```
echo Set CTRL (Logic Analyzer Control register) @ DP_1 AP_0 (APB):0x80910000 => 0x00000001
echo ***** [0]:RUN = 0x1 -> Run control: Enable ELA-600\n
memory set APB_0:0x80910000 0 0x00000001
```

### 6.1.7 Stop ELA-600 test code

```
echo Set CTRL (Logic Analyzer Control register) @ DP_1 AP_0 (APB):0x80910000 => 0x00000000
echo ***** [0]:RUN = 0x0 -> Run control: Disable ELA-600\n
memory set APB_0:0x80910000 0 0x00000000
```

### 6.1.8 Stop trace capture test code

```
echo Manually flush System ETF
echo Set FFCR (Control register) @ DP_1 AP_0 (APB):0x80020304 => 0x00000040
echo ***** [6]:FlushMan = 0x1 -> Manually generate a flush
memory set APB_0:0x80020304 0 0x00000040
```

```
echo Manually flush ETR
echo Set FFCR (Control register) @ DP_1 AP_0 (APB):0x80120304 => 0x00000040
echo ***** [6]:FlushMan = 0x1 -> Manually generate a flush
memory set APB_0:0x80120304 0 0x00000040
```

```
echo System ETF
echo Set CTL (Control register) @ DP_1 AP_0 (APB):0x80020020 => 0x00000000
echo ***** [0]:TraceCaptEn = 0x0 -> Disable trace capture\n
memory set APB_0:0x80020020 0 0x00000000
```

```
echo ETR
echo Set CTL (Control register) @ DP_1 AP_0 (APB):0x80120020 => 0x00000000
echo ***** [0]:TraceCaptEn = 0x0 -> Disable trace capture\n
memory set APB_0:0x80120020 0 0x00000000
```

### 6.1.9 Memory dump to file test code

```
echo Write ELA-600 trace capture to ela600_trace_capture_etr.bin
dump memory ela600_trace_capture_etr.bin 0x82000000 0x820027AC

echo Write ELA-600 trace capture to ela600_trace_capture_etr.vhx
dump vhx memory ela600_trace_capture_etr.vhx 0x82000000 0x820027AC
```