

High Throughput Hardware Accelerated CoreSight Trace Decoding

Anonymous

dept. name of organization (of Aff.)

name of organization (of Aff.)

City, Country

email address or ORCID

Anonymous

dept. name of organization (of Aff.)

name of organization (of Aff.)

City, Country

email address or ORCID

Anonymous

dept. name of organization (of Aff.)

name of organization (of Aff.)

City, Country

email address or ORCID

Abstract—A single tracing component embedded into a high-frequency processor may produce up to 1 GB/s of trace data or more. These data are vital in debugging, monitoring, verification, and performance analysis in System-on-chip and heterogeneous system development. Hardware trace decoders and analyzers have emerged to support online processing of trace data to perform the aforementioned use cases in real time. However, the existing hardware trace decoders designed for the Embedded Trace Macrocell version 4 (ETMv4), a standard feature in most modern ARM processors, can only process trace data at a maximum rate of 125 MB/s - 250 MB/s. This paper proposes an optimized and parallelized trace decoder for the ETMv4 specification implemented on a Xilinx Ultrascale+ processing up to 1 GB/s of trace data from a single ETM.

Index Terms—Tracing, Coresight, ETM, Decoding, FPGA

I. INTRODUCTION

Runtime traces collected from dedicated hardware components are an extremely detailed form of profile data. Such traces are used for non-invasive debugging to find non-deterministic bugs [10], verify runtime properties of safety-critical applications in real-time [12, 6, 5, 8], automate software testing with native on-device fuzzing [9], and optimize or monitor applications with detailed performance metrics [11]. Traces are further integrated into compiler toolchains to enable optimizations [4, 3]. Given the high volume of trace data produced by dedicated hardware, usually upwards of hundreds of MB/s, real-time trace decoders implemented on FPGAs have emerged to support processing traces online instead of accruing data in buffers and performing analysis offline [14, 13, 8].

ARM’s CoreSight subsystem exposes a family of tracing and debugging components, most prominently the Embedded Trace Macrocell (ETM) [1], that is tightly coupled to a Processing Unit (PU) to produce zero-overhead traces. Current real-time trace decoders and analyzers are limited by the volume of trace data (from a single source) they can decode as they are geared toward lower-frequency cores such as microcontrollers. To the best of our knowledge, the highest throughput decoder for the ETMv4 specification can handle a minimum of 125 MB/s and up to 250 MB/s depending on the trace data [15]. Prior work is not able to decode in real-time traces that are produced by high-frequency cores with inbuilt ETMs like ThunderX-1, Neoverse N1, or any cores from the Cortex-A series, where even a single Cortex-A53 core can produce 1 GB/s of trace

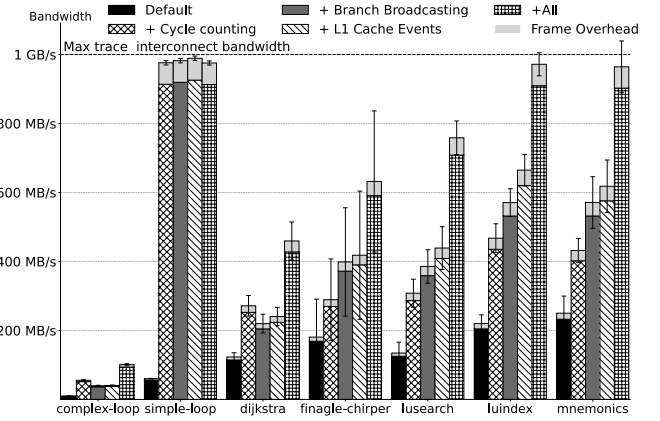


Fig. 1: Volume of trace data produced by a single ETM tracing a Cortex-A53@1.3GHz running different benchmark and ETM configuration combinations. We observe a large trace bandwidth variety depending on the setup. Simple-loop is a binary that performs only branching with `nop` instructions inside the loop and quickly overflows buffers and saturates interconnect bandwidth, indicating a bandwidth of more than 1 GB/s could be produced. Complex-loop is an endless loop with long expensive CPU operations inside and each loop iteration is slow, generating low bandwidth trace data. Other benchmarks are a selection of large-scale workloads with language runtimes reflecting common application workloads.

data (Fig. 1). Without the capability to decode these high bandwidths, valuable information may be discarded.

Achieving high decoding throughput is non-trivial since trace data are typically heavily compressed to minimize bandwidth. As a result, parallelizing the decoding process is difficult, a fact reflected in prior work where most decoders can only handle a single trace element per cycle in common scenarios, heavily throttling the effective throughput [15].

In this work, we introduce a novel ETMv4 instruction trace decoder to address these issues. We apply a trace stream *unrolling* technique that allows the decoder to *always* process multiple trace elements in the same clock cycle, no matter the inter-element dependencies caused by compression. The decoder using this technique supports a data rate of up to 1 GB/s and is implemented for the ETMv4 protocol. This paper makes the following contributions:

- A novel decoder for the ETMv4 instruction specification that can handle multiple bytes of trace data in parallel *every cycle*.

- An implementation of the decoder on a Xilinx Ultrascale+ xczu5ev-sfvc784-2-i that can handle 1 GB/s (handling 4 bytes each cycle) of trace data with an operating frequency of 250 MHz while using around 8.4% of the device resources.
- A correctness validation of the implementation against open CoreSight Decoding library (OpenCSD) [7].

II. RELATED WORK

The initial work on hardware trace decoding for the ETM protocol was presented by Weiss & Lange [13]. Later real-time PTM trace decoders follow a similar concept as well [8]. This approach involves collecting trace data in a buffer and employing an evaluation window to extract full trace elements from within this window. To enhance throughput, multiple overlapping evaluation windows are used to determine trace element boundaries (Fig. 2). Once trace element boundaries are known, multiple decoders begin decoding from a synchronization element. The synchronization element guarantees that no trace information depends on any trace element sent prior to the synchronization. This decoder was primarily tailored to the ETMv1 - ETMv3 or the Program Trace Macrocell (PTM) specifications and is not directly suitable for ETMv4. Trace elements in the ETMv4 specification may be of unbounded size and a finite evaluation window will no longer be able to guarantee that a full trace element is contained by the window.¹ Furthermore, the parallelization concept of using multiple decoders at synchronization elements is orthogonal to the decoder introduced in this paper, and both can be used simultaneously. Multiple parallel decoders require more resource utilization, larger amounts of trace data (multiple synchronization periods) to be stored in buffers, and will likely incur higher synchronization overhead to break the trace stream into smaller decodable chunks.

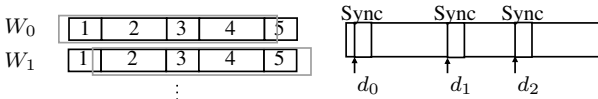


Fig. 2: Simplified parallelization principle of the ETM decoder introduced by Weiss & Lange [13]. Trace element boundaries (1-5) are determined by multiple overlapping evaluation windows W_i . Each window assumes that a trace element starts at the window boundary and speculatively determines all trace element boundaries within the window. The speculation is fully resolved at a later stage. To reconstruct program flow, multiple decoding units d_i start at a fresh synchronization element to decode in parallel.

Zeinolabedin, Partzsch, & Mayr introduce a decoder to support unbounded trace element sizes and thus ETMv4 [15]. Instead of an evaluation window, they use a Control Core that analyzes the trace data held within a small buffer. If multiple small trace elements are seen in succession, both elements are handled in parallel. However, if a trace element is large (more than 1 byte) each payload byte is processed individually.

¹Maximum trace element sizes *can* be determined for a subset of configurations and given a concrete ETM/PU combination.

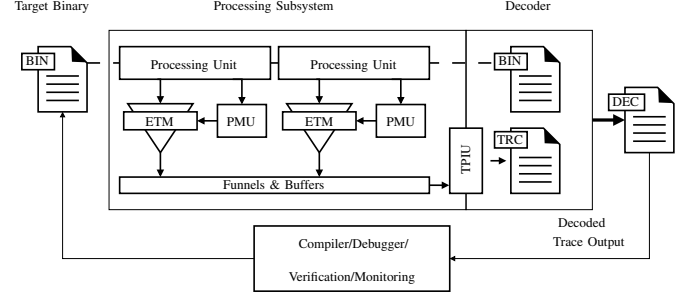


Fig. 3: CoreSight component overview and typical trace workflow: Trace data are produced at a trace source, in our case an ETM, driven to the TPIU through the trace bus, buffers, and funnels to the TPIU. Data from the TPIU can be processed by a trace analyzer in either hardware or software to be decoded and integrated into the surrounding toolchain.

Processing each byte sequentially makes it impossible to deal with the high bandwidths we wish to decode.

The decoder introduced in this work can support any trace element type or length with consistent parallelization, allowing for higher throughputs when decoding the ETMv4 specification. Multiple decoders can be used in parallel as the decoder by Weiss & Lange for even further throughput increases.

III. CORESIGHT TRACING SUBSYSTEM

The CoreSight subsystem is a network of components to enable the tracing of a system without interfering (zero-overhead) with the running application [2]. Typically, each PU has a tightly coupled tracing unit, in our case an ETM. The tracing unit is further connected to other CoreSight components, for example to a Performance Monitoring Unit (PMU) to optionally embed hardware events like cache misses into the trace stream.

Further components — a simplified block diagram is included in Fig. 3 — like buffers, funnels, and replicators are present in the subsystem and are responsible for driving trace data to a trace sink. For real-time trace analysis, the trace sink will typically be the Trace Port Interface Unit (TPIU), as this component supports the highest data rate [2]. The TPIU also interleaves source identifiers and raw trace data into frames to multiplex the interconnect. Two types of decoders are required: one to extract individual trace streams from raw TPIU frames referred to as an L1 decoder, and a distinct decoder for each trace source, or an L2 decoder. The L1 decoder is not the bottleneck and can handle multiple GB/s [15] and this work exclusively discusses L2 decoder design.

A. ETMv4 Instruction Trace Stream Protocol

An ETM instruction trace is a *compressed* and *packet-based* stream. Each packet in the trace stream consists of a sequence of full bytes and comprises a header followed by a variable and unbounded number of payload bytes. The ETM specification has around 45 unique headers [1] and is reported to have over 400 packet subtypes [15]. The main purpose of the trace data is to convey the sequence of virtual addresses of the instructions that are being executed on a PU to a trace analyzer. The ETMv4 specification optionally supports a wide range of features that provide additional details regarding the execution context or execution metrics. This includes tracing accurate

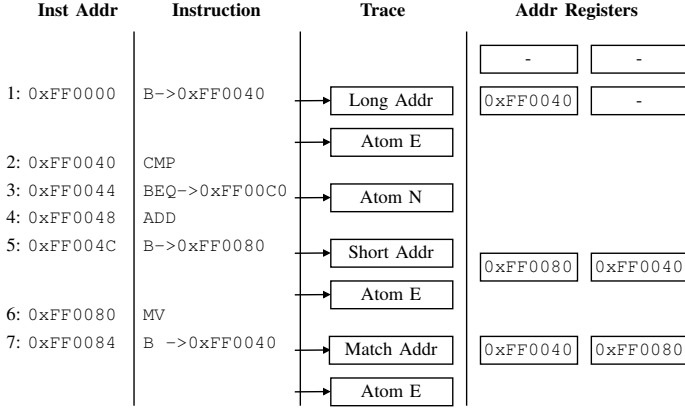


Fig. 4: A sequence of instructions being traced alongside their virtual memory addresses and the ETM packets that are produced by each instruction. Line 5 will only produce a Short Address packet with one byte of address data 0x80, and the address values held in the registers are shifted by one. Similarly, line 7 generates an Exact Match Address packet since the branch target is already held in the second address register.

processor cycle counts between basic blocks, tracing context and virtual machine identifiers to associate each instruction with its execution context, tracing hardware events, e.g. cache misses, and more.

A trace decoder must decompress the trace data, reconstruct the program flow, and associate it with any additional information encoded by raw traces. For this reason, understanding the compression scheme and the ETMv4 specification is critical to designing a high-throughput decoder. For reasons of brevity, we limit the description of the trace decoding process in this work to reconstructing the program flow, but our decoder supports the full specification.

Atom packets and Address packets together encode the program flow. The Atom packet acts as a signpost and is sent out whenever a program flow-changing instruction (P0 element), is executed. The information carried in an Atom packet only signals if a P0 instruction was taken (E) or not taken (N). But Atom packets alone are not enough for a trace decoder to reproduce the program flow — when the branch target address is unresolvable, the address is sent over the trace stream for a given P0 element in the form of an Address packet. This encoding principle is illustrated in Fig. 4, showing a short sequence of instructions executed on a PU and the corresponding packets produced by an ETM. Emitting full addresses over the trace stream is redundant. The ETM has three internal address registers storing the addresses contained by the last three address packets issued by the ETM. If the succeeding address packets’ more significant bits match with previous addresses, a smaller packet is sent with only the distinguishing lower significant bits.

As mentioned, branch target addresses are only sent out if the branch target cannot be inferred by the decoder. We note that the ETM specification intends for the trace data to be decoded alongside the binary, as is typical in debugging environments. This means the branch target of the branching instruction on line 1 *can* be inferred without sending any address data whatsoever, but this requires access to the target binary. ETMs have a *branch-broadcasting* feature that sends address packets

for every P0 element, allowing for decoding without the binary at the cost of higher trace volume. Our current ETM decoder supports only decoding with the branch-broadcasting feature enabled. Extending the decoder to work with a copy of the binary in hardware is left as future work.

The precise output format of a processed stream of trace data depends on both the ETM configuration and decoded trace application. Regardless of the configuration, this typically includes a stream of resolved branch target addresses as the basic building block. Table I illustrates an example output stream.

TABLE I: Decoded trace example

Type	Cycle	Value
Br	0x004	0xFF0040
Evt	0x010	L1D_CACHE_REFILL
Evt	0x050	Br_MIS_PRED
Br	0x055	0xFF0080
Evt	0x0455	L2D_CACHE_REFILL
Br	0x0460	0xFF0040

This stream is generated by the same sequence of instructions as in Fig. 4, with cycle-counting and event tracing².

IV. DECODING THE ETMv4 SPECIFICATION

Decoding the ETMv4 trace can be broken down into two main tasks: first determining packet boundaries and then decompressing the information encoded in these packets. To achieve a decoding throughput in the GB-range at FPGA frequencies (few hundred MHz), multiple bytes of trace data must be processed in parallel. However, this directly clashes with the fact there are two types of dependencies inherent to the ETM specification: Inter-packet and intra-packet dependencies. These both mandate sequential processing. We define and illustrate by example:

a) *Packetization*: Packets can be either variable-sized, fixed-sized, or header-only packets and because of this each byte needs to be processed sequentially to mark packet boundaries. For example, the Short Address packet in Fig. 5 is a variable-sized packet, meaning each payload byte contains a continuation bit to denote whether the current byte is the final byte of the payload. To determine if byte b_9 should be interpreted as a header byte of a new packet or an additional payload byte of the Short Address packet, byte b_8 must be, at least partially, processed. Accordingly, this implies the sequential processing of bytes $b_1 \dots b_n$.

b) *ETMv4 compression*: A similar scenario presents itself at the packet level. Observe the Atom packet p_2 in Fig. 5. This packet encodes a jump to the address stored in the first address register and is therefore determined by previous address packets. Before the Atom packet is processed the address registers must be properly mirrored, as a result, this again implies sequential processing of packets $p_1 \dots p_n$.

A. Overview & Key Idea

Considering these dependencies, our approach to achieving high-throughputs is to decode at the byte level and forego full repacketization, and mirror the ETM registers after each byte. Decoded output is produced once a full packet is processed. To achieve high throughput, the bitwise decoding function is optimized and pipelined such that the decoding circuit can be applied multiple times every cycle. The trace stream is *unrolled*

²This information is encoded into additional packets not shown in Fig. 4.

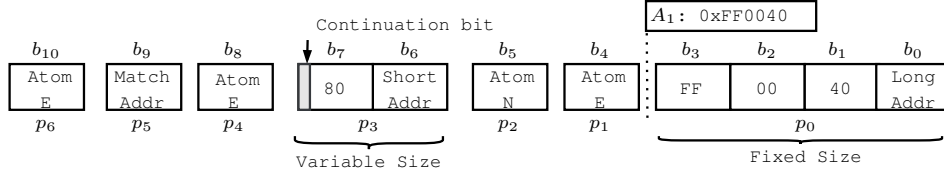


Fig. 5: Byte stream representation of packets produced in Fig. 4. The Short Address packet has a continuation bit to mark the last payload byte, while the Long Address packet has a fixed size known when the header is resolved. The address register, the first shown as A_1 after packet p_1 , must be kept up to date with the internal register of the ETM acting as the trace source and is required for decoding.

such that the decoding function can be simultaneously applied to multiple incoming bytes of the trace stream.

We begin with a brief overview of the byte-wise decoding function where we introduce two state components, the *stream state* for packet context (repacketization) and the *trace state*, mirrored ETM registers, for decompression. We continue with unrolling the trace stream to parallelize the decoding function and later describe our optimization approach with pipelining and speculative preprocessing.

B. Decoding function

More formally, a decoding function \mathcal{D} consumes a byte from the trace stream alongside a stream state S and trace state T to produce an updated trace and stream state S', T' :

$$\mathcal{D}(b, S, T) \rightarrow S', T' \quad (1)$$

This means that decoding a full trace, a stream of n bytes, is a successive chaining of the decoding function:

$$\mathcal{D}(b_n \mathcal{D}(b_{n-1}, \dots \mathcal{D}(b_0, S, T))) \quad (2)$$

This naive decoder can handle one byte per cycle. Implementing a decoding function that takes more than one byte with combinational logic is infeasible due to the sheer number of packet types in the specification. Instead, the stream of bytes from the trace stream are *unrolled*³ with an unroll factor u , such that u bytes are input to the full decoding unit that processes these bytes with throughput equal to $\frac{u \text{ bytes}}{\text{cycle}}$. The decoding function \mathcal{D} is used as a subcomponent for the unrolled decoder and \mathcal{D} is chained based on the position of the unrolled byte:

$$\begin{aligned} S^{i+1}, T^{i+1} &\leftarrow \mathcal{D}(b_0, S^i, T^i) \\ S^{i+2}, T^{i+2} &\leftarrow \mathcal{D}(b_1, \mathcal{D}(b_0, S^i, T^i)) \\ &\vdots \\ S^{i+u}, T^{i+u} &\leftarrow \mathcal{D}(b_{u-1} \dots \mathcal{D}(b_1, \mathcal{D}(b_0, S^i, T^i))) \end{aligned} \quad (3)$$

We refer to a full unrolled decoder with unroll factor u as \mathcal{D}_u .

C. Pipelining & Optimizing

The unroll factor provides flexibility in the number of bytes per cycle the decoder can process and determines the degree of parallelization. The key to achieving high throughput subsequently is optimizing the decoding function. The critical path is the last line in Equation (3), as the decoding function must be applied u times in a single clock cycle.

The circuit for our unrolled decoder is visualized in Fig. 6 and shows the four stages of the decoding process, *header preprocessing*, *stream state processing*, *action preprocessing*, and *trace state processing*. To minimize the critical path, the stream

state and the trace state are split into separate stages, as they can be computed independently from one another. Importantly, the computation to update the stream state or trace state cannot be further broken down, as full updates to both states must be made within a single cycle to make progress. Furthermore, any computation that can be precomputed without state context is speculatively executed in a preprocessing stage. For example, the header preprocessing stage looks up the header type of an incoming byte. This process is done speculatively and for every byte, regardless of whether the byte should be interpreted as a header. In the case the byte really should be interpreted as a header, the additional level of logic required to compare bits has already been performed outside of the critical path.

The stream state and trace state units are further elaborated upon in simplified Algorithms 1 and 2 respectively. The SSU is responsible for computing packet boundaries and setting current indices and header types. The logic is kept to a minimum, e.g. checking if a byte is the last byte in a payload requires at most 3-bit comparisons due to one-hot encoding of payload indices stream mode types. When a header is seen, payload sizes and stream mode are determined through hardcoded values based on the packet header type.

The TSU is correspondingly optimized but has the luxury of more preprocessing in the form of *action codes* that encode an update to the trace state. For example, byte b_0 (Long Address header) from Fig. 5 generates a `shift_address_registers` code, as the address registers must be shifted to prepare for the incoming address values. From the stream state, all action codes can be resolved. Using the payload index value and current header value, the action code can be generated on what bits to overwrite

Algorithm 1: Stream State Unit (SSU)

Input : data byte b , resolved header h , stream state S
Output : Updated Stream State S'
Record Stream State is
 $\text{mode} \in \{\text{Header}, \text{PldFixedSize}, \text{PldContinuous}\};$
 $\text{header};$
 $\text{index}; /*\text{Reverse one-hot payload index}*/$
Process Stream State is
 switch $S.\text{mode}$ **do**
 case *Header*
 $S'.\text{mode} \leftarrow \text{mode}(h)$
 $S'.\text{index} \leftarrow \text{size}(h)$
 case *PayloadFixedSize*
 $S'.\text{mode} \leftarrow \text{Header}$ **if** $S.\text{index}[0]$ **is** 1
 else $S'.\text{index} \leftarrow S.\text{index} \gg 1$
 case *PayloadContinuous*
 $S' \leftarrow \text{Header}$ **if** $\text{index}[0]$ **is** 1 **or** $b[0]$ **is** 0
 else $S'.\text{index} \leftarrow S.\text{index} \gg 1$

³Reminiscent of loop unrolling without vectorization, hence the name

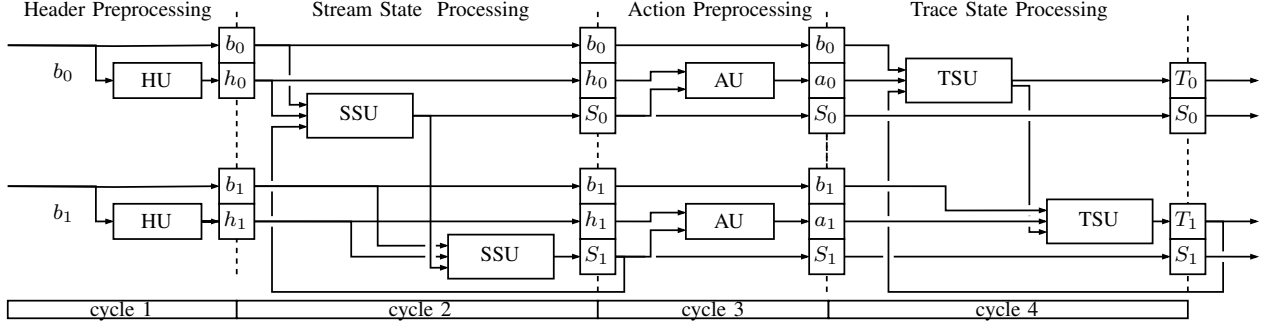


Fig. 6: Fully pipelined and \mathcal{D}_2 . HU=Header preprocessing Unit, SSU=Stream State processing Unit, AU=Action preprocessing Unit, and TSU=Trace State processing Unit. The unrolled decoder takes two bytes b_0 and b_1 at an initial trace and stream state to produce an updated stream and trace state S_0, S_1, T_0 and T_1 which together hold all information encoded into the trace stream. Both the SSU and the TSU have to be applied twice in one cycle as the input of the second unit depends on the first unit, and these are the critical timing paths.

Algorithm 2: Trace State Unit (TSU)

Input : Trace data byte b , stream state S at b , resolved action code a
Output : Updated Trace State T
Record Trace State is
 $\text{address_regs}[2] \leftarrow$
 \vdots
Process Trace State is
 switch a **do**
 case $\text{shift_address_registers}$
 $\text{address_regs}[1] \leftarrow \text{address_regs}[0]$
 $\text{address_regs}[2] \leftarrow \text{address_regs}[1]$
 case $\text{update_address_8_2}$
 $\text{address_reg}[0][8 : 2] \leftarrow b[6 : 0]$
 case $\text{update_address_15_9}$
 \vdots

in the address registers with the incoming bytes, e.g. with an `update_address_8_2` code for index 0 in a Short Address packet. Intuitively, every possible update to the trace state is encoded by a single action and updating the trace state (having a sequential dependency) is performed for u different actions in succession within a cycle.

We note that, for brevity, many state components are omitted from Algorithm 1 and action codes are omitted from Algorithm 2. One example of such an omission from the stream state is the inclusion of lookahead states required to handle more complex packets with optional subpackets or composite packets.

V. IMPLEMENTATION RESULTS

Our evaluation of the proposed decoder is twofold — a performance and resource utilization breakdown of the implemented design in Table II, and a correctness validation against the openCSD [7].

A. Performance & Resource Utilization

The performance and resource utilization of the unrolled decoder with different unroll factors are compared in Table II to a baseline implementation of Zeinolabedin, Partzsch, & Mayr [15], which, to the best of our knowledge is the only other reported implementation of an ETMv4 specification decoder. We emphasize the decoders are implemented on different FPGA devices and a direct comparison are not exact as the maximum operating frequency and device utilization depend on device characteristics.



The throughput of the decoders is determined by the bytes per cycle it can process and maximum operating frequency f . The maximum operating frequency of the unrolled decoder is inversely correlated with the unroll factor, as the required logic to be performed within a single cycle increases with the unroll factor. The highest throughput is achieved with an unroll factor of 4 running at 250 MHz.

The unrolled decoder \mathcal{D}_u can handle more throughput, up to $8\times$ more in the case of \mathcal{D}_4 , than the decoder by Zeinolabedin, Partzsch, & Mayr. The improvement is in part due to handling multiple bytes per cycle, and higher operating frequencies, although without taking into consideration the different FPGA devices.

Resource utilization of our implemented design is low, using a maximum of 14.5% with unroll factor 6 of the FPGA resources and 2.51% with unroll factor the 1. For the highest throughput at 1 GB/s, the decoder uses 8.38% of the device's resources. Overall, the utilization is kept comparable and even lower than prior work while increasing throughput. Critically, as the decoder can operate at fixed a line rate, no BRAM space is required for our implementation. These results also show that using an unroll factor >4 has no benefit while it uses more device resources.

B. Correctness

To ensure correctness, the design was both simulated and run on a Zynq Ultrascale+ device with a tracing session active on a single Cortex-A53 core. The latter process is visualized in Fig. 7. The actively maintained openCSD is used as an oracle to match the outputs of the implemented decoder. The openCSD is designed for decoding a trace alongside the target binary and provides more details than is contained by the trace itself. To directly compare against the trace output produced by our decoder, we added a translation step to transform openCSD output to the same format as produced by the decoder (Table I). The input stimulus for openCSD is collected from real tracing sessions on the Cortex-A53, where raw frames are collected directly from the TPIU and made accessible to the PS through AXI-DMA. For the simulation test bench, the raw frames were also used as a test vector. Using the TPIU and AXI-DMA we can collect much larger test vectors than with reading only from internal trace buffers of the CoreSight subsystem, giving us multi-GB datasets.

TABLE II: Performance and resource utilization  = Xilinx Virtex xc6vcx75t-2ff784  = Xilinx Ultrascale+ xczu5ev-sfvc784-2-i. The values for the decoder by Zeinolabedin, Partzsch, & Mayr (Baseline) are directly taken from the L2 decoder performance/utilization reports [15].

	Performance				Resource Utilization			
	$min(\frac{\text{bytes}}{\text{cycle}})$	$max(\frac{\text{bytes}}{\text{cycle}})$	$max(f)$	Throughput	LUT	Reg	BRAM	CLB/Slice
Baseline [15]	1	2	125 MHz	125-250 MB/s	3160(6%)	1006(1%)	8(5%)	1028(8%)
Unroll factor 1 (\mathcal{D}_1)	1	1	550 MHz	550 MB/s	521(0.44%)	631(0.25%)	0(0%)	368(2.51%)
Unroll factor 2 (\mathcal{D}_2)	2	2	400 MHz	800 MB/s	1701(1.44%)	953(0.40%)	0(0%)	689(4.71%)
Unroll factor 3 (\mathcal{D}_3)	3	3	300 MHz	900 MB/s	2375(2.03%)	1324(0.57%)	0(0%)	967(6.61%)
Unroll factor 4 (\mathcal{D}_4)	4	4	250 MHz	1000 MB/s	3075(2.62%)	1614(0.69%)	0(0%)	1227(8.38%)
Unroll factor 5 (\mathcal{D}_5)	5	5	180 MHz	900 MB/s	5702(4.87%)	1965(0.84%)	0(0%)	1998(13.64%)
Unroll factor 6 (\mathcal{D}_6)	6	6	130 MHz	780 MB/s	5727(4.88%)	2282(0.98%)	0(0%)	2128(14.53%)

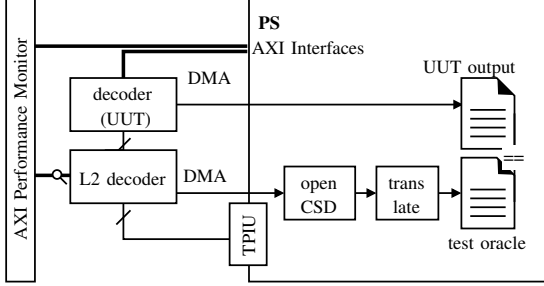


Fig. 7: Setup of throughput measurement and validation against openCSD. Raw trace data and decoded trace data are collected simultaneously until the main memory of the device is full. Once the trace capture session is complete, the OpenCSD is run on the raw trace data and translated to the same output format as the hardware decoder and matched. Packet type and length counter registers are incremented inside the decoder and are accessible through an AXI memory-mapped interface. Performance and trace bandwidth metrics reported throughout this work is measured by both an AXI performance monitoring unit and the internal decoder counters.

The same principle is used to validate the decoder running on a device. Fig. 7 shows how both raw frames and the decoded trace is collected in parallel and validated as soon as the target tracing session is complete. Both simulation and validation against openCSD were successful.

VI. CONCLUSION

We have presented a high-throughput decoder for the ETMv4 instruction trace specification. The decoder is implemented on a Xilinx xczu5ev-sfvc784-2-i and can handle up to 1 GB/s of trace data produced by a single trace source. This is enough to decode at the maximum data rate of the TPIU on a Zynq Ultrascale+ and supports all high-bandwidth features of an ETM coupled to a Cortex-A53@1.33 GHz e.g. cycle-counting, branch-broadcasting and event tracing simultaneously. In addition, we have outlined a framework for decoding any CoreSight-compliant stream. Our decoder design ensures that multiple bytes are processed every cycle regardless of packet boundaries and despite the inter-byte dependencies. This is achieved by unrolling the trace stream and employing an optimized bitwise decoding function. All of this means that our decoder can be used for online analysis of trace streams for

real-time runtime verification, feedback-directed optimization, and monitoring among other applications. For future work, we intend to support decoding the trace alongside a compressed form of the trace target binary to reconstruct every executed instruction. This also eliminates the requirement for branch-broadcasting support on the ETM and gives us full control flow reconstruction while also reducing trace data volume. Furthermore, our decoder can be simultaneously extended to support the ETM data trace specification and we aim to add stages to synchronize the ETM instruction trace with both an ITM/STM trace and the separate ETM data.

REFERENCES

- [1] ARM Ltd. *Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.6* ARM IHI0064H. 2020.
- [2] ARM Ltd. *ARM® CoreSight™ SoC-400 DDI0480G*. 2015.
- [3] Dehao Chen, David Xinliang Li, and Tipp Moseley. “AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications”. In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 2016, pp. 12–23.
- [4] *Coresight AutoFDO Collect ETM data for AutoFDO*. https://android.googlesource.com/platform/system/extras/+master/simpleperf/doc/collect_etm_data_for_autofdo.md. Accessed: 2023-4-4.
- [5] Normann Decker et al. “Rapidly adjustable non-intrusive online monitoring for multi-core systems”. In: *Formal Methods: Foundations and Applications: 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29–December 1, 2017, Proceedings 20*. Springer. 2017, pp. 179–196.
- [6] Hannes Kallwies et al. “TeSSLa—an ecosystem for runtime verification”. In: *International Conference on Runtime Verification*. Springer. 2022, pp. 314–324.
- [7] Linaro. *Linaro/openCSD: Coresight Trace Stream decoder developed openly*. URL: <https://github.com/Linaro/OpenCSD>.
- [8] Pirmin Schmid. “Runtime verification with tessla on enzian”. MA thesis. ETH Zurich. 2019.
- [9] Haoqi Shan et al. “CROWBAR: Natively Fuzzing Trusted Applications Using ARM CoreSight”. In: *Journal of Hardware and Systems Security* (2023), pp. 1–11.
- [10] Alan P Su et al. “Multi-core software/hardware co-debug platform with ARM CoreSight™, on-chip test architecture and AXI/AHB bus monitor”. In: *Proceedings of 2011 International Symposium on VLSI Design, Automation and Test*. IEEE. 2011, pp. 1–6.
- [11] Adrien Vergé, Naser Ezzati-Jivan, and Michel R Dagenais. “Hardware-assisted software event tracing”. In: *Concurrency and Computation: Practice and Experience* 29.10 (2017), e4069.
- [12] Conal Watterson and Donal Heffernan. “Runtime verification and monitoring of embedded systems”. In: *IET software* 1.5 (2007), pp. 172–179.
- [13] Alexander Weiss and Alexander Lange. *Trace-data processing and profiling device*. US Patent 9,286,186. Mar. 2016.
- [14] Seyed Mohammad Ali Zeinolabedin, Johannes Partzsch, and Christian Mayr. “Analyzing ARM CoreSight ETMv4. x Data Trace Stream with a Real-time Hardware Accelerator”. In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021, pp. 1606–1609.
- [15] Seyed Mohammad Ali Zeinolabedin, Johannes Partzsch, and Christian Mayr. “Real-time hardware implementation of arm coresight trace decoder”. In: *IEEE Design & Test* 38.1 (2020), pp. 69–77.