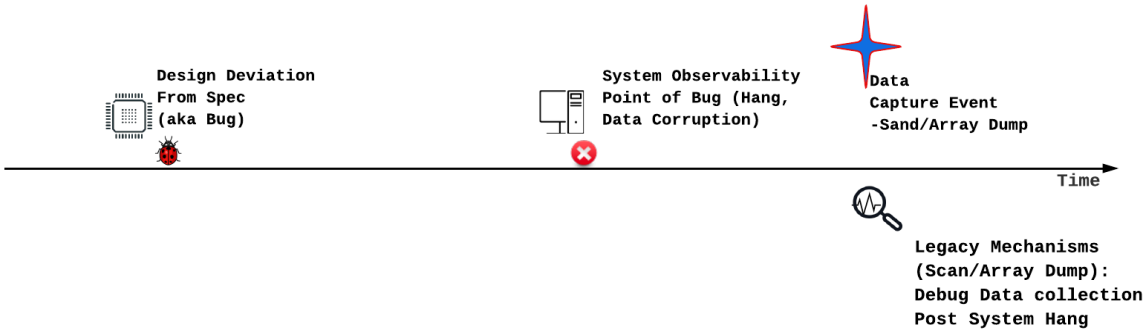
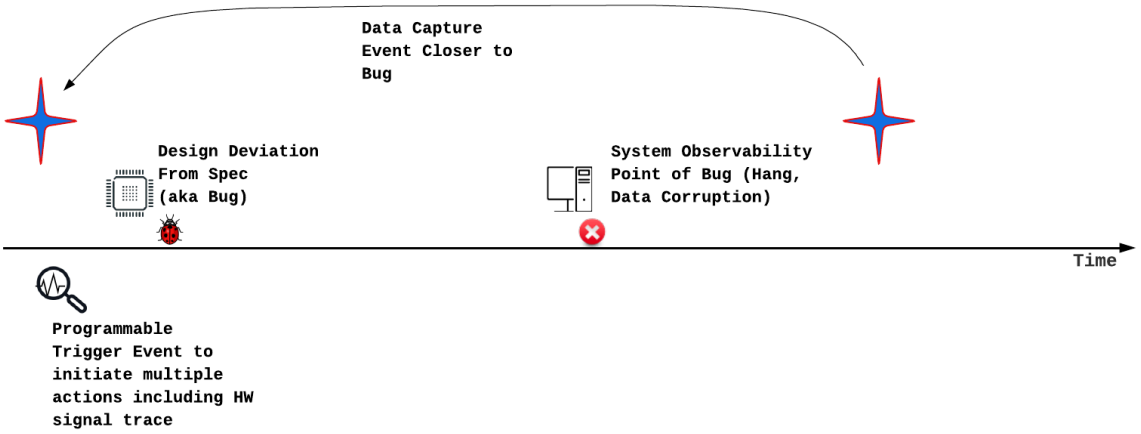
tt-dfd

Introduction

Post Silicon debug requires observability, design control for experimenting and data capture at the point of failure. Traditional data capture methods, such as scan and array dumps, only allow for capturing the design state when the failure is observed at system level, losing all transitional data.



Current RISC-V debug specifications, which include debug module and instruction trace, can track design transitions. However, visibility is confined to architectural components and frequently necessitates a functional core. Effective debugging requires the tracing of micro-architectural events, initiated by trace capture events that occur as close as possible to the point of design deviation. The specific debug scenario dictates the definition of these trigger events. Debugging can be significantly accelerated if these trigger events can also modify design behavior, thereby facilitating debug experiments.



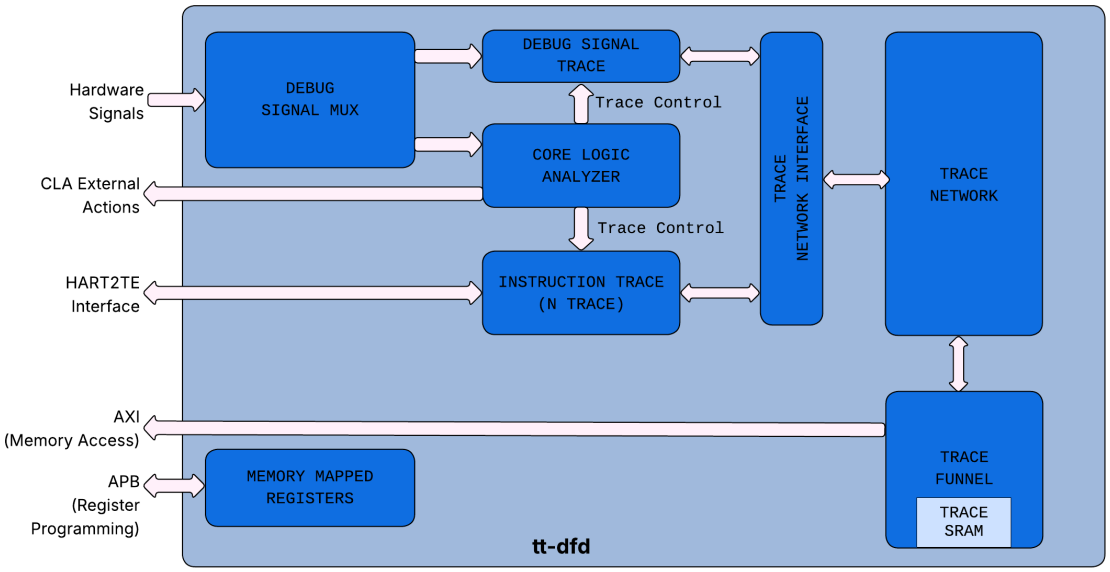
Subsequent chapters detail Core Logic Analyzer and Debug Signal Trace design elements, which are designed to achieve this objective. Beyond Debug, Core Logic Analyzer can also be used for performance analysis, hardware bug workarounds and generating reactive stimuli.

These Design for Debug (DfD) blocks are provided as open-source collateral, known as **tt-dfd**. The tt-dfd package also incorporates RISC-V defined Instruction Trace, based on N-Trace.

This document details the component of tt-dfd, integration steps,

tt-dfd

Tenstorrent offers tt-dfd as customizable IP to meet the Dfd requirements of SOC/Chiplet designs. A detailed block diagram is provided. Tenstorrent is open-sourcing the tt-dfd IP to promote the development of the post-silicon debug software and hardware ecosystem, including standardization. The IP is available for download at https://github.com/tenstorrent/tt-dfd.



The t-dfd implement the following:

1. **Core Logic Analyzer:** This component observes hardware signals and initiates actions when specific, programmable conditions (triggers) are met. Actions include starting or stopping the instruction and debug signal traces.
2. **Instruction Trace:** This feature traces retired program counters (PCs) to analyze software execution, which is also useful for debugging purposes.
3. **Debug Signal Trace (DST):** DST allows users to trace selected hardware signals, storing them either on-chip or in memory for later analysis, primarily for hardware debugging.

4. **Trace Network Interface (TNIF):** TNIF allows arbitrates and interfaces DST and Instruction Trace blocks to Trace Network.
5. **Trace Network & Funnel:** This on-chip network is responsible for transferring data from the Instruction Trace and Debug Signal Trace modules to either on-chip SRAM or main memory.
6. **Debug Signal Mux:** The Debug Signal Mux is a programmable multiplexer designed to select 8 8-bit wide debug buses from a total of 16 available 8-bit signals. Its output is subsequently utilized by the CLA and DST.

Core Logic Analyzer

The Core Logic Analyzer (CLA) offers capabilities for observing hardware signals, establishing user-defined triggers based on these signals, and executing actions when triggers are met. Beyond its primary debugging function, the CLA can be expanded to include performance monitoring, generating reactive stimuli based on runtime feedback, and implementing workarounds. For the initial silicon version of the IP, debugging will be the most prominent use. Other usage models are expected to provide return on investment over several silicon generations.

CLA High Level Overview

The CLA offers three key features:

1. **Debug Bus:** The CLA features a 64-bit debug bus, which is a dedicated set of hardware signals for observation. To increase observability beyond 64 bits, a programmable mux tree can be used to precede the debug bus input to the CLA.
2. **Event-Action Pairs:** CLA allows users to take “actions” when a set of events meet user defined functions. Subsequent sections details on each of the element of event-action pairs
3. **Nodes:** CLA supports the concept of nodes to allow sequential execution of “Event-Action Pairs”. This allows users to program conditions like Event-A followed by Event-B.

Event

The Core Logic Analyzer (CLA) enables users to program events based on the status of debug signals and internal CLA state elements, such as counters. These events are evaluated within a single cycle. The table below lists the CLA supported events.

Event Select	Name	Description

0x0	Disable	Event not active.
0x1	Always On	The event is active all the time. Useful for default actions.
0x2	Match1 (positive filter)	Match Debug Signals with a given mask and value. +ve Filter: Debug Signals & Mask == Match Value -ve Filter: Debug Signals & Mask! = Match Value. Use case example: Detect a state-machine to transition out of a given state.
0x3	No Match1 (negative filter	
0x4	Match2 (positive filter))	
0x5	No Match2 (negative filter)	
0x6	Edge Detect Set 0	For Debug Signals listed in c_dbg_signal_edge_detect_cfg[signal0_select] and look for a transition. The nature of transition (pos-edge or neg-edge) is selected by c_dbg_signal_edge_detect_cfg[pos_edge_signal0]
0x7	Edge Detect Set 1	For Debug Signals listed in c_dbg_signal_edge_detect_cfg[signal1_select] and look for a transition. The nature of transition (pos-edge or neg-edge) is selected by c_dbg_signal_edge_detect_cfg[pos_edge_signal1]
0x8	Transition	Look for transition of Debug Signals from Value A (with Mask A) to Value B (with Mask B). Useful for tracking state machine transitions.
0x9	Cross Trigger In 1	Cross Trigger Input from adjacent CLA. Cross triggers are daisy chained (to minimize wiring)
0xa	Cross Trigger In 2	
0xb	1s Count	Trigger if sum of all unmasked bits of debug bus equals a value (useful for one-hot rules)
0xc	Debug Signals Change	Any change in Debug Signals.
0xd-0xe	Future Use	Future Use
0xf	Core time match	Input time value matches (greater than or equal to the programmed value) with the programmed value in CLA Time match register. To deassert this event, write 0 to the time match value MMR.
0x10	CLA Counter0 Target Match	CLA Counter0 == Counter Target
0x11	CLA Counter0 Target Overflow	CLA Counter0 > Counter Target
0x12	CLA Counter0 Below Target	CLA Counter0 < Counter Target
0x13	CLA Counter1 Target Match	CLA Counter1 == Counter Target
0x14	CLA Counter1 Target Overflow	CLA Counter1 > Counter Target
0x15	CLA Counter1 Below Target	CLA Counter1 < Counter Target
0x16	CLA Counter2 Target Match	CLA Counter2 == Counter Target
0x17	CLA Counter2 Target Overflow	CLA Counter2 > Counter Target
0x18	CLA Counter2 Below Target	CLA Counter2 < Counter Target
0x19	CLA Counter3 Target Match	CLA Counter3 == Counter Target
0x1A	CLA Counter3 Target Overflow	CLA Counter3 > Counter Target
0x1B	CLA Counter3 Below Target	CLA Counter3 < Counter Target
0x1C-0x	Future Use	Future Use

3F		

Trigger on User Defined Function

For additional flexibility, CLA supports observing 3 events simultaneously and a user defined function (UDF) among the three events. A trigger is created when the UDF returns true.

Action

Actions are steps taken by Core Logic Analyzer when a trigger is fired. The purpose of actions can broadly be defined into local and global actions. Local actions are implemented within Core Logic Analyzer. Global actions are implemented outside of CLA. The components of Debug Signal Trace implement actions related to trace. The table below lists the CLA supported actions.

Select #	Name	Scope	Description
0x0	Null	-	
0x1	Clock Halt	Global	See Clock Halt Section.
0x2	Debug Interrupt	Global	This action asserts o/p pin ***external_action_debug_interrupt_out*** which is routed as an interrupt to the core.
0x3	Reserved		
0x4	Start Trace	Global	Action to start trace indicated by o/p pin: ***external_cla_action_trace_start***
0x5	Stop Trace	Global	Action to start trace indicated by o/p pin: ***external_cla_action_trace_stop***
0x6	Trace Pulse	Global	Action to trace for a single cycle (when trigger fires). Indicated by o/p pin: ***external_cla_action_trace_pulse***
0x7	Cross Trigger Out 1	Global	Trigger to other CLAs. Triggers are daisy chained to minimize wiring.
0x8	Cross Trigger Out 2	Global	
0x9	Reserved		
0x10	Incr. CLA Counter0 by 1	Local	Increment the CLA counter0.
0x11	Clear CLA Counter0	Local	Clear the CLA counter0.
0x12	Auto Incr CLA Counter0	Local	Increment CLA counter0 for every CLA clock.
0x13	Stop Auto Incr Counter0	Local	Stop auto increment of CLA counter0.
0x14	Incr. CLA Counter1 by 1	Local	Increment the CLA counter1.
0x15	Clear CLA Counter1	Local	Clear the CLA counter1.
0x16	Auto Incr CLA Counter1	Local	Increment CLA counter1 for every CLA clock.
0x17	Stop Auto Incr Counter1	Local	Stop auto increment of CLA counter1.

0x18	Incr. CLA Counter2 by 1	Local	Increment the CLA counter2.
0x19	Clear CLA Counter2	Local	Clear the CLA counter2.
0x1A	Auto Incr CLA Counter2	Local	Increment CLA Counter2 for every CLA clock.
0x1B	Stop Auto Incr Counter2	Local	Stop auto increment of CLA Counter2.
0x1C	Incr. CLA Counter3 by 1	Local	Increment the CLA counter3.
0x1D	Clear CLA Counter3	Local	Clear the CLA counter3.
0x1E	Auto Incr CLA Counter3	Local	Increment CLA Counter3 for every CLA clock.
0x1F	Stop Auto Incr Counter3	Local	Stop auto increment of CLA Counter0.

Clock Halt Action

A clock halt action effectively freezes the design when predefined trigger conditions are met. If these conditions involve monitoring internal microarchitectural states, halting the clock and then performing a scan allows for the capture of the design's state very close to the point of interest. Local clock gating is faster than global clock halt (like PLL shutdown).

The Clock Halt action activates both the *external_action_halt_clock_out* signal for global clock gating and the *external_action_halt_clock_local_out* signal for local clock halting. Users have the ability to enable or disable these functions using the *CDbgClaCtrlStatus[DisableLocalClockHalt]* & *CDbgClaCtrlStatus[DisableGlobalClockHalt]* settings, respectively.

Custom Actions

In addition to the standard actions, CLA offers "customer actions," which are defined & implemented by users. A 16-bit wide output bus from the CLA - *external_action_custom* - allows external (non-CLA) logic to implement the desired actions. When the trigger conditions are met, the user can program specific custom actions.

Event-Action Pair (EAP)

Each action is linked to a particular event using an "event-action" pair register. The EAP register layout is as below. It broadly contains - events, User Defined Function, Actions, Custom Actions and Destination Node.

Field	Description	Comments
Event 0	Select an event.	
Event 1	Select an event	
Event 2	Select an event	

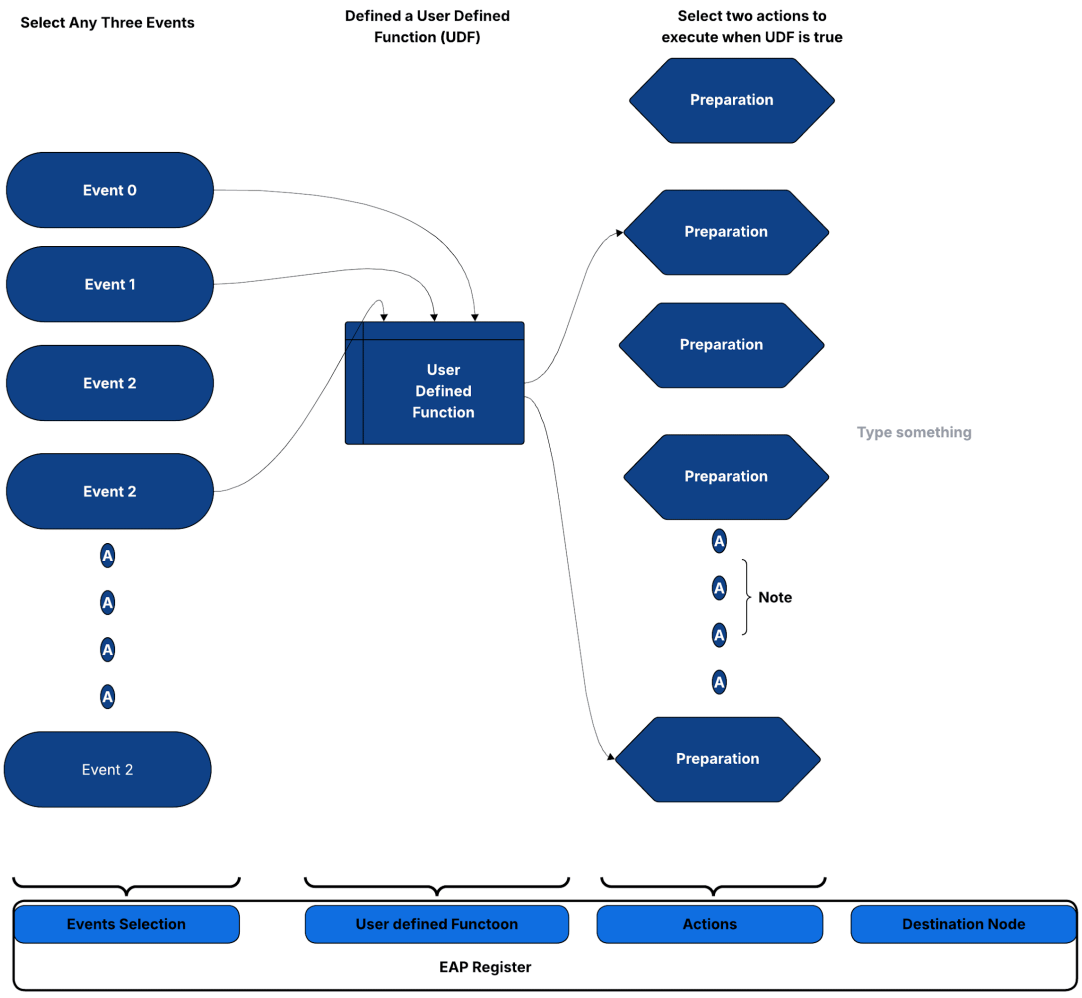
Reserved	Program 2'b10. Other values reserved.	
User Defined Function	*8-bit lookup table to implement a function w/ 3 events*	*Relation to be satisfied among Event0, Event1and Event 2 to activate the actions*
Action 1	*Select an Action*	
Action 0	*Select an Action*	
Destination Node	*Select "Destination Node"*	*See "Linking of EAPs"*
CustomAction1	*Select bit position for custom action*	*Select the bit position of the custom action bus to be set when the result of Logical Operation (see above) of events is TRUE.*
CustomAction0	*Select bit position for custom action*	*Select the bit position of the custom action bus to be set when the result of Logical Operation (see above) of events is TRUE.*

Note on UDP:
Here is a reference example of UDP. Assume you want to implement a logical operation of (Event2 && Event1) || Event0 .
Here is the truth table for the same:

Event2	Event1	Event0	(Event2 && Event1) || Event0 .	UDP Entry
0	0	0	0	UDP[0] (LSB)
0	0	1	1	UDP[1]
0	1	0	0	UDP[2]
0	1	1	1	UDP[3]
1	0	0	0	UDP[4]
1	0	1	1	UDP[5]
1	1	0	1	UDP[6]
1	1	1	1	UDP[7] (MSB)

In the UDP field program a value of 8'b11101010 (8'hEC)

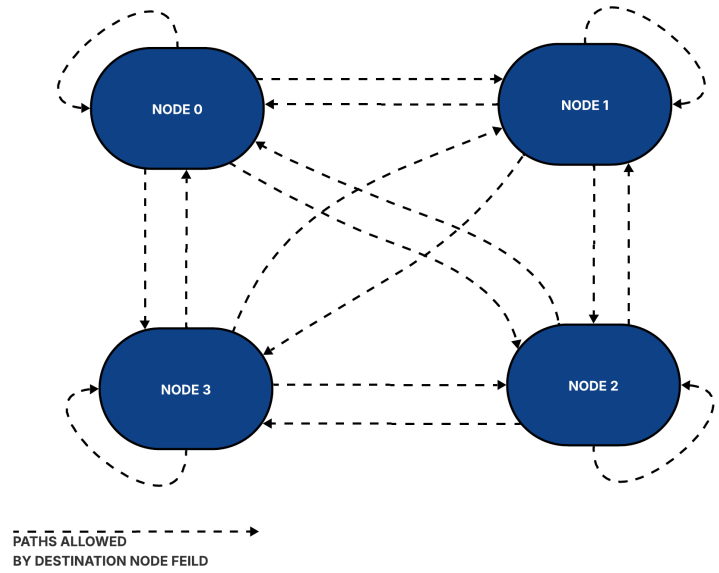
Pictorial representation of EAP.



Sequential Linking of EAPs

Debugging often necessitates observing a series of events over multiple cycles, as single-cycle events alone are usually inadequate. For example, imagine a design is stalled, waiting for a timer interrupt after executing a Wait For Interrupt (WFI) instruction. To debug such situations effectively, it's essential to first monitor Debug Signals for a WFI opcode match, and then subsequently monitor the debug signals for a Timer Interrupt match. In other words we want to have two different trigger matches in sequence.

To facilitate sequential triggers, CLA utilizes the concept of nodes. Each node comprises three EAPs (Event Action Processors). Each EAP can be programmed with a "destination node." With this mechanism, CLA can switch from any one node to another node using the destination node.

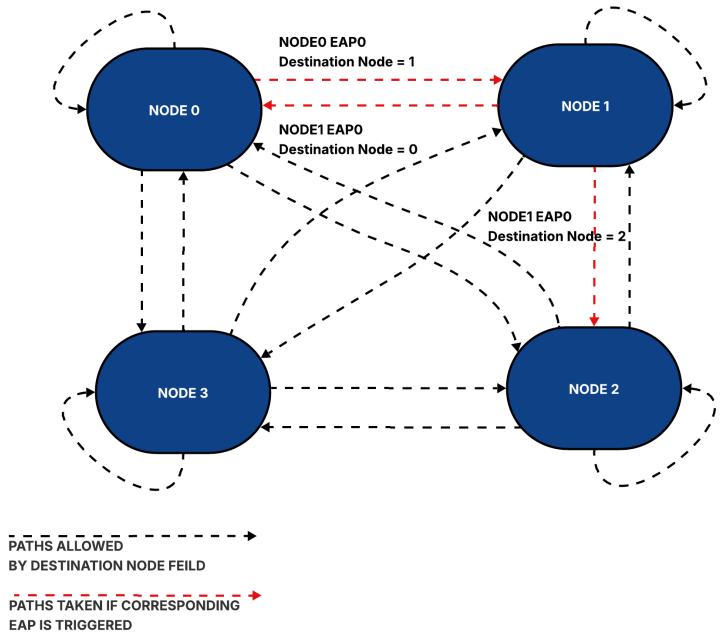


At any given time, only one node, referred to as the current node, and its associated 3 EAPs are active. An EAP can initiate a switch from the current node to its destination node based on its pre-programmed conditions. Upon reset, Node0 is designated as the current node. Once the EAP events within Node0 are met, the CLA transitions to the EAP's destination node, along with the execution of specified actions. With the transition, the EAPs of destination nodes are activated.

To illustrate the linking, let us continue with the example of debugging a hang with design stuck waiting for a timer interrupt for WFI exit. Let us say that a timer interrupt is programmed to occur within X clocks of WFI. If the interrupt is not received within this timeframe, CLA should be programmed to freeze the design (using clock stop action), and its state should be captured (e.g., using a Scan). Here is pseudo-code CLA programming for such a debug case, using nodes.

```
CLA Counter 0 Target is set to TIMEOUT
Set “Clr. CLA Counter0” in the CLA Ctrl & Status
#Start a CLA Counter on WFI Retire….and switch to Node 1
Node0 EAP0
Event0                :Debug Bus Mask0 & Match0 == WFI Indication
Event1 & Event2       :Always ON
UDF                   :AND of Event1, Event2 and Event3
Action0               :AutoIncr CLA Counter0
Dest. Node            :Node1
#Clear the Counter on Interrupt and switch-back to Node1 (back to default state)
#If (Counter0 ==TIMEOUT), Freeze the clock and Switch to Node2
Node1 EAP0
Event0                :Debug Bus Mask1 & Match1 == Interrupt Indication
Event1 & Event2       :Always ON
```

```
UDF                 :AND of Event1, Event2 and Event3
Action0             :Clear CLA Counter0
Dest. Node          :Node0
Node1 EAP1
Event0              :CLA Counter0 == TIMEOUT
Event1 & Event2     :Always ON
UDF                 :AND of Event1, Event2 and Event3
Action0             :Clock Halt (for Subsequent Scan Dump)
Dest. Node          :Node2
```



Note: In scenarios where multiple EAPs are triggered simultaneously, the CLA prioritiz
es and switches to the destination node associated with the lowest-numbered EAP. If no EAP events are triggered, the CLA remains in its current node. Users also have the option to configure the CLA to remain in the current node even when an event is triggered.

CLA Registers

Following is the table of CLA registers. Please refer to CR Registers.html for register fields.

Register	Description
Counter Config Registers: CDbgClaCounter<0,1,2,3>Cfg	Configuration for CLA counters used for counting cycles after an event match, and trigger an action on match.
EAP Registers: CDbgNode0Eap<0,1,2,3> CDbgNode1Eap<0,1,2,3>	Action(s) are tied to events(s) using "event-action pairing" registers. CLA supports 4, Each Node has 4 EAPs.

CDbgNode2Eap<0,1,2,3> CDbgNode3Eap<0,1,2,3>	
Debug Signal Mask & Match Registers: CDbgSignalMask<0,1,2,3> CDbgSignalMatch<0,1,2,3>	CLA Supports 4 sets of Debug Signal Mask and Match. These are used to define debug bus match event. Match event is triggered when debug_bus & mask = match.
Edge Detect: CDbgSignalEdgeDetectCfg	Register to configure debug bus for edge triggers.
EAP Status: CDbgEapStatus	Register to indicate if EAP Pair was activated. Use the corresponding w2c register to reset the status. There are 2 bits for each EAP. The bit corresponds to action-0 and action-1 of the EAP. The register is used by SW to know if an action was taken (ex: NMI ISR needs to know which of the EAP triggered the NMI)
CLA Control/Status: CDbgClaCtrlStatus	Ctrl/Status register to Enable EAP after EAP programming is complete, and read the current node.
Transition Event Registers: CDbgTransitionMask CDbgTransitionFromValue CDbgTransitionToValue	3 registers to configure "transition event". TransitionMask register is to select the debug signals of interest for transition events. The transition event is triggered on transition from CDbgTransitionFromValue to CDbgTransitionToValue. This register specifies Value A.
One Count Event: CDbgOnesCountMask CDbgOnesCountValue	2 registers to configure "ones count" event. CDbgOnesCountMask register is to select the debug signals of interest using a mask. Event triggered when the sum of the unmasked selected signals match the value specified in this register.
Debug Bus Change Event: CDbgAnyChange	This register is used to trigger an event when the debug bus changes value. The register value is used to select the subset of the debug signals.
Debug Bus Snapshot: CDbgSignalSnapshotNode0Eap<0,1,2,3> CDbgSignalSnapshotNode1Eap<0,1,2,3> CDbgSignalSnapshotNode2Eap<0,1,2,3> CDbgSignalSnapshotNode3Eap<0,1,2,3>	When the UDF of a EAP returns true, CLA captures the current value of the debug bus in this register. Since we have 16 EAPs, we have 16 “Snapshot” registers, one per EAP.
TimeMatch Event: CDbgClaTimeMatch	If the value in the register is greater than or equal to the to time register value of the core (core timer is an input to CLA), CLA generates a time match event. Needs to be non zero for the time match event signal to trigger. To stop the event trigger write 0s to this register.
CDbgSignalDelayMuxSel	CLA takes 8 lanes of debug bus. Each lane is 8-bit wide. This register can be used to add 0-8 additional clock delays

	on each lane before CLA samples the lane.
CDbgClaTimestampsync	This value is loaded to a timestamp counter on a cross trigger. In a system with multiple CLAs, combination of cross trigger and this register is used to sync the time value across the CLA blocks
CDbgClaXtriggerTimestretch	Multiple CLAs can “talk” to each other using cross triggers. The value in this register extends pulse duration to detect the cross trigger..

CLA Programming Support

To ease CLA programming, a CLA Compiler is part of an open source package. CLA compiler compiles an abstract description of a CLA program into csr register values. This makes programming the CLA easier and less error-prone, since you do not need to manually calculate mux selects, match, or mask CSRs.
Please refer to https://github.com/tenstorrent/tt-dfd/tree/main/scripts/cla_compiler for more details.
Coming Soon:
A leading industry vendor is actively working on a GUI based programmer for CLA, with initial release planned by end of 2024.

Debug Mux

Debug Signal Trace

Debug Signal Trace (DST) enhances traditional post-silicon hardware debug data collection by providing improved visibility into design states. Unlike conventional methods (scan and array dump) that capture a single snapshot at the point of failure, DST traces a selected set of design signals over multiple cycles leading up to the failure. This trace data can be stored in on-chip memory (SRAM) or off-chip system memory.

The post-processing of this stored trace data not only aids in debug visibility but also enables the creation of post-silicon coverage points. DST data is timestamped, allowing for correlation with instruction trace data, thereby extending its utility to software performance analysis.

To facilitate adoption and ease of use, DST's control register definition mirrors that of the RISC-V Trace Control Interface, a standard familiar to the RISC-V debug community. DST minimizes memory storage requirements through signal compression. It leverages existing

triggers from the RISC-V Debug Specification and also incorporates user-configurable triggers based on a select set of design signals.

Debug Signal Trace Data

DST Trace Data is generated when input debug bus signals change. It utilizes a "Variable Length Transmission" method, which only transmits bytes that have changed, using Byte Enables to indicate their positions. The term "variable length" indicates that the number of bytes transmitted varies based on the debug bus input.

Frames

Both Debug Signal Trace and N-Trace instruction data produce variable-length packets. The trace generator must arbitrate with the Trace Sink (SRAM/SMEM) for transfer size allocation due to this variability. Implementing a configurable, fixed transfer size simplifies this arbitration. Aligning the transfer size to the maximum memory or on-chip bus transfer boundary further allows software to minimize penalties from unaligned access and optimize memory bandwidth. To achieve a configurable fixed transfer size, the Tt-dfd hardware packs packets into frames before transmitting them to the trace sink. Tt-dfd also incorporates "filler packets" for both instruction and debug signal traces to generate a complete frame upon "trace flush.".

DST Packets

DST packets are of two types- Trace Data Packets and Trace Support Packets. Trace Data Packets are used for transmitting debug signals. Trace Support Packets provide additional data to assist with debugging and decoding, such as timestamps. Future definitions can be added to Trace Support Packets as needed.

Trace Data Packet

Packet Format

Header1 (Byte-1)	Header0 (Byte-0)			
Byte Enables (7 bits)	**Pkt Type (1’b0)**	Src Id (4 bits)	Pkt Loss (1 bit)	Trace Info (2 bits)
Payload0 (Byte -3 - ..[9])	Payload0 (Byte -2)			
Payload: Variable size, May go up to 8 bytes	Payload: Variable size, May go up to 8 bytes			

Header Decode for Trace Data Packet

Name	Bits	Meaning
Packet Type	Hdr0[7]	1’b0: Trace Data Packet. Use this table for decoding Trace Packet.
Source ID	Hdr0[6:3]	Source ID. Can support up to 16 sources.
Packet Loss	Hdr0[2]	Set to 1 if prior packet(s) were lost. Packets can be lost due to trace network bandwidth constraints.

Trace Info	Hdr0[1:0]	2'b01: Trace Start 2'b10: Trace Stop 2'b11: Trace Sync (periodic sync w/ all BE-s set to 1 and all debug bus bytes transmitted, irrespective of byte lane content. This resets the SW decoding logic. Useful to recover from losses.
Byte Enables	Hdr1[7:0]	Each bit indicates if the corresponding byte lane of the 8-byte debug bus is changing and transmitted. The total number of bytes in the payload will equal the number of Byte Enables set to 1. The total number of bytes in the trace data packet will be the length of payload + 2 (number of bytes in the header).

Trace Support Packet

Packet Format

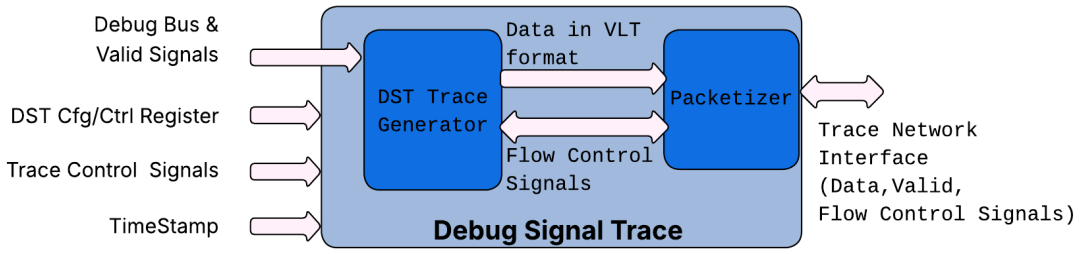
Header1 (Byte-1)		Header0 (Byte-0)				
Support Form (4 bits)	Support Info (4 bits)	**Pkt Type (1'b1)**	Src Id (4 bits)	Pkt Loss (1 bit)	Header Extended (1 bit)	Null Packet (1bit)
Payload0 (Byte -3 - ..[9])		Payload0 (Byte -2)				
Payload: Variable size, May go up to 8 bytes		Payload: Variable size, May go up to 8 bytes				

Header Decode for Trace Support Packet

Name	Bits	Meaning
Packet Type	Hdr0[7]	1'b1: Trace Support Packet. Use this table for decoding Trace Packet.
Source ID	Hdr0[6:3]	Source ID. Can support up to 16 sources.
Packet Loss	Hdr0[2]	Set to 1 if prior packet(s) were lost.
Header Extended	Hdr0[1]	If set to 1, next byte is header, else data
Null Packet	Hdr0[0]	If set to 1, this is a null packet. Used to fill frames.
Support Form,	Hdr1[7:4]	4'b0000: Time Stamp. Use of other fields: Support Info: Not Used Payload: 64-bit timestamp counter 4'b0001: Trace Info Update Use of other fields: Support Specific Info: Not Used Payload: None
Support Info	Hdr1[3:0]	4'b0001: Trace Start, 4'b0010: Trace Stop

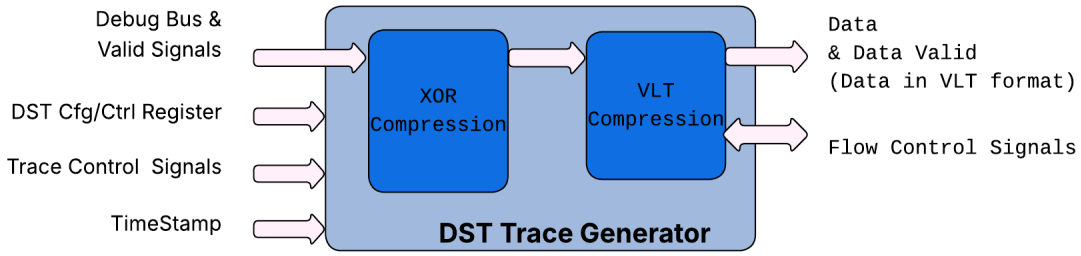
Debug Signal Trace Hardware

Debug Signal has two high level blocks - DST Trace Generator and Packetizer. DST Trace Generator takes incoming debug bus and trace control signals, and compresses the packet to generate “VLT Packets” (details in next section). The Packetizer (reused for Instruction Trace as well) interfaces the Trace Generator with Trace Network.



DST Trace Generator

The DST trace generator converts incoming debug signals, timestamps and generates a compressed data stream called “VLT” packets. Details on different VLT packet formats are listed in the DST Packets section. The DST produces traces based on configuration registers and trace control signals (start, stop, and flush).

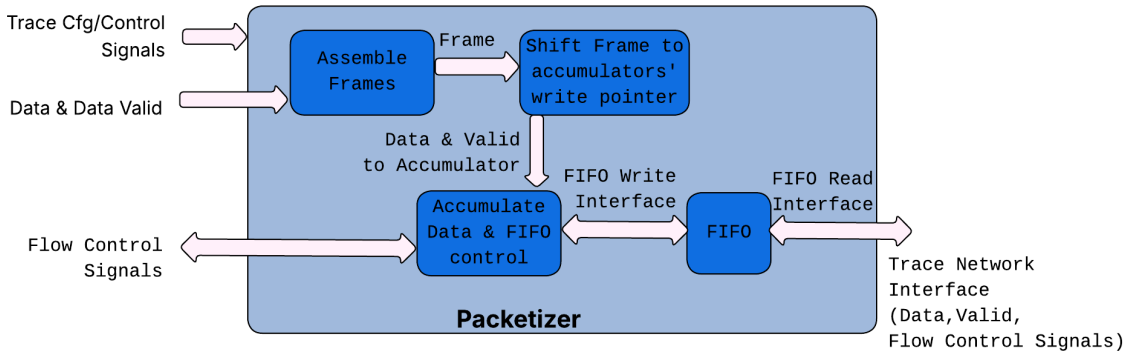


The DST Trace Generator employs a two-stage compression process: XOR compression and Variable Length Transmission (VLT) compression. XOR compression transmits data only when incoming data changes. However, this method is less effective for debug buses that monitor multiple, often uncorrelated hardware signals. For instance, if signals A, B, and C (part of a debug bus) each change on consecutive clock cycles (e.g., A on cycle 0, B on cycle 1, C on cycle 2), and this pattern repeats, XOR compression provides minimal benefit. VLT offers an improvement over XOR. Instead of tracking data changes across the entire width of the debug bus, VLT monitors changes at the byte-lane level. Only the changed bytes are transmitted, with Byte Enables indicating their positions. The "variable length" aspect refers to the fact that the number of transmitted bytes varies according to the debug bus input.

XOR Compression is implemented by ***dfd_xor_compression*** & VLT compression by ***dfd_vlt_packet_compression***.

Details on different VLT packet formats and compression schemes are listed in the chapter DST Packets.

Packetizer



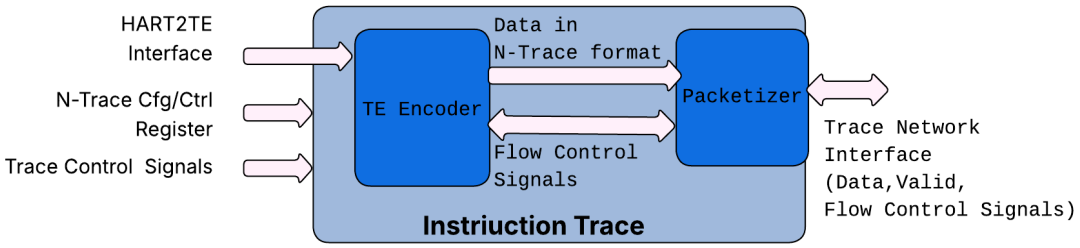
The Packetizer transforms variable-length packets into a continuous data stream, which is then sent to a FIFO. This FIFO's read port serves as the interface to the trace network. The Packetizer comprises four primary blocks:

- ***dfd_frame_filler***: Responsible for assembling incoming packets into frames.
- ***dfd_cross_connect:*** Shifts frames to align with the valid data currently held in the accumulator.
- ***dfd_accumulator***: Accumulates and assembles data, then transfers it for writing to the FIFO.
- ***generic_FIFOMN***: A FIFO used for buffering data and interfacing with the trace network.

Instruction Trace

Instruction tracing enables tracking program flow through program counters. It utilizes the N-Trace format, as detailed in the RISC-V N-Trace (Nexus-based Trace) Spec.

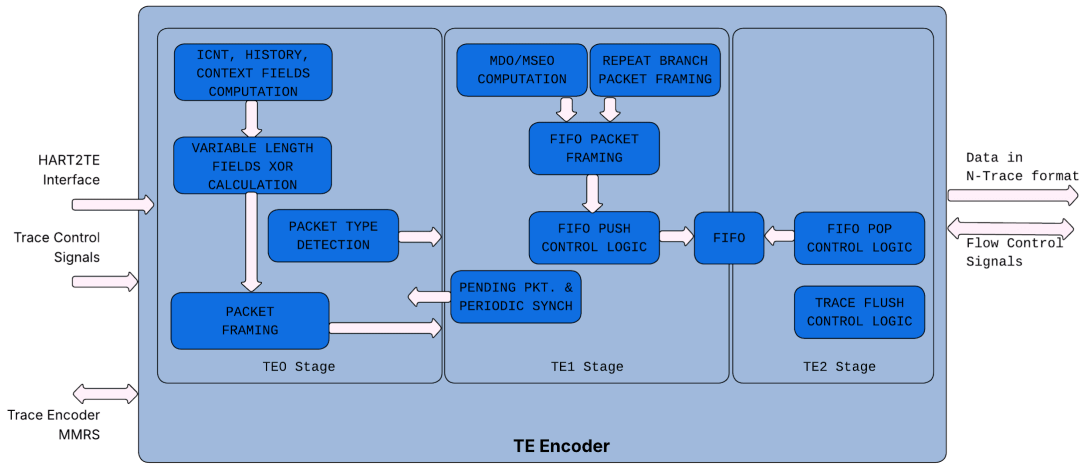
The Instruction Trace comprises a Trace Encoder and a Packetizer. The Trace Encoder interfaces with the Core via the HART2TE Interface, generating N-Trace Packets for the Packetizer. These n-Trace packets are then transmitted by the Packetizer to the Trace Sink over the TNIF interface. Further details about the Packetizer can be found in the dedicated section: Packetizer.



Trace Encoder

The trace encoder block creates an instruction trace, which includes history generation, according to the N-trace packet specification. The trace encoder (TE) processes a maximum of two branches per cycle and operates in two distinct modes. In lossless (throttle) mode, the TE asserts the ***backpressure*** signal to the core (when ***Trtecontrol.Trteinststallena*** == 1) to prevent FIFO overflow in the encoder. The core must stop instruction retire when ***backpressure*** is asserted. Conversely, in loss mode, the TE signals a packet loss using an N-Trace error message if an overflow occurs.

The following figure describes the overview of the interface from hart to the Trace Encoder block:



HART2TE Interface

When the Core executes a jump to a PC that is not aligned with the current instruction's boundary, it sends a "block" to the Trace Encoder. This action commonly occurs during interrupt, exception, or branch instructions. The core communicates details of the block using HART2TE Interface as listed below (This is borrowed from HART2TE interface as listed in RISCV Debug Spec).

List of Trace Encode signals exposed at *tt-dfd* top level.

Core to TE	Description
IRetire	Number of halfwords represented by instructions retired in this block
IType	Termination type of the block
IAddr	The address of the 1st instruction retired in the block
ILastSize	The size of the last retired instruction in the block. 0: Compressed 1: Uncompressed
Priv	Privilege level for all instruction in all the block(s)
Context	Context for all the instructions in the block
Tval	The associated trap values
Tstamp	Time generated by core

TE to Core	Description
Active	Indicates state of Trace Cfg register *Trtecontrol.Trteactive.*
StallModeEn	Indicates state of Trace Cfg register *Trtecontrol.Trteinststallena.*
StartStop	Start Trace. (*Cr4BTrtecontrol.Trteenable & Cr4BTrtecontrol.Trteactive* ==1).
Backpressure	Backpressure to core to avoid packet loss due to TE Encoder FIFO overflow.

Encoder Build Time Parameter

Trace Encoder has following configurable parameters:

Parameter	Description	Default Value
TE_EN_PKT_LEN	Whether packets are generated with packet length sign or with the standard MSEO/MDO bits in the packet 0: MSEO/MDO bits are used 1: Packet length is inserted	0
TE_ICNT_W	Number of bits in the instruction count (I-CNT) field	22 (max)
TE_HIST_W	Number of bits in the HIST field	32 (max)

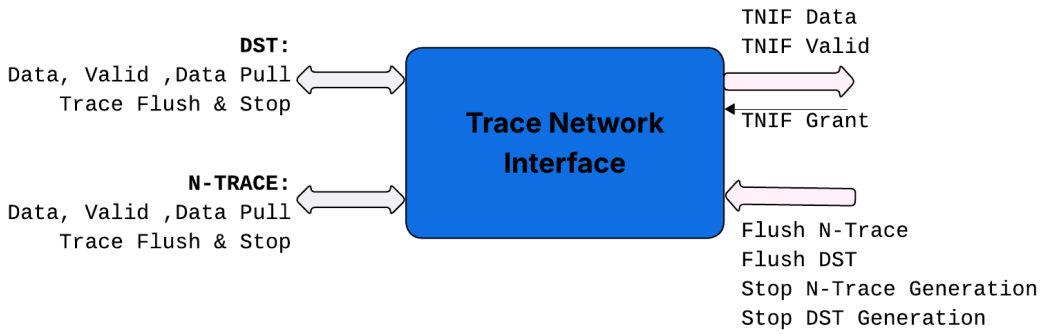
Trace Generator Packets

The following packets are supported by the trace encoder:

Packet	Comments
Ownership Packet	Sent on context and privilege change
ProgTraceSync	Start of the trace
Error	Supported when stalling is disabled
ResourceFull	For both HIST and ICNT bits
IndirectBranchHist(Sync)	HTM mode supported
RepeatBranch	No HRepeat support
ProgTraceCorrelation	Trace stop
VendorDefined	Used for xcause and xtval reporting

Trace Network Interface (TNIF)

The Trace network manages and grants requests from various blocks. Blocks, like a RISC-V core can produce both instruction trace and debug signal trace. The trace network interface locally arbitrates between these two sources before transmitting the trace information.



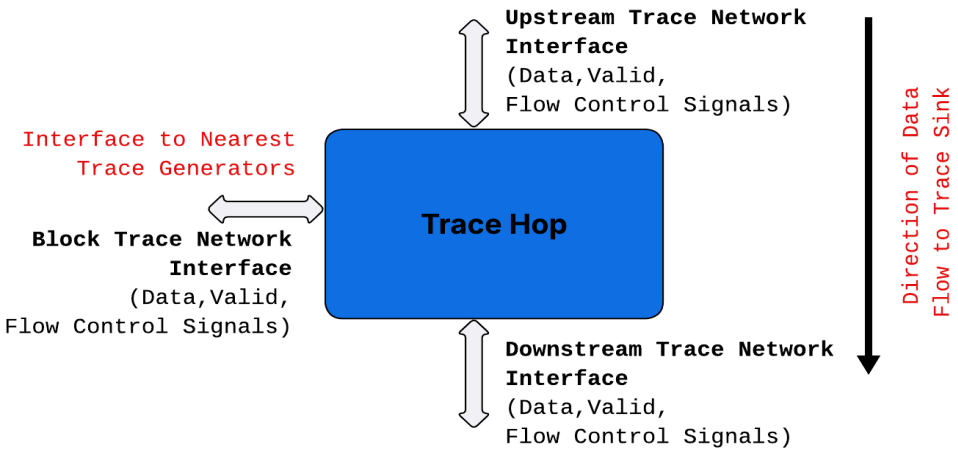
Flush signals force intermediate buffers between the trace generator and sink to empty out. Stop signals, enable the trace sink to apply back-pressure, preventing the generator from producing more data and thus avoiding overflow.

Trace Network

The trace network facilitates the transfer of packets from trace generators to the trace sink. These packets are then stored in trace SRAM or System Memory. The trace network itself consists of multiple trace hops, each with pre-defined build-time parameters for adding repeater flops, which helps in minimizing routing timing constraints.

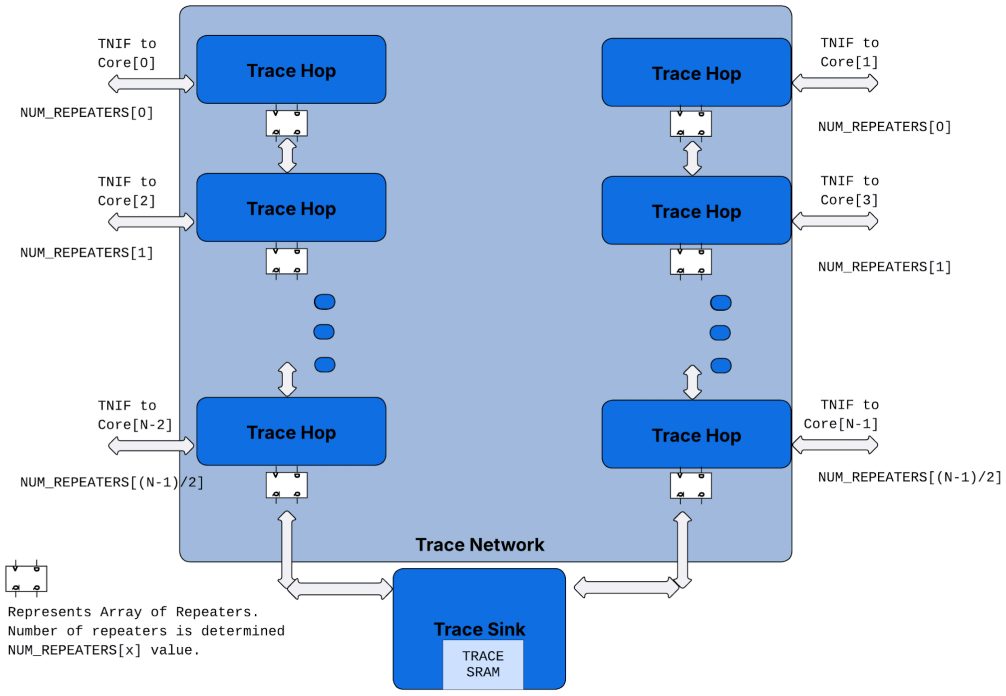
Trace Hops

Trace Hop consolidates trace traffic from upstream Trace hops and the closest trace generator. This combined traffic is forwarded downstream to the trace funnel. Trace Hop includes parameters (*NUM_REPEATER_STAGES*) that allow for the addition of repeater flop stages downstream, which helps to alleviate routing constraints.



Building a trace network

The Trace Sink is designed to accommodate two trace network interface channels. The *tt-dfd* block facilitates the construction of a trace network, offering two symmetrical channels configurable at build time. Currently, the trace network supports up to four blocks per channel, with plans to parameterize this for increased capacity in the future. The diagram below gives pictorial representation of building the trace network.



Trace Grant

Data is “pulled” from the Block (Trace Generator) on "Trace Grant." This grant is generated locally at each trace hop, utilizing time-slots and back pressure signals from the funnel. At each hop, a grant is provided to the connected block based on pre-defined time slots.

If a channel has N/2 enabled blocks, each block receives one grant every N/2 clock cycle. The grant delivery is synchronized across all blocks on the channel, with a delay parameter adjusted according to the physical layout and the number of repeaters.

Trace Funnel

The trace funnel consolidates trace traffic from two channels, directing it to the trace sink. It features a Trace RAM for "SRAM Mode" storage. In "SMEM Mode," the trace funnel utilizes an AXI interface to transfer trace data to system memory.

Trace Sink SRAM

The trace sink SRAM, a 32KB memory, is currently structured as eight 512x64 macrocell instances. This organization facilitates the distribution of macros between DST and N-trace

when both are active. Furthermore, it enables additional banking schemes, allowing for simultaneous read and write operations across different banks.

Programming Guide

Instruction Trace

N-Trace Encoder, and components of Trace Funnel use control and data flow as described in RISC-V Trace Control Interface Specification.

Debug Signal Trace

The RISC-V Trace Control Specification, originally designed for instruction trace, also governs the programming sequence for Trace components for Debug Signal Trace. The registers defined in the Trace Control Interface Specification are duplicated for Debug Signal Trace (DST). The table below details the registers reused by the DST implementation from the Trace Control Specification. Standardizing register definitions for the Debug Signal Trace extension, as per the Trace Control Specification, allows for reuse of SW stack across instruction trace and debug signal trace.

Trace Control Registers		**Name and Fields of equivalent registers for DST**	
Name	*Fields*	*Name*	*Fields*
ttrTeControl		trDstControl	
	trTeActive		trDstActive
	trTeEnable		trDstEnable
	trTeInstTracing		trDstInstTracing
	trTeEmpty		trDstEmpty
	trTeInstMode		
	trTeContext		
	trTeInstTriggerEnable		trDstInstTriggerEnable
	trTeInstStallOrOverflow		
	trTeInstStallEna		
	trTeInhibitSrc		
	trTeSyncMode		trDstSyncMode
	trTeSyncMax		trDstSyncMax

	trTeFormat		trDstFormat
trTeImpl		trDstImpl	
	trTeVerMajor		trDstVerMajor
	trTeVerMinor		trDstVerMinor
	trTeCompType		trDstCompType
	trTeProtocolMajor		trDstProtocolMajor
	trTeProtocolMinor		trDstProtocolMinor
	trTeVendorFields		trDstVendorFields
trTeInstFeatures		trDstInstFeatures	
	trTeInstNoAddrDiff		
	trTeInstNoTrapAddr		
	trTeInstEnRepeatedHistory		
	trTeSrcID		trDstSrcID
	trTeSrcBits		trDstSrcBits
trRamControl		trDstRamControl	
	trRamActive		trDstRamActive
	trRamEnable		trDstRamEnable
	trRamEmpty		trDstRamEmpty
	trRamMode		trDstRamMode
	trRamStopOnWrap		trDstRamStopOnWrap
trRamImpl		trDstRamImpl	
	trRamVerMajor		trDstRamVerMajor
	trRamVerMinor		trDstRamVerMinor
	trRamCompType		trDstRamCompType
	trRamHasSRAM		trDstRamHasSRAM
	trRamHasSMEM		trDstRamHasSMEM
	trRamVendorFrameLength		trDstRamVendorFrameLength
trRamStartLow		trDstRamStartLow	
	trRamStartLow		trDstRamStartLow
trRamStartHigh		trDstRamStartHigh	
	trRamStartHigh		trDstRamStartHigh
trRamLimitLow		trDstRamLimitLow	
	trRamLimitLow		trDstRamLimitLow
trRamLimitHigh		trDstRamLimitHigh	

	trRamLimitHigh		trDstRamLimitHigh
trRamWPLow		trDstRamWPLow	
	trRamWrap		trDstRamWrap
	trRamWPLow		trDstRamWPLow
trRamWPHigh		trDstRamWPHigh	
	trRamWPHigh		trDstRamWPHigh
trRamRPLow		trDstRamRPLow	
	trRamRPLow		trDstRamRPLow
trRamRPHigh		trDstRamRPHigh	
	trRamRPHigh		trDstRamRPHigh
trRamData		trDstRamData	
	trRamData		trDstRamData
trFunnelControl			
	trFunnelActive		
	trFunnelEnable		
	trFunnelEmpty		
trFunnelImpl			
trFunnelDisInput			
	TrFunnelDisInput [7:0]		trFunnelDisInput[15:8]

Programming Sequence

The programming guide follows trace control spec. It is summarized here for completeness.

Enabling Trace

- Release all needed components from reset by setting tr??Active = 1 (Set tr??Active and read back the same)
- RAM Sink Configuration
    - Program the RAM Mode config (RMW)
    - Discovery of trRamStart and trRamLimit (Incase of SRAM mode)
    - Program the required trRamStart/Limit (Both the modes, RMW)
    - Program StoponWrap or Overflow condition
    - Clear the trRamWpLow and trRamWpHigh, trRamRpLow/High (Read back 0’s)
    - Enable RAM (trRamEnable, RMW)
- Funnel Configuration
    - Program the Funnel disinput (If needed to disable few cores for funnel)

    - Enable Funnel (trFunnelEnable)
- Trace Encoder
    - Program Periodic Sync, Stall/Error configs etc, (RMW)
    - Program Vendorfields (Frame length – If needed)
    - Set Enable and InstrTracing (RMW)

Trace Stop and Reading out Trace Data

- Trace Encoder
    - Clear Enable bit (RMW)
    - Wait for trTeEmpty to be High
- Funnel
    - Clear Enable bit (RMW)
    - Wait for trFunnelEmpty to be High
- Sink
    - Clear RamEnable bit (RMW)
    - Wait for trRamEmpty to be High
    - Start the trace RAM reads (in case of SRAM mode)
    - In case of SMEM mode, empty is set when all AXI writes are done
- SRAM RAM Reading
    - StoponWrap Mode
        - Poll for trRamEnable to check for StopOnWrap (Later version)
        - Read WpLow, RpLow
        - Read Ramdata till WpLow == RpLow
    - Overflow Mode (WpLow.Wrap is set)
        - Read from WpLow till LimitLow
        - Read from StartLow to WpLow