High Throughput Hardware Accelerated CoreSight Trace Decoding

Matthew Edwin Weingarten
Systems Group, D-INFK,
ETH Zurich
Zurich, Switzerland
matthew.weingarten@inf.ethz.ch

Nora Hossle
Systems Group, D-INFK,
ETH Zurich
Zurich, Switzerland
nora.hossle@inf.ethz.ch

Timothy Roscoe
Systems Group, D-INFK,
ETH Zurich
Zurich, Switzerland
troscoe@inf.ethz.ch

***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 for real-time applications. 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 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 [12], verify runtime properties of safety-critical applications in real-time [14, 7, 6, 10], automated software testing with native on-device fuzzing [11], and optimize or monitor applications with detailed performance metrics [13]. Traces are further integrated into compiler toolchains to enable optimizations [5, 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 [17, 16, 10].

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 either do not support ETMv4 [10, 16] or are unable to handle the upper range of trace data bandwidth of high-frequency CPUs [18]. ETMv4 encompasses a major overhaul of the trace specification over previous versions and requires a complete redesign of the decoder. The newer specification also seems to have become the de facto standard among CPUs since ARM-V8, including the likes of the Cortex-A, Cortex-R, Cortex-M, the ThunderX-1, and Neoverse series. As such, a decoder for ETMv4 covers a wide range of ARM CPUs on the market. However, to the best of our knowledge,

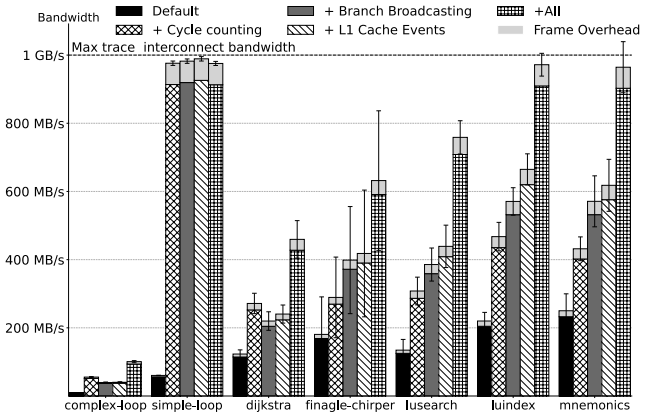

Fig. 1: Volume of trace data produced by an ETM tracing a single Cortex-A53@1.3 GHz core running different benchmark and ETM configurations. Simple-loop is a busy-looping application, whereas Complex-loop performs expensive operations each iteration, loosely representing upper and lower bounds. Finagle-chirper, lusearch, luindex and mnemonics are part of the renaissance server-grade benchmark suite representing a set of typical applications [9].

the highest throughput ETMv4 decoder can handle a minimum of 125 MB/s and up to 250 MB/s depending on the type of trace data packets [18]. Critically, this throughput is too low to cover all use cases and valuable information may be discarded — even a Cortex-A53 with a modest frequency of 1.3 GHz can produce *1 GB/s* of trace data per core, varying based on the ETM configurations and the running application (Fig. 1). Internally, this problem is exacerbated when tracing server-grade processors such as the ThunderX-1 running at around 2.5 GHz on the Enzian heterogeneous platform [4]. A high-throughput decoder is necessary to enable rack-scale tracing for runtime verification [10], or detailed performance profiling and optimization [15].

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.

In this work, we introduce a novel ETMv4 instruction trace decoder to address the lack of throughput, allowing for more detailed real-time trace analysis with more ETM

features enabled on high-frequency CPUs. We apply a trace stream *unrolling* technique that allows the decoder to unconditionally process multiple trace elements in the same clock cycle, no matter the inter-element dependencies caused by compression. This paper makes the following contributions:

- A novel decoder design for the ETMv4 instruction trace specification that can handle multiple bytes of trace data in parallel *every cycle* at high frequencies. (Section IV)
- An implementation of the decoder on a Xilinx Ultrascale+ xczu5ev-sfvc784-2-i able to process 1 GB/s (4 bytes each cycle) of trace data with an operating frequency of 250 MHz while using around 8.4% of the device resources. (Section V-A)
- A correctness validation of the implementation against open CoreSight Decoding library (OpenCSD) [8]. (Section V-B)

II. Related Work

The initial work on hardware trace decoding for the ETM and Program Trace Macrocell (PTM) protocol was presented by Weiss & Lange [16]. Later real-time trace decoders follow a similar concept as well [10]. This approach involves collecting trace data in a buffer and employing an evaluation window to mark the boundaries of trace elements, referred to as trace *packets*, within the window. To enhance throughput, multiple overlapping windows are used (Fig. 2-a). Once the trace stream is divided into packets, a decoder can begin decompressing and extracting trace information, starting at a synchronization packet. Additional decoders can be added to increase throughput, each starting at a separate synchronization packet (Fig. 2-b). Weiss & Lange's decoder is tailored to the ETMv1 - ETMv3 or the PTM specifications and is not suitable for ETMv4. Trace packets in the ETMv4 specification may be of unbounded size and a finite evaluation window cannot guarantee that a full trace element is contained by the window. Furthermore, using numerous parallel decoders incurs higher resource utilization and requires additional trace data to be stored in buffers while also straining the trace port interface with extra synchronization packets.

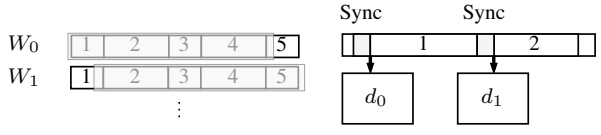

Fig. 2: Multiple overlapping evaluation windows W_i speculatively determine all trace packet boundaries within each window. Subsequently, decoding units d_i start decoding at a synchronization element in parallel.

Zeinolabedin, Partzsch & Mayr introduce a new decoder to support ETMv4 [18], mainly adding support for dealing with unbounded packet sizes and decoding the increased number of packet types in ETMv4. Instead of an evaluation window, they use a Control Core that analyzes the trace data held within a small buffer. If two single-byte trace packets appear in succession, the Control Core decodes them in parallel. However, if a trace packet is comprised of more than one byte, each payload byte is processed sequentially, making it impossible to achieve the desired high throughputs.

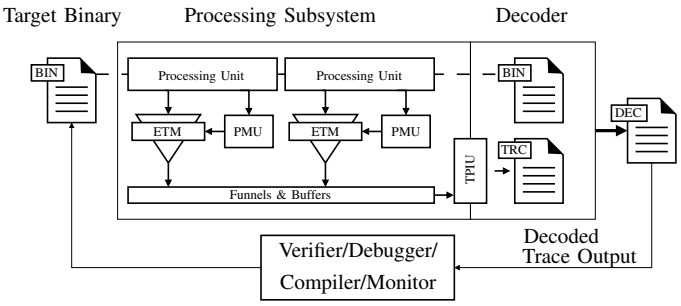

Fig. 3: CoreSight component overview and typical trace workflow: Trace data are produced at a trace source (ETM) and driven to the TPIU through the trace bus. A trace analyzer processes data from the TPIU in either hardware or software and integrates the results into a surrounding toolchain.

The decoder introduced in this work can support any trace packet 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 to support bandwidths higher than 1 GB/s, with the tradeoff of incurring additional resource utilization and buffer space.

III. CoreSight Tracing Subsystem

The CoreSight subsystem is a network of components that enable the tracing of a system without interfering 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, such as a Performance Monitoring Unit (PMU) to optionally embed hardware events like cache misses into the trace stream.

Further components — Fig. 3 illustrates a simplified block-diagram — like buffers, funnels, and replicators, are present in the subsystem and are responsible for driving trace data to a trace sink, commonly a Trace Port Interface Unit (TPIU) for real-time applications. The TPIU also interleaves source identifiers and raw trace data into frames to multiplex the interconnect. A first-layer decoder (L1 decoder) is needed to extract raw trace streams from frames before a second-layer decoder (L2 decoder) can decode a raw stream from a single source individually. This work exclusively discusses L2 decoder design since the L1 decoder is never the bottleneck and can handle multiple GB/s [18].

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 over 400 packet subtypes [18]. The purpose of the trace data is to convey the sequence of virtual addresses of the instructions executed on a processor to a trace analyzer. The ETMv4 specification optionally includes additional details like tracing accurate processor cycle counts between basic blocks, tracing context

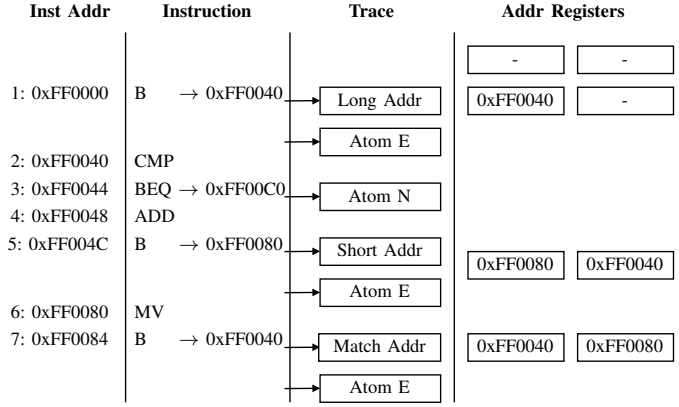

Fig. 4: A sequence of traced instructions alongside their virtual memory addresses and the corresponding ETM packets. Line 5 will only produce a `Short Address` packet with one byte of address data `0x80` and the address values held in the registers are shifted by one. Similarly, line 7 generates an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

and virtual machine identifiers to associate each instruction with its execution context, and tracing PMU events.

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 details of 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.

Together, `Atom` packets and `Address` packets encode the program flow. The `Atom` packet acts as a signpost and is emitted whenever a program flow-changing instruction (P0 element), is executed. The information carried in an `Atom` packet only signals whether a P0 instruction was taken (E) or not taken (N). The branch target address is sent over the trace stream separately in the form of an `Address` packet, illustrated in Fig. 4. Emitting full addresses over the trace stream is redundant, therefore the ETM has three internal address registers storing the last three branch target addresses. If the address of a jump target shares the most significant bits with the address already held in the first address register, the following `Address` packet will contain only the disparate least significant bits. When the jump target address matches one of the addresses in the registers, no address bits are required, but instead, a single-byte packet pointing to one of the registers is sent.

`Address` packets only appear in the stream if the decoder cannot infer the jump target. We note that the ETM specification intends for the trace data to be decoded alongside the binary, as is typical in debugging environments. With access to the binary, a decoder can resolve all direct branch targets without the `Address` packets. For example, the branch target of the instruction on line 1 in Fig. 4 is unnecessary since the address is encoded in the instruction. However, real-time trace decoders in hardware do not necessarily have 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 the surrounding toolchain. Regardless of the configuration, this typically includes a stream of resolved branch target addresses. Table I illustrates an example output stream. This stream is generated by the same sequence of instructions as in Fig. 4, with cycle-counting and event tracing enabled.[1]

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`

IV. Decoding the ETMv4 specification

Decoding the ETMv4 trace can be broken down into two main tasks: First determining packet boundaries, and second, decompressing the information encoded in these packets. To achieve a decoding throughput in the GB range at FPGA frequencies, multiple bytes of trace data must be decoded in parallel. However, this directly clashes with the existence of two types of dependencies inherent to the ETM specification: Inter-packet and intra-packet dependencies:

a) Packetization: Packets can be either variable-sized, fixed-sized, or header-only packets. For example, the `Short Address` packet p_3 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, bytes $b_1 \dots b_n$ must be decoded sequentially.

b) 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 a target address. The target address is determined by the value in the first address register and, therefore, by a previous packet. Before the `Atom` packet is decoded, the address registers of the trace source must be properly mirrored. As a result, $p_1 \dots p_n$ must be decoded sequentially.

A. Overview & Key Idea

Considering these dependencies, our approach to achieving high-throughputs is to decode at the byte level, forego full repacketization, and mirror the ETM registers after each byte. The decoder then produces an output once a complete packet is processed. To achieve high throughput, the bytewise decoding function is optimized and pipelined such that the decoding circuit can be applied multiple times every cycle. The key

[1]Cycle-counting and PMU are encoded into additional packets not shown in Fig. 4.

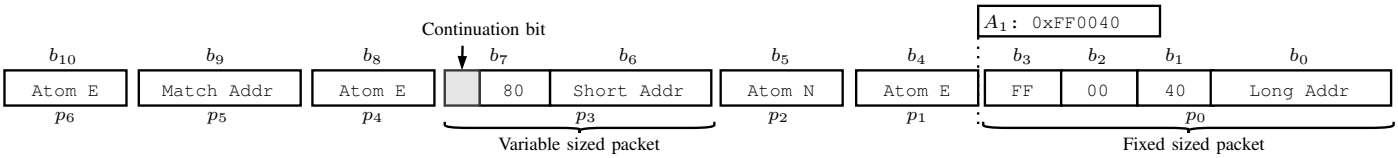

Fig. 5: Byte stream $b_0 \dots b_{10}$ representation of packets $p_0 \dots p_6$ produced in Fig. 4. The `Short Address` packet has a continuation bit to mark the last payload byte, while the `Long Address` packet has a fixed size known as soon as the header is resolved. The first address register A_1 after packet p_0, must be updated before decoding packet p_1.

idea is that the trace stream is *unrolled*, 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*, in other words, the mirrored ETM registers required for decompression. Packet boundaries can only be determined in the context of the stream state, and addresses can only be determined in the context of the trace state. 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':

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

From this follows that decoding a full trace, a stream of n bytes, is a successive chaining of the decoding function:

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

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 number of packet types in the specification. Instead, the stream of bytes from the trace stream is *unrolled*[2] with an unroll factor u, such that u bytes are input to the 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:

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

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 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 critical path is broken down into four pipeline stages: *Header preprocessing*, *stream state processing*, *action preprocessing*, and *trace state processing*, visualized in Fig. 6. 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. If 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; 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 and stream mode types. The decoded header directly determines the expected payload size and the current stream mode.

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

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
    mode ∈ {Header, PldFixedSize, PldContinuous};
    header;
    index; /*Reverse one-hot payload index*/
Process Stream State is
    switch S.mode do
        case Header
            S'.mode ←mode(h)
            S'.index ←size(h)
        case PayloadFixedSize
            S'.mode ←Header if S.index[0] is 1
            else S'.index ←S.index >> 1
        case PayloadContinuous
            S' ←Header if index[0] is 1 or b[0] is 0
            else S'.index ←S.index >> 1
```

[2]Reminiscient of loop unrolling without vectorization, hence the name.

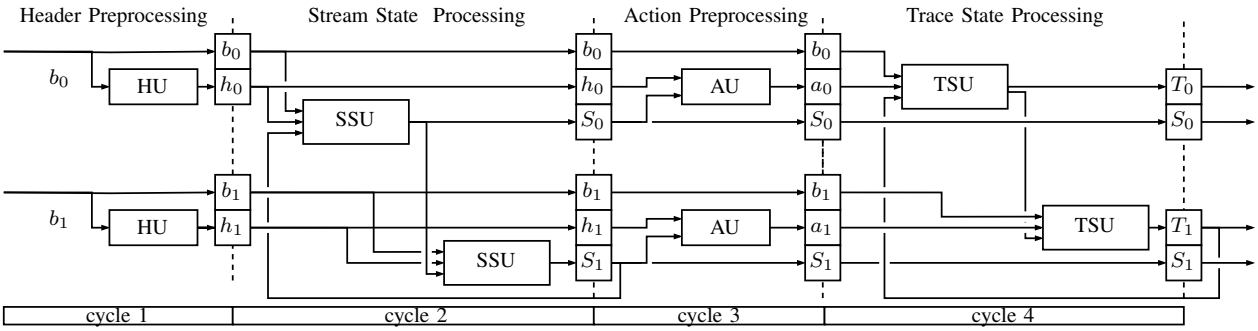

Fig. 6: Fully pipelined unrolled decoder $\mathcal{D}_2$ with HU=Header preprocessing Unit, SSU=Stream State processing Unit, AU=Action preprocessing Unit, and TSU=Trace State processing Unit. $\mathcal{D}_2$ two bytes b_0 and b_1 to produce updated state S_0, S_1, T_0 and T_1. Both the SSU and the TSU must be applied twice in one cycle to support the dependencies of the trace stream.

`shift_address_registers` code, as the address registers must be shifted to prepare for the incoming address values. All action codes can be resolved from the stream state. Using the payload index value and current header value, the action code can be generated on what bits to overwrite in the address registers with the incoming bytes, as 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 lookahead states required to handle more complex packets with optional subpackets or composite packets. Reference [15] provides additional details on handling more complex packet types.

Algorithm 2: Trace State Unit (TSU)

```
Input  : Data byte b, Stream State S at b, Resolved action
         code a
Output : Updated Trace State T'
Record Trace State is
    address_regs[2];
        ⋮
Process Trace State is
    switch a do
        case shift_address_registers
            address_regs[1] ← address_regs[0]
            address_regs[2] ← address_regs[1]
        case update_address_8_2
            address_reg[0][8 : 2] ← b[6 : 0]
        case update_address_15_9
                    ⋮
```

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 [8], a library to decode collected CoreSight traces in software.

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 [18], which, to the best of our knowledge, is the only other reported implementation of an ETMv4 decoder. We emphasize that the decoders are implemented on different, albeit similar, FPGA devices. Experiments on the same device are not possible, as their decoder is not public, and all metrics are taken directly from reference [18]. The direct comparison is not exact, as the maximum operating frequency and device utilization depend on device characteristics. Nevertheless, we believe a comparison is still warranted and shows a significant increase in performance due to unrolling and guaranteeing that multiple bytes are processed every cycle.

The throughput of the decoders is determined by the bytes per cycle it can process and the 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 in the worst case. 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.

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. OpenCSD is used as an oracle to match the outputs of the implemented decoder. We chose to compare our outputs to openCSD, as this is the only decoder available for public use and is actively maintained by ARM developers. 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.

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 [18]. The maximum operating frequency is determined by Vivado timing analysis, and device utilization reports are exported from Vivado. Throughput is experimentally validated (Fig. 7).

	Performance				Resource Utilization			
	$min(\frac{\text{bytes}}{\text{cycle}})$	$max(\frac{\text{bytes}}{\text{cycle}})$	$max(f)$	Throughput	LUT	Reg	BRAM	CLB/Slice
Baseline [18]	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%)

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.

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. The presented decoder has been successfully validated in both simulations, and verification runs on the implemented design.

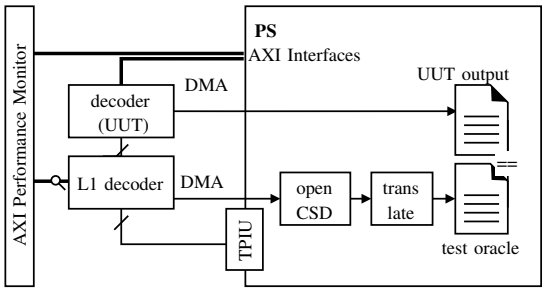

Fig. 7: Setup of throughput measurement and validation. Raw trace data and decoded trace data are collected simultaneously. OpenCSD is run on the raw trace data and compared to the decoder output. Performance and trace bandwidth metrics are measured by an AXI performance monitoring unit.

VI. Conclusion

We have presented a high-throughput decoder for the ETMv4 instruction trace specification. Our design ensures that multiple bytes are processed every cycle regardless of packet boundaries and despite the inter-byte dependencies, achieved by unrolling the trace stream and employing an optimized bytewise decoding function. We have implemented the decoder on a Xilinx xczu5ev-sfvc784-2-i, where it 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@1.3 GHz with cycle-counting, branch-broadcasting, and event tracing simultaneously, making it ideal for runtime verification, feedback-directed optimization, monitoring, and other real-time applications. In addition, we have outlined a framework for decoding any CoreSight-compliant stream. For future work, we intend to support decoding the trace alongside a compressed form of the trace target binary to reconstruct every executed instruction, also eliminating the reliance on branch-broadcasting. Furthermore, our decoder can be 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 trace.

References

[1] ARM Ltd. *Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETM4.6 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] David Cock et al. "Enzian: an open, general, CPU/FPGA platform for systems software research". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2022, pp. 434–451.

[5] *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.

[6] 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.

[7] Hannes Kallwies et al. "TeSSLa–an ecosystem for runtime verification". In: *International Conference on Runtime Verification*. Springer. 2022, pp. 314–324.

[8] Linaro. *Linaro/opencsd: Coresight Trace Stream decoder developed openly*. URL: https://github.com/Linaro/OpenCSD.

[9] Aleksandar Prokopec et al. "Renaissance: Benchmarking suite for parallel applications on the jvm". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 31–47.

[10] Pirmin Schmid. "Runtime verification with tessla on enzian". MA thesis. ETH Zurich, 2019.

[11] Haoqi Shan et al. "CROWBAR: Natively Fuzzing Trusted Applications Using ARM CoreSight". In: *Journal of Hardware and Systems Security* (2023), pp. 1–11.

[12] 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.

[13] 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.

[14] Conal Watterson and Donal Heffernan. "Runtime verification and monitoring of embedded systems". In: *IET software* 1.5 (2007), pp. 172–179.

[15] Matthew Edwin Weingarten. "Hardware Accelerated Trace Analysis for Compiler Optimizations". MA thesis. ETH Zurich, 2023.

[16] Alexander Weiss and Alexander Lange. *Trace-data processing and profiling device*. US Patent 9,286,186. Mar. 2016.

[17] Seyed Mohammad Ali Zeinolabedin, PartzschJohannes, 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.

[18] Seyed Mohammad Ali Zeinolabedin, PartzschJohannes, and Christian Mayr. "Real-time hardware implementation of arm coresight trace decoder". In: *IEEE Design & Test* 38.1 (2020), pp. 69–77.