# tt-dfd

[**tt-dfd 1**](#_h23bq4qzso0i)

[**Introduction 2**](#_p592b1nn8b5b)

[**tt-dfd 3**](#_s8f2brxsi1z6)

[**Core Logic Analyzer 4**](#_8d8mi1v4mlbe)

[CLA High Level Overview 4](#_c3bhl76mrfow)

[Event 4](#_pw3ufzpwc7e1)

[Trigger on User Defined Function 6](#_33j4wnissi9o)

[Action 6](#_r4o7eg7qjup4)

[Clock Halt Action 7](#_rozmsb4wpiiz)

[Custom Actions 7](#_r9zaid6e24jh)

[Event-Action Pair (EAP) 7](#_o6d0i34ddh1x)

[Sequential Linking of EAPs 9](#_d2vcrtd64vwq)

[CLA Registers 11](#_hr1ypc4brx6v)

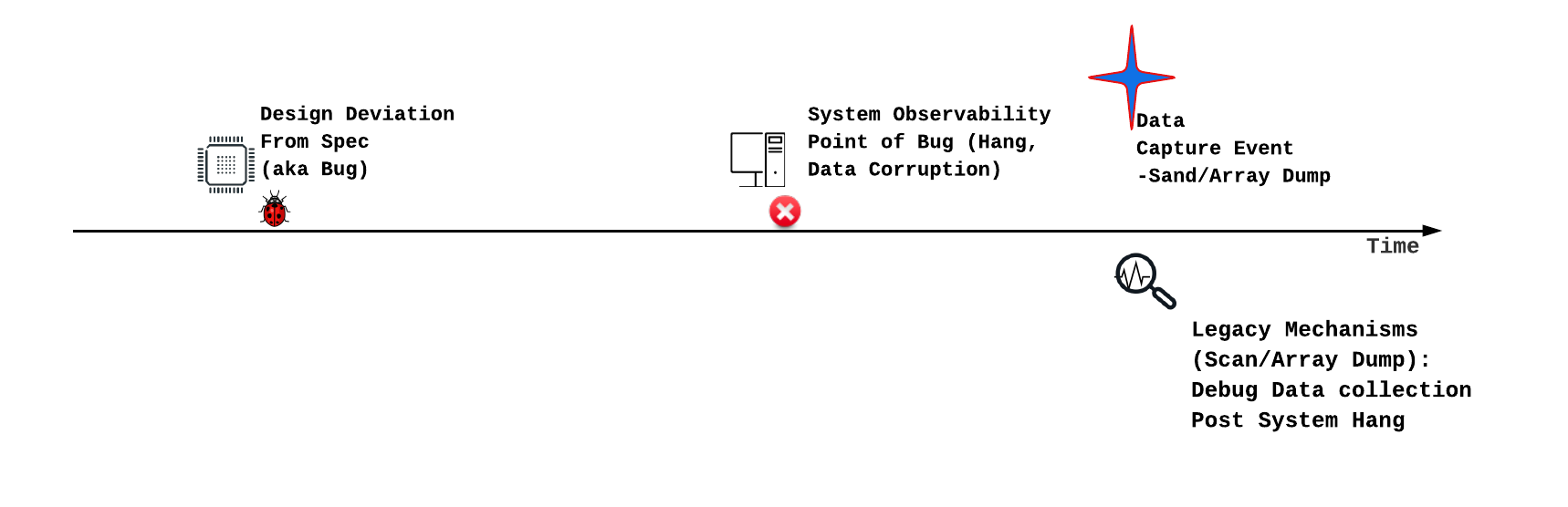
[CLA Programming Support 13](#_wu0xz9itect)

[**Debug SIgnal Trace 13**](#_xk9qdgpnpvmk)

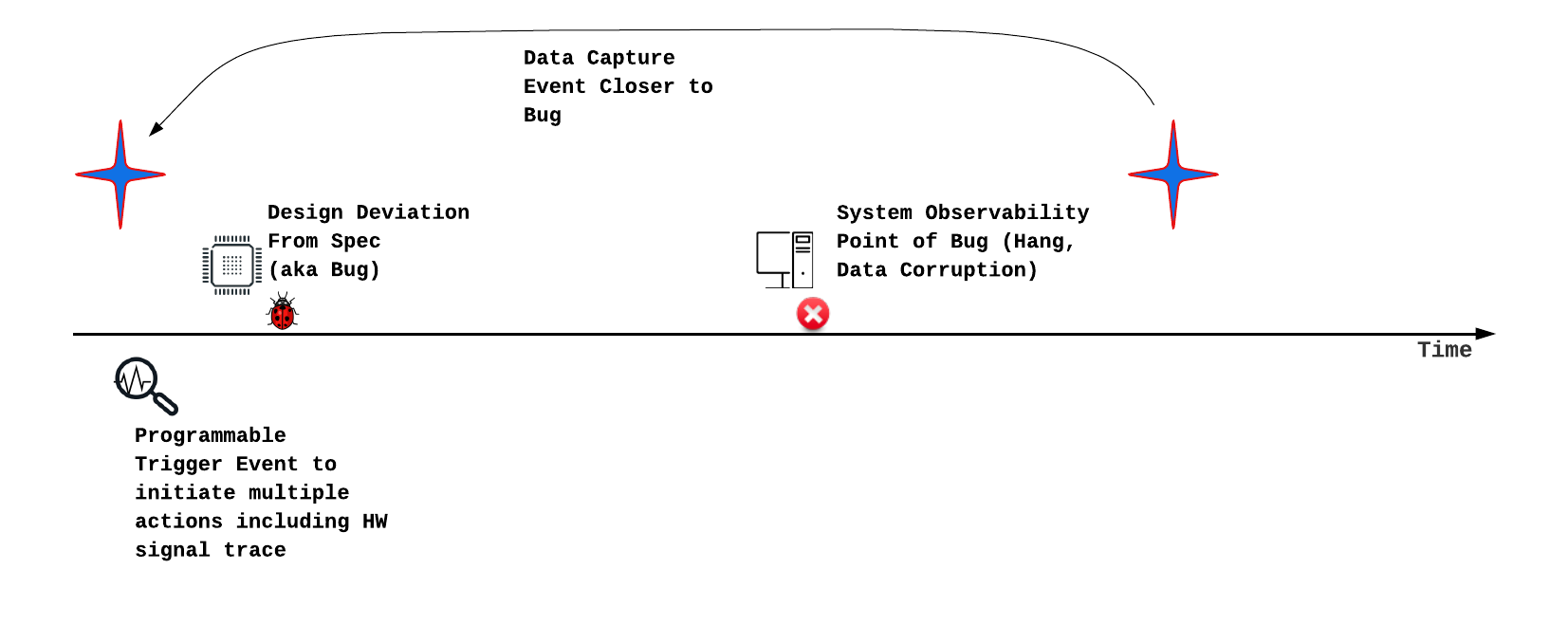
# 

# 

# 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.

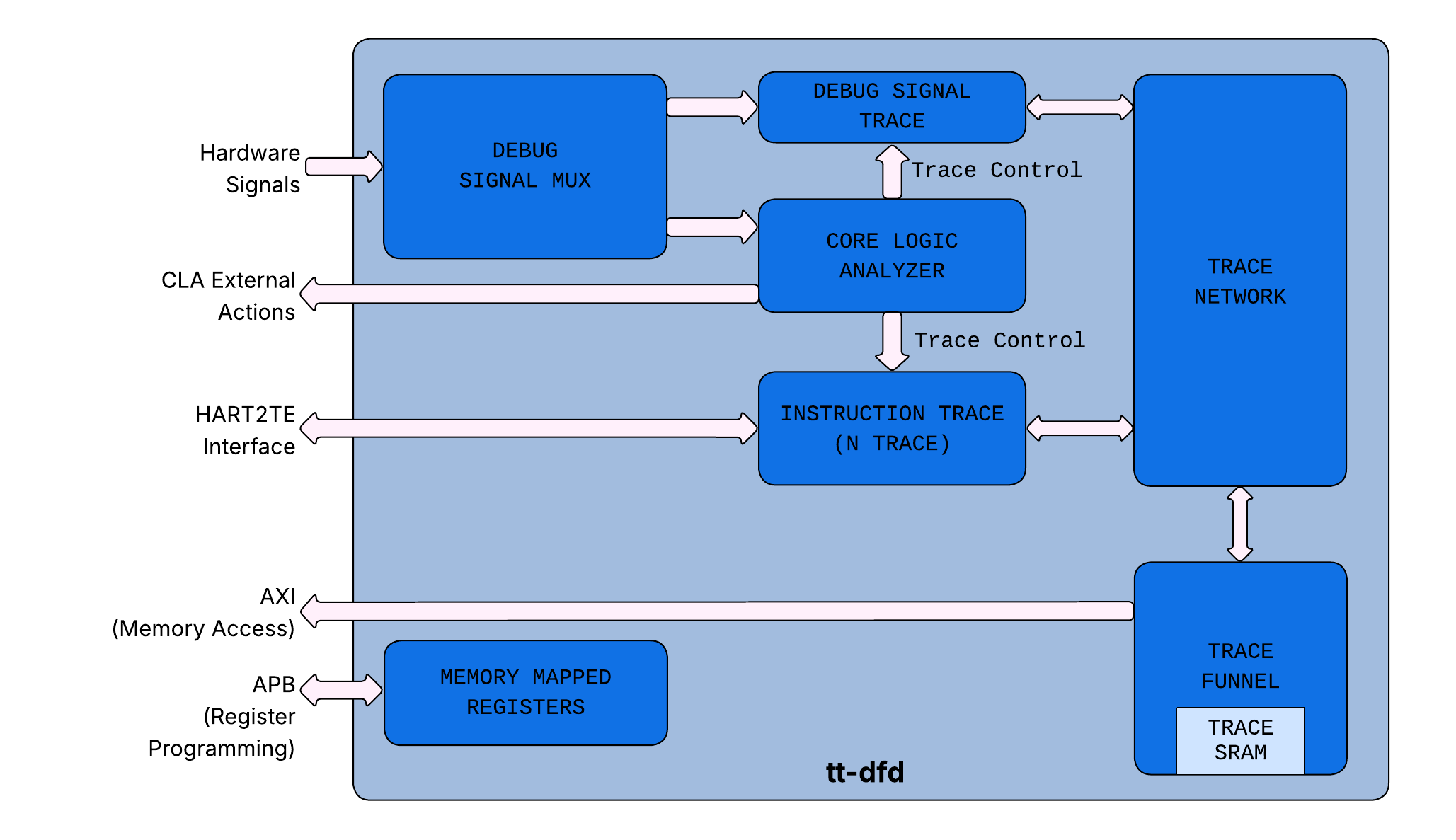


Subsequent chapters detail Core Logic Analyzer and Debug Signal Trace design elements, which are designed to achieve this objective. Beyond Debug, Core Logic Analyzer can also be used for performance analysis, hardware bug workarounds and generating reactive stimuli. These Design for Debug (DfD) blocks 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:** This 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 & 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.

# 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-0x3F | Future Use | Future Use |

# 

# 

## 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](#_rozmsb4wpiiz) 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 .

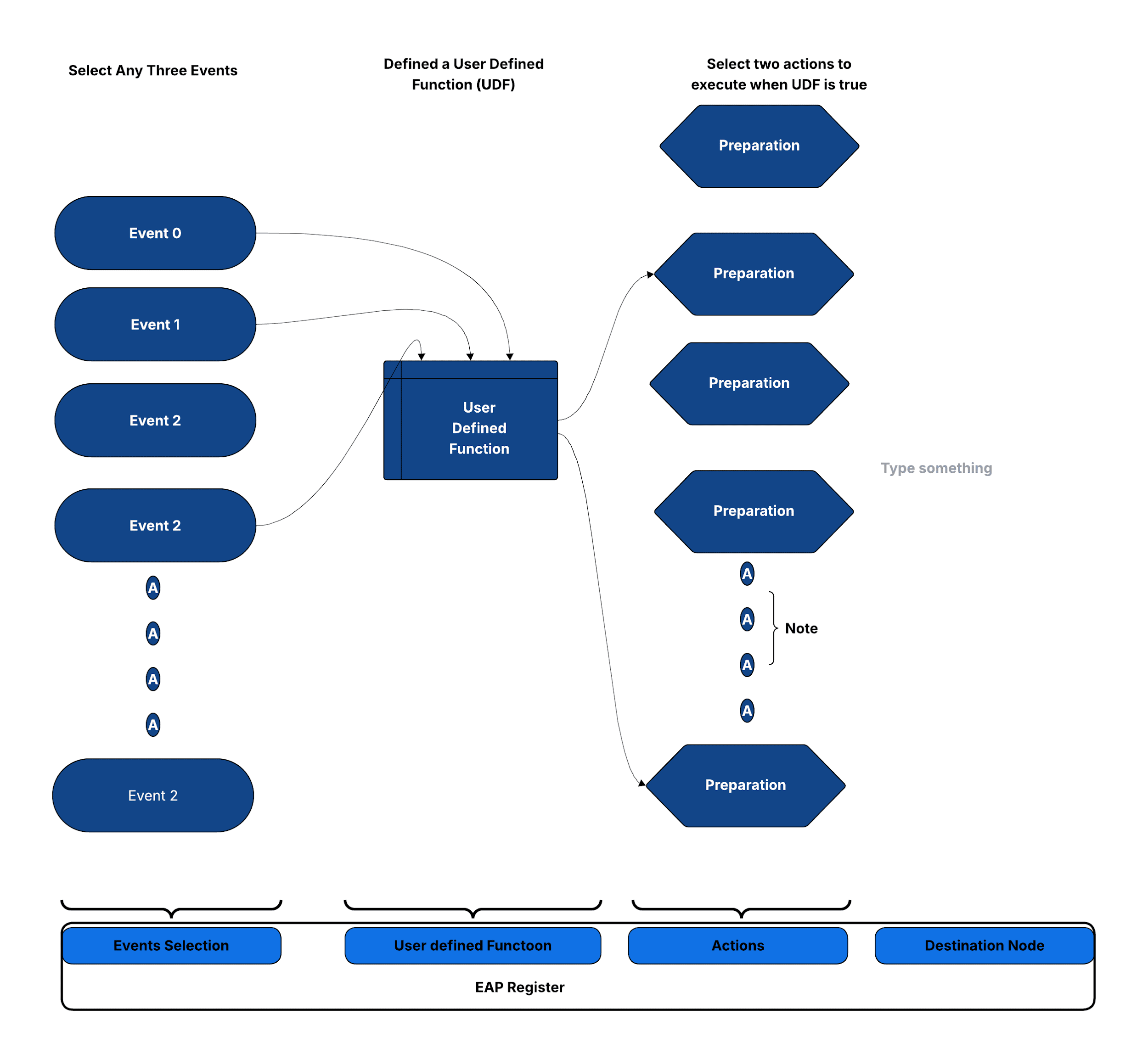
Here is the truth table for the same:

| 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)

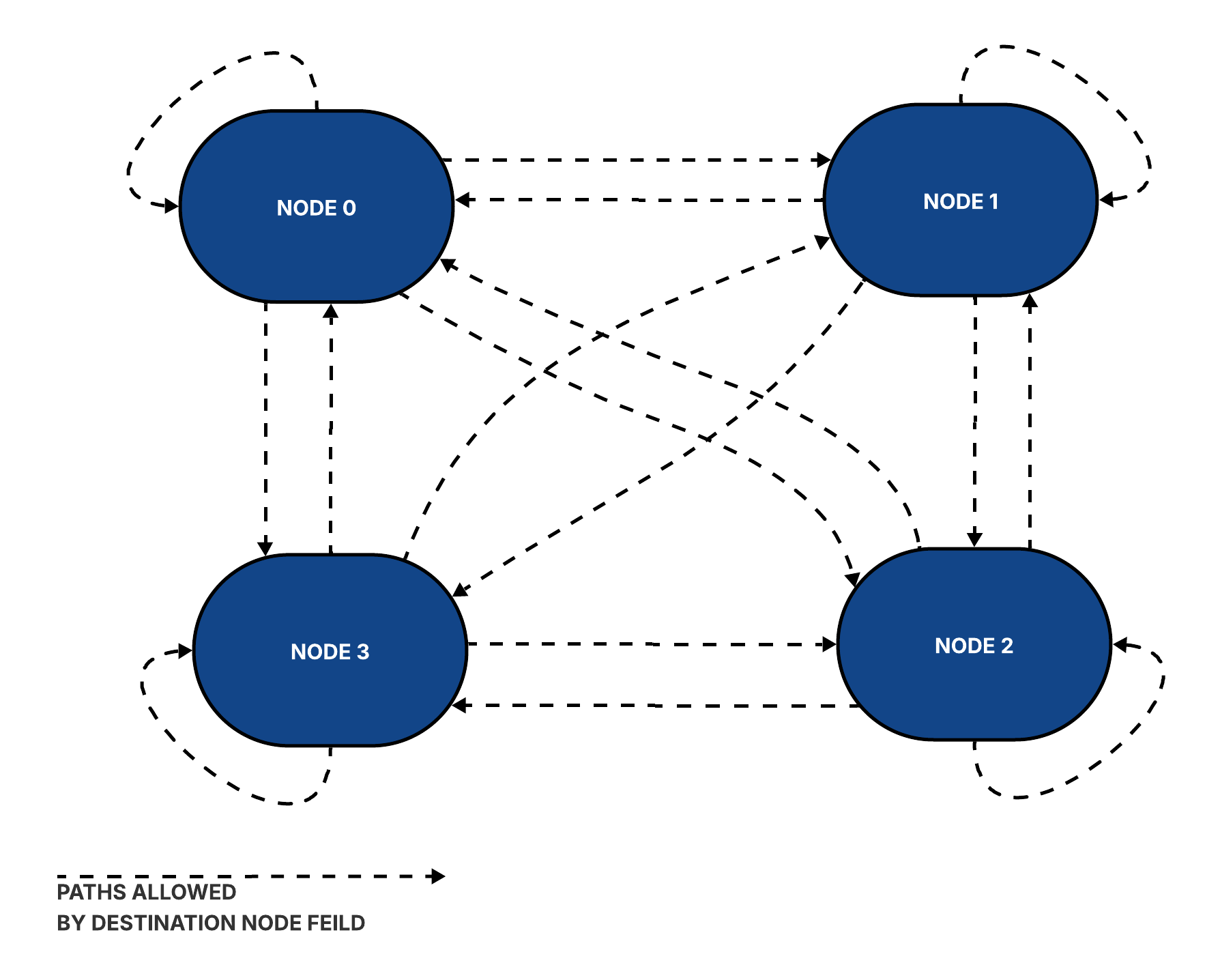
# 

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*

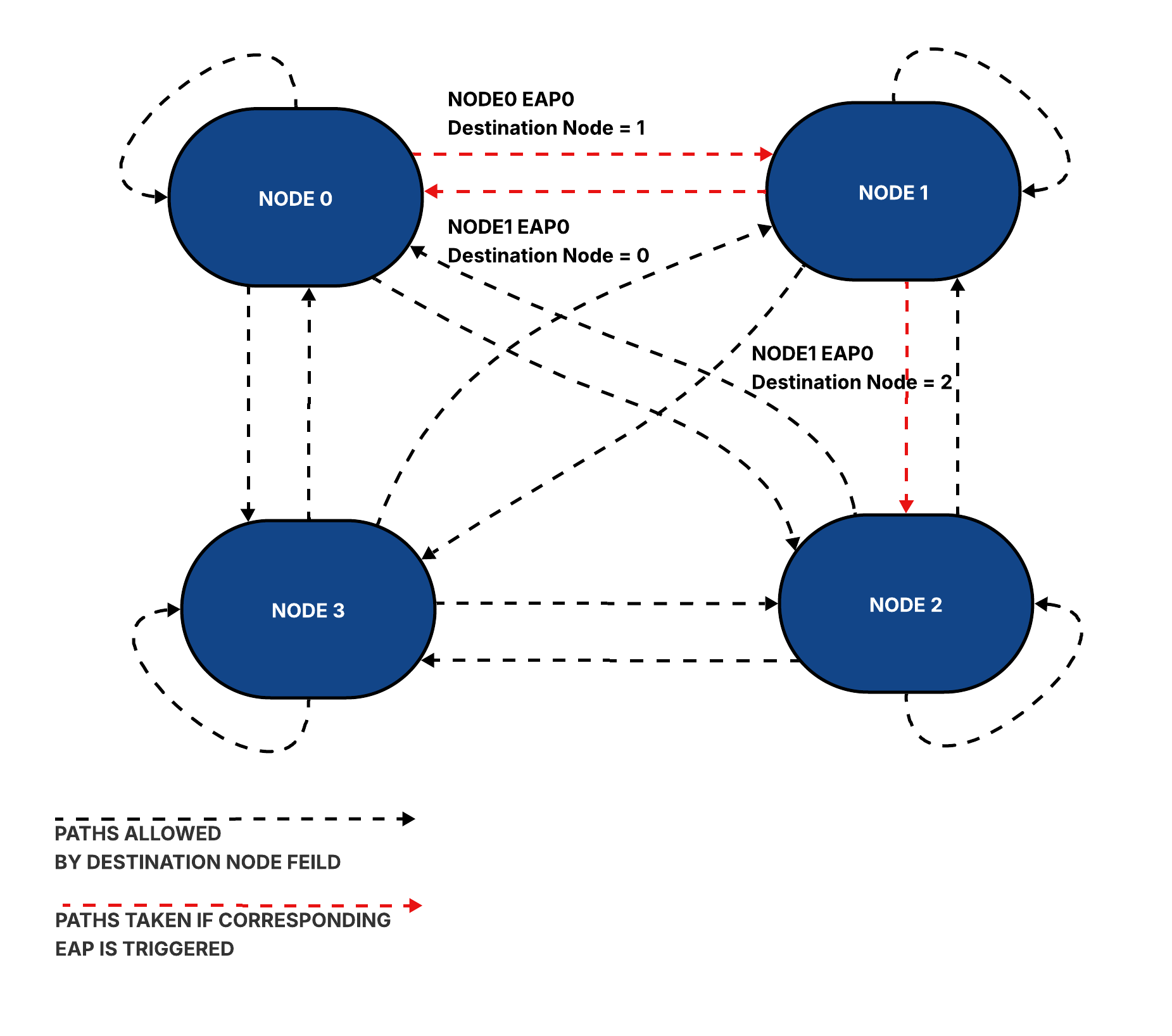
*Node1 EAP1*

*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](https://github.com/tenstorrent/tt-dfd/blob/main/rtl/mmr/html/cla_csr.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> CDbgNode2Eap<0,1,2,3> CDbgNode3Eap<0,1,2,3> | Action(s) are tied to events(s) using "event-action pairing" registers. CLA supports 4, Each Node has 4 EAPs. |
| 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 the finegrain timestamp counter based on a cross trigger to sync the time value across the multiple cluster's 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.

CLA ports for input/output

# Debug Signal Trace

TBD