Trace and Debug Specification

| Revision # | Author | Date | Notes |
| --- | --- | --- | --- |
| 0.1 | Jim Bauman | 6 Apr 2015 | Initial Strawman |
| 0.2 | Jim Bauman | 6 May 2016 | First pass refinement to kick off implementation. |
| 0.3 | Jim Bauman | 10 Jun 2016 | Major update – first cut at detailed description of trace probes, incorporates MIPI STP v2 compliance. |
| 0.4 | Jim Bauman | 21 Jun 2016 | Added detail for NetSpeed Trace Protocol for transporting ATB over NoC via trace masters and trace slave. |
| 0.5 | Jim Bauman | 18 Jul 2016 | Updated address packet description to reflect refinements in format consistent with first SVN check-in of coresight.pl. |
| 0.51 | Jim Bauman | 25 Jul 2016 | Added section 4.1.9 describing NocStudio properties and parameters needed for trace probes (for initial implemementation).  Section 4.1.10: initial list of interface signals for trace probes.  Updated arch overview diagram at beginning of chapter 2 to reflect current direction. |
| 0.52 | Jim Bauman | 6 Nov 2016 | Updated probe interface signal list.  Updated trace framing for AR and AW channels to cover all control fields.  Completed definition of trace framing for AC and response channels (R, B and CR).  Added some detail about trace filtering. |
| 0.53 | Jim Bauman | 9 Dec 2016 | Refined definitions of trace framing for all channels.  Updated probe interface signal list with (hopefully) final signal naming to be used in RTL.  Added section 4.1.7 documenting use of MERR to communicate FIFO overflows.  Highlighted need to prioritize response channel servicing over address channel in section 4.1.4.4. |
| 0.54 | Jim Bauman | 15 Dec 2016 | Section 4.1.4.5.5: Packet content select – filled in more detail.  Section 4.1.5: Timestamps - overhauled with detail about how timestamps are to be implemented.  Section 4.1.6: Sync requests – updated with more requirements for being precise about how these are handled (driving by timestamping and address compression requirements).  Section 4.1.9.1: Low Power – added, describes how probes behave with power state changes, including insertion of GERR packet. |
| 0.55 | Jim Bauman | 30 Dec 2016 | Section 4.1.2: Registers – added detail  Section 4.1.9: added this section about how we used authentication interface.  Misc smaller updates. |
| 0.56 | Jim Bauman | 10 Feb 2017 | Section 4.1.4.5.1.1: include AWUNIQUE with AWSNOOP.  Section 4.1.4.5: packet header updated so bit 0 is Format Mode, bit 1 is packet type (always 0 currently).  Section 4.1.4.5.5: updated to document finalized trace format parameters, mechanisms, etc. |
| 0.57 | Jim Bauman | 21 Feb 2017 | Section 4.1.3: Trace Filtering – updated with rough description of registers and logic for first release.  Section 4.1.4.5.5.3: rough notes on various internal id signals used in NS RTL. |
| 0.58 | Jim Bauman | 27 Feb 2017 | Section 4.1.3.3: Trace Filter Control – completed documentation of the 3rd (control) layer in the filtering logic. |
| 0.59 | Jim Bauman | 28 Feb 2017 | Section 4.1.4.6: Special handling of AC/CR in slave bridges – added. |
| 0.60 | Jim Bauman | 8 Mar 2017 | Section 4.1.9.2: Authentication Control – overhauled to comply with ARM Arch Spec C5.4. |
| 0.61 | Jim Bauman | 14 Mar 2017 | Added Section 4.1.4.5.1.4: Address LSB Adjustment  Refined Section 4.1.4.6.4: AC chan common field suppression  Refined Section 4.1.9.2: Authentication Control  Refined Section 4.1.9.3: Graceful transition… |
| 0.62 | Jim Bauman | 15 May 2017 | Added Section 4.1.5.4: Reduced Width Differential Timestamp in Channels  Substantial Updates, still a work in progress, to sections 4.1.3.4 through 4.1.3.7 regarding Events, Start/Stop, Trigger packets and XSYNC packets. |

Table of Contents

[1 Trace and Debug High Level Requirements 7](#_Toc482624285)

[1.1 CoreSight Compliance 7](#_Toc482624286)

[1.1.1 High-Level Requirements of CoreSight Visible and Reusable Components 8](#_Toc482624287)

[1.1.2 Multiple CoreSight Reusable Components 8](#_Toc482624288)

[1.1.3 Debug Access Port 9](#_Toc482624289)

[1.1.4 Trace Port Interface Unit (TPIU) and Embedded Trace Buffer (ETB) 9](#_Toc482624290)

[1.1.5 Cross triggering – Event Interfaces: Cross Trigger Interface (CTI) and Channel Interface 10](#_Toc482624291)

[1.2 MIPI STP v2.2 Compliance 10](#_Toc482624292)

[2 NetSpeed Debug Architecture 10](#_Toc482624293)

[2.1 Debug APB Interface 11](#_Toc482624294)

[2.1.1 Security 11](#_Toc482624295)

[2.2 Tracing 11](#_Toc482624296)

[2.2.1 NS Trace Probes 12](#_Toc482624297)

[2.2.2 NS Trace Master 13](#_Toc482624298)

[2.2.3 NS Trace Slave 13](#_Toc482624299)

[2.2.4 NS Trace Protocol 13](#_Toc482624300)

[2.2.5 ATB Interfaces 15](#_Toc482624301)

[2.3 Error Capture, Interrupts, Status and Statistics 15](#_Toc482624302)

[2.4 Fencing and Draining 15](#_Toc482624303)

[2.4.1 Regbus Debug Fencing Control 16](#_Toc482624304)

[2.4.2 Centralized Debug Fencing Control 16](#_Toc482624305)

[2.4.3 NocStudio Interface 17](#_Toc482624306)

[3 MicroArchitectural Building Blocks 17](#_Toc482624307)

[3.1 Multi-Write-Port FIFO 17](#_Toc482624308)

[3.2 ATB Funnel 17](#_Toc482624309)

[3.3 Internal Event Bus Filter 18](#_Toc482624310)

[4 MicroArchitectural Details 18](#_Toc482624311)

[4.1 Trace Probes 18](#_Toc482624312)

[4.1.1 ACE/AXI Channels Map to STP v2 Master IDs 18](#_Toc482624313)

[4.1.2 Registers 19](#_Toc482624314)

[4.1.3 Trace Filtering 20](#_Toc482624315)

[4.1.4 Trace Framing 30](#_Toc482624316)

[4.1.5 Timestamp 44](#_Toc482624317)

[4.1.6 Sync Requests and Sync Packet Sequences 48](#_Toc482624319)

[4.1.7 Error Conditions Signaled by STP MERR Packets 48](#_Toc482624320)

[4.1.8 Trace Sampling Point 49](#_Toc482624321)

[4.1.9 Tracing Enable, Authentication Control and Invasive Mode 49](#_Toc482624322)

[4.1.10 Reset, Clocking and Power 52](#_Toc482624323)

[4.1.11 Miscellaneous 53](#_Toc482624324)

[4.1.12 Software (NocStudio) Support 53](#_Toc482624325)

[4.1.13 Trace Probe Interface Signals 54](#_Toc482624326)

[5 Implementation Plan 55](#_Toc482624327)

[5.1 Basic ATB Address Stream – AXI4 Bridge 55](#_Toc482624328)

[5.2 Basic Trace Master and Trace Slave 55](#_Toc482624329)

[6 References 56](#_Toc482624330)

Table of Figures

[Figure 1 NetSpeed Debug Architecture 10](#_Toc482624331)

[Figure 2 Packing MIPI STP v2 into ATB 31](#_Toc482624332)

# Trace and Debug High Level Requirements

This specification covers the set of features provided by NetSpeed NoCs that assist with error logging and handling and post silicon debug. There are several aspects to our solution:

1. Error event capture and reporting: generation and delivery of maskable interrupts, register access via regbus to extract details and to support SW intervention in error handling.
2. Status and statistics: regbus reporting of interesting status, filters and counters (outside of error events).
3. Packet tracing: non-invasive, real time capture of packet data entering and exiting the NoC, transmission of that data through the NoC to a target that forwards the data to on chip storage or to an off chip interface.
4. Debug port access to regbus: a means of supporting connection of an external agent that can access the NoC register space.
5. ARM CoreSight Compliance: our solution meets the requirements of a “CoreSight reusable component” to allow seamless integration in a platform level CoreSight trace and debug implementation.

This last aspect, CoreSight compliance, drives many of the details of our solution, so it is discussed further in the next section. There are a number of ARM documents and specifications that are relevant and should be read as background. These are listed in “section 6: References.” Of particular interest are the first three:

1. ARM Whitepaper: CoreSight Technical Introduction - A quickstart for designers
2. CoreSight Technology System Design Guide
3. ARM CoreSight Architecture Specification v2.0

## CoreSight Compliance

CoreSight provides a framework for implementing debug visibility in SoCs with a standard interface that supports a large ecosystem of compliant 3rd party HW and SW debug tools. Many of NetSpeed’s customers use CoreSight components in their SoCs, and it is important that our debug features can be easily integrated into this platform. While much of this ecosystem is built around ARM CPU based SoCs, the CoreSight platform is generic enough that it may be used in non-ARM-centric systems as well.

ARM provides a number of components for building a CoreSight debug infrastructure, including the debug access port, compatible processor cores, processor trace units, system trace units (e.g., AHB Trace Macrocell aka HTM), trace interconnect and buffers, etc. These components all provide a standard programming model with features supporting topology detection, described by ARM as the “visible component architecture.” Debug IP that is designed for integration into a CoreSight system must furthermore be compliant with ARM’s “CoreSight reusable component architecture,” which specifies the physical interfaces and rules that must be followed for a component to be used with other CoreSight components. Being a reusable component implies compatibility with the visible component architecture.

NetSpeed’s debug features are packaged as CoreSight reusable components.

### High-Level Requirements of CoreSight Visible and Reusable Components

Details of the CoreSight “visible component architecture” and “reusable component architecture” are provided in Parts B and C respectively of the ARM CoreSight Architecture Specification v2.0.

The visible component architecture defines the programmer’s model, which is a set of 32-bit registers, some required, some optional, and others component specific, that occupy a well-defined address map consisting of one or more contiguous 4KB blocks. Our solution must present the register API for controlling our debug features in this format.

The reusable component architecture defines several key aspects:

1. Debug APB: all CoreSight components connect to a dedicated 32-bit APB bus with its own address space into which the visible component register sets are mapped. See “AMBA APB Protocol Specification v2.0.”
2. AMBA ATB interface: ARM’s protocol for carrying trace data around an SoC, and any component that generates trace data is a master, and any component that receives trace data is a slave. See “AMBA 4 ATB Protocol Specification (ATB v1.0 and ATB v1.1)” for details.
3. Several other interfaces: event interface, channel interface, authentication interface and timestamp interface. We likely need to support some, maybe all of these.
4. Topology detection: interface requirements to support this.

Since we implement CoreSight reusable components that support tracing, we provide both a debug APB interface to the programming model (register set) as well as an ATB interface which delivers the captured trace data. We also provide an event interface to support cross-triggering, a timestamp interface (note: this does not appear to be implemented in ARM HTM), and we support the authentication interface.

### Multiple CoreSight Reusable Components

Debug functionality will be distributed throughout the elements of the NoC, each piece with its own local set of related registers formatted to meet the requirements of ARM’s specification of a visible component interface. NoC-wide debug instrumentation (e.g., debug fencing controls) will appear as another visible component with an associated register interface.

Access to this register space will be provided via an APB port in our regbus tunnel, though we may also offer local APB slaves to directly access registers within each component. Some investigation is required to determine how best to segregate and translate the debug address space from mission mode address space.

As indicated by the following text, it might be feasible to represent all the NS CoreSight components deployed within a NoC as a single large custom debug component. From p C1-66 of ARM CoreSight Architecture Specification v2.0:

“You can create a component that performs a number of functions internally as separate components, but presents only one set of the re-usable component interfaces externally. This is encouraged as a means to implement pre-built platforms with the CoreSight infrastructure already integrated. You can integrate the platform into a larger system as if it is a single CoreSight component.”

However, we will instead present debug module as a separate CoreSight reuseable component (each with its own visible component register map). This is far more consistent with the CoreSight ecosystem that exists in the world today.

### Debug Access Port

The Debug Access Port (aka DAP) is a standard ARM CoreSight component that provides several bridges between external debug ports (e.g., JTAG, 2 wire serial, etc.) and on-chip debug resources. As such, it is central to implementation of CoreSight on any SoC. Presumably any CoreSight compatible implementation will provide an instance of this component.

It provides the following bridges to internal resources (not all may be deployed in a given implementation):

1. Debug APB Master (APB-AP): all CoreSight reusable components, including the debug feature set of our NoC, are accessible over this interface.
   1. The DAP provides an APB input that is multiplexed onto the Debug APB interface to allow system access to debug functionality (e.g., the CPU subsystem could drive this port to gain internal access to CoreSight components).
2. System Access AXI or AHB Master (AXI-AP or AHB-AP): this interface can be connected into the SoC system interconnect to gain read and write access to system address space.
3. JTAG Access Port (JTAG-AP): this can be connected internally to components that provide some debug access and control via a JTAG interface.

A CoreSight implementation using a NetSpeed NoC would connect the Debug APB Master interface to the NoC debug APB interface and potentially other components, possibly via connections through the NoC. It might also connect the System Access AHB Master to an AHB bridge to the NoC to gain access to system address space in the design.

Sections 2.2.2 and 4.2.2 of the CoreSight Technology System Design Guide provide additional detail, and also see section C2.1 of the ARM CoreSight Architecture Specification v2.0.

### Trace Port Interface Unit (TPIU) and Embedded Trace Buffer (ETB)

The Trace Port Interface Unit (aka TPIU) and the Embedded Trace Buffer (aka ETB) are 2 CoreSight trace sink components. The components receive trace data over an ATB interface and either forward it to an external trace port in the case of the TPIU, or capture the data in an internal buffer in the case of the ETB.

Any SoC implementing tracing functionality will presumably instantiate one or the other of these components, possibly both. Our tracing features will deliver trace data over an ATB interface so that it may be seamlessly consumed by such components. Note, there are other CoreSight components such as funnels and replicators that might be used in an SoC in the trace processing datapath. Our ATB interface may be connected to such elements as well.

See sections 2.2.4 and 2.2.5 of the CoreSight Technology System Design Guide for additional detail.

### Cross triggering – Event Interfaces: Cross Trigger Interface (CTI) and Channel Interface

CoreSight defines a couple of simple protocols for passing event information between components over a small number of wires. There are some standard CoreSight components that handle multiplexing and interconnection of these signals amongst multiple components (called a “cross trigger matrix”) that are called collectively the Embedded Cross Trigger (aka ECT). This signaling is generally outside the scope of the NetSpeed NoC, but our tracing modules provide a set of trigger/event interfaces that are compliant with these protocols so that customers may connect this logic into a cross trigger matrix.

See sections C3 and C4 of the ARM CoreSight Architecture Specification v2.0 for additional detail.

## MIPI STP v2.2 Compliance

MIPI has published a set of specifications related to trace and debug, certainly influenced by ARM as well as other vendors. ARM’s more recent specifications for CoreSight components, e.g., ARM CoreSight STM-500 System Trace Macrocell (STM) TRM, claim compliance to MIPI System Trace Protocol (STP) Specification (v2.2). MIPI STP is a “base protocol” that standardizes common features of the protocols implemented by each trace source, which each have their own specific definitions. Our trace probes define their own protocols, and these are built upon (and are therefore compliant with) MIPI STP v2.

# NetSpeed Debug Architecture

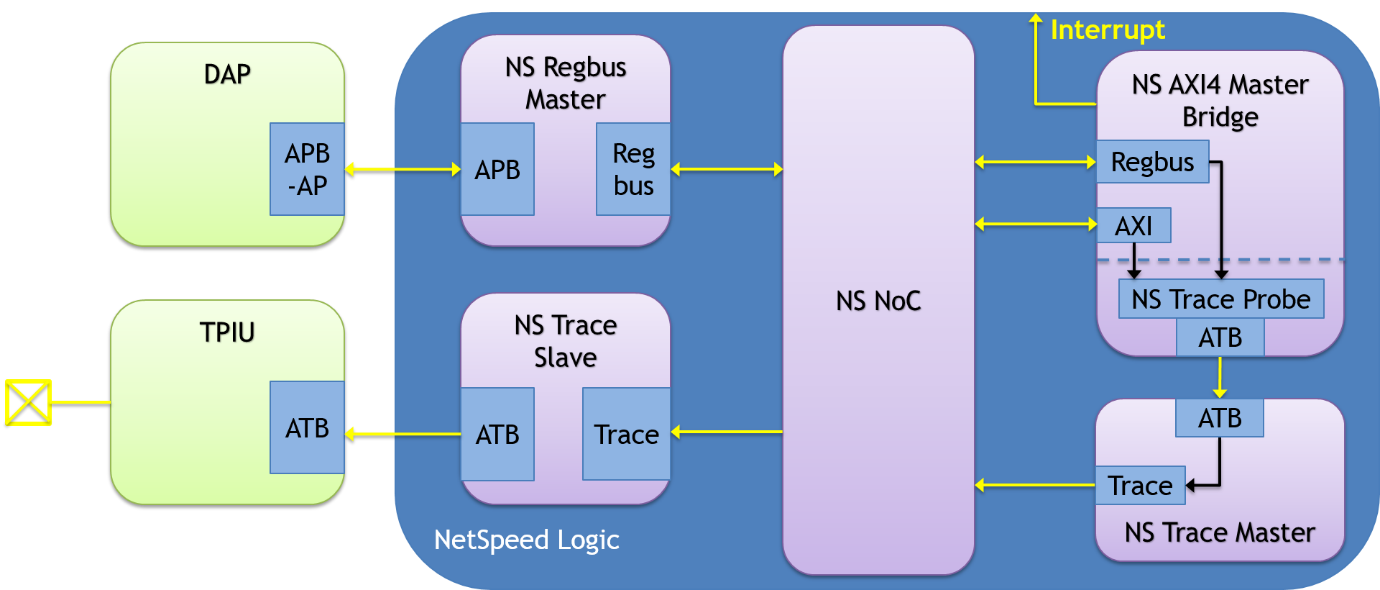


Figure 1 NetSpeed Debug Architecture

1. Debug APB Interface: 32-bit APB bridge to regbus (as shown above, integrated into regbus master) that provides access to the register space that controls our debug features.
   1. Register map for debug features is compliant with CoreSight visible architecture, and it presents the full set of debug features as one or more CoreSight reusable components.
   2. Some of the registers might be locally implemented within the bridge, others will be local to NoC elements and access is provided over regbus.
2. Tracing
   1. NS Trace Probes: instrumentation that is instantiated as companion modules at our master and slave bridges that can capture transactions in real time, filter and format the data into CoreSight compliant ATB streams.
   2. NS Trace Masters: translate ATB streams into NoC protocol for transportation across the die.
   3. NS Trace Slaves: receivers of trace data sent across NoC from NS Trace Masters, multiplexed and formatted for delivery over ARM ATB interface.
3. Error capture and maskable interrupts
4. Status and statistics registers

## Debug APB Interface

This is a 32-bit APB bridge to our regbus. This needs further definition…

Some issues to work through

1. How should debug functionality interact with normal system functionality in the regbus (ideally for non-invasive debug, not at all)?
   1. How does the debug bus address map overlay the regbus address map? Are they one and the same?
   2. If there are multiple regbus address maps (debug and normal operation), how is access to normal state provided for debug purposes?
2. To the extent debug transactions occur at the same time as normal operational transactions, how do we isolate them, prioritize, arbitrate, etc. to minimize the amount the system is perturbed by debug activity?

### Security

Per section 1.4.3 of the AMBA AHB Trace Macrocell (HTM) TRM, all unprivileged accesses must be blocked from accessing Debug APB.

## Tracing

Our tracing features support non-invasive monitoring of transactions at the master and slave interfaces to the NoC. Due to the decentralized nature of NetSpeed NoCs, it is not generally possible to monitor at a single location to observe all or even a significant portion of network activity. That said, we may want to also implement tracing functionality at router nodes in the network, perhaps as a longer term enhancement, as these can provide a more centralized view of network activity, varying with specific implementations.

Our solution is comprised of 3 basic components:

1. NS Trace Probes: these capture transactions at master and slave bridges and generate Coresight compliant ATB trace streams.
2. NS Trace Masters: these are master bridges that take in ATB trace streams and transmit them over the NoC to a single NS Trace Slave.
   1. CoreSight networks are structured as a tree funneling into a root where trace data is captured and/or shipped off-chip. The flushing mechanism of the ATB protocol does not obviously support a structure where trace streams do not all converge into a single multiplexed stream.
3. NS Trace Slave: this receives traces from one or more NS Trace Masters and delivers them in a multiplexed stream over an ATB interface.

To be truly non-invasive, the trace data should be passed over an isolated layer in the NoC, with dedicated bridge interfaces. However, it might be useful to support an area optimized implementation option that allows trace data flows to share common NoC resources with normal operational traffic.

### NS Trace Probes

Our initial implementation supports two trace probe modules, one for our ACE master bridge and its derivatives, and one for our ACE slave bridge and its derivatives. The functionality is largely common between the two and is implemented in a modular fashion to maximize design reuse. Both non-invasively monitor the host interface of the bridge being probed and provide a programmable set of filtering and triggering features to give some selectivity in what transactions are captured in a trace and what portions of those transactions are included (i.e., address, control, data, etc.). The trace output is delivered over an ATB interface.

ARM provides a standard CoreSight component for tracing an AHB bus, described in detail in the AMBA AHB Trace Macrocell (HTM) TRM. Our tracing functionality will provide a similar set of features, tuned to make optimal use of HW we already have (e.g., address lookup tables). Matching the feature set and register interface of the HTM as closely as possible may provide value in minimizing effort to integrate our solution into existing CoreSight deployments, so that is the direction we are taking.

Following is a brief enumeration of filtering and event/trigger generation features of the HTM that we should consider including in our own implementation. Each feature should be implemented to match the HTM as closely as possible. We should provide some configurability of which features are included (and in what quantities) to provide users flexibility in trading off feature richness for area.

* Address Comparators
  + Masters: leverage address map logic for address range events
    - This is one aspect where our functionality must differ from the HTM, but there is huge value in reusing the existing address table logic.
  + Masters and Slaves: optional configurable address & address range match registers
* Control Comparators: set of configurable control field registers
* Error Status
* Counters
* Sequencer
* External trigger inputs, event outputs (CTI)
* Events may be combined via Boolean functions and used to generate derived events

The AHB interface probed by the HTM module represents only a single shared read/write channel. One key challenge in probing at our ACE/AXI bridge interfaces is that they implement multiple channels of simultaneous activity. This ultimately may limit how closely the functionality of our probes may match that of the HTM.

Internally the HTM captures trace data into a circular buffer that can sustain the write bandwidth required to keep up with a fully active host interface. The ATB output goes over a narrower interface that is clocked by a dedicated ATB clock. Our probes will need to implement a similar structure that can capture the full bandwidth across all the interface channels, and it will need to serialize simultaneously occurring events in a multiplexed trace output stream. A single buffer with multiple write ports is proposed as it should allow best utilization of storage capacity.

Another consideration is how tightly to integrate the probing functionality with our ACE master and slave bridges. Implementing the probes as independent modules instantiated outside the bridges would potentially allow the probes to be shared across multiple bridge instances (as indicated in one customer spec). It will also be desirable to provide the option of mapping debug logic into a separate power domain from the mission mode logic, which would be difficult to do if it is not separated into a distinct parallel RTL module. However, in practice, it is likely that there are significant opportunities to optimize the probe implementation via a tight integration with the bridge where mission mode functionality may be used to support the probing functions. Furthermore, deployment of a SIB to consolidate mission mode infrastructure may be interesting in such circumstances, and probe sharing would come for free. Some investigation is warranted to drive a final decision about this.

Over time, additional probes may be developed to provide more specialized visibility (e.g., coherency tracing).

<Details to be refined>.

### NS Trace Master

The NS Trace Master takes in trace streams in ATB format and formats the data for transmission over the NoC to one or more of our NS Trace Slaves. The ATB streams may originate from our NS Trace Probe, or they may come from ARM or other 3rd party Coresight IP.

### NS Trace Slave

The NS Trace Slave receives data from one or more NS Trace Masters via a trace slave port on the NoC, and it multiplexes and reformats the data as necessary to transmit over an ATB interface to downstream CoreSight components.

It provides functionality similar to that of the standard CoreSight funnel component, which is described briefly in section 2.2.4 of the CoreSight Technology System Design Guide and in more detail in section 6.2 of the ARM CoreSight SoC-400 TRM.

### NS Trace Protocol

The trace bridge NoC ports of the Trace Masters and Trace Slave will communicate via a NetSpeed Trace Protocol (aka NSTP) that packetizes ATB transactions for transmission over the NoC. ATB interfaces are primarily unidirectional, streaming trace data in the forward direction, relying on simple ready/valid signaling for flow control, with no other form of explicit acknowledgement. ATB transactions are single cycle, tagged with an ATID that indicates the source, with no interdependencies between cycles, so trace masters simply create fixed length NSTP packets whose payload represents a single cycle of ATB transfer.

There are a couple of signals that communicate requests in the reverse direction: AFVALID, which indicates a flush request, and SYNCREQ, which indicates a synchronization request. These signals are essentially broadcast in nature, flowing to all upstream elements. These could be packetized and sent via the NoC datapath, but the overhead would be excessive for just two signals. Furthermore, these packets would need to be replicated to each trace master. Instead, a special high fanout network is created for each of these signals to connect them between the trace slave and all the masters that feed it.

As currently defined, NSTP does not implement any end to end flow control. It relies on the hop by hop credit based flow control of the NoC to provide back pressure from the trace slave into the Noc and ultimately back to the trace masters. The benefit of this approach, aside from simplicity, is that the storage in the NoC can be fully used for trace buffering, potentially reducing the size of capture buffers at trace sources. Traffic driven NoC construction in NocStudio plus QoS features of the NoC can be used to provide some level bandwidth allocation and prioritization amongst trace sources. However, if more precise control were required (ARM’s trace funnel component provides its own dynamically configurable prioritized arbitration scheme), some form of end to end flow control might need to be added.

One subtlety to consider is the following regarding prioritizing trace flushing over other trace traffic as described in the ARM ATB Protocol Spec on p. 4-31:

“When a flush occurs, indicated by AFVALID HIGH, the protocol expects that trace funnels give the highest priority to trace sources that have not yet asserted AFREADY.”

To address this our Trace Masters could dynamically boost the priority of their NSTP streams while they have a pending flush to serve.

#### NSTP Trace Data Packet Format (Forward Direction)

This is a NetSpeed Streaming Packet with payload containing an ATB transaction plus AFREADY bit as follows:

|  |  |
| --- | --- |
| **NSTP Trace Data Field** | **Width** |
| AFREADY | 1 |
| ATID[6:0] | 7 |
| ATBYTES[m:0] | m |
| ATDATA[n:0] | n |

**Table 1** NSTP Trace Data Packet Format

Note that there is no size field to allow for variations in ATB interface widths. A single size trace data packet is defined for a given NoC (i.e., *n* and *m* above are the same in all NSTP packets flowing in the NoC). It is expected that all trace masters and/or the trace slave will do ATB width conversion as necessary to match the trace data packet size transported across the NoC.

#### NSTP Flush/SyncReq Signal Propagation (Reverse Direction)

The NoC routing network that delivers NSTP Trace Data packets to the trace slave is inherently a tree structure, with all masters communicating to the single slave. Flush requests and SYNCREQs are returned to all masters as single bit broadcast packets (pulses) via a network built by adding special back propagation stages at each routing hop of the forward network. Each stage captures an incoming pulse and handles any clock crossings before transmitting the pulse to one or more next hops until all trace masters are reached.

The AFVALID/AFREADY handshake at the trace slave’s ATB interface guarantees that there will only be one outstanding AFVALID pulse propagating back through the NoC at any time. AFREADY, the master side response to AFVALID, is incorporated as an additional bit of payload in the forward direction packets. The trace slave waits to receive an NSTP packet with the AFREADY bit set from each of the connected masters before asserting AFREADY at its ATB interface (performing the same aggregation function as ARM’s CoreSight Funnnel component). Therefore, there is no concern about pulse combining as AFVALID (possibly) passes through multiple clock domains back to the trace masters.

There is no such handshake with SYNCREQ. SYNCREQs are passed upstream to trace sources, which respond by adding time stamps into the trace data stream. However, the ATB protocol provides leeway in handling SYNCREQs such that if multiple SYNCREQs arrive before a trace source has a chance to respond, it may merge the requests and return only a single timestamp in response. It is possible that for some topologies involving multiple clock domains and a rapid sequence of SYNCREQs, pulse merging may occur in the network. This is acceptable, while at the same time it is not expected to be a condition that is likely to occur in practice.

### ATB Interfaces

ATB protocol defines the reset/idle/clamp status of certain control signals to be 1’b1, in particular ATREADY and AFREADY. This is so that inactive parts of a CoreSight deployment just blindly ack requests so connected components are unblocked. To support this we will need to update our low power implementation to support clamping specific signals to 1’b1.

## Error Capture, Interrupts, Status and Statistics

We have already implemented this functionality, but it deserves a round of review and likely some expansion and enhancement.

Following is a list of planned enhancements:

* Create new LP related interrupts
  + Separate DECERR flag when this is caused by a fencing action.
  + Add flag for coherency connection that is stalled due to powered down PD that is not autowake capable.
* Make bridge idle status visible over regbus.

## Fencing and Draining

This feature comes at the request of a particular customer (deemed a useful suggestion): provide a mechanism to cleanly hold off some masters from the NoC to allow isolated debug of traffic between components of interest. A simple interface was requested such that a set of bridges could be controlled via a single register write as opposed to requiring a sequence of regbus transactions to each bridge. We support such a centralized interface as well as local per master bridge regbus control, making use of existing self-fencing logic that is part of low power operation in our master bridges.

A *debug\_fence\_enable* is added to eligible master bridges, and it causes similar behavior to the LP case where a bridge’s own pd\_active signal goes low (self-fencing). When this control is asserted, the master bridge stalls its host interface at a clean transaction boundary, while any accepted transactions are allowed to complete normally. The difference in behavior vs. pd\_active is that the bridge will never deny this request (i.e., there is no debug\_fence\_deny). Correspondingly, there is no explicit acknowledgment of acceptance of a debug fencing request (i.e., there is no debug\_fence\_ack\_n either). QACTIVE status or local regbus idle status flags may be polled to determine when a fully drained and idle state has been achieved.

As with self-fencing in low power operation, in a multi-stage fencing scenario, this control must be applied at the furthest upstream master interface. For example, in the case of an AXI4 master bridge that has a voltage domain boundary at its host interface, the debug fencing is implemented at the host interface of the VDC FIFO that sits in front of the AXI4 master bridge. However, ahb2axi is handled differently for debug fencing than for low power fencing. In this case, the AXI4 master bridge does the debug fencing as the ahb2axi converter has no protocol compliant way to stall its interface.

Care must be taken to avoid creating blockages in the NoC with this feature. We will only allow it to be enabled at host boundaries at the original injection point of transactions into the NoC. It should not be enabled at downstream hops in multi-hop scenarios (e.g., regbus master bridge in a tunnel configuration, reorder bridge, Gemini agents, etc.). Furthermore, this feature is not supported for ACE or ACE-lite+DVM masters because the coherency protocols do not allow for blocking of the read or write interfaces.

Powering down the PD containing a master bridge that is actively doing debug fencing is generally possible, with the exception of AHBLM, in which case there could be transactions held pending in ahb2axi that prevent draining (though this would result in a fence\_deny if the PD were autowake capable). In all cases, local register state, including the local debug\_fencing\_enable, will be lost if the bridge is powered down (assuming no deployment of retention registers), so upon power-up, the bridge will not be executing debug fencing unless driven to do so by a *debug\_group\_fencing\_enable* signal.

### Regbus Debug Fencing Control

A regbus control register in the ACE Master Bridge (and all derivatives) drives a local *debug\_fence\_enable* bit that activates this functionality. When the ACE master is not in a downstream position in a multi-stage fencing scenario, it implements self-fencing locally when this bit is set. In a multi-stage fencing scenario, this bit drives an output that is connected by NocStudio to the appropriate upstream component (e.g., VDC FIFO, ahb2axi, SIB, etc.) which executes the self-fencing.

### Centralized Debug Fencing Control

When specified by the config, a set of one or more *debug\_group\_fencing\_enable* signals are broadcast to groups of bridges. These signals originate in a register that lives in the regbus master. Debug fencing groups are defined via *add\_debug\_fencing\_group* commands in NocStudio, and each master bridge may be assigned to any number of these groups (a bridge may be part of multiple debug groups). Each master bridge receives the *debug\_group\_fencing\_enable* signal for each group in which it participates. These signals are locally synchronized before being combined with the bridge’s register driven *debug\_fencing\_enable* signal to activate debug fencing.

For multi-stage fencing scenarios, the *debug\_group\_fence\_enable* signals are connected to the upstream component that is executing the fencing, and it must synchronize and combine these and the *debug\_fence\_enable* coming from the downstream AXI master’s register.

### NocStudio Interface

Debug fencing is always available regardless of the state of *mesh\_prop low\_power\_enabled*, as long as regbus exists and the *debug\_fence\_enable* signal is available in the set of registers deployed in the ACE master bridge.

Debug fencing groups may be created and destroyed with the following commands.

* add\_debug\_fencing\_group <dbg\_group\_name> <list of 1 or more master bridges>
* del\_debug\_fencing\_group <dbg\_group\_name>

NocStudio should check the list of master bridges and make sure that it does not include any ACE or ACE-lite+DVM masters nor any downstream hops in a multi-hop traffic flow. Specifically, the re-order buffer master bridge and the regbus master are disallowed. Master bridges of Gemini agents are also not allowed.?

The following properties are added to fencing master components (ACE master bridge and derivatives, VDC FIFO, SIB, etc.):

* P\_DEBUG\_FENCE\_ENB: master enable for debug fencing functionality, uses to control generation of necessary self-fencing logic. Must be set at the most upstream fencing component.
* P\_NUM\_DBG\_GROUPS: specifies the number of *debug\_group\_fence\_enable* signals this master receives. May be 0, even if P\_DEBUG\_FENCE\_ENB is set to 1 (meaning only local regbus register bit is used).

The following property is added to the regbus master bridge:

* P\_NUM\_DBG\_GRP\_OUTPUTS: specifies the number of *debug\_group\_fence\_enable* outputs driven by the regbus master bridge. May be 0.

# MicroArchitectural Building Blocks

There is common functionality that is required in multiple places in the trace and debug architecture, so it makes sense to create modular building blocks that implement these functions.

## Multi-Write-Port FIFO

ACE/AXI probing likely require this, and it might be useful in the ATB funnel (below) as well.

## ATB Funnel

Both NS Trace Master and NS Trace Slave may be required to multiplex multiple incoming trace streams into a single output stream. This is the function performed by ARM’s ATB Funnel component. We should explore whether we could simply use the ARM component or if there would be benefit in developing our own rendition.

## Internal Event Bus Filter

Multiple aspects of the trace probe use a common event selection mechanism to select and combine events from the internal event bus. This function should be implemented module that can be re-used. Section 2.5 of the ARM HTM TRM describes the selection function, and section 2.7 describes the format of the selection control register contents. There are multiple selection control registers, one for each resource that monitors the event bus, and there would be one of these modules associated with each such register.

# MicroArchitectural Details

## Trace Probes

Implementation is heavily modeled from ARM’s HTM, so this section relies on the AMBA AHB Trace Macrocell (HTM) TRM as a reference. However, the HTM, at least the specification to which we currently have access, is relatively old, and it predates MIPI debug standards, which define a base protocol for formatting trace data (STP). Therefore, for guidance in defining the formatting of our trace output, we refer to the ARM CoreSight STM-500 System Trace Macrocell (STM) TRM and the MIPI System Trace Protocol (STP) Specification (v2.2). ARM’s STM generates MIPI STP v2 compliant trace output.

At a high level, the HTM implements a set of “resources” that generate events. These events are broadcast onto an “internal event bus.” Functions like trace start/stop, trigger generation, and some of the resources themselves select and combine events from this bus to drive their functionality. Section 2.5 of the ARM HTM TRM provides a more detailed overview of this implementation.

### ACE/AXI Channels Map to STP v2 Master IDs

ACE/AXI interfaces implement multiple split transaction channels that operate simultaneously whereas AHB implements a single channel that is shared for reads and writes with no real transaction splitting (each transaction must complete with response before the next one can proceed). Each channel can be generating trace data in parallel with the others, and each will need its own trace filter and will produce its own trace stream.

MIPI STP v2 defines hierarchical data streams in terms of masters and channels, with masters being the higher (outer) level identifier, and channels being the lower (inner) level identifier, i.e., each master may have multiple channels. In our application, the ACE/AXI channels map to unique STP master IDs. Initially, STP channel IDs will be unused (default channel ID == 0). Eventually we may use channel IDs to distinguish transactions originating from different sources, e.g., in our slave probe we may map source ID to channel ID.

The downside to this approach is that synchronization information such as timestamps and perhaps other control information might need to be replicated across all active master IDs. As long as that information only represents a small percentage of the overall trace bandwidth, this is an acceptable trade-off.

Each trace stream will need a write port into storage, and that write port may vary in width depending on trace filtering options (e.g., include vs. don’t include data) and compression. It may be possible to implement a shared FIFO with multiple variably sized write ports and a single read port for ATB output. This would maximize efficiency in sharing the local storage resources, but it would be quite complex. To limit complexity, our implementation dedicates a FIFO to each channel with round robin selection to transmit data out the ATB interface.

Streams from response channels need to be correlated with the forward command streams. For this purpose we include the transaction ID number used in our internal tracking structures in both command and response traces.

### Registers

The trace probes implement a 4KB block of control registers that comply with the requirements for a Coresight component, including 32-bit access and alignment (which differs from other regbus space). This 4KB block occupies the address offsets 0x2000-0x2fff within the 16KB regbus space of the bridge that contains the probes, but only when an ATB probe has been specified to be implemented within the bridge.

The Coresight Arch Spec section B2.2 describes expected behavior when accessing reserved and unimplemented addresses within the 4KB block, and the implication is that no errors should be generated, with reads returning 0’s and writes silently ignored. Section B2.5 regarding the Coresight Management Registers is more explicit, stating that “any reads from unimplemented or reserved registers in 0xF00 to 0xFFF must return zero, and writes must be ignored.” Therefore, if an ATB probe has been implemented with a bridge, no errors will be generated for accesses to unimplemented register addresses within the 0x2000-0x2fff Coresight block. If no probe is implemented in the bridge, DECERR should be generated within this address range.

The NocStudio generated HTML reference manual describes the probe registers in detail. Note that only implemented registers are described there (and in noc\_registers.csv). As noted above, these registers are all 32-bits in size, and they generally have 32-bit address alignment (i.e., some registers base address modulo 8 will equal 4 – they occupy the upper half of a 64-bit location). 64-bit wide regbus accesses (required to be 64-bit address aligned) will succeed, reading or writing the pair of registers that occupy the accessed 64-bit space.

All ATB probe registers are secured, only accessible via secure regbus transactions (PROT[1] == 0).

#### Control Register (HTMGLBCTRL 0x000) - Deprecated

We have customer input that inclusion of data capture in traces (bit 3 – DATAEN) should be gated by security status. As long as this DATAEN resets to disable data capture and this register can only be accessed by privileged SW, maybe there is nothing special to do. However, it may be that this bit should be qualified by SPIDEN, etc. or some other signal, perhaps custom to our implementation to provide this function.

#### Filter Control Register (CS\_TRACE\_CTRL 0xXXX) - TBD

This register determines which resources are used to filter which transactions are included in the trace output.

* Start/Stop Enable
* Include
  + Table Match
  + Table Miss
  + Address/Control Comparator
* Exclude
  + Table Match
  + Table Miss
  + Address/Control Comparator

### Trace Filtering

The trace probe provides programmable resources that may be used to select which transactions that arrive at the bridge interfaces are to be included in the trace output, aka trace filtering. Filtering is done primarily on the request interfaces – AR, AWW and AC, and responses are included in trace output when the request that generated them has been included. The CR response channel in ACE slave bridges requires special handling, described further below.

Filtering is implemented in 3 layers…

#### Channel Comparator Resources

Each of the request channel’s trace probe logic contains a set of comparators for address and control fields. They are organized in a specific way described below that allows sharing of configuration register controls, but they all have a common implementation, where a mask register is ANDed with the incoming transaction data to zero out bits that are don’t cares, and the resulting value is compared for equality against a corresponding value register.

hit = ((incoming\_data & csr\_mask) == csr\_value);

Note that the user must zero out bits in the csr\_value register that correspond to zero values in the csr\_mask register.

Following is a list of comparator resources…

* Address: 3 copies, AxADDR width, in AR, AWW and AC channels
  + incoming\_data: AxADDR
  + csr\_mask: cs\_addr\_match\_mask\_<0|1|2>[63:0]
  + csr\_value: cs\_addr\_match\_val\_<0|1|2>[63:0]
  + Register Names: CS\_ADDR\_MATCH\_HI\_MASK\_<0|1|2>, CS\_ADDR\_MATCH\_LO\_MASK\_<0|1|2>, CS\_ADDR\_MATCH\_VAL\_<0|1|2>, CS\_ADDR\_MATCH\_LO\_VAL\_<0|1|2>
* AC Control: 2 copies, width of incoming data (max 15), in AR, AWW and AC channels
  + incoming\_data: {Axagnid[7:0], AxSNOOP[3:0], AxPROT[2:0]}
    - Note: for AWW, use {AWUNIQUE, AWSNOOP[2:0]}
    - Note: Axagnid only applies to AC channel, AR and AW just PROT and SNOOP.
  + csr\_mask: cs\_acctrl\_match\_mask\_<0|1>[14:0]
  + csr\_value: cs\_acctrl\_match\_val\_<0|1>[14:0]
  + Register Names: CS\_ACCTRL\_MATCH\_MASK\_<0|1>, CS\_ACCTRL\_MATCH\_VAL\_<0|1>
* Control: 2 copies, width of incoming data (max 26), in AR and AWW
  + incoming\_data: {AxQOS[3:0], AxREGION[3:0], AxCACHE[3:0], AxLOCK, AxBURST[1:0], AxSIZE[2:0], AxLEN[7:0]}
  + csr\_mask: cs\_ctrl\_match\_mask\_<0|1>[25:0]
  + csr\_value: cs\_ctrl\_match\_val\_<0|1>[25:0]
  + Register Names: CS\_CTRL\_MATCH\_MASK\_<0|1>, CS\_CTRL\_MATCH\_VAL\_<0|1>
* AxID: 1 copy, width of AxID upto 32 lsbs, in AR and AW
  + incoming\_data: AxID[31:0] (or smaller)
  + csr\_mask: cs\_id\_match\_mask[31:0]
  + csr\_value: cs\_id\_match\_val[31:0]
  + Register Names: CS\_ID\_MATCH\_MASK, CS\_ID\_MATCH\_VAL
* Origid, Logid, Mstrid: 1 copy, width of these fields, in AR and AW
  + incoming\_data: {Axorigid[7:0], Axlogid[7:0], Axmstrid[7:0]}
    - Note: for master bridges, Axmstrid is unused.
  + csr\_mask: cs\_lom\_id\_match\_mask[23:0]
  + csr\_value: cs\_lom\_id\_match\_val[23:0]
  + Register Names: CS\_LOM\_ID\_MATCH\_MASK, CS\_LOM\_ID\_MATCH\_VAL

For all of these comparators, the logic may be generated so that bits of incoming data that don’t exist (because fields are narrower in that particular bridge instance) are completely left out of the logic, and the corresponding bits from the csr registers may also be left out. However, it is important to keep the fields lined up such that they are masked and compared against the same fixed position within the csr register. For example, if P\_TRACE\_LOG\_ID\_WIDTH is 4, then Axlogid[3:0] should be masked/compared with cs\_lom\_id\_match\_mask[19:16]/cs\_lom\_id\_match\_val[19:16].

#### Match Signals

A set of registers controls logic within each request channel that selects which comparator outputs to use to generate 3 different match signals. As opposed to the comparator controls, which are shared across all 3 request channels, each channel gets its own match signal control register.

The match signal control registers are: CS\_MATCH\_AR\_CFG, CS\_MATCH\_AW\_CFG and CS\_MATCH\_AC\_CFG.

The RTL signals are: cs\_match\_ar\_cfg[26:0], cs\_match\_aw\_cfg[26:0], and cs\_match\_ac\_cfg[26:0].

There are 3 sets of 9 control bits within these registers, 1 for each of the 3 match signals. The bits are:

[8] M0\_LOM\_ID – selects origid/logid/mstrid comparator output

[7] M0\_ID – selects AxID output

[6] M0\_CTRL1 – selects control comparator 1 output

[5] M0\_CTRL0 – selects control comparator 0 output

[4] M0\_ACCTRL1 – selects AC control comparator 1 output

[3] M0\_ACCTRL0 – selects AC control comparator 0 output

[2] M0\_ADDR2 – selects ADDR comparator 2 output

[1] M0\_ADDR2 – selects ADDR comparator 1 output

[0] M0\_ADDR2 – selects ADDR comparator 0 output

Bit’s 8:5 are UNUSED for AC channel (the exist in registers as read-only 0’s). Bits 17:9 repeat this set of bits for match signal 1, and bits 26:18 are for match signal 2.

The function for the match signal is:

match0 = (~M0\_LOM\_ID | hit\_lom) & (~M0\_ID | hit\_axid) & (~M0\_CTRL1 | hit\_ctrl1) & (~M0\_CTRL0 & hit\_ctrl0) & (~M0\_ACCTRL1 | hit\_acctrl1) & (~M0\_ACCTRL0 | hit\_acctrl0) & (~M0\_ADDR2 & hit\_addr2) & (~M0\_ADDR1 & hit\_addr1) & (~M0\_ADDR0 & hit\_addr0);

#### Trace Filter Control

The register CS\_TRACE\_FILT\_CTRL configures how the above match signals are used to include or exclude transactions. It also configures whether the address lookup table match signals are used to control inclusion. This register contains bit fields for each of the 3 request channels AR, AW and AC (each channel trace probe uses the appropriate subset of bit fields).

##### CS\_TRACE\_FILT\_CTRL Bit Fields

|  |  |  |  |
| --- | --- | --- | --- |
| Bit Range | Name | RTL | Description |
| 30 | AC\_EXC\_ONLY | cs\_filt\_ac\_exc\_only | 1: Include all AC transactions, only apply exclusion filters. 0: Apply both inclusion and exclusion AC filters. |
| 29 | AW\_EXC\_ONLY | cs\_filt\_aw\_exc\_only | 1: Include all AW transactions, only apply exclusion filters and AW\_ERR\_INC. 0: Apply both inclusion and exclusion AW filters. |
| 28 | AR\_EXC\_ONLY | cs\_filt\_ar\_exc\_only | 1: Include all AR transactions, only apply exclusion filters and AR\_ERR\_INC. 0: Apply both inclusion and exclusion AR filters. |
| 27 | AW\_ERR\_INC | cs\_filt\_aw\_err\_inc | 1: Include AW transactions that miss the address table or generate other errors in address decode, even if they are otherwise excluded. 0: Do not force inclusion of AW error transactions. |
| 26 | AW\_TBL\_EXC | cs\_filt\_aw\_tbl\_exc | 1: Exclude AW transactions do NOT hit an address table entry enabled in the CS\_TBL\_MATCH registers. This bit is only settable in master bridges. 0: Do not exclude any transactions based on AW table lookup results. |
| 25 | AR\_ERR\_INC | cs\_filt\_ar\_err\_inc | 1: Include AR transactions that miss the address table or generate other errors in address decode, even if they are otherwise excluded. 0: Do not force inclusion of AR error transactions. |
| 24 | AR\_TBL\_EXC | cs\_filt\_ar\_tbl\_exc | 1: Exclude AR transactions do NOT hit an address table entry enabled in the CS\_TBL\_MATCH registers. This bit is only settable in master bridges. 0: Do not exclude any transactions based on AR table lookup results. |
| 23 | UNSD\_23 |  |  |
| 22:20 | AC\_EXC | cs\_filt\_ac\_exc[2:0] | bit\_sel: When high, selects corresponding AC channel match signal (0-2) to exclude AC transactions. |
| 19 | UNSD\_19 |  |  |
| 18:16 | AC\_INC | cs\_filt\_ac\_inc[2:0] | bit\_sel: When high, selects corresponding AC channel match signal (0-2) to include AC transactions. |
| 15 | UNSD\_15 |  |  |
| 14:12 | AW\_EXC | cs\_filt\_aw\_exc[2:0] | bit\_sel: When high, selects corresponding AW channel match signal (0-2) to exclude AW transactions. |
| 11 | UNSD\_11 |  |  |
| 10:8 | AW\_INC | cs\_filt\_aw\_inc[2:0] | bit\_sel: When high, selects corresponding AW channel match signal (0-2) to include AW transactions. |
| 7 | UNSD\_7 |  |  |
| 6:4 | AR\_EXC | cs\_filt\_ar\_exc[2:0] | bit\_sel: When high, selects corresponding AR channel match signal (0-2) to exclude AR transactions. |
| 3 | UNSD\_3 |  |  |
| 2:0 | AR\_INC | cs\_filt\_ar\_inc[2:0] | bit\_sel: When high, selects corresponding AR channel match signal (0-2) to include AR transactions. |

Table 2 CS\_TRACE\_FILT\_CTRL Register Bit Fields

##### Trace Filter Control Function

The filter control function for the AR channel is as follows:

assign filter\_en = (tbl\_err & cs\_filt\_ar\_err\_inc) |  
 (~(|(match\_ar[2:0] & cs\_filt\_ar\_exc[2:0]) | (~tbl\_match & cs\_filt\_ar\_tbl\_exc)) &

(|(match\_ar[2:0] & cs\_filt\_ar\_inc[2:0]) | cs\_filt\_ar\_exc\_only));

When filter\_en is 1, the transaction is included in trace output (subject to start/stop filtering described in section 4.1.3.5 and to authentication control described in section 4.1.9.2), otherwise the transaction is not included in trace output. This function prioritizes exclusion over inclusion, except in the case of table error (local decode error) inclusion, which has highest precedence. Note that if *cs\_filt\_ar\_inc[2:0]* == 3’b000, then *cs\_filt\_ar\_exc\_only* must be set or only tbl\_err transactions (local decode errors) will be included.

The function for AW channel is the same as for AR. For AC, and for all channels in slave bridges, since there is no address table, the *cs\_filt\_\*\_err\_inc* signals and the *cs\_filt\_\*\_tbl\_exc* signals do not exist or will always be 0 (and could be locally tied off in the above filter function).

##### Address Table Matching

A set of registers, CS\_TBL\_MATCH\_<0:7>, hold a mask of bits that correspond to entries in the AR/AW address lookup table. The address table match signal goes high if a transaction hits an address table entry for which the corresponding bit is set in these registers. The registers are common for both AR and AW channels, but since address table lookups happen separately for AR and AW channels, there are AR and AW channel specific address table match signals.

The address table has 2 levels of hierarchy, where each entry can be designated as high or low priority. If there is a high priority match in the address table, the results of that that lookup are used, and any low priority match is ignored. A low priority match is only used if there was not a high priority match. Table match filtering adheres to this priority scheme by using the prioritized range match vector “range\_match\_mem\_combined” as follows…

assign tbl\_match = |(cs\_tbl\_match[P\_NUM\_OF\_ADDRS\_RANGES\_MEM-1:0] & range\_match\_mem\_combined[P\_NUM\_OF\_ADDRS\_RANGES\_MEM-1:0]);

#### Events

The trace probe provides a set of resources, some of which are shared with the filtering logic, for generating events. The events produced by the various resources are aggregated by configurable logic controlled by CS\_TRACE\_EVENT\_# registers to create aggregated event signals. These aggregated events are in turn used by several mechanisms to control probe behavior. The mechanisms using the aggregated events are:

* Start/Stop
* Trigger Packet Generation
* TRIGOUT[1:0] signaling
* Counter Resource (which is itself an event resource)
* Trace Disable

The types of resources that generate events that feed into the event aggregation logic are:

* Request channel events: at least 2 (provision for 4).
  + 3 match signals from each channel
    - Do we need to be able to invert the sense?
  + Tbl match signals
    - Including tbl miss
* Response channel eventscomparator: at least 1 (provision for 2)
* Channel samples: mask of pulses for each channel sample, really only useful for counter.
* Counter
  + Programmable start value
  + Configurable what is counted
    - clocks
    - valid ATB bytes queued (not necessarily flushed)
      * Stops counting down within ATB WIDTH of bytes
    - an event: can be different than reset event
  + Reset event uses all these resources except itself
  + Generates event pulse when it reaches 0
  + Configurable 1 time or repeat
* External signals
  + All synchronous to noc\_clk, outputs single cycle pulse
  + Configurable edge detect or level: does this matter, seem like it has to be edge based.
  + It would be nice to be able to select direction of change

Use the same filtering per channel, OR in other resources.

Each channel has additional filters to generate events: start event, stop event, trigger event, counter reset event… 4 events total.

Event combining:

* 4 events per channel: 12 bits
* 1 response event per channel: 6 bits
* 2 counter events: 2 bits
* 2 external inputs: 2 bits

Total: 22 bits

To keep filter enable simple, channel event 0 can be start event.

##### Request Channel Events

Each request channel (AR, AW, and AC) reuses for event generation the same transaction comparator resources that are used for filtering. In master bridges, this includes the address table filtering resources.

The functionality for generating events is the same as the trace filter control logic described in section 4.1.3.3, and in fact copies of the same RTL submodule may be used for event generation. The control signals for the event generation are driven by the CS\_TRACE\_<CHAN>\_EVENT\_<#> registers.

Architecturally, control registers are defined to provide for up to 4 configurable events per request channel, but only 2 events per request channel are implemented in the first production version of the probe.

##### Response Channel Events

Unlike request channels, response channels do no transaction filtering (with the special case exception for CR channel in slave bridges as described in section 4.1.4.6).

##### HWEVENT Input Signal Events

The trace probe implements 2 external event signals: HWEVENT[1:0]. These are synchronous to noc\_clk, but otherwise the probe does not constrain timing of transitions on these signals (i.e., they may change state at any time, with pulses of arbitrary duration). The trace probe implements edge detection in both directions on these signals, and it internally generates separate single cycle event pulses for each of positive and negative edge transitions for each signal. A total of 4 distinct single cycle events are created that are available to the event aggregation logic.

#### Start/Stop

The start/stop mechanism is an additional filtering control that globally applies a “stop” filter (effectively “trace none”) to all request channels. That state of the filter is driven by events. This is distinct from the global trace enable which fully disables all trace probe functionality when it is de-asserted. By contrast, when tracing is “stopped,” the logic is still fully active, including responding to SYNCREQ and AFVALID and maintaining CRTID tracking in the ACE slave bridge, but no new request channel transactions are included in the trace (all are filtered).

Start/stop is configured via the CS\_START\_STOP register. This register contains a bit, SS, that indicates the start/stop status and is used to qualify (AND) the *filter\_en* signal described above in section 4.1.3.3.2. There is latency between a start event originating from a channel event and the SS bit being updated such that a transaction causing the SS bit to be set (transitioning from stop to start) would not be included in the trace output. This is not desirable, so some logic is added to qualify the SS within each request channel so that a transaction that will set SS is included in the trace output.

When tracing is “stopped” (SS transitions from 1->0), a flush is generated so that all data captured up to the point of stopping gets out to the ATB interface. Otherwise, until some other flush event occurs, or tracing is started again, some captured transactions could be stuck in channel probe FIFOs indefinitely.

*Need to think about timestamping. Do we want the channels issuing flag packets to cover wraps? Do we want syncreqs happening while we are stopped? Do we need to throw out some timestamp update upon every start?*

* If flag packets are suppressed, we will need to issue a new full timestamp upon start.
  + If other stuff is filling the trace buffer, you should probably be regularly generating syncreqs anyway.
    - Could support a mode where internal syncreq only sends full timestamp, but not obvious there is a lot of value to this.
  + If not, it may be cheaper to just keep sending flag packets. At worst only incrementally more expensive.
* How would we do a flush when response packets are still pending?
  + Don’t do a flush. It is up to the customer to take care of flushing either with AFVALID or trace\_enb.
  + I think there is value in being able to drop trace\_enb based on an event, including a stop. Does stop need to be an event that feeds into trace\_enb event, or can it just be that trace\_enb event is configured to be the same. I am leaning toward the latter.

#### Trigger Packet

If enabled, when either of the trigger outputs asserts, a TRIG packet will be inserted into the trace stream, with timestamp if timestamping is enabled. The timestamp that is transmitted with the TRIG packet shall be captured in the same cycle as the assertion of the associated TRIGOUT.

The TRIG packet includes an 8-bit payload, which contains an encoded value that indicates which event caused the trigger. TBD how the encoding will handle cases where multiple simultaneous trigger events have occurred.

Only one TRIG packet may be outstanding at the trace probe bridge arbiter at a time. If a subsequent trigger event arrives while there is already a pending TRIG, it is ignored.

STP encoding:

* TRIG: 0xF 0x0 0x6 <2 nibbles encoded cause>
* TRIG\_TS: 0xF 0x0 0x7 <2 nibbles encoded cause> <TS nibbles>

#### XSYNC Packet

A register enables either of the HWEVENT inputs to cause an XSYNC packet to be inserted, with timestamp if timestamping is enabled. The timestamp that is transmitted with the XSYNC packet shall be captured in the same cycle as the event that causes the XSYNC to be generated.

Do both rising edge and falling edge detection.

The XSYNC packet includes an 8-bit payload, the upper 4-bits of which are always 0, and the lower 4 bits are a bit mask indicating which event(s) caused the XSYNC.

Only one XSYNC packet may be outstanding at the trace probe bridge arbiter at a time. If a subsequent XSYNC event arrives while there is already a pending XSYNC, it is ignored. *Should we instead merge the event masks, perhaps with an extra bit indicating that multiple events occurred?*

STP encoding:

* XSYNC: 0xF 0x0 0xA <0x0> <mask>
* XSYNC\_TS: 0xF 0x0 0xB <0x0> <mask> <TS nibbles>

#### Old Text…

We implement a model similar to ARM’s HTM where we provide multiple types of event resources whose outputs can be programmatically combined to generate triggers and to start/stop tracing. A subset of these resources (address comparators) can be used to filter trace content. The event resources have 2 classes, primary and derived. Primary events are generated from primary inputs, derived events are generated from Boolean combinations of primary or derived events. Unlike the HTM, we have multiple channels operating simultaneously, so we have a set of primary event resources associated with each individual channel (e.g., address and control comparators), while the derived event resources are shared across all channels.

We should implement parameters that control quantities of each type of resource that is included to the extent that it is practical. However, we need to be conscious of the verification burden this presents. ARM offers 3 fixed configurations of HTM – HTM64, HTM32 and HTM32L. We should probably limit ourselves in a similar fashion.

#### Address Lookup Table Match

For AR and AW channels, the address table look-up is a resource that can generate events. A set of registers (CS\_ADTBL\_MASK) specify a mask which indicates which entries will generate an event if they are matched and result in forwarding the transaction. This mask is shared between the AR and AW channels, just as the address table configuration registers are. Another register (CS\_ADTBL\_ERR) controls whether an event is generated when the result of a look up is an error.

#### Address and Control Comparators

The HTM separates address from control comparators, though the address comparators can be configured to include the results of one of the control comparators in determining a match. Our implementation unifies address and control comparison.

The HTM programmers model supports up to 16 of single address comparators, and these can be paired together to create up to 8 range comparators, but ARM’s HTM configurations implement either 2 or 4 single address comparators.

* HTM has a type register to configure additional aspects of the transaction to match, including a pointer to one of the control comparators.
* These have a normal mode and a sticky mode (stay high until next transfer arrives).
* Range comparators: registers are shared with range comparators. Single address comparators may be simultaneously active. Type registers of upper and lower bound must be programmed to be the same.

Our implementation implements address and control comparison with value/mask registers that are shared across channels (AR, AW, AC).

#### Control Comparators

As described above, our implementation unifies address and control comparison, but for reference a brief description of these resources in the HTM follows here.

These have the same normal vs. sticky modes. The programmers model supports 8, but all HTM permutations only have 1.

A 4-bit control register field selects which control signals are multiplexed into the comparator. Our implementation will necessarily differ as some fields don’t apply for the ACE interfaces, and there may be additional fields that we do need to monitor. Some work needs to go into defining the mapping options appropriate for our implementation, which will likely vary across the multiple interfaces of our ACE bridges.

AXI Address Control Signals

* AxLOCK[1:0]
* AxCACHE[3:0]
* AxPROT[2:0]
* AxQOS[3:0]
* AxREGION[3:0]
* AxUSER[n:0] – only supported with address, not data

ACE Address Control Signals

* AxSNOOP[3:0]
* AxDOMAIN[1:0]
* AxBAR[1:0]

AXI/ACE Response Signals

* xRESP[4:0]
  + AXI: RRESP[1:0] and BRESP[1:0]
  + ACE: RRESP[3:0], BRESP[1:0], CRESP[4:0]
* xUSER[n:0] – only 1st cycle

#### Data Comparators

We should provide at least simple pattern matching on DATA to generate a trigger.

#### External Inputs

Typically connected to ECT, but could be connected to other things. HTM supports 2 external inputs and it drives 2 external outputs. We will do the same.

#### Derived Resource – Event Selection

Each derived resource selects 2 events and combines them via a selectable Boolean function.

#### Counters – HTM TRM sec 2.7.1

The HTM programmers model supports up to 4 16-bit countdown counters. Each counter outputs an event when it reaches 0. All permutations of HTM support 1 counter.

We should provide some mechanism that would help identify hang events.

#### Sequencer – HTM TRM sec 2.7.3

A programmable 3-state sequencer is supported by the programmers model. This is only available in the HTM64 configuration.

#### Trace Filtering – HTM TRM sec 2.8

See fig 2-12 on p 2-22.

#### Trigger Generation – HTM TRM sec 2.9

The HTM can insert trigger packets into the trace stream.

#### Bus Endianness and Bus Width

The HTM implements 2 single bit input signals that select between endianness modes and bus width respectively. These are provided as signals to support external bus selection (driven by the HTMBUSSELECT register), when different buses might need different settings. In our case, unless we implement a probe sharing mechanism such as HTMBUSSELECT, these properties are likely to be fixed at design time. Bus width is determined by the basic bridge configuration, but endianness is probably requires a new NocStudio parameter to allow it to be specified by the customer to configure probe behavior. *Are there any circumstances where endianness of a bus would be dynamic at runtime?*

### Trace Framing

ARM’s HTM frames trace data into numerous byte oriented packet types that are transported over the ATB interface. However, its format is not MIPI STP v2 compliant. Our trace probes will produce similar classes of trace data but will need to frame the data somewhat differently to produce a MIPI STP v2 stream. Following is a list of packets types that may be generated with relevant HTM TRM section numbers and a mapping to the MIPI STP packet types that our probes will use to contain each class of information with relevant STP spec section numbers.

* Address (4.3.12, 4.3.4) – STP Data (D\*: 6.4.12 – 6.4.15)
* Auxiliary (4.3.13) – STP Data (D\*: 6.4.12 – 6.4.15)
* Data (4.3.11) – STP Data (D\*: 6.4.12 – 6.4.15)
* Response (N/A) – STP Data (D\*: 6.4.12 – 6.4.15)
* Control
  + A-Sync (4.3.2, 4.10) – STP ASYNC (6.4.2) + VERSION (6.4.3)
  + Trigger (4.3.3) – STP TRIG (6.4.8)
  + Ignore (4.3.5) – STP NULL (6.4.1)
  + Trace Off (4.3.6) – STP USER (6.4.6)
  + Data Suppressed (4.3.7) – STP MERR (6.4.17)
  + FIFO Overflow (4.3.8) – STP MERR (6.4.17)
  + Reset (4.3.9) – STP USER (6.4.6)
  + CycleCount (4.3.10) – STP TIME (6.4.7)
  + Master ID (N/A) – STP M8 (6.4.11)

#### MIPI STP v2 VERSION

The latest version (as of this version of this document) of the MIPI STP specification is 2.2, dated 8/21/2015 and adopted by the MIPI board on 2/11/2016, and it is listed as “new” at the MIPI web site, whereas version 2.1, dated 3/8/2013 and adopted 6/12/2013, is listed as “recommended.” We should comply with version 2.1 at a minimum, and we will comply with version 2.2 unless we find a compelling reason not to.

The supported version is communicated in the trace stream in the VERSION packet, which always follows the ASYNC packet, which is issued in response to Sync Reqs over ATB and possibly also periodically in response to an internal timer maintained within the trace probe. Section 6.4.3 starting on p. 17 of the STP Spec describes the VERSION packet.

The VERSION packet communicates 2 pieces of information describing implementation choices made by the trace probe.

1. 3-bit Timestamp Format Field: specifies which of several format options is used for timestamps in this STP stream. We implement STPv2NAT (same as ARM STM), which is encoded as 0x3.
2. Is\_LE\_Flag: specifies how nibbles within STP packets are ordered. Versions prior to 2.2 support only big-endian ordering, and this flag is missing from VERSION packets prior to 2.2. We will use big-ending ordering, so this flag will be 0 in our VERSION packet.

The full nibble sequence of our VERSION packet output (encoding v 2.2 compliance) is:

Nibble0 = 0xF, Nibble1 = 0x0, Nibble2 = 0x0, Nibble3 = 0xB, Nibble4 = 0x0, Nibble5 = 0x1

#### Data Ordering Within MIPI STP v2

We are using STP in big-endian mode, so all data fields, including NetSpeed defined trace contents transported in STP data (D\*) packets, will be emitted in big-endian order. For example, a 32-bit address included in trace output inside an STP D32 packet would be ordered as follows:

D32\_payload\_nibble[0] = address[31:28]  
D32\_payload\_nibble[1] = address[27:24]  
D32\_payload\_nibble[2] = address[23:20]  
…

This example is representative only. Details of how addresses and other fields are packed in trace output appear in sections that follow.

#### Mapping MIPI STP v2 onto ATB

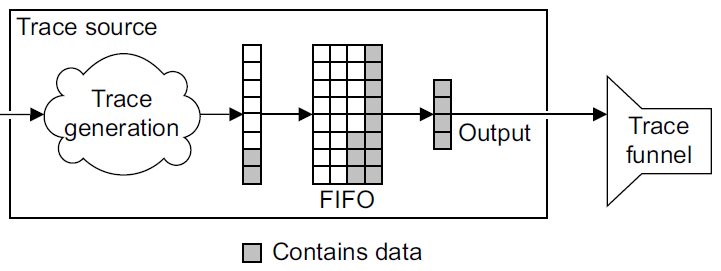


Figure 2 Packing MIPI STP v2 into ATB

We need to transmit a MIPI STP v2 stream over ATB. ATB is a byte oriented streaming protocol, and the contents of its data field are not defined in any way by the protocol. MIPI STP v2 is nibble oriented, with numerous packet types and associated formats specified. Some choice needs to be made about how to pack the STP stream onto the ATB data bus. Fortunately, the STP spec provides a recommended scheme for packing into a byte stream, which is to put first nibble in byte[3:0] and second nibble in byte[7:4]. See Annex E on p. 51 of the MIPI System Trace Protocol (STP) Specification (v2.2) for full detail.

ATB packs data LSB first, so byte 0 (containing STP nibbles 0 and 1) occupies ATDATA[7:0], byte 1 is in ATDATA[15:8], and so on.

ATDATA[7:0] = Byte0[7:0] = {STP\_nibble1[3:0], STP\_nibble0[3:0]}  
ATDATA[15:8] = Byte1[7:0] = {STP\_nibble3[3:0], STP\_nibble2[3:0]}  
ATDATA[23:16] = Byte2[7:0] = {STP\_nibble5[3:0], STP\_nibble4[3:0]}  
…

See section 3.2.1 on pp 3-26/3-27 of the AMBA 4 ATB Protocol Specification (ATB v1.0 and ATB v1.1) for more detail. Also see section 2.12 of the ARM CoreSight STM-500 System Trace Macrocell (STM) TRM for a detailed example of how the STM uses the same packing order as we do.

#### Channel Switching

In STP v2 streams, a master ID and channel ID are set, and all frames that follow are assumed to be associated with the prior settings. The STP VERSION packet, which is part of the synchronization sequence, resets the master ID and channel ID both to 0x0. When a trace packet is queued for transmission from an ACE/AXI channel that differs from the previous one, we must insert an STP v2 M8 frame which updates the master ID bits [7:0] to indicate that subsequent frames are associated with this new source. Because the M8 frame represents overhead in the trace, it may be desirable to implement an arbitration scheme amongst the trace data FIFOs for the different ACE/AXI channels that transmits a minimum quantity of data from a single channel before switching (when there remains pending data to send from the current channel). It would also be desirable to prioritize serving RESP channels ahead of address channels because the RESPs all correlate to previously captured address channel transactions, and if we lose a single RESP, we will lose the ability to reliably correlate RESP transactions with the corresponding address channel transactions. In other words, it is more harmful to lose a RESP transaction than to lose an address channel transaction.

|  |  |
| --- | --- |
| **ACE/AXI Channel** | **Master ID[7:0]** |
| AR | 0x0 |
| AW/W (w/data) | 0x1 |
| R Response (w/data) | 0x2 |
| B Response | 0x3 |
| AC | 0x4 |
| CR/CD (w/data) | 0x5 |

**Table 3** ACE/AXI Channel to STP v2 Master ID Mapping

#### Address, Auxiliary, Response & Data Packet Framing

Like ARM’s HTM, we pack the main trace stream into several distinct types of packets: Address, Auxiliary, and Data. Because responses in ACE/AXI are returned on a different channel and generally separated in time from the requests, our probes add the Response packet type which is not produced by the HTM (instead responses values are embedded in the data). The packet types share a common encoding in the first nibble of the packet which indicates the packet type, format mode and some additional information described in the tables and text that follow:

|  |  |  |  |
| --- | --- | --- | --- |
| **Header Nibble Bits** | | | |
| 3 | 2 | 1 | 0 |
| Type Specific/Reserved | Continued | Packet Type | Format Mode |

**Table 4** Header Nibble for Address, Auxiliary, Response & Data Packets

The continued field (bit 2), when set, indicates that the packet is continued in the next STP data frame. The usage of the type specific/reserved field (bit 3) varies with packet type. The format mode (bit 0) indicates which of the two format modes was active when the specific transaction was capture (format modes described further in section 4.1.4.5.5).Packet type (bit 1) is encoded as:

* Address: 0x0
* Auxiliary: 0x1
* Data: 0x1
* Response: 0x0

These trace packet types are all wrapped in STP v2 Data (D\*) frames. In STP, there are 3 sets of Data frame types: data-only, data w/timestamp, and data w/marker and timestamp. All encode the length of the data payload that follows in a power of 2 number of bits ranging from 4-64. Packets of each type will be wrapped in the smallest size data frame that can contain the complete contents. Where it makes sense, accounting for the overhead of STP framing, the trace packets may be split across more than one STP data frame by setting the Continue bit in the header nibble. For example, if an address trace packet contains a 32-bit address + 4-bit header nibble, the total is 36-bits in length. The only size STP data frame that can contain the entire contents is D64, which would require 28-bits of padding to complete – lots of wasted space. This can be more efficiently framed in a D32 with Continue bit set followed by a D4. The second frame of a continued packet has no header – its payload holds only the remaining contents of the trace packet. Therefore we only support data splitting into a maximum of 2 frames. It is expected that for most practical cases, there will be no efficiency to be gained by further splitting when accounting for the overhead of the STP data frames.

In most cases, the trace packet payload will be compressed by suppressing leading 0’s.

##### Address Packets

The format of Address packets is described in the following table:

|  |  |  |
| --- | --- | --- |
| Nibble #’s | Field | Contents |
| 0 | Packet Type Header | {Seq Addr, Cont, 0x0} |
| 1-*C*\* | Control | Bit packed field containing AxID, AxLEN, AxSIZE and AxBURST as required. Highest order bits first. |
| *C*+1… | Address | AxADDR, highest order bits first. May be compressed. |

**Table 5** Address Packet Format

The Type Specific bit of the header nibble for address packets is a Sequential Address flag. The flag and its implications are described below in section 4.1.4.5.1.2.

For non-sequential transactions (Sequential Address flag of header nibble is 0), both control and address fields are included. The address field itself is compressed (also on a per ACE/AXI channel basis) by transmitting only the least significant nibbles that have changed from the previous address.

One and only one address packet is issued for each traced AR/AW/AC command regardless of burst length, with intermediate per beat addresses inferred by the trace decoder. Unlike ARM’s HTM implementation, address packets are always issued for each new transaction that is initiated at the host interface, even when the address is the next sequential address and could otherwise be inferred.

*Note: as presently defined and implemented in RTL, the content of trace packets is limited to a maximum of 124 bits (the payload of 2 MIPI STP D64 packets), including a maximum width timestamp. It is possible to create bridge configurations in NocStudio where this limit would be exceeded if full widths of all potentially captured interface signals were used. NocStudio enforces this limit, preventing IP generation until the violation is resolved by adjusting trace format and/or timestamp width via interface properties (see sections 4.1.4.5.5 and 4.1.5 for more detail).*

###### AR/AW Channel Address Packet Control Field

The control field of address packets contains configurable sets of ACE/AXI address channel signals that are considered interesting to include with every traced address. These signals are bit packed together into the control field using the minimum number of bits necessary to capture all the information. Since these signals vary in width for different bridges, the control field will vary in size for different bridge instances down to a nibble granularity. For a given bridge instance, the sizes of of the captured fields are fixed – there is no compression or other mechanism that would dynamically change their sizes.

In addition to protocol standard ACE/AXI signals, several NetSpeed proprietary ID fields may be included within the control field. The composition of the control field for a given bridge/probe combination is determined at NoC configuration time and is visible in the *XXX-TBD-Fixme* register.

The format of the control field is described in the table below from lsb at the top to msb at the bottom. Some signals may have zero width in the control field if they hold static values at all times, and it is possible that no control field is required at all – e.g., AXI4-lite.

|  |  |  |
| --- | --- | --- |
| Width (# of bits) | Field | Contents |
| 0-50 | AxID | # of bits based on actual used AxID bits, indicated by P\_[MST|SLV]\_AID\_[R|W]\_WIDTH parameters in RTL, 0 if static/unused. |
| 0-8 | AxLEN | # of bits based on max burst length, indicated by P\_ALEN\_WIDTH in RTL, 0 if static/unused. |
| 0-3 | AxSIZE | # of bits based on max data size, inidicated by P\_TRACE\_SIZE\_WIDTH in RTL, 0 if static/unused. |
| 0-2 | AxBURST | # of bits 0 if static/unused, 1 = AxBURST[1] if no FIXED, 2 otherwise (indicated by P\_TRACE\_BURSTTYPE\_WIDTH in RTL). |
| 0-3 | AxPROT | # of bits indicated by P\_TRACE\_PROT\_WIDTH, 0 if static/unused. |
| 0-2 | AxLOCK | # of bits indicated by P\_TRACE\_LOCK\_WIDTH, 0 if static/unused. *Current RTL hardwired with 1 bit.* |
| 0-4 | AxCACHE | # of bits indicated by P\_TRACE\_CACHE\_WIDTH, 0 if static/unused. |
| 0-4 | AxREGION | # of bits indicated by P\_TRACE\_REGION\_WIDTH, 0 if static/unused. |
| 0-4 | AxQOS | # of bits indicated by P\_TRACE\_QOS\_WIDTH, 0 if static/unused. |
| 0-8 | Axorigid | # of bits indicated by P\_ORG\_ID\_WIDTH, 0 if static/unused. |
| 0-8 | Axlogid | # of bits indicated by P\_LOG\_ID\_WIDTH, 0 if static/unused. |
| 0-32 | AxUSER | # of bits indicated by P\_TRACE\_USER\_WIDTH, 0 if static/unused. |
| 0-4 | (AWUNIQUE,) AxSNOOP | Only present for ACE bridges, # of bits indicated by P\_TRACE\_SNOOP\_WIDTH, 0 if static/unused. For AW channel, AWUNIQUE is combined as msb with AWSNOOP. |
| 0-2 | AxDOMAIN | Only present for ACE bridges, # of bits indicated by P\_TRACE\_DOMAIN\_WIDTH, 0 if static/unused. |
| 0-2 | AxBAR | Only present for ACE bridges, # of bits indicated by P\_TRACE\_BAR\_WIDTH, 0 if static/unused. |
| 0-8 | Axmstrid | Used internally to track destination of responses, only present on host interface for ACE slave bridge connected to CCC, # of bits indicated by P\_MST\_ID\_WIDTH, 0 if static/unused. For single hop transactions, this should always have the same value as Axorigid, but it will differ for multi-hop transactions. |

**Table 6** AW/AR Channel Address Packet Control Field Format

Transaction ID (AxID): AxID is an 8-bit bus in ACE/AXI4, but our bridges support widths up to 50 bits in size. Actual deployments often will use fewer bits, possibly none at all if a single transaction ID is used.

Length (AxLEN): AxLEN is an 8-bit bus in ACE/AXI4, so it may be up to 8 bits in size in the trace. Actual implementations generally constrain max burst length such that fewer bits are required (e.g., AXI3 only uses AxLEN[3:0]). If all transactions use a fixed burst length, 0 bits are required in the trace output.

Size (AxSIZE): AxSIZE is a 3-bit bus in ACE/AXI4, but actual number of bits required depends on data bus width of the interface. If all transactions are the same size, which may often be the case, 0 bits are required in trace output.

Burst Type (AxBURST): AxBURST is 2-bits wide in ACE/AXI4. There are 3 burst types: FIXED, INCR and WRAP. FIXED is rarely seen in practice, but if it is supported for a particular deployment, both bits of AxBURST will be included in the trace. Otherwise, only 1 bit, AxBURST[1] will be included to distinguish between INCR and WRAP burst types, unless burst type is static, in which case 0 bits will be sampled.

In all cases, the subfields are sized to support the maximum value that can appear in a given deployment, even if only a small subset of values are possible, i.e., we do not do more aggressive encoding to further reduce the number of bits required.

###### AC Channel Address Packet Control Field

|  |  |  |
| --- | --- | --- |
| Width (# of bits) | Field | Contents |
| 0-3 | ACPROT | # of bits indicated by P\_TRACE\_ACPROT\_WIDTH, 0 if static/unused. *Is there a reason to capture anything but ACPROT[1]?* |
| 0-4 | ACSNOOP | Only present for ACE bridges, # of bits indicated by P\_TRACE\_SNOOP\_WIDTH, 0 if static/unused. |
| 0-8 | ACcrtid | Only present for ACE master bridge connected to CCC or ACE slave bridge connected to CCC and DVM, # of bits indicated by P\_CRT\_ID\_WIDTH, 0 if static/unused. |
| 0-8 | ACslvid  or ACagnid | ACslvid is only present for ACE master bridge connected to CCC, # of bits indicated by P\_AGN\_ID\_WIDTH, 0 if static/unused.  ACagnid is only present for ACE slave bridge connected to CCC and DVM, # of bits indicated by P\_MST\_ID\_WIDTH, 0 if static/unused. |

Table 7 AC Channel Address Packet Control Field Format

###### Sequential Address Compression

The trace encoder performs sequential address compression by computing an expected sequential address from the address, length, size and burst type fields of most recently traced transaction within the channel and comparing it against the address of the next traced transaction. If the newly arriving transaction has identical settings for all control fields except AxID, and the address matches the expected sequential address, sequential address compression will be used and the Sequential Address flag will be set*.*

If the Sequential Address flag is set, it indicates that transmission of the address field and most or all of the control field has been suppressed, and the trace decoder calculates the address from the prior traced transaction instead, and control fields are taken from the prior transaction as well. If the AxID of the currently traced transaction is the same as the prior traced transaction, the entire control field is suppressed, otherwise a reduced control field containing only AxID is included. The trace decoder distinguishes presence or absence of AxID based on the size of the MIPI STP data packet that contains the header nibble – if it is D4, then no control field is present and the prior AxID is used, otherwise remaining nibbles after the header are decoded as the AxID. The sequential calculation is maintained per ACE/AXI channel, so sequential address suppression may still apply even if there have been intervening transactions associated with other master IDs. Note also that the calculation applies between pairs of traced transactions (those passing any active trace filters), so it is possible to have intervening non-traced/non-sequential transactions while still generating a sequential address packet in the trace output.

The expected sequential address is calculated as follows:

If (AxBURST == FIXED) {  
 Expected\_Seq\_Addr = AxADDR + decoded(AxSIZE);  
}  
else {  
 Expected\_Seq\_Addr = AxADDR + (decoded(AxLEN) \* decoded(AxSIZE));  
}

###### Address LSB Adjustment

In many applications, the lsbs of the address may not be very interesting (e.g., all transactions are cache line reads/writes), and it would be desirable to leave them out of the captured data to save bandwidth. A proposed enhancement is to implement a property in NocStudio that would allow the user to specify the lsb of the sampled address field as something other than 0. This would be communicated to RTL via a parameter that would control address sampling and also be reported in a Coresight CSR.

##### Auxiliary Packets

The HTM generates auxiliary packets to capture miscellaneous information related to a transaction, including some signals from the host interface that are not captured in other packets. Only one auxiliary packet is generated per burst transaction. The HTM implements a control register that selects among 16 pre-configured 12-bit sets of information.

Given the multichannel requirements, our probes use auxiliary packets for different purposes depending on the channel that is being traced.

###### Auxiliary Packets for Address Channels (AR, AW and AC)

For address channels, auxiliary packets may be used to indicate an alternate control field format. However, at present this mode is not fully defined.

###### Auxiliary Packets for Response Channels (R and B) Capture of RACK and WACK

Read and write response channel tracing of ACE master bridges use auxiliary packets to indicate toggling of RACK and WACK signals respectively. The format is a single D4 STP packet that contains only the header nibble with the following content…

|  |  |  |  |
| --- | --- | --- | --- |
| **Header Nibble Bits** | | | |
| 3 | 2 | 1 | 0 |
| Type Specific/Reserved | Continued | Packet Type | |
| 0 | 0 | 2’b01 | |

**Table 8** Response Channel (R & B) Auxiliary Packet for RACK and WACK

##### Data Packets

The format of Data packets is described in the following table:

|  |  |  |
| --- | --- | --- |
| Nibble #’s | Field | Contents |
| 0 | Packet Type Header | {Includes WSTRB, Cont, 0x2} |
| 1+\* | WSTRB | WSTRB – if capture enabled, variable #’ of nibbles based on WDATA width. |
| 2+\* | DATA | DATA, highest order bits first. |

**Table 9** Data Packet Format

The Type Specific bit of the header nibble for data packets indicates whether or not the WSTRB field is present. It is always 0 for read data. A *TBD* config register controls whether or not WSTRB is captured and included in a write data trace.

Data compression is done by omitting leading 0’s (most significant nibbles that are all 0 until first non-zero nibble).

*TBD:* per beat user bits should be included here (at least as an option). Per transaction user bits probably belong in Aux packet.

##### Response Packets

The format of Response packets is described in the following table:

|  |  |  |
| --- | --- | --- |
| Width (# of bits) | Field | Contents |
| 4 | Packet Type Header | {Extended, Cont, 0x3} |
| 0-50 | xID | # of bits based on actual used AxID bits, indicated by P\_[MST|SLV]\_AID\_[R|W]\_WIDTH parameters in RTL, 0 if static/unused (always 0 for CRRESP) |
| 2-5 | xRESP | BRESP – 2 bits in all cases CRRESP – 5 bits for ACE bridges, 0 bits for AXI RRESP – 4 bits for ACE bridges, 2 bits for AXI |
| 0-32 | xUSER\* | Per transaction user bits, width always 0 for CR. |
| 0-8 | xcrtid\* | Only present for ACE master bridge connected to CCC or ACE slave bridge connected to CCC and DVM, # of bits indicated by P\_CRT\_ID\_WIDTH, 0 if static/unused. |
| 0-8 | xslvid\*  or xmstrid\* | xslvid is only present for ACE master bridge connected to CCC, # of bits indicated by P\_AGN\_ID\_WIDTH, 0 if static/unused.  xmstrid is only present for ACE slave bridge connected to CCC and DVM, # of bits indicated by P\_MST\_ID\_WIDTH, 0 if static/unused. |

**Table 10** Response Packet Format

The Type Specific bit of the header nibble indicates whether whether the “Extended” fields (those marked by \* above) are included in this trace packet. The Cont bit of the header nibble is used to indicate that the packet extends to a second STP D\* packet.

Note only the last beat of a multi-beat RRESP is captured (i.e., RLAST == 1). BRESP and CRRESP are always single beat.

As with address packets, the exact sizes of the fields within the response packet are determined at NoC construction time, and we will implement a *TBD* read-only config register that reports the these sizes.

A response packet may be followed by an auxiliary packet and/or one or more data packets (e.g., RDATA or CRDATA) to complete capture of a transaction response.

##### Packet Content Selection

This section describes support of user control of the content of trace packets provided via interface properties in NocStudio that drive parameters to RTL the control the trace format for each channel.

* 2 packet capture modes exist: which mode is operating for each channel is controlled by a configuration register bit from the CS\_TRACE\_CFG\_STS.FMT field (*cfg\_trace\_fmt\_sel[5:0]* in RTL – channel encode from 5:0 is CR, AC, B, R, AW, AR).
  + The trace packet header indicates the state of this setting for each captured transaction in the format mode bit (0) (see section 4.1.4.5).
* The format of each capture mode is statically defined via NoC configuration in NocStudio. NocStudio provides the *trace\_req\_capture\_mask* interface property for AR, AW and AC interfaces, and the *trace\_resp\_capture\_mask* interface property for R, B and CR interfaces. These directly control the corresponding RTL FMT0\_ENB parameters, are bit masks that indicate which fields are to be included in the captured trace output. NocStudio constrains which bits may be set within the mask based on the exact type of bridge (e.g., the user may not set the AXID enable for AXI4LM bridges as that field is irrelevant for that bridge type).
  + As currently implemented, the FMT1\_ENB RTL parameters are derived by applying a fixed reduced mask to the capture\_mask properties – they always contain a subset (perhaps the exact set) of the bits set in the FMT0\_ENB parameters. *However, the intent is to separate the* trace\_<req|resp>\_capture\_mask *parameters into a pair of 0|1 versions that will each explicitly drive the corresponding FMT0|1\_ENB parameters.*
  + Trace field widths are indicated to RTL via P\_TRACE\_<FIELD>\_WIDTH parameters. Some of these are localparams that pass through a design parameter setting or a static value where the width is always the same, others are parameters that are configured via NocStudio in RTL generation. Corresponding bridge and/or interface properties with NocStudio may be provided to allow user control over the capture widths in these cases.
    - *Note: it may be desirable to provide additional control which bits of a given field are captured, e.g., bit mask or range (msb:lsb). This would require additional RTL parameters. As of yet, this has not been done. Likely candidates include Address, AXID and USER fields.*
  + NocStudio also programs P\_TRACE\_<CHAN>\_FMT<0|1>\_WIDTH parameters that indicate the total width of the trace data to be captured for a given format mode.
* Format settings are reported in read-only Coresight registers (CS\_TR\_FIELDS\_\*) so that a debugger can learn directly from the DUT how to decode trace output. These registers indicate the width of each possible field that might be included in the trace output, with copy of these registers for each of FMT0 and FMT1 settings. Where the FMT<0|1>\_ENB bit for a given field is 0, a 0 width is reported in these registers for the corresponding field.

Note: the HW datapath for each channel trace probe limits the maximum width of the trace capture vector (either FMT0 or FMT1) plus differential timestamp at maximum width to be <= 124 bits for request channels, and <= 60 bits for response channels. NocStudio checks at *gen\_*ip and prevents RTL generation if this constraint is violated in and enabled trace probes. Users can address any violations by making adjustments to the *trace\_<req|resp>\_capture\_mask* and or *timestamp\_width* properties.

###### P\_TRACE\_<REQ\_CHAN>\_FMT<0|1>\_ENB Parameters

The request channels AR, AW and AC share a common format for the P\_TRACE\_<REQ\_CHAN>\_FMT<0|1>\_ENB[18:0] parameters, enumerated below. Each bit corresponds to a field that may be included in the trace capture, with a value of 1 indicated the field is to be included in the trace, a value of 0 indicating the field is left out. These parameters cover a superset of fields that are available across all 3 channel types. Since AC channel has some fields that are only available to it, and likewise there are many AR/AW fields that are not available to AC, there are some bits within a given channel type vector that will never be set for that channel (and should be completely ignored in RTL).

[18] Address  
[17] Acslvid AC only  
[16] Accrtid AC only  
[15] Axmstrid AR/AW only  
[14] AxBAR AR/AW only (ACE only)  
[13] AxDOMAIN AR/AW only (ACE only)  
[12] AxSNOOP (ACE only)  
[11] AxUSER AR/AW only  
[10] AxLogid AR/AW only  
[9] Axorigid AR/AW only  
[8] AxQOS AR/AW only  
[7] AxREGION AR/AW only  
[6] AxCACHE AR/AW only  
[5] AxLOCK AR/AW only  
[4] AxPROT  
[3] AxBURST AR/AW only  
[2] AxSIZE AR/AW only  
[1] AxLEN AR/AW only  
[0] AxID AR/AW only

(bits are encoded in the same order as the tables in the sections above describing address packet formats.)

As mentioned above, a configuration register bit selects between which of the 2 capture formats to use. In RTL code, two versions of the captured data are built within generate blocks that statically interpret the FMT<0|1>\_ENB parameters, and the setting of the config register bit dynamically controls a 2-to-1 mux that selects between the 2 capture vectors.

###### P\_TRACE\_<RESP\_CHAN>\_FMT<0|1>\_ENB Parameters

The response channels R, B and CR share a common format for the P\_TRACE\_<RESP\_CHAN>\_FMT<0|1>\_ENB[4:0] parameters, enumerated below. Like request channels, the response channel parameters contain a superset of fields available across all response channels, and not every field is necessarily available within a given channel.

[4] xslvid/xmstrid  
[3] xcrtid CR only  
[2] xUSER R/B only  
[1] xRESP  
[0] xID R/B only

###### Content Notes Regarding origid, logid, mstrid, slvid, crtid, agnid, etc.

In ns\_acemstrbrdg\_core:

mstrid: this is a constant sent to NoC, always P\_MASTER\_ID.

origid: in middle hop scenarios, this is extracted from mstrid in preceding slave and passed into host interface of this master. Only present for master bridges used in this situation.

logid: this is a value that may be set by user logic to communicate which of multiple originators sourced the transaction. Presence configured by user in NocStudio.

crtid: coherent transaction ID, assigned by CCC (and possibly DVM?). Likely 0’d out for transactions not involving CCC (and possibly DVM).

slvid: ID of source of a given transaction (AC, R, B, returned over CR). P\_AGN\_ID\_WIDTH.

AR/AW: origid, logid

* mstrid: constant as described above.
* origid: comes from host pins as input, but only for some bridges.
* logid: comes from host pins as input, but only for some bridges.
  + In ns\_acmb\_<AR|AW\_W>\_ch.v, P\_<AR|AW>\_LOG\_ORG\_ID\_ENB[1:0] determines if origid and/or logid are used or not.

R: crtid, slvid

* Come from NoC via reorder buffer: in ns\_acmb\_R\_ch.v
  + r\_ch\_crtid\_int = rrob\_stage\_rresp\_info
  + r\_ch\_slvid\_int = rrob\_stage\_rresp\_info
* Only goes to host pins acmb\_Rcrtid and acmb\_Rslvid if P\_CCCM\_ACE is set
* Something about going to RACK queue…
  + assign noc\_fasttap\_r\_ch\_slvid = r\_ch\_slvid\_int ;
  + assign noc\_fasttap\_r\_ch\_crtid = r\_ch\_crtid\_int ;

B:

* Come from NoC via reorder buffer: in ns\_acmb\_B\_ch.v
  + crtid\_int = wrob\_bresp\_info
  + slvid\_int = wrob\_bresp\_info
* Only goes to host pins acmb\_Bcrtid and acmb\_Bslvid if P\_CCCM\_ACE is set
* I think these get looped back for WACK

AC: crtid and slvid

* Come from NoC: in ns\_acmb\_AC\_ch.v
  + ac\_ch\_crtid\_int = noc\_ac\_ch\_info
  + ac\_ch\_slvid\_int = noc\_ac\_ch\_info
* P\_CCCM\_ACE set (in ns\_acmb\_AC\_ch.v):
  + Host pins acmb\_ACcrtid and acmb\_Acslvid driven
  + CR\_CD loop back values are 0’d out.
* P\_CCCM\_ACE clear
  + Host pins acmb\_ACcrtid and acmb\_Acslvid 0’d out
  + CR\_CD loops back values

*Note: in RTL as of 2/10/17, these are only going to trace probe when P\_CCCM\_ACE is set. That should be changed to rely exclusively on FMT\*\_ENB.*

CR\_CD: crtid and slvid

* P\_CCCM\_ACE determines if these come from host pins (when set) or are looped back through response FIFO from AC channel (when cleared).

In ns\_aceslvbrdg\_core.v:

mstrid: for AR/AW, this comes from the NoC, is the ID of the previous hop initiator. Depending on the mode, it as appended as either msb or lsb of transaction AID, though there is a P\_UNIQ\_AID\_MODE where unique AIDs are locally recycled from a free list. So, depending on the configuration and the exact width of the host AID bus, it may be redundant to separately trace mstrid. For R/B, this comes from host pins.

origid: in middle hop scenarios, this is extracted from mstrid in preceding slave and passed into host interface of this master. Only present for bridges used in this situation.

logid: this is a value that may be passed by user logic master bridge to communicate which of multiple originators sourced the transaction. Comes into slave via NoC, presence configured by user in NocStudio.

crtid: output pins for CR, RWACK. Input pins for R, B, AC. Not present for AR, AW, WACK.

agnid: output pin for CR, input pin for AC. P\_MST\_ID\_WIDTH. For CR, this comes from NoC where it has been set to P\_MASTER\_ID at the master side source. For AC, this is the ID of the target of the snoop request, and it doesn’t go into the NoC, it is consumed locally to look up the route.

slvid: for B/R/AC, this is a constant P\_SLAVE\_ID, but only inserted in B/R when P\_SYS\_ACE is set.

So I think that mstrid/agnid are only relevant in slaves, and they are the same thing (just named differently for AC/CR vs other channels). Slvid is only relevant in master bridges. All should have the same system set width I believe.

#### Special Handling of AC/CR channels in ACE Slave Bridges (attached to CCC/DVM)

The AC/CR channels of ACE slave bridges require some special handling to be able to track and correlate requests and responses so that some filtering is possible. When these are driven by CCC or DVM agents, multiple copies of a snoop request may be issued, each copy going to a different target. The snoop requests are uniquely identified by NetSpeed specific fields *crtid*, which is a an identifier for the AR or AW transaction that generated 1 or more snoop requests, and the *agnid*, which indicates the target of each particular snoop (AC transaction).

To support filtering of the AC channel, we need some mechanism for tracking which outstanding AC requests have been traced so we can filter the CR requests accordingly. Fortunately we can take advantage of some properties of the AC requests to implement an area efficient mechanism.

##### Tracking AC Traced Status

We take advantage of the following aspects of the way that CCC (and presumably DVM) issue AC requests to implement a lightweight tracking mechanism:

* All CR responses associated with AC requests that have used the same *crtid* value must have returned before that *crtid* value is used for a new transaction. In other words, all outstanding AC requests using the same *crtid* value belong to the same originating transaction.
  + Furthermore, the CCC does not interleave AC requests with different *crtid* values. If multiple snoops to different agents are required to process a transaction that arrived over AR/AW, these are issued sequentially by the CCC to the AC channel interface. It turns out that even if CCC did interleave across *crtid* values, the tracking mechanism would still work, though address suppression would not.
* ACADDR, ACPROT and ACSNOOP have the same value for all outstanding AC requests using the same *crtid* value.
  + This means that filtering based on these fields yields the same result for all outstanding requests associated with the same *crtid.*
* ACagnid is unique to each AC request using the same *crtid* value. However, it is returned with the CR response.
  + This means that filtering based on ACagnid if the same filter is applied to returning CR responses.

The tracking mechanism is very simple: a vector of flags capturing trace filtering results of common the common fields is written by the AC channel trace probe, one bit per *crtid* value. When new AC requests arrive, trace filtering is done on the fields that are common within a given *crtid* value on each and every incoming transaction, and the result is written to the tracking vector in the appropriate bit position. When multiple AC requests arrive for the same *crtid* value, this bit gets written multiple times – however, the answer is the same for each of the multiple requests (because of above properties).

The tracking vector is read by the CR channel probe, which uses the *crtid* value of the CR response to look up the correct flag. This flag indicates whether or not to trace the response.

*Agnid* filtering is done separately in both AC and CR channels, and the result of this filtering is not captured in the tracking vector. In both AC and CR, the decision from filtering all the other fields, which is written by AC channel into the per *crtid* tracking mechanism and read by CR channel, is ANDed with the results of the *agnid* filter to create the filter enable for each transaction. Note this is the only place in response channels where filtering is applied on incoming transaction information.

The tracking vector is only written by the AC channel probe and read by the CR channel probe. It only needs to be reset when tracing is disabled.

##### Filtering on Common AC Fields

As mentioned above, filtering based on the following common fields is applied to every incoming AC transaction with the result recorded in the tracking vector:

* ACADDR
* ACPROT
* ACSNOOP

##### Filtering on AGNID

As mentioned above, filtering on *agnid* is done separately from the common field filtering in AC channel, its result combined with the common field filtering to determine whether or not to trace the transaction, but the combined result is not recorded in the tracking vector. In the CR channel probe, *agnid* filtering is again performed and combined with the bit read from the tracking vector to determine whether or not to trace the response.

##### Common Field Suppression in AC Channel Trace

To the extent that multiple snoops per *crtid* is common, substantial trace bandwidth reduction can be achieved with a simple field suppression mechanism. Since most fields don’t change among subsequent transactions for a given *crtid*, we keep track of the *crtid* value from the most recently traced transaction, and if the current transaction’s *crtid* value matches, and it also passes all filters so that it is traced, we suppress capture of the fields that don’t change (ACADDR, ACPROT & ACSNOOP).

A simple means of doing this would be to have NocStudio force the AC channel trace FMT1 parameter to disable the common fields, and the trace probe would select between capturing format 0 or format 1 based on the result of the above comparison (ignoring the CSR bit that normally controls this choice). A new CSR bit should be added that controls this feature (so that it can be disabled if need be), in which case the original CSR control should operate.

### Timestamp

Timestamps may be added to trace packets captured in any of the channels. A timer module generates a running time value, and based on configuration settings, each channel uses this time value to attach a timestamp to captured packets.

#### Timer Module

The timer module produces a timer value either from an internal free running counter or from an externally provided value delivered via the *dbg\_TSVALUE* input pins. This choice is determined at NoC construction and communicated to RTL via the P\_TRACE\_TS\_EXT\_TIMER\_ENB parameter. The *dbg\_TSVALUE* interface, if enabled, is a CoreSight compliant TSVALUE interface that operates synchronously to noc\_clk. Otherwise, a local counter incrementing by 1 every noc\_clk provides the timer value. In either case, the width of the timer value is specified by the P\_TRACE\_TS\_WIDTH parameter, a modulo 4 value up to a maximum of 64 bits.

Two timer value modes are implemented, a fine grained (default) mode, which delivers the full timer value directly to the channel capture logic, and a coarse grained mode, which right shifts the timer value by P\_TRACE\_TS\_SHIFT, also a modulo 4 value. The operating mode is controlled by a configuration register bit.

#### Per Channel Timestamping

MIPI STP v2 supports multiple formats for timestamp output. Our probes use the STPv2NAT format (same format as ARM’s STM), which is natural binary compressed as the logical difference from previous timestamp. See chapter 7 of the STP spec for more detail.

Timestamping is dynamically enabled by configuration bits in the CS\_TRACE\_TS\_CFG register on a per channel basis. Each channel probe records the last timestamp that it issued as a timestamp baseline, and it encodes the timestamp appended to the end of each captured packet as the logical difference between the current timer value and the baseline value, suppressing leading (MSB) 0 nibbles. If there is no difference, the timestamp is suppressed altogether (i.e., normal data opcode is used instead of timestamp opcode).

Timestamps are framed into the trace data by using a D\*MTS opcode in place of the equivalently sized D\* opcode that would be used for MIPI STP v2 framing. Where multiple D\* opcodes are required to hold the capture data, only the last D\* opcode is converted to a D\*MTS, so the timestamp value is always placed at the end of the captured data.

For example, consider a newly captured transaction that occupies 48-bits of payload (including header). This would be packed into a D32 followed by a D16 packet. If timestamping is enabled, and the logical difference is non-zero, the D16 is changed to a D16MTS opcode, and an encoded timestamp value is included after the 4 nibbles of data payload for that second packet.

Timestamp value encoding is described in detail in section 7.2 of the MIPI STP v2 spec. Our probes use the STPv2NAT format in big-endian mode. Briefly, the timestamp value begins with a nibble that specifies the number of timestamp difference nibbles that follow in big-endian order – the first nibble is most significant, followed by next most significant, down to least significant (bits 3:0 of value). The count value in the first nibble is slightly encoded such that 0x0-0xC represent the exact count, 0xD == 14, 0xE == 16, and 0xF is an illegal value.

Consider the following example of timestamp encoding:

P\_TRACE\_TS\_WIDTH = 48  
Timestamp baseline = 0x0000\_2745\_3EA7  
Current time value = 0x0000\_2746\_1E01  
Timestamp value = 0x5 0x6 0x1 0xE 0x0 0x1

In this case, the first nibble of the time stamp value contains the count (5) of nibbles that differed, followed by those differing nibbles in big endian order.

More background information…

* Chapter C6 of the ARM CoreSight Architecture Specification v2.0 describes the timestamp interface, including the fact that fewer than 64-bits may be used in a given system.
  + ARM’s STM implements the full 64-bit interface.
* Per section 7.2 of the STP v2 Spec, the first timestamp issued after a VERSION packet must be full width (and these will be issued in response to sync requests over ATB).

#### FREQ Packet

ARM’s STM issues an STP v2 FREQ packet after the ASYNC/VERSION sequence if timestamping is enabled. The value that is emitted comes from the STMTSFREQR register as defined in section 2.3.12 on p 2-32 of the ARM System Trace Macrocell (STM) Programmers’ Model Architecture Specification. We implement this register as CS\_TSFREQR to support the same behavior.

When timestamping is enabled for any channel, all ASYNC/VERSION sequences are immediately followed by a FREQ\_TS packet (see MIPI STP v2 spec section 6.4.4) with contents of CS\_TSFREQR and with the full width of the current timer value as the timestamp. This meets the requirement that the first timestamp transmitted after any ASYNC/VERSION sequence must be the full uncompressed value at width indicated by P\_TRACE\_TS\_WIDTH (reduced by P\_TRACE\_TS\_SHIFT if operating in coarse grained mode).

The last transmitted timestamp values maintained within each channel probe are updated with this value such that their subsequent logical different timestamps are all relative to it. See the following section for further discussion about properly synchronizing distribution of this timestamp baseline.

#### Reduced Width Differential Timestamp in Channels

Physical restrictions on the width of trace samples generated within the channel probe datapaths constrain the maximum size of the differential timestamp that can be included with any sample, and it is necessary to implement an enhancement that allows the trace stream to support a larger timestamp.

The enhancement is to support 2 timestamp widths, a smaller per channel maximum differential timestamp width and the full timestamp width (existing parameter P\_TRACE\_TS\_WIDTH). Channel probe timestamping logic behaves largely as before, but uses new parameters to constrain the width of differential timestamps captured with trace samples. Technically, the physical datapath limit may be different for each channel based on which fields are configured to be sampled for that channel. In practice, all request channels can always support 28 bits of differential timestamp, so there is little need to make these individually configurable, and a single parameter is provided to cover all 3 request channels: P\_TRACE\_REQ\_TS\_WIDTH. Response channels can have more variability in datapath physical constraints, with max width possibly being as low as 12 bits (due to STP encoding overhead when sample width is > 36 bits). However, to keep things simple, we will share a single parameter for all response channels: P\_TRACE\_RESP\_TS\_WIDTH. The central timestamp logic continues to use the full P\_TRACE\_TS\_WIDTH to size the counter and/or the external TSVALUE interface, and sync requests include the full width timestamp with FREQ\_TS.

The maximum value for P\_TRACE\_REQ\_TS\_WIDTH and P\_TRACE\_RESP\_TS\_WIDTH is 32 (this is the maximum differential timestamp width that needs to be supported by the channel probe logic). NocStudio is responsible for determining the actual values assigned to P\_TRACE\_REQ\_TS\_WIDTH and P\_TRACE\_RESP\_TS\_WIDTH based on the maximum possible sample sizes across all request channels and response channels respectively. Of course these will never bet set to be larger than P\_TRACE\_TS\_WIDTH. NocStudio must prevent the user from specifying channel mask properties that lead to a maximum sample size that when combined with maximum channel timestamp width exceeds the channel probe datapath limits. To satisfy this, NocStudio limits request channel sample size to a maximum of 92 bits (which with header fits into D64 + D32 STP opcodes, leaving room for 28 bits differential timestamp + 4 bit length field), and it limits response channel sample size to a maximum of 36 bits (which with header fits into D32 + D8 opcodes, leaving room for 20 bits of differential timestamp + 4 bit length field).

Trace channel probe hardware should be implemented to support P\_TRACE\_REQ\_TS\_WIDTH and P\_TRACE\_RESP\_TS\_WIDTH down to 4 bits, which while not practical for production applications, will make it easier to test this feature in simulation.

##### Flag Packet to Signal Differential Timestamp Wrap

When there is sufficient activity within a channel trace stream, it is possible for the trace decoder to reliably recognize timestamp wraps and compensate. To do this, the decoder needs to know about every transition of the msb of the channel probe timestamp. If there are long idle periods such that multiple msb transitions occur (one or more complete wrap arounds) without any intervening timestamped samples or SYNCREQs, the compensation will fail. To address this, a mechanism is added to the channel probe logic that keeps track of whether any activity has occurred since the last change to the timestamp msb. If no activity has occurred by the next timestamp msb transition, a FLAG packet (opcode 0xF 0xE) is injected into the channel stream to indicate there has been a full wrap of the msb. Note that the FLAG packet counts as activity seen since the latest msb value, so another FLAG packet is not needed until the msb has toggled back and forth without any other intervening activity.

Following is a representative trace stream with 16-bit channel timestamp that demonstrates the operation of this mechanism…

TRACE SAMPLE + TS = 0x1234 (msb = 0) – set activity reg  
(TS increments to 0x8000, msb 0->1) – activity was logged -> No FLAG, clear activity reg  
TRACE SAMPLE + TX = 0xff35 (msb = 1) – set activity reg  
(TS increments to 0x0000, msb 1->0) – activity was logged -> No FLAG, clear activity reg  
(TS increments to 0x8000, msb 0->1) – NO activity was logged -> ***Issue FLAG***, set activity reg  
(TS increments to 0x0000, msb 1->0) – activity was logged -> No FLAG, clear activity reg  
(TS increments to 0x8000, msb 0->1) – NO activity was logged -> ***Issue FLAG***, set activity reg  
(TS increments to 0x0000, msb 1->0) – activity was logged -> No FLAG, clear activity reg  
TRACE SAMPLE + TS = 0x79a2 (msb = 0) – set activity reg  
(TS increments to 0x8000, msb 0->1) – activity was logged -> No FLAG, clear activity reg  
(TS increments to 0x0000, msb 1->0) – NO activity was logged -> ***Issue FLAG***, set activity reg  
(TS increments to 0x8000, msb 0->1) – activity was logged -> No FLAG, clear activity reg  
TRACE SAMPLE + TX = 0xaf38 (msb = 1) – set activity reg  
(TS increments to 0x0000, msb 1->0) – activity was logged -> No FLAG, clear activity reg  
TRACE SAMPLE + TS = 0x49d2 (msb = 0) – set activity reg  
(TS increments to 0x8000, msb 0->1) – activity was logged -> No FLAG, clear activity reg  
SYNCREQ w/TS = 0xc04b (msb = 1) – set activity reg  
(TS increments to 0x8000, msb 0->1) – activity was logged -> No FLAG, clear activity reg  
…

There may be circumstances where the user would prefer not to have FLAG packets issued, even if it leads to incorrect interpretation of timestamp information by trace decoding software. To support this, 2 configuration register bits are provided that allow the user to control issuing FLAG packets: *cfg\_trace\_ts\_req\_flag\_enb* for request channels and *cfg\_trace\_ts\_resp\_flag\_enb* for response channels. One scenario where a user might clear these enables would be if a trace is expected to be captured over an extended period of time with filters configured such that samples are sparsely captured. In this case, FLAG packets could consume a large portion of valuable trace buffer resources, and it would be preferable to live with imperfect timestamp interpretation. Another scenario, perhaps derived from the first one, would occur when P\_TRACE\_REQ\_TS\_WIDTH is substantially larger than P\_TRACE\_RESP\_TS\_WIDTH. If FLAG packets were disabled for response channels but still enabled from request channels (which would need to issue them much less frequently), the trace decoding software could still correct response channel timestamp interpretation using information it has received over the related request channel.

### Sync Requests and Sync Packet Sequences

STP v2 alignment synchronization requires sending an ASYNC packet followed by a VERSION packet. We implement the same basic policy as ARM’s STM as described in section 2.4.2 of the STM TRM, where we issue a synchronization sequence when tracing is initially enabled and also in response to ATB sync requests (signaled via the ATB SYNCREQM input pin and/or internal trace byte counter). As with STM, if timestamping is enabled, we also issue a FREQ\_TS packet after the VERSION packet.

Some coordination is required when responding to sync requests. Time stamp and address compression is reset following emission of the ASYNC/VERSION/FREQ\_TS sequence, and the timestamp baselines maintained within each channel probe must be updated in a precise manner. This requires that sync requests must internally generate a flush to all channels and only issue the ASYNC/VERSION/FREQ\_TS sequence after acknowledgement that all channels have been drained. Furthermore, new trace data from all channels must be held pending until after the ASYNC/VERSION/FREQ sequence has been inserted. A detailed sequence follows:

* SYNCREQM received
* Channel flush issued to all active channel probes
  + In this cycle, the current timer value is captured into the timestamp baseline registers within each channel, and the same value is also captured for use in the FREQ\_TS packet that is to follow.
  + All pending packets within the channel buffers must be drained prior to issuing ASYNC/VERSION/FREQ\_TS.
  + Any newly arriving packets have timestamp generated relative to the updated baseline, but these packets must not be transmitted prior to the ASYNC/VERSION/FREQ\_TS sequence.

#### Internally Generate Sync Requests

The probes can internally generate sync requests at period intervals driven by a count of clock cycles (noc\_clk) or ATB valid bytes transmitted. The CS\_TRACE\_SYNC\_CFG register controls this behavior with an enable bit, and counter reset value, and a mode select that chooses whether to count clock cycles or transmitted ATB bytes. When enabled, a decrementing counter is reset with the reset value from the configuration register, and when it reaches zero (or >= ATB width # of bytes if counting ATB bytes) an internal sync request is generated. This is combined with the captured SYNCREQM signal to be handled the same way as an externally signaled sync request would be.

### Error Conditions Signaled by STP MERR Packets

Error conditions are signaled in the trace data via STP MERR packets. Per section 6.4.17 of the STP v2 Spec, these are associated with the “current\_master” (as set via M8 packets), which we use to distinguish our channels. In the context of STP, MERR packets reset the “current\_channel,” which for our implementation is always 0, so this has no material. Therefore, these errors are associated with the channel indicated by the most recent M8 or ASYNC/VERSION sequence.

The MERR packet has an 8-bit payload. In our implementation, the 2 msbs of the payload indicate an *error\_type*, while the lower 6 bits are used to convey error\_type specific information.

|  |  |  |
| --- | --- | --- |
| Payload [7:6] | Error Type | Payload [5:0] Content |
| 0 | Channel FIFO Overflow | Count of number of dropped transactions (saturates at 63). |
| 1-3 | Reserved | Reserved |

**Table 11** MERR Packet Encodings

### Trace Sampling Point

Trace data is sampled in the noc\_clk domain at the internal bridge interfaces after any channel synchronization and any clock crossing structures.

### Tracing Enable, Authentication Control and Invasive Mode

#### CS\_TRACE\_CFG\_STS Register

The CS\_TRACE\_CFG\_STS register provides a global tracing enable in the EN bit. When this bit is set to 0, all trace probe logic (except configuration registers) is disabled, and interfaces are driven to idle states. Specifically…

* AFREADY is driven to 1’b1, immediately acknowledging any AFVALID cycles.
  + Note: in our current low power implementation, we have no way to indicate that this signal should be clamped to 1’b1, so powering down the bridge will force AFREADY low.
* SYNCREQs are ignored and will not generate any ASYNC/VERSION sequences on the ATB interface.
* ATVALID is driven to 1’b0.
* Clocks are gated for all trace probe logic (except configuration registers, though they be separately gated based on regbus activity).

The CS\_TRACE\_CFG\_STS register also provides per channel pair enable bits that are combined with the EN bit to provide channel level enable control.

* AR: enables AR/R channel tracing.
* AW: enables AW/B channel tracing.
* AC: enables AC/CR channel tracing.

#### Authentication Control

The 4 input pins *dbg\_DBGEN, dbg\_NIDEN, dbg\_SPIDEN* and *dbg\_SPNIDEN* implement the Coresight authentication interface, which controls whether secure transactions may be traced and whether invasive debug (in our case, backpressure) is allowed. These signals are combined according to the recommendations in section B2.5.5 and C5.4 of the Coresight Architecture Spec and results reported in the CS\_AUTHSTATUS register as dictated there. The Coresight Arch Spec defines asynchronous and synchronous versions of this interface, then goes on to say that the asynchronous version is deprecated. These appear to be synchronous signals in ARM’s STM component. It is unclear what clock reference is assumed for these signais in ARM’s HTM component. Our probes treat these as synchronous ATCLK.

There are 4 levels of debug that are enabled by these signals, encoded in 2 bit fields of the CS\_AUTHSTATUS register. Section B2.5.5 and Section C5.4 (specifically Table C5-4) of the ARM Coresight architecture spec document the following equations that decode the authentication interface signals in to the debug levels that are reported in the AUTHSTATUS register. We implement these equations verbatim as follows:

* SID: secure invasive debug – (*dbg\_SPIDEN* & *dbg\_DBGEN)*
* SNID: secure non-invasive debug – ((*dbg\_SPNIDEN* | (*dbg\_SPIDEN* & *dbg\_DBGEN*)) & (*dbg\_NIDEN* | *dbg\_DBGEN*))
* NSID: non-secure invasive debug – *dbg\_DBGEN*
* NSNID: non-secure non-invasive debug – (*dbg\_NIDEN* | *dbg\_DBGEN*)

When none of the 4 levels above are enabled, tracing is completely disabled, the effect being the same as setting CS\_TRACE\_CFG\_STS.EN = 0. At a minimum, NSNID must be active for any debug to be allowed.

Note that the ARM Coresight arch spec is self-contradictory between Table C5-4 and Table C5-5, specifically on line 9 of the latter. We are using the equations of C5-4 and control our probes behavior according.

##### Invasive Tracing

Invasive tracing, where each channel probe may backpressure the source interface when its trace buffers are full, is controlled by a combination of the <CHAN>\_V bits of the CS\_TRACE\_CFG\_STS register and the debug levels above. The register provides a separate enable bit for each channel, so invasive debug is enabled on a channel by channel basis.

Backpressuring is considered invasive since it may change the observable behavior of the system. Since backpressuring can perturb secure processes, even if the transaction that stalls is non-secure (since secure transactions behind it would also be stalled), our probes cannot support the “non-secure invasive debug” mode. Therefore, for invasive tracing to be enabled for a channel, the corresponding bit in CS\_TRACE\_CFG\_STS must be set to 1, and either SID must be enabled via the authentication interface.

inv\_en\_<chan> = CS\_TRACE\_CFG\_STS.<CHAN>\_V & SID;

##### Secure Tracing

Secure tracing is the capture of transactions that have AxPROT[1] == 0. When secure tracing is disallowed by the authentication settings, the probe will not include such transactions in the trace output, i.e., it will filter them. Secure transactions will be traced if either SID or SNID is active, otherwise they will be filtered…

sec\_filter\_en = AxPROT[1] | SID | SNID;

Transactions are traced if sec\_filter\_en evaluates to 1. Note this filter takes precedence over all other filter decisions, i.e., if this signal is 0, the transaction will not be traced regardless of the state of all other filters.

##### AUTHSTATUS Register

We implement the AUTHSTATUS register as specified in section B2.5.5 of the ARM Coresight arch spec. Because we do not support non-secure invasive debug, we set the NSID field of this register to 2’b00, indicating we do not support this debug level.

|  |  |  |
| --- | --- | --- |
| Bit Range | Name | Description |
| 7:6 | SNID | 2'b11: Secure non-invasive debug enabled: ((SPNIDEN | (SPIDEN & DBGEN)) & (NIDEN | DBGEN)) == 1. Secure transactions may be traced. 2'b10: Secure non-invasive debug disabled: ((SPNIDEN | (SPIDEN & DBGEN)) & (NIDEN | DBGEN)) != 1. Secure transactions are not traced. |
| 5:4 | SID | 2'b11: Secure invasive debug enabled: (SPIDEN & DBGEN) == 1. Invasive debug is allowed. Secure transactions may be traced. 2'b10: Secure invasive debug disabled: (SPIDEN & DBGEN) != 1. Invasive debug is not allowed. Secure transactions may be tracied if SNID is active. |
| 3:2 | NSNID | 2'b11: Non-secure non-invasive debug enabled: (NIDEN | DBGEN) == 1. Must have this status for any tracing to be done. 2'b10: Non-secure non-invasive debug disabled: (NIDEN | DBGEN) != 1. No tracing at all is possible when NSNID is not active. |
| 1:0 | NSID | 2'b00: Non-secure invasive debug is not supported. Invasive debug is only allowed when SID is active - when (SPIDEN & DBGEN) == 1. |

#### Graceful Transition from Enabled to Disabled State

The trace probes implement a simple state machine to track the “busy” status of the tracing logic. This status is reported in the CS\_TRACE\_CFG\_STS.BSY bit. Whenever tracing is enabled, where CS\_TRACE\_CFG\_STS.EN is 1’b1 and any of the debug levels are enabled by the authentication interface, CS\_TRACE\_CFG\_STS.BSY returns 1’b1. When changing from enabled to disabled state (CS\_TRACE\_CFG\_STS.EN 1’b1->1’b0, the authentication interface changes such that none of the debug levels are enabled, or sleep\_req\_n is asserted), the probes remain in the busy state until they have gracefully completed all outstanding tracing activities. The following must happen before CS\_TRACE\_CFG\_STS.BSY is deasserted…

* Trace capture on all request interfaces (AR, AW and AC) is halted.
* Trace capture on all response interfaces (R, B and CR) is halted.
  + Note: there may be pending trace flags in the tracking structures for these channels. Ideally we would continue response capture until all pending response channel traced events had been received and captured (until there were no more trace flags). However, that is not practical for the initial implementation.
* An internally generated flush has executed and completed leading to…
* All per channel trace FIFOs are empty.
* ATB FIFO is empty.

When CS\_TRACE\_CFG\_STS.BSY == 1’b0, trace probe clocks may be gated.

Trace probe configuration registers should generally not be written while CS\_TRACE\_CFG\_STS.BSY is 1’b1, except to change CS\_TRACE\_CFG\_STS.EN to 1’b0. The trace probe hardware does not prevent register updates while CS\_TRACE\_CFG\_STS.BSY is 1’b1, but unpredictable behavior may result.

### Reset, Clocking and Power

The probe logic, as integrated with the bridges, uses the same pair of reset pins (reset\_n and reset\_pd\_n) as the rest of the bridge logic. No dedicated ATRESETn is provided.

For clocking, the ATB interface operates synchronously with the dbg\_ATCLK input, but the core trace capture and processing logic operates off of the bridge’s noc\_clk, and clock crossing structures are inferred as necessary where dbg\_ATCLK and noc\_clk are not synchronous.

Historical info about ARM HTM…

* There are 3 clock domains and 4 asynchronous reset signals that come into the HTM as described in sections 1.3.1 and 5.3 of the ARM HTM TRM.
  + Clocks
    - HCLK (AHB)
    - PCLKDBG (Debug APB)
    - ATCLK (ATB)
  + Resets
    - PRESETDBGn: reset for Debug APB.
    - HTMHRESETn: resets HTM circuits that are in the AHB clock domain, including clearing contents of the FIFO.
    - HRESETn: AHB reset. Has no effect on HTM logic, will cause a reset packet to be inserted into an active trace.
    - ATRESETn: ATB reset.

#### Low Power

The trace probes as integrated with the bridge logic are part of the bridge’s power domain. If the bridge is powered down, tracing capability is lost. There is limited opportunity for the trace probe logic to prevent power down – only in master bridges that are fencing for themselves is there the possibility to return a fence\_deny. In such cases, a configuration bit controls whether or not the probe will deny the power down request.

If tracing is active when sleep\_req\_n is received by the bridge, the probe must prevent assertion of sleep\_ack\_n until tracing has been cleanly stopped. The following steps are taking by the probe before allowing sleep\_ack\_n to be returned.

1. Tracing of all channels is stopped.
2. An internal flush of all channels is performed, all data transmitted over ATB.
3. A GERR packet with payload 0x00 is transmitted over ATB to signal power down has occurred.
4. Trace probe is placed in idle mode.
5. Probe allows sleep\_ack\_n to be returned.

### Miscellaneous

ATB packing: the probes attempt to fully pack the ATB interface, so they will generally hold trace data pending capture of sufficient nibbles to fill a full line of ATDATA. Flush requests and other events that internally generate flushes may force the probes to issue incompletely filled ATDATA lines in order to immediately transmit any pending trace data.

### Software (NocStudio) Support

Trace probes are optional functions of our bridges that are configured via bridge and interface properties. A master “trace\_enable” bridge property controls whether any tracing functionality will be present in the HW, and none of the values of any of the other trace related interface properties should matter if this is set to “no.”

|  |  |  |  |
| --- | --- | --- | --- |
| Bridge Prop | Default | RTL Param | Description |
| trace\_enable | no | P\_TRACE\_ENB | Master enable for tracing functionality. |

**Table 12** Trace Probe Bridge Properties

Each of the existing interfaces of bridges that support tracing has additional interface properties that configure aspects of tracing functionality for that interface, including an interface level “trace\_enable” that controls whether HW is provided to trace that particular interface.

|  |  |  |  |
| --- | --- | --- | --- |
| Interface Prop | Default | RTL Param | Description |
| trace\_enable | yes | P\_<IFCE>\_TRACE\_ENB | Master enable for tracing functionality for this particular interface. |
| trace\_fifo\_depth | varies | P\_<IFCE>\_TRCFIFO\_DEPTH | Depth in bytes of trace capture FIFO for this interface. |

**Table 13** Trace Probe Interface Properties

All bridges that support tracing will have an additional RX interface: ATB. Its presence in RTL (at wrapper level) is determined by the “trace\_enable” bridge property. The “trace\_enable” interface property does not apply to this interface. Following is set of interface properties that apply for the ATB interface.

|  |  |  |  |
| --- | --- | --- | --- |
| Interface Prop | Default | RTL Param | Description |
| data\_width | 32 | P\_ATB\_DATA\_WIDTH | Specifies the width of the ATDATA interface. Valid values are 32, 64 & 128. |
| atid | varies | P\_ATB\_ATID | Specifies the reset default value for ATID[6:0]. Valid values are 0x1-0x6F inclusive. NocStudio should enforce that these are unique across the NoC and refuse to generate IP if that is not the case. |

**Table 14** ATB Interface Properties

### Trace Probe Interface Signals

Bridges that are enabled for tracing will bring out an ATB master (TX) interface as well as some other trace related interfaces. The <brdg> prefix below is replaced with appropriate prefix for the bridge type driving the output (e.g., “acmb” for ACE master bridge).

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Direction** | **Width** | **Description** |
| dbg\_ATCLK | input | 1 | Clock input for ATB interface. Interface operates synchronously w/respect to this clock. |
| <brdg>\_ATBYTES | output | log2(P\_ATB\_DATA\_WIDTH) - 4 | Number of valid bytes in ATDATA. |
| <brdg>\_ATDATA | output | P\_ATB\_DATA\_WIDTH | Trace data. |
| <brdg>\_ATID | output | 7 | ID that uniquely identifies source of trace. |
| dbg\_ATREADY | input | 1 | Downstream ATB slave is ready to accept data. |
| <brdg>\_ATVALID | output | 1 | Transfer valid this cycle. |
| dbg\_AFVALID | input | 1 | Flush request. |
| <brdg>\_AFREADY | output | 1 | Flush acknowledgement. |
| dbg\_SYNCREQ | input | 1 | Synchronization request. |

**Table 15** ATB Interface Signals

There are some additional interface signals that will be added to support tracing.

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Direction** | **Width** | **Description** |
| dbg\_TSVALUE | input | 64 | External timestamp value, encoded as natural binary number. Synchronous to noc\_clk. |
| dbg\_HWEVENTS | input | 2 | CTI inputs. |
| <brdg>\_TRIGOUT | output | 2 | CTI outputs. |
| <brdg>\_ASYNCOUT | output | 1 | CTI output, asserted for one cycle when ASYNC-VERSION goes out on ATB interface (see STM-500 TRM p A-12). |
| dbg\_DBGEN | input | 1 | Invasive debug enable. Synchronous to dbg\_ATCLK. |
| dbg\_NIDEN | input | 1 | Non-invasive debug enable. Synchronous to dbg\_ATCLK. |
| dbg\_SPIDEN | input | 1 | Secure invasive debug enable. Synchronous to dbg\_ATCLK. |
| dbg\_SPNIDEN | Input | 1 | Secure non-invasive debug enable. Synchronous to dbg\_ATCLK. |

**Table 16** Other Trace-Related Signals

# Implementation Plan

This section enumerates a high-level phased implementation plan, focused on bringing up basic functionality initially where choices are unambiguous, refining with additional features over time as requirements clarify.

## Basic ATB Address Stream – AXI4 Bridge

1. Implement CoreSight Register Block with basic set of registers – *TBD*: need to complete definition of this “basic set.”
2. Capture/Trace only addresses from channels
   1. Start with read channel only
   2. Add write channel
   3. Add address packet timestamping
3. Trace Filtering
   1. Initially no filtering, no trace start/stop – trace all addresses from reset deassertion
   2. Add filtering based on address table look up
4. Implement ATB Interface
   1. All pins, parameterized data width – powers of 2 from 32-128 bits
   2. Separate ATB clock used for this interface
   3. Support ATREADY/ATVALID only, no flush, no sync requests

## Basic Trace Master and Trace Slave

1. Initial Trace Master and Trace Slave RTL - simply take incoming ATB packets, frame them for NoC and send them, return them to ATB egress.
   1. Same width ATB interface at all bridges, no internal upsizing or downsizing.
      1. Initially, slave supports single master.
      2. Next, slave supports multiple masters – multiplexes incoming NSTP into ATB performing funnel function.
   2. No support for back-channel: no AFVALID/AFREADY, no SYNQREQs.
   3. Synchronous ATB/NoC clocking.
   4. NocStudio
      1. NoC datapath matches ATB width – NSTP frames fit in single flits.
      2. Simple tree NoC construction.
2. Incremental Features
   1. Implement back-channel network: need RTL building blocks and NocStudio code to instantiate and stitch.
   2. AFVALID/AFREADY
      1. Master bridge: receives pulse from NoC, sets AFVALID at its interface, holds it until AFREADY returned.
         1. If pending ATB data to send – set AFREADY bit in last remaining ATB frame.
         2. If no pending ATB data to send – set AFREADY and send an otherwise empty NSTP frame (signaled with ATID=0x0 per D4.2.4 of ARM CoreSight Architecture Specification v2.0)
      2. Slave Bridge: on rising edge of AFVALID, sets internal flag and launches AFVALID pulse into NoC back-channel. Tracks receipt of AFREADY bit from all master bridges, and when all have returned those bits and all prior trace data has been transmitted out its ATB interface, pulse AFREADY to complete handshake.
   3. SYNCREQ:
      1. Master Bridge: propagate pulse from NoC out ATB interface.
      2. Slave Bridge: propagate pulse from ATB interface into NoC.
   4. Async ATB/NoC clocks
3. Yet Further Incremental Features
   1. Allow NSTP packets to be split across multiple flits to provide flexibility in NoC datapath widths.
   2. Support Trace traffic BW specifications, including traffic profiles for different scenarios, construct trace data layer to sustain these BW’s.
   3. Support priority and QoS in trace data layer.
      1. Support dynamic source priority adjustment in Trace Master.
   4. Configurable ATB widths at Masters and Slave.
      1. Need some means of determining native NoC NSTP width – masters and/or slave perform ATB upsizing/downsizing as needed to match this (see descriptions of CoreSight upsizer/downsizer components in sections 6.3 and 6.4 of ARM CoreSight SoC-400 TRM).

# References

1. [ARM Whitepaper: CoreSight Technical Introduction - A quickstart for designers](https://www.assembla.com/code/netspeed_noc/subversion/nodes/HEAD/doc/external_specs/coresight_technical_introduction_EPM_039795.pdf)
2. [CoreSight Technology System Design Guide](https://www.assembla.com/code/netspeed_noc/subversion/nodes/HEAD/doc/external_specs/DGI0012D_coresight_dk_sdg.pdf)
3. [ARM CoreSight Architecture Specification v2.0](https://www.assembla.com/code/netspeed_noc/subversion/nodes/HEAD/doc/external_specs/IHI0029D_coresight_architecture_spec_v2_0.pdf)
4. [AMBA APB Protocol Specification v2.0](https://www.assembla.com/code/netspeed_noc/subversion/nodes/HEAD/doc/external_specs/IHI0024C_AMBA_APB.pdf)
5. [AMBA 4 ATB Protocol Specification (ATB v1.0 and ATB v1.1)](https://www.assembla.com/code/netspeed_noc/subversion/nodes/HEAD/doc/external_specs/IHI0032B_amba_atb_protocol_spec.pdf)
6. [AMBA AHB Trace Macrocell (HTM) TRM](https://www.assembla.com/code/netspeed_noc/subversion/nodes/HEAD/doc/external_specs/DDI0328E_amba_htm_r0p4_trm.pdf)
7. [ARM CoreSight SoC-400 TRM](https://www.assembla.com/spaces/netspeed_noc/subversion/source/HEAD/doc/external_specs/DDI0480G_coresight_soc_trm_r3p2.pdf)
8. [CoreSight Components TRM](https://www.assembla.com/spaces/netspeed_noc/subversion/source/HEAD/doc/external_specs/DDI0314H_coresight_components_trm.pdf)
9. [ARM CoreSight STM-500 System Trace Macrocell (STM) TRM](https://www.assembla.com/spaces/netspeed_noc/subversion/source/HEAD/doc/external_specs/DDI0528B_coresight_system_trace_macrocell_r0p1_trm.pdf)
10. [ARM System Trace Macrocell (STM) Programmers’ Model Architecture Specification](https://www.assembla.com/spaces/netspeed_noc/subversion/source/HEAD/doc/external_specs/IHI0054B_stm_architecture_specification_v1_1.pdf)
11. [CoreSight Program Flow Trace Architecture Specification](https://www.assembla.com/spaces/netspeed_noc/subversion/source/HEAD/doc/external_specs/IHI0035B_cs_pft_v1_1_architecture_spec.pdf)
12. [MIPI System Trace Protocol (STP) Specification (v2.2)](https://www.assembla.com/spaces/netspeed_noc/subversion/source/HEAD/doc/external_specs/mipi_STP_specification_v2-2.pdf)