ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Ingegneria dell'Energia Elettrica e dell'Informazione "Guglielmo Marconi"

Implementation of the Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V specification

Supervisor:
Prof.
Andrea Bartolini

Co-supervisor:
Simone Manoni

Presented by:
Umberto Laghi

Abstract

The RISC-V *Efficient Trace* (E-Trace) specification offers a standardized and efficient branch tracing mechanism, defining algorithms, input ports, and trace output formats. However, while the Trace Encoder (TE) can produce trace packets as per the specification, its output remains inaccessible outside simulation environments.

To bridge this gap, the RISC-V consortium introduced the *Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V*, a module that transforms TE output into standardized packets suitable for off-chip transmission. For our specific use case, the *ARM Trace Bus* (ATB) protocol was selected as the output format, owing to its suitability for trace data transport and seamless conversion to advanced protocols like AXI and PCIe for debugging purposes.

The resulting architecture is modular and designed for adaptability, enabling easy integration with alternative communication protocols as required. This work provides an efficient and flexible solution for trace data handling, facilitating the extraction of real-time tracing data in modern RISC-V-based systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In modern CPUs, comprehending program behavior presents significant challenges. It is not surprising that software in such environments may occasionally deviate from expected performance and behavior. This unpredictability may depend on various factors, including interactions with other cores, software components, peripherals, real-time events, suboptimal implementations, or a combination of these factors.

To tackle this challenge, was developed a less intrusive and efficient technique to performance debug: tracing. The RISC-V consortium developed the open-source RISC-V ISA and also connected non-ISA specification, among these is present the *RISC-V Efficient Trace (E-Trace)*. This specification is a standardized processor trace method that defines an efficient compressed branch trace algorithm, input port specifications, and trace output packet formats.

To implement the E-Trace specification, the Trace Encoder (TE) was developed. To get more detail about branch tracing and the trace encoder design refer to the associated documentation.

Now, the TE produces the output packets as defined in the specs, but there's a problem: right now it is not possible to read the output produced by the TE outside simulation. To overcome this issue the RISC-V consortium also developed the *Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V*, that describes a module that transforms the tracing data into standard packets sendable off-chip.

The specification does not specify what are the inputs and the outputs: for our specific usecase the input is obviously the TE output. The output is not specified either, but the protocol more practical for our usecase is the *ARM Trace Bus* (ATB).

This protocol was chosen due to the fact that is design to transport tracing data

and that it can be easily converted to other more capable protocol such as AXI. From AXI it can be converted to PCIe and data can be sent to the debug device.

The architecture chosen is designed to be easily adapted to other communication protocols on the output.

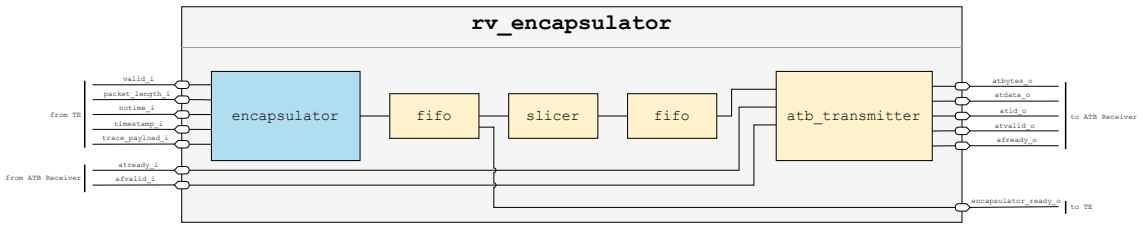

Figure 1.1: Top level module internal architecture

Chapter 2

Encapsulator

This module is designed to create the encapsulated modules as described the specification. The standard encapsulation structure is the following:

Field name	Bits	Description
header	8	Encapsulation header.
srcID	0 - 16	Source ID.
timestamp	T*8	Time stamp.
payload	1 - 248	Packet payload

Table 2.1: Encapsulation field groups

As clearly explicited in the specification: the uppermost field must be transmitted first and multi-bit fields are transmitted least significant bit first.

This module is a simple combinatorial network that takes the inputs from the TE and configuration registers and puts them inside the associated structs (when necessary).

2.1 Header

The header is a byte long and works as an introduction to the payload. It is organized as follows:

Field name	Bits	Description
length	5	Encapsulated payload length. A value of L indicates an L byte payload. Must be ¿ 0.
flow	2	Flow indicator. This can be used to direct packets to a particular sink in systems where multiple sinks exist, and those sinks include the ability to accept or discard packets based on the flow value.
extend	1	Indicates the presence of timestamp when 1. Must be 0 if timestamp width is 0.

Table 2.2: Header fields

The header is implemented as a `packed struct` in system verilog, this way each field can be accessed via dot notation.

2.2 srcID

This fields serves as an identification for the packet source and its size varies from 0 to 16 bits. As explained in the specification, this field can be omitted for two particular scenarios:

- There's only one source in the system.
- The transport scheme uses a sideband bus for the source ID (for example, ATB).

Since in this implementation we have only one source - the TE - and the transport used is ATB, there's no need for this field.

2.3 Timestamp

THis field is used to include time information with every packet. It is included in the encapsulation if header.extend is 1. When included it is T bytes long. The timestamp field can be omitted if the user is not interested or it is already included inside the payload. To verify if the timestamp is included within the payload it looks into the TE configuration parameters.

2.4 Payload

The encapsulation payload has the following structure:

Field name	Bits	Description
type	≥ 0	Packet type. May be eliminated for sources with only on packet type.
trace_payload	$\leq R$	Packet payloads such as those defined for E-Trace. Maximum value of R is defined as 248 - Y - srcID%8 where Y is the length of the type field.

Table 2.3: Payload fields

As explained in the specification, if the TE implemented supports only the *instruction tracing* (like in our case), the *type* field can be omitted from payload. The length of the *trace_payload* field in this implementation is the maximum allowed (248) beacause both the type and srcID fields are omitted.

2.5 Null Encapsulation

The specification also describes the *null packets*, they can be distinguished by standard packets from the length field field of the header that is set to 0. This family of packets is used for those transport schemes that do not use a ready/valid handshake, but instead they a require a continuous flow of data and when there is no useful data the gaps are filled by null packets.

In this implementation it was chosen to use the ATB transport protocol, that uses a ready/valid handshake and so the null packets are not emitted at all.

For completeness, it is the reported the two kinds of null packets:

- *null.idle*, used to fill gaps.
- *null.alignment*, used for synchronization.

To discriminate between the two types it is used the extend field of the header, that otherwise it would have been useless.

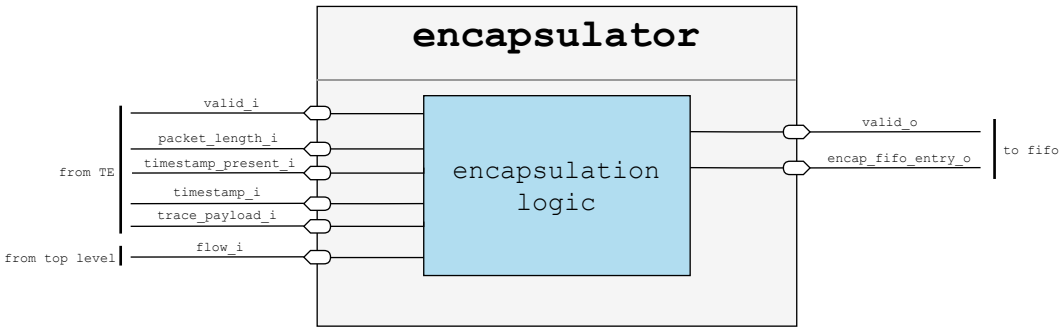

Figure 2.1: Encapsulator internal architecture

Chapter 3

Slicer

This module has the duty to turn the encapsulator output into user chosen chunks of bits. This module is necessary because the ATB interface has a specific sized data bus on which data has to travel.

The *slicer* reads the outputs of the encapsulator module: header, timestamp and payload. Then, they are all put in a single array and starting from the least significant byte, they are output one slice per cycle, along with the number of valid bits inside each slice. The output of this module is connected to a fifo, and when it is full, the module stops outputting slices and waiting for the fifo to have a free entry. The number of valid bytes per slice is necessary for the ATB interface, as required by the AMBA ATB specification.

On instantiation, the *SLICE_LEN* parameter can be selected to make it compliant with the data bus size used by the ATB interface.

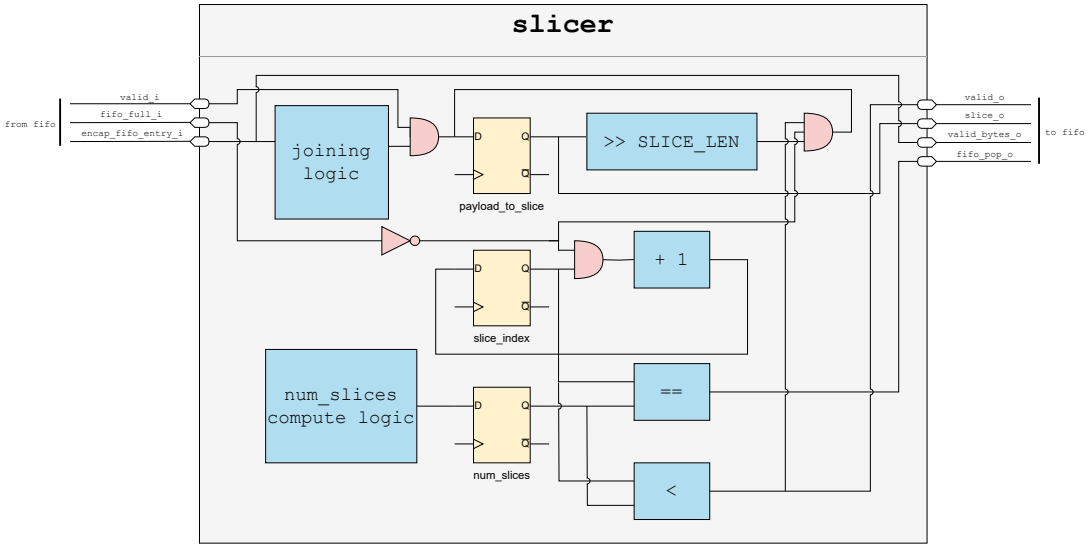

Figure 3.1: Slicer module internal architecture

Chapter 4

ATB Transmitter

The *ATB Transmitter* module performs transactions using the AMBA ATB protocol. This module is fundamental to make the TE communicate with the outside world.

4.1 Signals necessary

The AMBA ATB specification requires the following signals, grouped by usage.

Signal	Transmitter	Receiver	Description
ATCLK	Input	Input	Global ATB clock.
ATCLKEN	Input	Input	Enable signal for ATCLK domain. This signal is optional.
ATRESETn	Input	Input	ATB interface reset, active LOW. This signal is asserted LOW asynchronously, and deasserted HIGH synchronously.

Table 4.1: ATB global signals

Signal	Transmitter	Receiver	Description
ATBYTES[m:0]	Output	Input	The number of bytes on ATDATA to be captured, minus 1.
ATDATA[n:0]	Output	Input	Trace data.
ATID[6:0]	Output	Input	An ID that uniquely identifies the source of the trace.
ATREADY	Input	Output	The Receiver is ready to accept data.
ATVALID	Output	Input	A transfer is valid during this cycle. If LOW, all other AT signals must be ignored.

Table 4.2: ATB data signals

Signal	Transmitter	Receiver	Description
AFVALID	Input	Output	The flush signal to indicate that all buffers must be flushed because trace capture is about to stop.
AFREADY	Output	Input	The flush acknowledge to indicate that buffers have been flushed.

Table 4.3: ATB flush control signals

Signal	Transmitter	Receiver	Description
SYNCREQ	Input	Output	The synchronization request signal to request the insertion of synchronization information in the trace stream.

Table 4.4: ATB synchronization request signals

Signal	Transmitter	Receiver	Description
ATWAKEUP	Input	Output	The wake-up signal indicating any activity associated with an ATB interface.

Table 4.5: ATB wake-up signals

4.2 Transactions

The fundamental transactions that can happen are the following:

- Data transfer.
- Flush control.
- Synchronization.

This module can be instantiated to use different sized data bus, given they are compliant with the ones defined in the specification.

Within the AMBA ATB specification are defined different functionalities and some of them are optional. Here is a table that shows which are implemented and which are not:

Feature	Supported
Flush control	yes
Synchronization	no
Wake-up	no

Table 4.6: ATB features supported by the ATB transmitter module

4.2.1 Data transfer

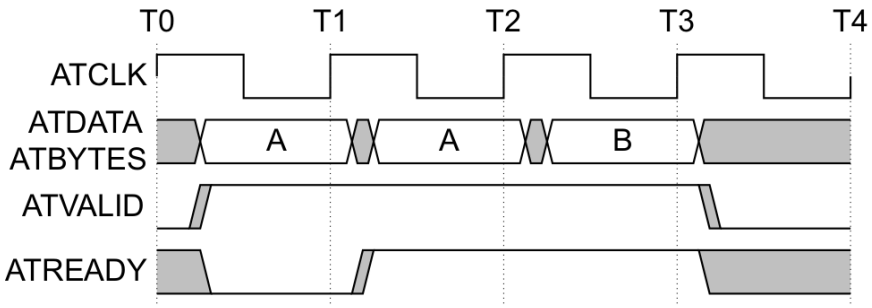

Figure 4.1: ATB data transfer

The transfer is initiated by the transmitter that asserts the `ATVALID` signal and puts data along with the number of valid bytes (`ATBYTES`) on the bus. The `ATVALID` signal is asserted and data are kept on the bus until the receiver asserts the `ATREADY` signal, meaning the data can be read.

4.3 Flush control

The buffer flush is used to clean the FIFO to prepare for new data as cycles progress. The handshake changes depending on the architecture used for the Transmitter:

with or without internal storage.

4.3.1 Transmitter with internal storage

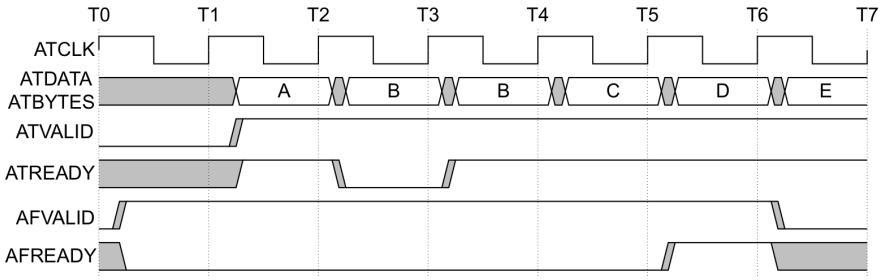

Figure 4.2: ATB flush transmitter with storage

When the `AFVALID` signal coming from the receiver is asserted, the Transmitter starts outputting the data stored and generated until the `AFVALID` signal was received. When all the data stored before the receiving of the `AFVALID` signal have been sent, the `AFREADY` signal is asserted. This signal does not mean the FIFO is empty.

4.3.2 Transmitter with no internal storage

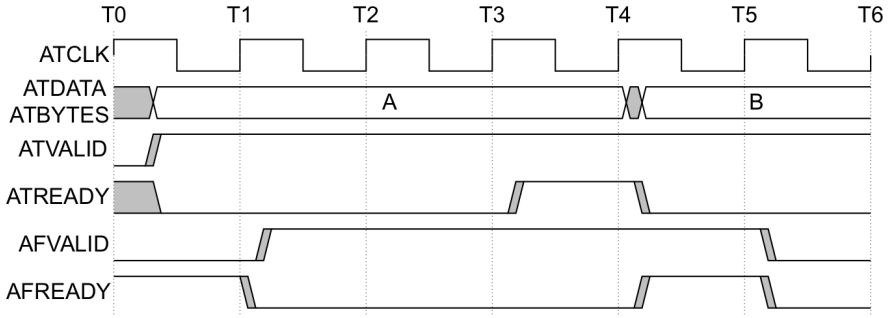

Figure 4.3: ATB flush transmitter with no storage

In case the Transmitter has no internal storage, the flush can be done. In this scenario, the packet considered "old" and to be flushed is the one that the Transmitter is trying to send when the `AFVALID` signal is received. The cycle after the "old" packet has been sent, the `AFREADY` signal is asserted communicating to the Receiver the flush has been completed.

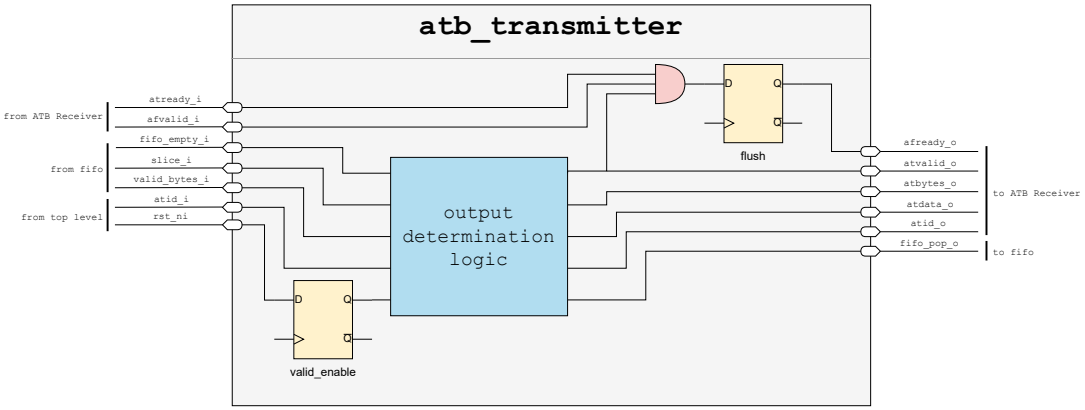

Figure 4.4: ATB Transmitter module internal architecture