CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Implementation of a Trace Encoder for NOEL-V RISC-V Processors

Master's thesis in Embedded Electronic System Design

Max Hessman
Oskar Sternvik

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Implementation of a Trace Encoder for NOEL-V RISC-V Processors

Max Hessman
Oskar Sternvik

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Implementation of a Trace Encoder for NOEL-V RISC-V Processors

Max Hessman
Oskar Sternvik

Supervisor: Per Larsson-Edefors, Department of Computer Science and Engineering
Company advisor: Martin Rönnbäck, Frontgrade Gaisler
Examiner: Lena Peterson, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Implementation of a Trace Encoder for NOEL-V RISC-V Processors

Max Hessman
Oskar Sternvik
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis proposes a solution to the problem of overwhelming trace data production compared to available bandwidth in systems-on-chip in space applications. The proposed solution revolves around compressing the trace data through instruction branch trace, which assumes sequential execution and reduces the trace to instructions and events that result in a non-sequential instruction flow. Given full compilation of a program at the receiving location, the instruction flow of the traced program can be fully reconstructed.

The thesis compares two known trace standards (Nexus and E-trace), and concludes that E-trace is the preferred standard for the stated problem, due to its higher potential compression rate. The thesis then presents an implementation of the trace data encoder in VHDL. The implemented encoder is fed by instruction trace from Spike RISC-V simulator running a test suite, and the correct encoding of the trace is proven through correct reconstruction of the instruction flow by a third-party decoder. The resulting compression rate in a worst case scenario indicates that a 12 core system could be traced simultaneously in a 100 MHz dual-issue system, given a high-speed link bandwidth of 6.25 Gbps. The encoder is then successfully integrated into Frontgrade Gaisler RISC-V implementation NOEL-V. Through synthesization, the thesis shows that the encoder does not introduce a new lower limit on the attainable clock frequency of the processor. Furthermore, the synthesis shows that the encoder falls within a reasonable boundary of the total available hardware in the chosen FPGA. The thesis concludes that the proposed solution to the trace data problem in space applications is valid and realizable.

Keywords: RISC-V Instruction-trace Trace Encoder E-trace Nexus N-trace Compression NOEL-V FPGA

Acknowledgements

We'd like to thank our academic supervisor Per Larsson-Edefors as well as our company supervisors Martin Rönnbäck, and Nils Johan Wessman for their great help with guiding us in this project. We'd also like to thank our examiner Lena Peterson for valuable feedback on our report. Additionally, we'd like to thank all the other kind persons at Frontgrade Gaisler, whos tips have been immensely helpful.

Max Hessman, Gothenburg, August 2024
Oskar Sternvik, Gothenburg, August 2024

Contents

Acronyms

$\overline{MSEI}$ Message Start/End In. 16
$\overline{MSEO}$ Message Start/End Out. 16, 17
.CSV comma separated values. 23
.ELF executable and linkable format. 23

AHB Advanced High-performance Bus. 27
AMBA Advanced Microcontroller Bus Architecture. ix, 19
APB Advanced Peripheral Bus. ix, 19, 27, 28
ASIC application specific integrated circuit. 36, 37
ATB arm-trace-bus. 10, 12, 18

bpi bits per instruction. 13, 23, 31, 32, 34, 36–38, 40, I, II
BTM Branch Trace Messaging. 16
BW bandwidth. 1, 2, 8, 12, 34

CPU central processing unit. 26
CSR Control Status Register. ix, 5–7, 10

ECALL Environment Call. 7
ESA European Space Agency. 20

FE instruction-fetch. 5
FPGA field programmable gate array. v, 1, 2, 24, 34, 37, 38

Gbps Gigabits per second. v, 40
GRHSSL Gaisler Research high-speed serial link. 38

hart hardware thread. ix, 5–7, 9–12, 17, 18, 26, 28, 37, 38
HTM Branch History Messaging. 16

IEEE Institute of Electrical and Electronics Engineers. 14
IP intellectual property. 24, 28
ISA instruction set architecture. 4–6, 8, 17

JAL jump and link. 7
JALR jump and link register. 7

LSB least significant bit. 17, 28
LUT look-up table. 37

M Machine Mode. 5, 11
MDO Multiple Data Operations. 17
MSB most significant bit. 28

NASA National Aeronautics and Space Administration. 39

PC program counter. 6–10, 28
PKG Package. 34, I
PL Payload. 31, 33, 34, I

ra return address. 7
RISC reduced instruction set architecture. 4
RISC-V reduced instruction set computer V. v, ix, 2–7, 9, 10, 13, 17, 18, 20, 22, 23, 26–28, 31

S Supervisor Mode. 5, 11
SIE Supervisor Interrupt Enable. 7
SIP Supervisor Interrupt Pending. 7

TCODE Transfer Code. 15, 16, 19

U User Mode. 5, 11

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language. v, ix, 6, 20, 22

WNS worst negative slack. 34

1

Introduction

Complex system-on-chip designs with processor cores can be analyzed by obtaining trace data of events such as instructions executed, results of operations and bus accesses. For systems with multiple processors running at high frequencies, the amount of data generated becomes very high and on-chip buffers fill up rapidly. The transfer link's bit rate limitations poses a problem transferring the trace data in real time as the amount of trace data generated is much larger than the available data transfer bandwidth (BW) and speed. The problem intensifies in extreme-long-range applications such as space, where the generated trace data is to be analyzed back on Earth, further limiting the achievable transfer rate. One solution is to minimize the amount of data to be transferred by compressing it. One way to achieve this is to remove all executed instructions bar the branch instructions from the trace data [1][2]. If the instruction sets and programs are available at the receiving location, matching the performed branch executions to the compressed trace data can enable reconstruction of all instructions executed.

There are several standards known for compressing trace data [1][2][3][4]. This thesis compares two of them (Nexus and E-trace) in terms of capabilities, compression rates, and available third-party tools in order to determine which standard best fits the system and if it can fully or partially solve the posed problem. The aim of this thesis is to implement the chosen standard on a field programmable gate array (FPGA) in order to practically prove the theoretical solution and to find and explain any discrepancies between theory and practice.

1.1 Related Work

The E-trace standard traces the instructions performed by the processor by reporting what branches are taken and if, and where, an interrupt has occurred. If an interrupt occurs, the final instruction is included in the trace in order to allow debugging [1]. Nexus is an open industry standard that includes several tools for enabling debugging of a processor [2]. One of these is the program trace which, similarly to E-trace, reports taken branches and at what instructions an interrupt has occurred. Nexus program trace also reports every 256th instruction issued for synchronization reasons. Nexus and E-trace also feature other debugging tools, one of which can report what data has been written into specific registers and allows a user to see the content of memory locations [1][2].

For trace encoders in general, there are several published and patented implementations. However, the research papers published in the field of reduced instruction set computer V (RISC-V) trace encoder implementations, are few. In [4] from 2022, an encoder implementation of E-trace standard is proposed and analyzed in terms of area and power. However, no comparison with other available standards is performed and the paper explicitly states that there is a lack of published literature in the subject.

Nexus has previously been the industry standard, but the forum dissolved in 2020, and their latest standard was published in 2012 [5]. Although the standard is fairly old, it is still being used and has even been adopted for RISC-V. The E-trace standard on the other hand is designed specifically for RISC-V processors [6], and is still being developed.

1.2 Purpose and Goal

As today's long range space applications of computers provide limited insight into the internal workings of actual program execution, localizing sources of error can be challenging. This is not optimal for the efficient spending of resources in space missions. The purpose of this project is to facilitate better debug tools for RISC-V processors in space applications. The reason for this is to enable greater detection of bugs and faults in programs running on systems fitting this description, and ultimately enable greater knowledge and efficiency in space computer implementations.

The main goal of the thesis is to implement one of the available encoder standards (E-trace, Nexus) on an FPGA, and to find a suitable third-party tool for decoding the compressed instructions. The project aims to prove that the proposed solution is viable in order to solve the stated problem. The choice of standard is to be performed by a comparison of capabilities and available third-party tools, where capabilities include compression rate and functionalities. The choice of standard poses a challenge as the functionalities and compression rate have to be weighted against availability of third-party tools, complexity of the implementation, attainable data transfer rate, industry trends, and the available time for the project.

A comparison should be made between the generated trace data of the processor and the theoretical, and practical, compression rate of the encoder in order to establish to what extent the processor can be traced. Can the entire processor be traced? If not, how many cores can be traced? This comparison also considers the BW of the following high speed link. Discrepancies between theoretical and practical compression rates can arise due to the implementation. An additional aim is to achieve the highest possible theoretical compression rate and if this is not met, explain why.

1.3 Limitations

This thesis does not focus on constructing an interface between the processor and the encoder, as one is provided by Frontgrade Gaisler. If a properly functioning interface is not provided, the verification of the encoder is limited to simulation results.

The thesis does not include any construction of a decoder (or debug tool), but it is limited to compiling a market survey of available decoders that comply with the two different standards Nexus and E-trace.

The project does not include any redundancy or other bit-flip protective measures that may be needed for ensuring the integrity of the information handled by the encoder. Any such measures are deemed beyond the scope of the project.

Any optional functionalities available to the chosen standard are only implemented if there exists excess time.

The construction of the encoder should comply with the chosen standard, so that any later additions in functionalities can easily be made.

1.4 Thesis Outline

Chapter 2 presents a technical background for the project. It provides necessary context for understanding the thesis, including information about the RISC-V standard and the Frontgrade Gaisler implementation of the standard called NOEL-V. It continues with an in-depth account of the E-trace and Nexus trace standards, including their distinctive individual designs, features, and functions. The chapter also provides basic explanations about other relevant subjects and concepts.

Chapter 3 is focused on describing the methods to be used for the different parts of the project such as the literature study, market survey, design, implementation, and analysis.

Chapter 4 gives a description of the design and motivations for the different design choices made. It also provides a motivation to what choice of standard has been taken.

Chapter 5 presents the results of the project, including proof of functionality and metrics of the achieved performance of the system.

In chapter 6, the results are discussed and related back to the stated goals of the project, as well as the theoretical context in the technical background. Sources of error are discussed, and future work are suggested. The chapter also includes a discussion on societal, ethical and ecological aspects of the project.

Finally, chapter 7 presents the reached conclusions based on the achieved results and discussion thereof.

2
Technical Background

In this chapter, a number of concepts, implementations, and standards are presented. These form a necessary technical background to understand the project.

2.1 RISC-V

RISC-V is an instruction set architecture (ISA) that is based on reduced instruction set architecture (RISC), which aims at reducing the amount of instructions stored in a microprocessor, or chip [7]. The V represents the roman numeral 5, and is meant as the fifth major generation of RISC, as well as to symbolize the support for variations and vectors [8]. It was developed by UC Berkeley, in order to enhance education aimed at hardware implementation by making an open-source ISA that can be used on all kinds of computers [9], since there was no available ISA for this purpose at the time [10].

RISC-V instructions are always executed sequentially between branches [1], meaning that the full trace of a RISC-V processor can be decoded from the branch instructions, given that the decoder has access to the code or a file containing the binary.

The RISC-V ISA has three base integer variants, RV32I, RV64I as well as RV128I which is aimed at future use for larger address spaces [8]. There are also a number of extensions that can be used as shown in Table 2.1.

Table 2.1: RISC-V extensions. Table reconstructed from RISC-V standard [8].

Extension	Description
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic
F	Standard Extension for Single-Precision Floating-Point Instructions
D	Standard Extension for Double-Precision Floating-Point
Q	Standard Extension for Quad-Precision Floating-Point
L	Standard Extension for Decimal Floating-Point
C	Standard Extension for Compressed Instructions
P	Standard Extension for Packed-SIMD Instructions
V	Standard Extension for Vector Operations
N	Standard Extension for User-Level Interrupts

Instructions are used for different purposes: one might need access to two registers, while another only needs access to one. Some might require a larger immediate

field than others. All RISC-V standard instruction formats are 32 bits long and are presented below in Table 2.2.

Table 2.2: RISC-V Formats. Table reconstructed from RISC-V standard [8].

Format	Description
R	Integer register-register operations
I	Integer register-immediate operations
S	Store instructions
U	Standard Extension for Double-Precision Floating-Point
B	Conditional branch instructions
J	Unconditional jump instructions

2.1.1 Hardware threads (harts)

A RISC-V core is a block able to execute RISC-V instructions, and is defined by its instruction-fetch (FE) unit. A RISC-V core may have several harts, which are defined as execution pipelines with separate sets of registers (in order to limit the number of context switches) [8]. Multiple harts allow multithreading inside a single core, and therefore increases the level of parallelism of the processor as it may have several cores with several individual harts, which may all be executing instructions concurrently.

2.1.2 Privilege levels

A RISC-V hart is always executing instructions with a certain privilege level. As specified by the RISC-V ISA manual [11], currently there are three defined levels: User Mode (U), Supervisor Mode (S) and Machine Mode (M). The software stack has different components and the privilege levels are used to give protection between them. Should an operation be attempted for a privilege level that is not allowed, an exception should be generated, which normally leads to a trap.

The highest privilege level is the M-level and this is the only level that is mandatory for RISC-V devices and should have access to the entire machine. Following the M-level, the second and third highest levels are U-level and S-level, respectively. M-level in RISC-V facilitates the management of secure execution environments. U-level and S-level are designated for standard application and operating system purposes, respectively [11].

2.1.3 Control Status Registers (CSRs)

The CSRs are designated for different privilege levels, and can only accessed from the same level, or higher levels. CSRs are used to store necessary information for a context switch e.g. when the instruction flow has to be halted in order to handle an interrupt or an exception. There is a 12 bit immediate field for these instructions, meaning that there are $2^{12} = 4096$ CSRs for each hart. CSR instructions involve

the instructions which are used to gain information about the current hart, or control/change the hart, and store addresses and information relevant to the present context in order to enable a return to the halted state after the handling of an interrupt or exception. CSRs additionally handle the base address of execution handler upon an interrupt or exception [8][11]. The trace function *data trace*, described later in Section 2.3, gets access to the contents of the CSRs [1].

2.1.4 Spike

Spike is an open-source ISA simulator for RISC-V that can model the functionality of one or more RISC-V harts [12]. Spike is provided by the international RISC-V community, and is widely used to simulate RISC-V behavior.

2.1.5 NOEL-V

NOEL-V is a synthesizable Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) model of a radiation-hardened 32- and 64-bit RISC-V processor implementation by Frontgrade Gaisler [13]. The modular nature of RISC-V has enabled the NOEL-V to be developed to meet the increasing demands on high-speed parallel computing in space posed by image processing, and artificial-intelligence tasks. It additionally supports the use of hypervisors, which increases the flexibility of the system. The current implementation hosts eight processing cores, and has the ability to correct radiation-induced bit-flips in its cache memories. The system has the ability to automatically detect and correct up to four bit adjacent errors, and is equipped with a scrubber mechanism which prevents accumulation of faults over greater periods of time [14]. The processor is dual-issue, meaning that there are two pipelines in each core in order to enable execution of two instructions per cycle. However, all instructions cannot be executed by both pipelines and all kinds of branches are executed in the same pipeline, limiting the branch instruction trace to a maximum of one per cycle, per core.

2.2 Branches

In the RISC-V specification, the term *branch* refers to an instruction that conditionally changes the sequence of executed instructions [8]. However, in instruction trace literature, it is often used to encompass all types of instructions that change the sequence of the executed instructions [1][15]. This looser definition of branches has led to terms that may cause confusion. Two examples of this are the *unconditional branch*, which is an instruction that does not exist in RISC-V, and is performed as a jump instruction, and the *conditional jump*, which does not either exist in RISC-V and is performed as a branch. In this report, the different types of instructions that will be generalized to "branches" are:

- **Branch instructions:** Branch instructions are conditional, meaning that they require a condition to be fulfilled in order to change the flow of execution. When executing the branch instruction, it sends the program counter

(PC) to a destination address either hardcoded into the instruction, or stored in a register. Common conditions are that the content of a register should be less than, greater than, or equal to the contents to another register. When compiling a program, For-loops can be implemented through branch instructions.

- **Jump instructions:** Jump instructions are unconditional, meaning that they do not require a condition to be fulfilled in order to change the flow of execution. Similar to branch instructions, jump instructions send the PC to a destination address. The flora of jump-instructions are differentiated by whether the destination address is stored in the opcode or in a register, and whether or not the return address should be stored, and if so, in what register.

  The flow of execution for jumps can be changed in two different ways: direct jumps and indirect jumps. Direct jumps change the PC by a constant value, while indirect jumps change the PC by a computed value, such as the contents of a register. Some jumps can be optimized by the compiler, such that they get their target address from a constant embedded in the jump instructions opcode. These are called inferable jumps. If the jump is not optimized, then it is called an uninferable jump.

- **Calls:** Calls are instructions that jump to a function or a subroutine.

- **Returns:** Returns are instructions that return from a function or subroutine. In RISC-V there is no designated return instruction, and a return to a specific address is instead implemented through the use of jump instructions such as jump and link (JAL) or jump and link register (JALR). When returning, it is important to return to the point where the next instruction following the call to the function is. This is done with the return address (ra) register, which is used to hold the address of the next instruction.

- **Interrupts:** Interrupts do not typically happen intentionally as a result of instructions. Instead, they usually occur asynchronously.

  Interrupts are triggered by the setting of the interrupt-enable registers, Supervisor Interrupt Enable (SIE) and Supervisor Interrupt Pending (SIP). Whenever a logical 'one' (if active high) is registered in one of the registers, information of the current PC and cause of the interrupt is saved in the CSRs, and the PC is jumped to the interrupt handler [11].

- **Exceptions:** These happen when a RISC-V hart experiences an unexpected condition during runtime associated with an instruction.

  Exceptions are handled similar to interrupts, but are raised through the Environment Call (ECALL) instruction instead of the setting of registers [8]. An exception can typically be linked back to a specific instruction address.

- **Trap:** In the context of branch tracing, not all interrupts or exceptions change the PC. Only the interrupts and exceptions that cause a transfer of control to a trap handler will be noted by the trace [8].

2.3 Instruction/program trace

One technique to debug a processor, or a program running on a processor, is to go through all the instructions that have been executed in order to localize where an error has occurred, a technique which can also be applied in order to validate code coverage [16]. Most modern ISAs have the ability to efficiently report all instructions, this is called instruction trace or program trace. But as a modern processor may execute billions of instructions per second, this generates vast amounts of data which may lead to the instruction trace quickly filling up the internal memory or buffer assigned for the generated data [17]. This problem has produced a need for compression of the stored data, in order to make better use of the storing capacity of the system. Similarly, solutions where the generated data is to be transmitted to an external machine or memory with bigger capacity may suffer from limited BWs of the transmission links, which again motivates the need for compression of the reported instructions. One way to achieve this is called branch instruction tracing, or simply branch tracing. Branch tracing presumes sequentially executed instructions and compresses the produced data by first reporting the first instruction, and then removing all instructions that are not some kind of branch (branch, jump, call, return, interrupt or exception). The branch trace must include information on whether the branch is taken or not, and if the destination of the branch is not hardcoded into the instruction opcode (uninferable), the destination of the branch. The module performing this compression is called the encoder [18].

If the compressed information is needed, it first has to be decompressed by a decoder module. The decoding of the information presumes that the decoder has access to the uncompressed instructions of the program if it would have run without any errors. The decoder then uses the first instruction in order to synchronize with the compressed data, and continues on to the next reported branch. As the instructions have been performed sequentially, the decoder can assume that all instructions between the first instruction and the first reported branch have been executed. If the branch is reported taken, the decoder knows where the PC has jumped, either from the hardcoded branch address, or the reported destination for an uninferable branch. This way, the decoder can adjust its own shadow PC to align with the original and continue to assume sequential execution until the next reported branch. If a branch is reported as non-taken, the decoder can continue to assume sequential operation without adjusting its shadow PC. This way, the compressed information can be decompressed in order to perfectly reconstruct the original execution flow, and an error can be located by identifying deviation from the original instruction sequence [18]. For example, if a program was supposed to execute a loop a certain number of times and then continue, a smaller or larger number of taken branches than expected may indicate errors in the construction of the loop, or faults in the registers containing the loop condition variables.

Whether the unpredicted behavior from the example above was due to a faulty register or a poorly constructed loop, it may be useful to see the data written to the registers. This is called data tracing, and poses a more complicated compression problem than instruction trace does, as a fault may have occurred in between

branches, and if it didn't, the reconstructed instruction trace would perfectly describe what happens in the registers [1][15]. The tracing of data and instructions is generally implemented as functions that can be turned on and off separately, and in that case, data tracing can be turned on just when running a part of a program where a problem has been identified to be occurring. For compression of data trace, one common technique is to allow for the user to inform the encoder to either only send the data written to a register, or the address of the register data is being written to. This allows for some compression of the data being sent, but assumes that the information that is not being sent is functioning correctly [1][15]. Data tracing can be implemented as an additional function of an encoder, but is not necessary for complying with the standards discussed in this report.

2.3.1 E-Trace

Efficient Trace, or E-trace, is a debug standard developed by the RISC-V Debug and Trace Working Group. The work was led by engineers from Siemens and the trace algorithm was given as a donation to the RISC-V community. E-trace is a non-intrusive debug standard, taking in instructions from the processor, categorizing them in a hart-to-encoder interface and sending information about the type of instruction and other necessary information to the encoder for packaging and transmission to the destined recipient. The recipient can either be a memory, a transmission link to an external system, or both simultaneously. The core of the trace algorithm is the search for PC discontinuities.

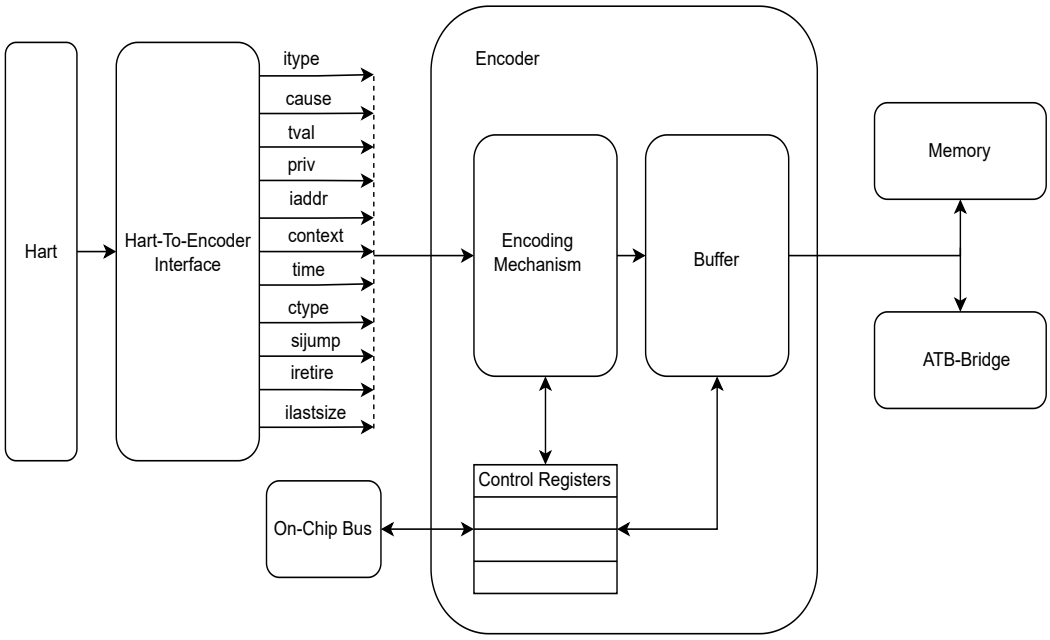

Figure 2.1: A block schematic of a single E-trace encoder implementation. Figure reconstructed from RISC-V Control Interface standard [19], but extended to include the hart-to-encoder interface signals.

The first instruction of a program cycle is always transmitted, and as RISC-V executes instructions sequentially, a PC delta greater than one indicates as branch of some kind. Whenever a branch is detected (taken or non-taken), the encoder interface transmits relevant information to the encoder for packaging. The relevant information differ for different kinds of PC discontinuities. The PC discontinuities can occur due to branches, jumps, returns, interrupts and exceptions. This way the instruction flow is continuously filtered from unwanted information, and the relevant information is packaged and sent by the encoder. A block schematic of a single E-trace encoder implementation can be seen in Fig. 2.1. Note how the encoder outputs to both or either of memory or arm-trace-bus (ATB) bridge. At least one of these are mandatory to fulfill the RISC-V Trace Control Interface standard, but none of them are required in order to fulfill the E-trace standard, and can therefore be exchanged for vendor specific sink solutions [19].

The list below shows the output signals from the hart-to-encoder interface:

- **itype:** Mandatory signal that signals the termination type of the instruction block according to the encoding shown in Table 2.3. Must be replicated for each hart-to-encoder interface.

Table 2.3: Encoding for itype. Table reconstructed from E-trace standard [1].

Value	Meaning
0	Final instruction in the block is none of the other named itype codes.
1	Exception following the final retired instruction in the block.
2	Interrupt following the final retired instruction in the block.
3	Exception or interrupt return.
4	Nontaken branch
5	Taken branch
6	Uninferable jump if itype_width_p is 3, reserved otherwise
7	Reserved
8	Uninferable call
9	Inferrable call
10	Uninferable jump
11	Inferrable jump
12	Co-routine swap
13	Return
14	Other uninferable jump
15	Other inferable jump

- **cause:** Mandatory signal that signals the cause of an exception or interrupt. The signal is ignored unless itype equals one or two.
- **tval:** Mandatory signal that communicates the associated trap value, e.g., the faulting virtual address for address exceptions, as would be written to the CSR. Ignored unless itype equals one.
- **priv:** Mandatory signal that signals the privilege of all instructions retired in this cycle. Encoding shown in Table 2.4.

Table 2.4: Encoding for priv. Table reconstructed from E-trace standard [1].

Value	Meaning
0	U
1	S/HS
2	Reserved
3	M
4	D (debug mode, optional)
5	VU (optional)
6	VS (optional)
7	Reserved (optional)

- **iaddr:** Mandatory signal that communicates the address of the first instruction retired in this block. Invalid if iretire equals zero unless itype equals one, in which case it reports the address of the instruction that caused the exception. Must be replicated for each hart-to-encoder interface.
- **context:** Optional signal that reports context for all instructions retired on this cycle.
- **time:** Optional signal that reports the time generated by the core.
- **ctype:** Optional signal that reports behavior for context. Encoding shown in Table 2.5.

Table 2.5: Encoding for ctype. Table reconstructed from E-trace standard [1].

Value	Meaning
0	Unreported, don't report context.
1	Report context imprecisely, precise point can be deducted from source code.
2	Report context precisely.
3	Report context as an asynchronous discontinuity.

- **sijump:** Optional signal that is used for showing if an uninferable discontinuity is sequentially inferable, and thus can be inferred by the sum of two instructions. If implemented, it must be replicated for each hart-to-encoder interface.
- **iretire:** Mandatory signal that shows the number of halfwords represented by instructions retired in this block. Must be replicated for each hart-to-encoder interface.
- **ilastsize:** Mandatory signal that shows the size of the last retired instruction through: instruction size = $2^{ilastsize}$ halfwords. Must be replicated for each hart-to-encoder interface.

The encoder itself is controlled by a set of signals that are stored in a set of 32-bit registers referred to as trace component control registers. In these registers, each bit corresponds to the activation, or deactivation, of a certain function in the encoder. It is possible to implement 19 trace component control registers, but only two are required in order to fulfill the basic requirements for the E-trace standard. The

possible extra registers are used for detailed control of optional implementations such as filters and their internal comparators [19].

If several hart-to-encoder interfaces and encoders are implemented (e.g. four cores having an hart-to-encoder interface and an encoder each), a trace funnel has to be implemented in order to administrate the shipment of information to the sink [19]. It may have to include a buffer in order to ensure not losing any trace for a multi-hart processor as the BW of the sink link and/or the ATB-bridge may be limited.

The encoder packages the information in different formats depending on what is being transmitted. There are four formats with several different sub-formats [1].

- **Format 3:**
  - *Subformat 0:* This packet is used for transmitting the first instruction in a cycle and contains all information necessary for the decoder to fully reconstruct the instruction. It can also be used for forced re-synchronization.
  - *Subformat 1:* This packet is used for transmitting traps and is sent following an exception or and interrupt, it contains all information necessary to reconstruct the instruction flow, such as cause and the address of the trap handler or the exception itself.
  - *Subformat 2:* This packet is used for reporting a context change, and contains only the context and optionally a timestamp.
  - *Subformat 3:* This packet is used for support information such as when trace is enabled or disabled, the operating mode is changed, or to signal that one ore more trace packets cannot be transmitted.
- **Format 2:** This packet contains only the instruction address and is used when reporting the address of an instruction is mandatory, while there are no unsent branches.
- **Format 1:** This packet contains branch information and is used whenever a branch is to be reported, or when the address of an instruction needs to be reported and there has been atleast one branch since the last packet.
- **Format 0:** This packet is assigned for optional extensions and functionalities of the encoder such as branch prediction and implicit jumps. It can be used to report the number of correctly predicted branches, or jump target cache index if implemented.

The features of E-trace are listed below:

Filtering: The standard includes an optional filter implementation, which may be used for additional filtering of the collected branch instructions. Each filter consists of three comparators which can be used to single out certain kinds of branches.

Timestamping: The standard includes a timestamping option for sending timestamps with each package sent, enabling further accuracy for debugging.

Delta address mode: The delta address mode is a mandatory feature of the E-trace standard, where the destination of a taken branch is reported as a delta from the branch address.

Full adress mode: An optional operational mode where the full addresses of branches taken are always reported, in contrast of delta address mode. This is a useful feature for debugging.

Implicit exception mode: An optional implementation where the decoder is assumed to know the destination addresses of exceptions, which can then be omitted from the trace, further reducing the package size. Requires functionality in decoder for full reconstruction.

Sequentially inferable jump mode: An optional function where the destination of some uninferable branches can be inferred through the combination of two instructions, rendering the branches inferable. This increases the compression rate as the jump address can be omitted from the trace. Requires functionality in decoder for full reconstruction.

Implicit return mode: A mode for an optional implementation of a call stack, so that the return addresses can be read from the call stack. This enables further compression, as the return addresses can be omitted from the trace. This function requires functionality in decoder for full reconstruction. In a simulated benchmark suite performed by one of the authors of E-trace, Iain Robertson, this function is clamied to have a 36% increase in bits per instruction (bpi) compression on average, making it the most significant contributor in compression rate amongst the different optional functions. Furthermore, Robertson claims that a call stack with only room for one address, achieves an average increased bpi compression rate of more than 30% compared to the minimal implementation [20].

Branch prediction mode: The standard includes the optional implementation of a branch predictor, in order to further increase the compression rate, as loops may then be predicted and therefore render the individual branches (iterations) of the loop unnecessary to send. Instead, the encoder can then use packages of format 0, subformat 0, to transmit the number of correctly predicted branches. This requires an identical copy of the branch-predictor in the decoder for full reconstruction. In the aforementioned simulated benchmark suite performed by Iain Robertson, this function is claimed to have a 6% increase in bpi compression rate on average, making it the second most significant contributor in compression rate amongst the optional functions [20].

Jump target cache mode: A mode for an optional implementation of a jump target cache, so that the address of a jump can be exchanged for an cache entry index, enabling further compression of the trace. Requires functionality in decoder for full reconstruction.

In the aforementioned simulated benchmark suite performed by Iain Robertson, all compression rate improving functions together is claimed to have increased the bpi compression rate by roughly 40% compared to the minimal implementation [20][21]. It should be noted that this is an unsupported claim, as the details of these tests remain unpublished in any available article or report, making it impossible to review. The claims were made at the RISC-V Summit North America 2023. The claims are admitted into this report solely because it is the only found public claim made by anyone regarding different E-trace optional function compression rates, and could

therefore give a hint of what optional functions should be prioritized in regards of maximizing compression rates.

2.3.2 Nexus

Nexus 5001 is an open standard that was administered by the Nexus 5001 Forum, which was a part of the Institute of Electrical and Electronics Engineers (IEEE) Industry Standards and Technology Organization, until it disbanded in 2020 [5]. It was published in three versions in 1999, 2003, and 2012, and has during its 20 year existence been a widely used standard for embedded debugging, trace handling and performance monitoring [5].

Nexus is a standard with many areas of use, and can be applied with several different intentions. The standard defines the I/Os and the package formats for a number of functionalities, some of which are mandatory and some of which are optional to implement in order to fulfill the standard [2]. Furthermore, the standard defines four classes that each has a clearly defined set of mandatory and optional features. These functionalities and classes are listed below in Table 2.6.

Table 2.6: The Nexus 5001 standard features and classes, and information on what features are mandatory or optional for what classes. Table reconstructed from the Nexus 5001 standard [15].

Feature	**Class 1**	**Class 2**	**Class 3**	**Class 4**
Read & write registers while in debug mode	Yes	Yes	Yes	Yes
Read & write memory while in debug mode	Yes	Yes	Yes	Yes
Enter debug mode from reset	Yes	Yes	Yes	Yes
Enter debug mode from user mode	Yes	Yes	Yes	Yes
Exit debug mode from user mode	Yes	Yes	Yes	Yes
Single-step instruction; reenter debug mode	Yes	Yes	Yes	Yes
Stop on breakpoint; enter debug mode	Yes	Yes	Yes	Yes
Set breakpoint or watchpoint	Yes	Yes	Yes	Yes
Device identification	Yes	Yes	Yes	Yes
Notify of watchpoint match	Yes	Yes	Yes	Yes
Monitor process ownership in real time (ownership trace)	No	Yes	Yes	Yes
Monitor program flow in real time (program trace)	No	Yes	Yes	Yes
Monitor data writes in real time (data trace)	No	No	Yes	Yes
Monitor data reads in real time	No	No	Optional	Optional
Read & write memory in real time	No	No	Yes	Yes
Execute program through Nexus port (memory substitution)	No	No	No	Yes
Begin trace on watchpoint	No	No	No	Yes
Begin memory substitution on watchpoint	No	No	No	Optional
Low-speed I/O port replacement	No	Optional	Optional	Optional
High-speed I/O port sharing	No	Optional	Optional	Optional
Transmit data-acquisition	No	No	Optional	Optional

The features listed in Table 2.6 can be used for many different things, as the Nexus standard claims the right to control the debug module even for the most basic classes. The higher classes can be made even more intrusive as they can be implemented with features such as I/O port replacement, memory substitution, which provide more,

and stronger, tools. As for program trace, data trace, and possible implementation of functionalities enabling further compression of these, the listed features in Table 2.6 provide limited possibilities for further optimization of the compression rate. However, the standard innately minimizes the program trace information needed to be sent in order to be perfectly reconstructed. This is done by reporting only whether conditional branches are taken by encoding each branch outcome in a single taken/non-taken bit.

Table 2.7: Descriptions of all possible Nexus Transfer Code (TCODE) 6-bit messages. Table reconstructed from the Nexus 5001 standard [15].

TCODE	Meaning	Direction
0	Debug Status	From device
1	Device ID	From device
2	Ownership Trace	From device
3	Program Trace - Direct Branch	From device
4	Program Trace - Indirect Branch	From device
5	Data Trace - Data Write	From device
6	Data Trace - Data Read	From device
7	Data Acquisition	From device
8	Error	From device
9	Program Trace - Synchronization	From device
10	Program Trace - Correction	From device
11	Program Trace - Direct Branch with Sync	From device
12	Program Trace - Indirect Branch with Sync	From device
13	Data Trace - Data Write with Sync	From device
14	Data Trace - Data Read with Sync	From device
15	Watchpoint Hit	From device
16-19	Reserved	–
20	Port Replacement - Output	From device
21	Port Replacement - Input	From tool
22	Auxiliary Access - Read	Bidirectional
23	Auxiliary Access - Write	Bidirectional
24	Auxiliary Access - Read Next	Bidirectional
25	Auxiliary Access - Write Next	Bidirectional
26	Auxiliary Access - Response	Bidirectional
27	Program Trace - Resource Full	From device
28	Program Trace - Indirect Branch History	From device
29	Program Trace - Indirect Branch History with Sync	From device
30	Program Trace - Repeat Branch	From device
31	Program Trace - Repeat Instruction	From device
32	Program Trace - Repeat Instruction with Sync	From device
33	Program Trace - Correlation	From device
34	In-Circuit Trace	From device
35	In-Circuit Trace with Sync	From device
36-55	Reserved	–
56-62	Vendor-Defined Message	Bidirectional
63	Vendor-Defined Extension Message	Bidirectional

Nexus packages information into *Public Messages*. The public messages each start with a 6-bit TCODE which contains information about the number of packets, size of packets, and type of information that is being sent. The TCODE is followed by different *fields*, which number and of what kind is predefined through 29 packet types. The packet types contain different numbers of fields, and are thus of different size. The definition of TCODEs are shown in Table 2.7.

Nexus packet transmission is administered by the Nexus Message Protocol, which is implemented by the Message Start/End In ($\overline{MSEI}$) and Message Start/End Out ($\overline{MSEO}$) signals, which are active low (as indicated by the bar over them). $\overline{MSEI}$ is used for sending information to the processor and $\overline{MSEO}$ for the processor to transmit information to the memory or an outside client (via bus or high-speed link), both can be implemented as parallel dual signals in order to enable the transmission of two packets in one cycle. The encoding is shown below in Table 2.8.

Table 2.8: The Nexus Message Protocol for $\overline{MSEO}$. Table reconstructed from the Nexus 5001 standard [15].

$\overline{\text{MSEO}}$ Function	**Single $\overline{\text{MSEO}}$ Data (serial)**	**Dual $\overline{\text{MSEO}}$ Data**
Start of message	110	11-00
End of message	001(s)	00/01-11-11(s)
End of variable-length packet	010	00-01
Message transmission	0(s)	00(s)
End of message/End of packet	11	10
Idle (no message)	1(s)	11(s)

A branch history buffer can be implemented in order to avoid sending direct branches. They will instead be logged as executed in the branch history buffer and the transmitted in the next *indirect branch message - with sync* packet.

The encoder is required to implement at least one of the following modes (both may be supported):

- **Branch Trace Messaging (BTM):** Every taken direct conditional branch is generating at least a two byte long message, but repeated branches may instead be counted and reported as a single message with a count (instead of many identical messages), increasing efficiency.
- **Branch History Messaging (HTM):** Every direct conditional branch (taken or not-taken) adds a single bit to the history buffer, which is, similarly to BTM, more efficient than reporting every branch.

2.3.3 N-Trace

N-Trace is based on the Nexus standard, but it is specific for the RISC-V ISA [22]. Since it builds on the Nexus standard, features such as branch predictions are not possible. Notably for N-Trace is that the standard uses the same hart-to-encoder interface and control interface as the E-Trace standard, making it possible to use a 100% Nexus compatible message protocol with the E-Trace hart-to-encoder interface. The N-Trace specification is not yet fully complete with all features that Nexus can utilize. This means that N-Trace is only able to perform program trace, although data and bus trace are likely to be implemented in the future.

While the N-Trace standard builds on the Nexus standard, the message protocol for N-Trace is a strict subset of the Nexus protocol. N-Trace requires two $\overline{MSEO}$ bits and six Multiple Data Operations (MDO) bits, meaning that MDO+$\overline{MSEO}$ is always one byte. While the transmission order of $\overline{MSEO}$ or MDO is vendor specific for Nexus, N-Trace requires that $\overline{MSEO}$ is sent before MDO and that the least significant bit (LSB) is sent first for each field. The maximum field and message sizes are specified as 64 bits and 38 bytes, respectively.

Since N-Trace specifies that $\overline{MSEO}$ needs to be two bits, this also changes the $\overline{MSEO}$ sequence. This is shown on Table 2.9.

Table 2.9: The N-Trace Message Protocol for $\overline{MSEO}$. Table reconstructed from the N-trace standard [22].

$\overline{\text{MSEO}}$ Function	**Transition, Previous-Current**
Start of message	11-00
Middle of field	00 (or 01)-00
End of variable-length packet	00 (or 01)-01
End of message	00 (or 01)-11
Idle (no message)	11-11
Reserved	any-10

The Transition column shows what sequence is needed to change state. For example, for the start of a message, the previous $\overline{MSEO}$ should be 11, followed by 00 in the current cycle.

Fig. 2.2 shows a block schematic of the N-trace encoder interface for a single hart. For each hart there needs to be the same number of hart-to-encoder interfaces, encoders and control interfaces. There should however only be one debug module and one control layer to control the control interface. Note how the encoder outputs to both or either of memory or ATB bridge. At least one of these are mandatory to fulfill the RISC-V Trace Control Interface standard, but none of them are required in order to fulfill the Nexus standard, and can therefore be exchanged for vendor specific sink solutions [19].

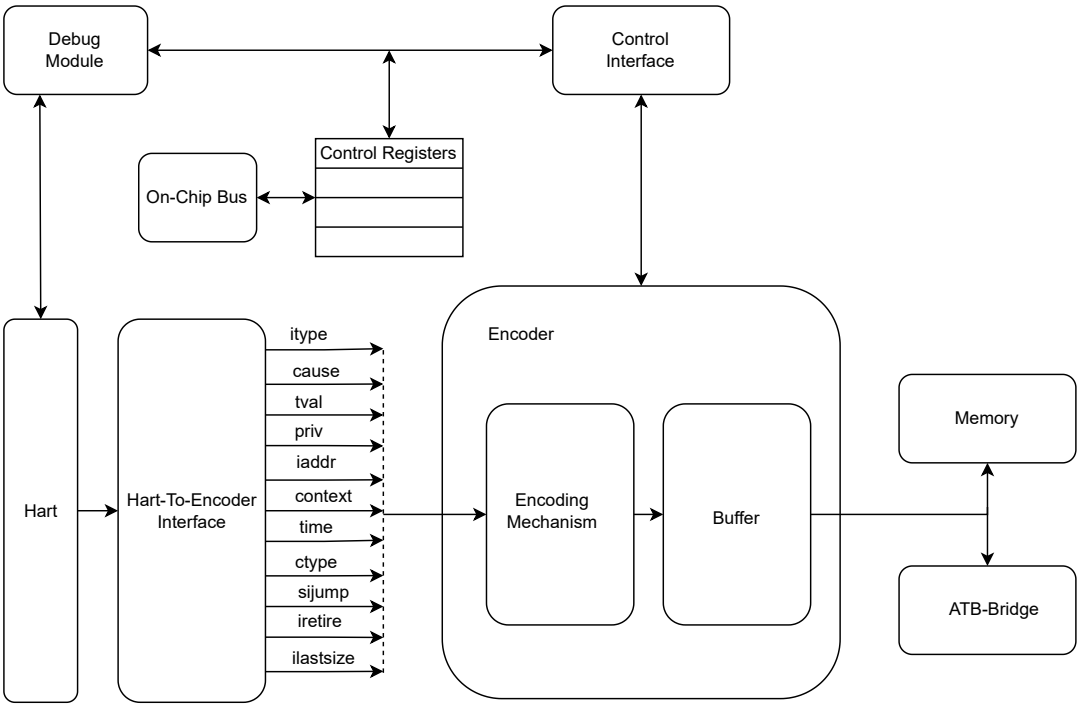

Figure 2.2: A block schematic of a single N-trace encoder implementation. Figure reconstructed from RISC-V Control Interface standard, but extended to include the hart-to-encoder signals.

The debug module is a significant part of the Nexus standard and likewise, it is needed for N-Trace to communicate between the hart and the control interface. The debug module also needs to communicate with the control registers. The hart-to-encoder interface is the same interface that is used for E-Trace, see section 2.3.1 for more info.

The possible messages for N-Trace are shown in Table 2.10. These are similar to Table 2.7 showing the messages for Nexus, the difference being that N-Trace supports much fewer messages. As can be seen from Table 2.10, many of the features that Nexus has, are not possible with the N-Trace specification. The only tracing that is possible is of the program trace, aswell as seeing the ownership.

Table 2.10: Descriptions of all possible N-Trace TCODE 6-bit messages. Table reconstructed from the N-trace standard [22].

TCODE	Meaning	Direction
2	Ownership Trace	From device
3	Program Trace - Direct Branch	From device
4	Program Trace - Indirect Branch	From device
8	Error	From device
9	Program Trace - Synchronization	From device
11	Program Trace - Direct Branch with Sync	From device
12	Program Trace - Indirect Branch with Sync	From device
27	Program Trace - Resource Full	From device
28	Program Trace - Indirect Branch History	From device
29	Program Trace - Indirect Branch History with Sync	From device
30	Program Trace - Repeat Branch	From device
33	Program Trace - Correlation	From device
56-62	Vendor-Defined Message	Bidirectional
other	Reserved	–

2.4 Advanced Microcontroller Bus Architecture (AMBA) Advanced Peripheral Bus (APB) protocol

The AMBA APB protocol is an open source standard describing an interface for connecting large numbers of peripherals and components to micro processors, and multi-core processor, via a bus system. The first official version 1.0 (issue A) of the protocol was released in 2003. The APB standard includes two separate busses for read and write operations, but can only do one of each at a time as the standard includes no individual identification system other than 32-bit addresses for the different components [23]. The bus is designed to complete each operation performed on the bus in two clock cycles. APB is one of several bus systems implemented on NOEL-V.

2.5 Available third-party decoders

One important aspect for comparing the standards is the availability of third-party decoders. Below is a list of all found decoder solutions for Nexus, N-trace, and E-trace:

- **Lauterbach:** Lauterbach have a software solution called TRACE 32 Powerview which can be used to decode both Nexus trace and E-Trace. The company provides a 10 hour free demo version of the program, any extended use must be purchased [24].
- **Siemens:** Siemens has developed Tessent Embedded Analytics which pro-

vide software tools for debugging. Tessent Embedded Analytics is E-trace complaint. Siemens offer no free demo version of the program [25].

- **IAR:** IAR provides a complete debug system called IAR Embedded Workbench, which contains a software decoder called C-SPY which is Nexus 5001 (and N-trace) compliant. They additionally provide a free C-based software decoder which is also Nexus 5001 compliant [26].

- **Ashling:** Ashling provides the RiscFree debug software which is able to decode both Nexus 5001 messages, and E-trace messages [27]. Ashling offers a free evaluation period of RiscFree.

- **Open-source Solution:** The survey conducted has only found two open-source solutions, one for Nexus 5001 and one for E-trace respectively. The available open-source software based decoder for Nexus 5001 is the same C-based software provided from IAR [26], and the only available E-trace solution is in the form of a python-based software provided by the RISC-V Debug and Trace Working Group itself [1]. This decoder solution contains no optional compression rate enhancing functionalities.

2.6 VHDL design method

The European Space Agency (ESA) states VHDL as their preferred hardware description language [28]. VHDL was developed to describe digital circuits, and is therefore inherently parallel in order to describe a multitude of signals and operations happening concurrently. In order for circuits being described in this dataflow style to implement sequential behavior, processes are often used, where *process* is a reserved word. Processes are triggered from events in chosen signals (stated in a sensitivity list), and start to perform their operations sequentially when triggered. For complex designs, this may lead to an overwhelming amount of sequential processes happening concurrently, which may lead to problems when reading or debugging the code. In order to clarify code, Jiri Gaisler has proposed the so called "two-process method" which has been adopted by ESA as a preferred design method.

As the name suggests, the method proposes a reduction of the number of processes to two, one describing combinational logic, and one describing sequential logic. The idea behind the method is to increase readability and decrease simulation time. Another claimed advantage of the method is the reduction in risk of creating latches, which may lead to timing issues within the circuit [29]. A block schematic representation of the two-process method can be seen below in Fig. 2.3. The output Q may, or may not, be dependent on the immediate input D.

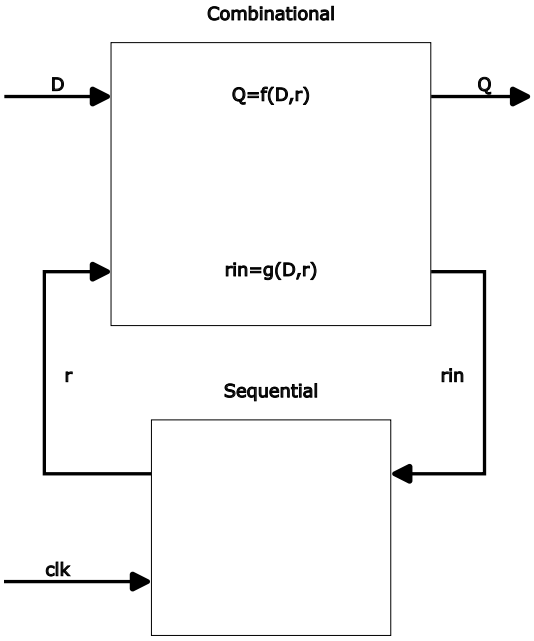

Figure 2.3: A block schematic representation of the two-process method. Figure reconstructed from [29].

3

Methods

The methods used in this project can be divided into five parts: literature study, market survey of decoders, design of encoder, verification through simulation, and simulation on hardware. These parts are presented below.

3.1 Literature study

In order to identify the characteristics of the trace standards, a literature study is performed on the subject. The main goal of the study is to identify the pros and cons of each standard in order to enable a fair and unbiased comparison in relation to this specific implementation. The study includes the understanding and description of the inner workings of each standard, and the analysis of the consequences of these in regards to this specific application. In order to fully understand the standards in a RISC-V application, the RISC-V standard itself is studied. Within the RISC-V standard, topics of particular interest is the instruction set, privilege levels, and context switch handling.

While much of the information is gathered from the standards themselves, published scientific articles regarding earlier implementations are also of relevance to the project.

3.2 Market survey of third-party decoders

The market survey is conducted through searching through specifications, brochures, and other relevant literature from companies working with the aforementioned standards. The standards themselves and their respective forums, forms an integral source of information about open-source solutions.

3.3 Designing the encoder

The method of designing the encoder is to first implement a minimal implementation of the chosen standard, and then to verify that it functions correctly in simulation, and with a third-party decoder. The description of the encoder is written in VHDL, following the "two-process method". The description is also, as frequently as possible, using *record types* in order to group signals together and make it easier to add, or remove, signals from different connections.

3.4 Verification through simulation

Verification through simulation is performed in two different stages. The first one is a stand-alone simulation of the encoder testbench, where the encoder is fed with the expected output trace from a generic RISC-V, and the result is decoded directly. The second one is a simulation where the encoder is connected to NOEL-V, which is then instructed to execute the test suite. Since connected, the trace generated from NOEL-V is automatically fed to the encoder, and the encoded trace is then decoded directly.

Stand-alone simulation

Trace data from the test suite is gathered by simulating the suite in Spike RISC-V simulator. The test suite is a number of benchmark tests in the form of executable and linkable format (.ELF) files. The outputted data from Spike is in the form of E-trace control interface signals in a comma separated values (.CSV) file. These are then read by a testbench, using Mentor Modelsim version 10.6a, and fed to the stand-alone encoder which in turn outputs a raw byte stream, which is saved to a file. The byte stream is translated to .CSV format by a Python program, and is then fed to the third-party decoder for reconstruction of the instruction flow. Upon reconstructed instruction flow, the decoder makes a check to see if the original instruction flow and the reconstructed instruction flow is identical. After assuring identical instruction flow, a statistics script (shell) is run, in order to extract measures for number of instructions, number of packets, total payload in bytes, and compression rate in bpi.

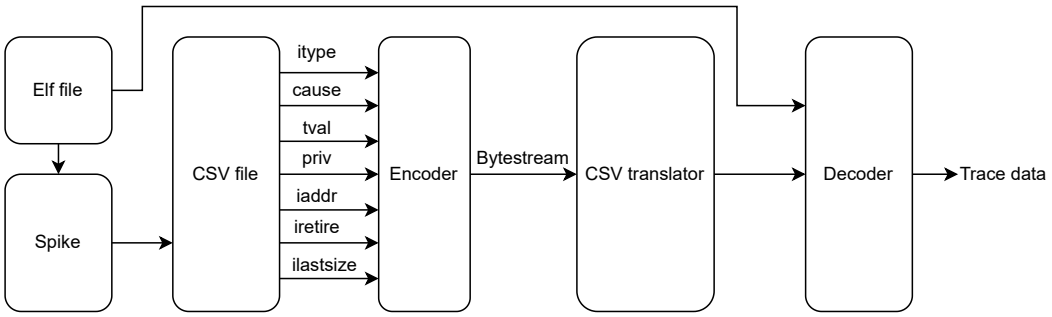

Figure 3.1: A block diagram of the verification workflow of the stand-alone simulation.

NOEL-V simulation

Simulation of NOEL-V with the encoder is carried out with Mentor Modelsim version 10.6a, altering an existing testbench in order to execute compiled programs in .srec format, containing RISC-V instruction sequences. The resulting encoded instruction trace data is then saved to a file, in order to verify correct encoding through the import/reading of said file into the designated third-party decoder. Upon achieving a working signal chain with this setup, the Frontgrade Gaisler Supertrace2 high-

speed link intellectual property (IP) is added to the signal chain (in between the buffer and the decoder) and the simulation is repeated.

3.5 Synthesis

NOEL-V is synthesized using Xilinx Vivado 2018.1 in order to establish baseline measurements of resources, power and critical path. The synthesis is performed for a Xilinx KCU-105 board with a XCKU040-2FFVA1156E FPGA. The synthesis is then repeated with the encoder connected to NOEL-V in order to gather information about required resources, power and critical path of the encoder. An additional synthesis is performed on the stand-alone encoder.

3.6 Verification on hardware

Upon verification of the encoder functionality in simulation, and synthesis, the verification on hardware is performed. The NOEL-V with the encoder is synthesized using Xilinx Vivado version 2018.1, and upon approved synthesis, a netlist is generated. The last stage of Vivado handling is generating a bitstream and loading it onto a Xilinx KCU-105 board with a XCKU040-2FFVA1156E FPGA. The FPGA is then connected to another FPGA through the Supertrace2 high speed link IP. The test suite is executed by the NOEL-V on the FPGA and the instruction trace data is saved to memory on the receiving end of the setup. The encoder is then verified by inputting the encoded instruction trace to the third-party decoder.

4

Design

The choice of standard was based on five main characteristics:

- **Functionalities:** The optional and mandatory functionalities that are available to the standard, and whether or not they are relevant for achieving the aim of the project.
- **Potential compression rate:** The potential compression rate excluding, and including optional compression rate enhancing functionalities.
- **Available third-party decoders:** To what extent there exists available third-party decoders on the market, and whether or not there are open-source solutions.
- **Available time compared to complexity of implementation:** A relative estimate of how much time and effort the implementation of the standard would require, with consideration taken to the available time of the project. What is the most realistic solution under the projects conditions and boundaries?
- **Industry trends:** An indication on how the market is evolving. Is one standard being heavily favored in contemporary solutions? Is there a noticeable change to the status quo?

A comparison of the two standards when applying the five chosen characteristics is shown below:

- **Functionalities:** Although Nexus has many more functionalities than both of the other standards, the only ones that are relevant for the aim of the project are available with the other standards as well. These are timestamping, "full-address mode" and data trace (except for N-trace, where data trace has not been added to the standard yet). The majority of optional functionalities of E-trace focuses on increasing the compression rate, and is only available through E-trace standard (branch prediction, implicit exception mode, sequentially inferable jump mode, implicit return mode, and jump target cache mode).
- **Potential compression rate:** The lower limit of branch instruction trace is available through all standards, and the major compression rate increasing functionalities are only available through E-trace. The encoding of the baseline compression (branch instruction trace) is done in the same way in all standards, with one bit representing a taken/non-taken branch, resulting in a similar compression rate. Therefore, this category heavily favors E-trace as it can

potentially be extended with optional functionalities to outperform both Nexus and N-trace in this regard.

- **Available third-party decoders:** The fact that Nexus is an older standard favors Nexus and N-trace in this category as there has been more time for the standard to penetrate the market, resulting in significantly more available decoders. Open-source solutions exist for all standards.
- **Available time compared to complexity of implementation:** The clear winner in this category is E-trace as it is a non-intrusive trace standard, and therefore requires little to no alteration of the central processing unit (CPU) or internal debugger. An additional factor that excludes pure Nexus is that it would require the development of a new hart-to-encoder interface, which is beyond the scope of this project. Therefore the only realistic choices are N-trace or E-trace, of which E-trace is favored in this category.
- **Industry trends:** The trend seems to be that if compatibility with Nexus compliant devices is needed, then N-trace is pursued. If that is not the case, E-trace is pursued.

Based on this comparison, E-trace is clearly the preferred standard for this implementation. However, this causes a potential problem with decoding as only three of the found E-trace compliant decoders are available for free. One of which, is an open-source solution. An additional problem is that non of these available decoders can be tested in advance since it requires E-trace encoded messages. Despite this, the E-trace standard is chosen for its relevant functionalities and superior potential compression rate. A full comparison of the standards is shown below in Table 4.1.

Table 4.1: A comparative chart of the three available trace standards.

	Functionalities	Compression	Decoder	Time and complexity	Industry trends
Nexus	+ Time-stamps + Debug capabilities	- Less	+ Common + Open source	- No interface - Intrusive - More time	+ Established + In use - Forum ended
N-trace	+ Time-stamps + Debug capabilities - No data trace	- Less	+ Common + Open source	+ Interface + For RISC-V - Intrusive - Less time	+ Developing + Established
E-trace	+ Time-stamps + Filtering + Compression - No vector ops.	+ More	- Uncommon + Open source	+ Interface + For RISC-V + Non-intrusive + Less time	+ Developing

4.1 Schematic

The encoder is made up of two components. A component for the control registers for the encoder called "cnt_reg" and a component for handling the functionalities of when and how to create different packages called "Pkg_format". A block schematic of the encoder can be seen below in Fig. 4.1.

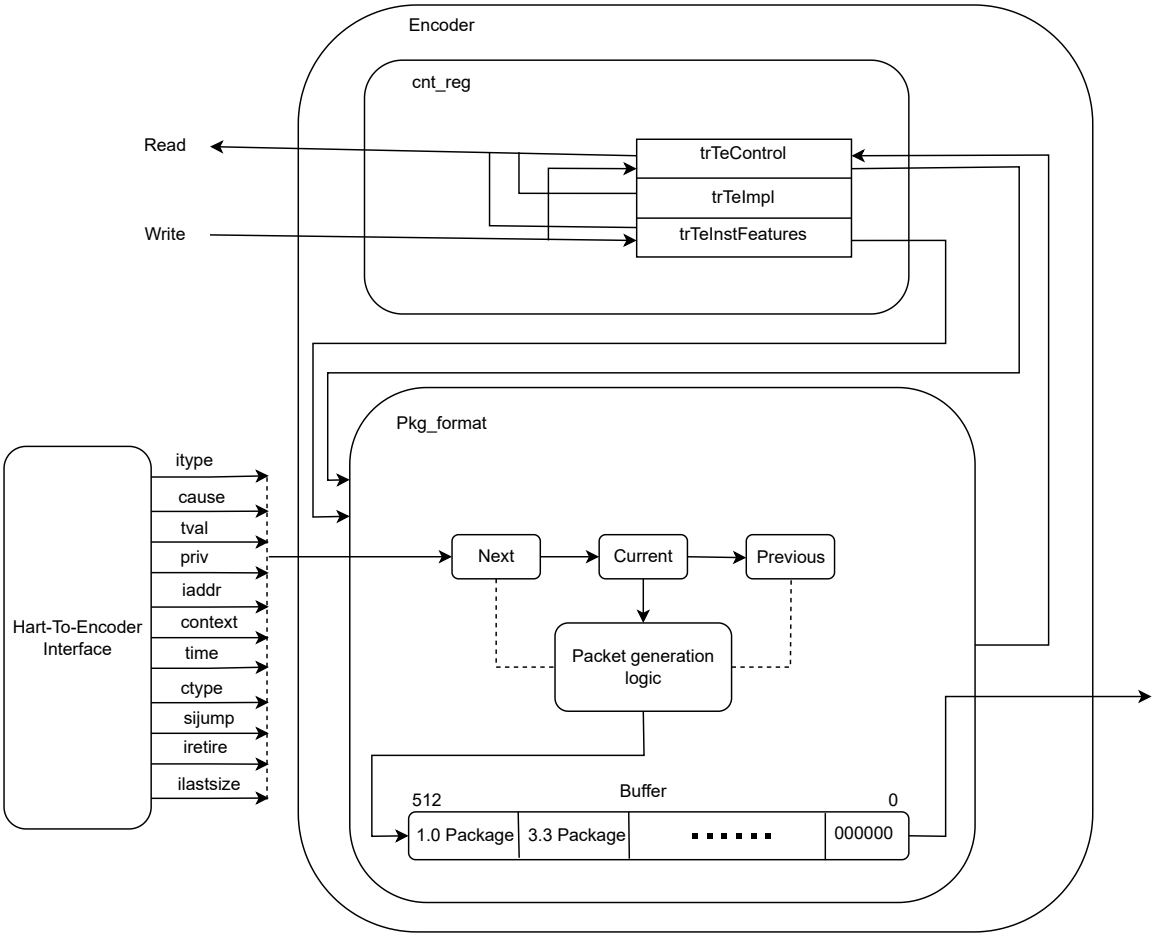

Figure 4.1: A block schematic of the encoder implementation.

4.1.1 cnt_reg

The encoder is controlled by a set of 32-bit component control registers that communicates with the system bus. In the implemented design this is named "cnt_reg". The registers control different functionalities within the encoder. How many registers are implemented per component is optional according to E-trace, with a required minimum of two. The implemented encoder has a total of three control registers, with two handling the encoder itself (trTeControl, trTeImpl), and one handling optional features (trTeInstFeatures). Within each control register, different bits control or communicate different actions or information to and from the system bus. The exact functions of each bit in each register can be found in the RISC-V Trace Control Interface standard under chapter six "Trace Encoder Control Interface" [19]. The implemented control registers are designed as an APB slave, as APB is the simplest of the two implemented system bus protocols on NOEL-V, the other one being Advanced High-performance Bus (AHB). In Fig. 4.1, note how trTeImpl is read only. This is because that the trTeImpl register only contains information about the system, which is hardcoded into the memory. Similarly, some bits in trTeControl

are set by Pkg_format, as they convey information about the process, such as when the buffer is empty.

For the trace encoder to be activated (according to the RISC-V Trace Control Interface standard [19]), a specific sequence of writes and reads on the encoder component control registers has to be issued via the system bus (in this case, the APB). First, a write to the control register releases the encoder from reset. Afterward, a read from the control register verifies that the encoder is active. Next, writes to the control and features registers configure the desired options and features. One of these options is the syncmode and syncmax register fields, which determine how and when to send a resync packet (format 3, subformat 0). Syncmax specifies whether to count trace packages, hart clock cycles, or instruction half-words, with trace packages and hart clock cycles currently implemented in the encoder. Syncmax defines the maximum interval between synchronization packets, ranging from 2^4 to 2^{19}, with each value being a power of two. After reading to ensure the registers have been updated, another write enables the encoder. Finally, a read confirms the encoder is enabled, and a last write turns on instruction trace.

4.1.2 Pkg_format

The control registers control the encoder packaging component called Pkg_format that takes input from the hart-to-encoder interface and packages the information according to the E-trace standard. The only implemented optional feature is "full address mode" which enables the packages to include the full address of each instruction instead of an offset to indicate how far the PC has progressed, or jumped. The packaging mechanism is implemented with a three-stage pipeline of previous, current, and next instructions. The reason for the pipelining is that the deduction of package type is dependent on the preceding and following packages (e.g. if an exception occurs at the arrival of the trap handler of a previous exception). Logically, Pkg_format boils down to a number of parallel, and nested, if-statements that deduces of what type the incoming instruction is, whether it should be packaged, and if so, in what format.

After the package is created, it is compressed using sign-based compression. This means that identical bits are removed from the most significant bit (MSB) side of the package. A decoder is able to sign extend this part to restore it back to its full length. After sign based compression is done, a one byte header is added to the LSB side of the package. This header gives information about how big the package is in bytes and if the package contains a timestamp.

When a header has been added to a package, this new package is placed in a 512 bit buffer. The implementation of this buffer serves two goals, it introduces flexibility in the system, and it ensures that the information sent to the sink is always of the same size, as the package format types are of varying length. When the buffer is full, the full content of the buffer, along with a valid signal is sent to the Frontgrade Gaisler IP Supertrace2 which is a high speed link working with double clock rate in order to achieve a high speed transfer of data. In order to accommodate the varying packet lengths inside the buffer, and packing them without padding in between, a

hardware implementation of binary search algorithm was implemented. This was done in order for the algorithm to find the exact byte length of the packet so that the buffer could be shifted the right amount of bytes. The binary search algorithm continuously splits the number of bytes into two parts, starting with the maximum number of bytes, and continuing with each of the resulting parts. In the end of the algorithm, the size of the packet is found within five steps. This solution was chosen due to its superior timing characteristics when synthesizing the design.

Since it's not always possible to store packages perfectly in the buffer because of different length of packages, there is also a second buffer of equal size. This is to help with two problems. First is that if a package is too large to fit into the remainder of the first buffer, it is instead placed in the second buffer, while the remaining bits in the first buffer are padded with zeros. This is shown in Fig. 4.2.

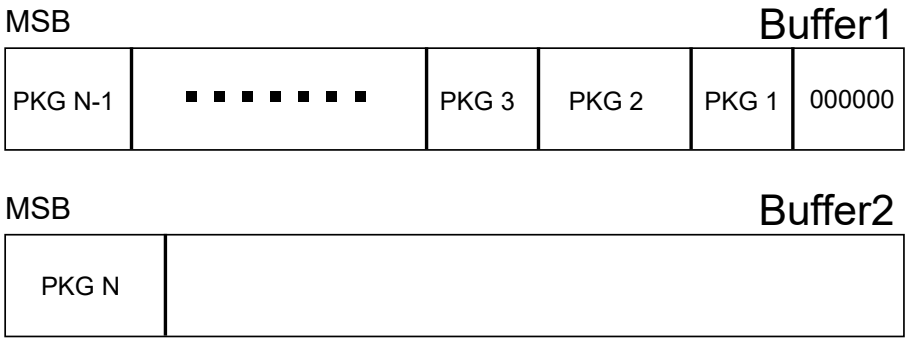

Figure 4.2: Storing packets with zero padding

When a buffer is full, either by being perfectly loaded with with packages, or by being padded with zeros beacause a package is too large to fully fit in the buffer, the buffer that the packages are loaded into is shifted. So when the first buffer is full, the next package is placed in the second buffer. The system then continues to load the second buffer with packages until the second buffer is full and then switches back to the first buffer. This way, only one package per cycle is created.

To compress the data even further it should be possible to split a package between the two buffers. Meaning that if a package is too large to fit entirely in the first buffer, it is split so that the lower bits of the package are placed in the first buffer and the other bits are placed in the second buffer.

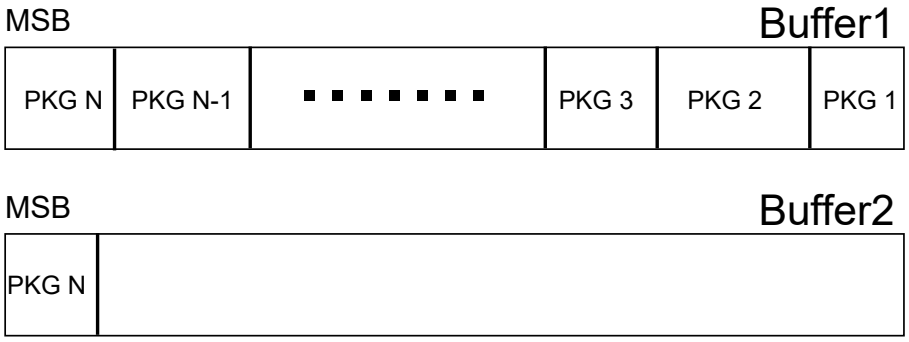

Figure 4.3: Storing packets without zero padding

This would utilize the full width of the 512 bit buffer, and remove some unnecessary zeros in the buffer, potentially giving a better compression.

5

Results

This chapter presents the compression results obtained from the simulated design, the compression rates achieved with different synchronization limits, and provides a theoretical estimate of how many cores could be traced based on the achieved compression. All tested programs have been decoded successfully using the publicly available open-source decoder provided by RISC-V Debug and Trace Working Group.

5.1 Simulation

Fig. 5.1 shows test results for the encoder using full address mode, syncmode set to count packages, and a syncmax value of 2^4. The bpi is calculated with

$$bpi = \frac{PL \cdot 8}{Instructions} \tag{5.1}$$

where PL is the payload, and is calculated by multiplying the number of 512-bit buffer outputs by 64 to obtain the total number of bytes. For each test we see two bars, one blue and one orange. The blue bar shows the bpi when sending a buffer with some unnecessary zeros and the orange bar shows the bpi when splitting some packages over two buffers, eliminating the unnecessary zeros. Since the last buffer is sent when tracing ends, it will still contain some zeros, which are included in this calculation. A full table of the values along with the total number of instructions, packages and bytes for each package is shown in Appendix A.1

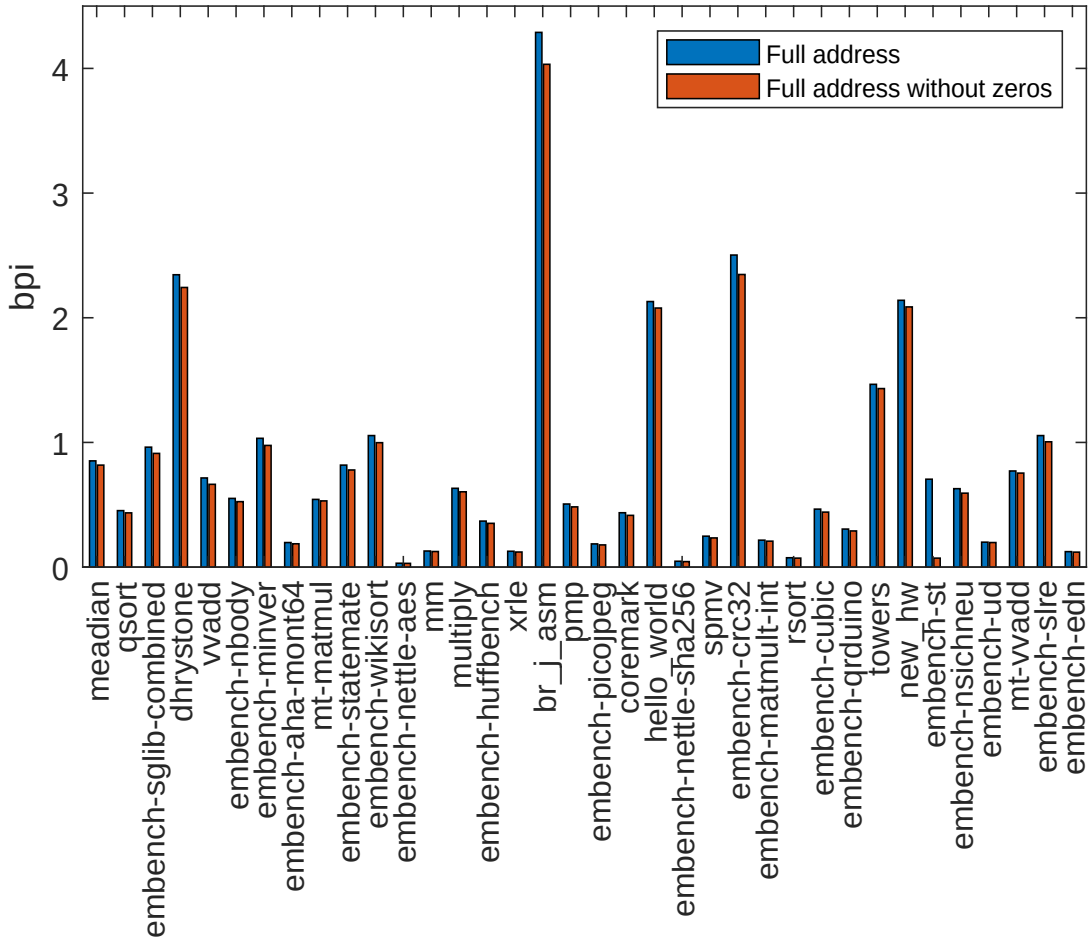

Figure 5.1: Compression of different test suites for full address mode

Fig. 5.1 shows that there is some improvement on the bpi to be gained by not using zero padding and instead splitting the package over two buffers, since this gives a better compression (lower bpi) for each test.

Fig. 5.2 below shows test results for the encoder using delta address mode, syncmode set to count packages, and a syncmax value of 2^4. The figure shows a 27.4% improvement on the average bpi by using delta addressing mode. Note how the five worst observed bpi from Fig. 5.1 is reduced by 44.2% on average, indicating a higher gain for the worst cases. A full table of the values along with the total instructions, packages and bytes for each package is shown in Appendix A.2

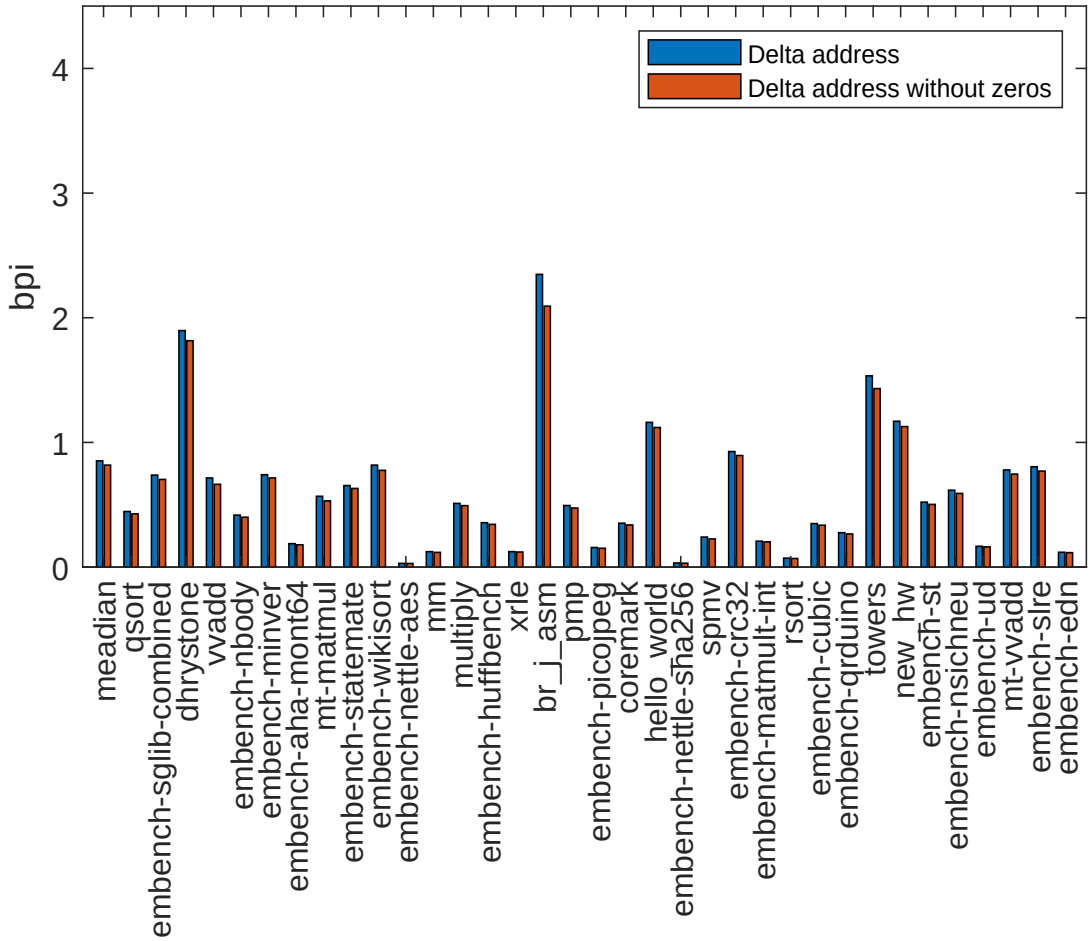

Figure 5.2: Compression of different test suites for delta address mode

Table 5.1 below shows the average statistics from the tests shown in Fig. 5.2 for different values of SyncMax with delta address mode. The average is taken from the average $PL \cdot 8$ divided by the average number of instructions. There is evidently a 12.4% increase in average compression rate to be gained from the case of having a synchronization packet sent after every 2^4 instruction, compared to after every 2^{19}. This indicates that the SyncMax value can be a useful parameter for achieving a certain compression rate goal. In practice, the upper limit of 2^{19} means that the vast majority of test suite programs will never send a synchronization packet (except at the start and end of tracing).

Table 5.1: Average encoder statistics for different SyncMax values.

SyncMax	Inst	PKG	PL [B]	bpi [b]	PL(w/o 0s)[B]	bpi(w/o 0s)[b]
2^4	2850881	27684	138490	0.3870	132965	0.3710
2^5	2860881	26168	129584	0.3620	124650	0.3480
2^6	2860881	25414	125299	0.3500	120556	0.3370
2^7	2860881	25038	123045	0.3440	118444	0.3310
2^8	2860881	24850	121886	0.3400	117392	0.3280
2^{11}	2860881	24685	120876	0.3380	116483	0.3250
2^{14}	2860881	24664	120800	0.3370	116371	0.3250
2^{19}	2860881	24662	120787	0.3370	116360	0.3250

5.2 Synthesis

The encoder integrated into NOEL-V was synthesized at 100 MHz for a Xilinx XCKU040-2FFVA1156E FPGA without any timing violations reading a worst negative slack (WNS) of 0.017 ns. The encoder turns up on second place of WNS with a slack of 0.0022 ns due to hold time, indicating that the encoder would have a limiting impact on the design for higher clock frequencies.

Table 5.2: Resource usage and power results from synthesis.

Part	LUTs	LUTs %	Power [W]	Power %
Encoder	21150	8.7%	0.018	1%

5.3 Traceable core count

The estimated minimum number of cores that can be traced by the system is calculated by Equation 5.2 below.

X = Worst observed compression rate (diff. mode) = 2.348 $\approx$ 2.5 bpi
Y = HSSL BW = 6.25 Gb/s [30]
Z = Worst case cache hit rate = 100%
f_{cl} = NOEL-V operating frequency
W = No. of instructions per second = $f_{cl} \cdot Z$
D = Worst possible dual issue factor = 2
W_{tot} = Worst total no. of instructions per second = $D \cdot W$
$T_{compressed}$ = Compressed BW required by a core = $W_{tot} \cdot X$ [bps]
N = No. of cores that can be traced

$$N = \frac{Y}{T_{compressed}} = \frac{Y}{W_{tot} \cdot X} = \frac{Y}{D \cdot W \cdot X} = \frac{Y}{D \cdot f_{cl} \cdot Z \cdot X} \tag{5.2}$$

Fig. 5.3 visualizes Equation 5.2 for clock frequencies of 100-1000 MHz, with a bar every 50 MHz. The red line highlights at which frequencies at least one processor core can be traced. It is clear that at least one core can be traced up to 1 GHz. For the encoder not to be able to trace a single core at 1 GHz, the compression rate

would have to increase to 3.125 bpi. Note the exponential decrease in number of cores that can be traced as the clock rate is increased. When tracing the graph backwards towards lower clock frequencies, the graph culminates in twelve cores with a considerable margin at 100 MHz.

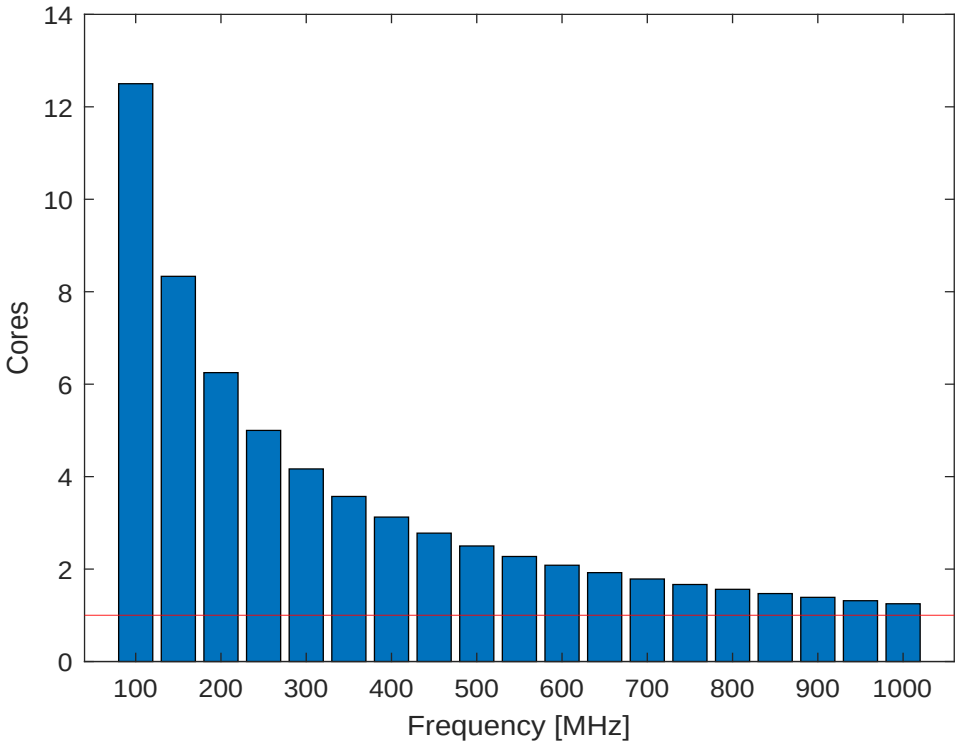

Figure 5.3: The number of cores that can be traced by the system at different clock frequencies, using Equation 5.2 and worst case parameters.

6

Discussion

In this chapter we analyze and discuss the results. We propose suitable design choices and give insight into how many cores the encoder could trace in a NOEL-V implementation. Further, we give some source of errors and suggest some future work for the encoder. We also present some societal, ethical and ecological implications of our work.

6.1 Encoder performance

We were expecting a big improvement in compression from using delta address mode instead of full address mode since the only difference between them is that most of the time, delta address sends a much shorter address (the difference is generally shorter than the full address). Although, there are cases where full address sends a shorter address. For example if we use a 64-bit addressing and the address changes from 0x80000004 to 0x80000000. Full address would report both addresses while delta address would have to report 0xfffffffffffffffc (should be shifted down so actual address sent is 0x7ffffffffffffffe). However, most of the time the PC will not be going backwards, and so, delta address should give a better result over time. This was confirmed by the results.

As expected, increasing the resync interval gave a better compression rate. However, the amount of compression decreases for every doubling of SyncMax. This also needs to be considered alongside the fact that increasing the resync interval runs the risk of losing more trace data in case some packets are lost. An appropriate value for SyncMax when counting packages might be 2^7 as the decrease in bpi does not outweigh the risk of losing a significant portion of the trace, should a package be lost. The results show that SyncMax could be a useful parameter to use in order to reach a certain goal of core coverage.

We expected the number of cores that could be traced simultaneously given the bandwidth of 6.25 Gb/s, to be much lower. The considerable margin at 1 GHz clock frequency indicates that the system could comfortably trace one core in a potential application specific integrated circuit (ASIC) implementation of NOEL-V running at 1 GHz. In a typical FPGA implementation of NOEL-V running at a clock frequency of 150 MHz or below, all eight cores could be traced simultaneously. This indicates that the system significantly outperforms our expectations, and solidifies the encoder as a viable solution to the trace data problem in space applications. This suggests

that the problem stated at the introduction of the report could be solved with the presented solution, which means that the goal of the project has been achieved.

An interesting note is that this is achieved without implementing any of the optional compression rate enhancing functionalities available through the E-trace standard. Therefore, several more tools for meeting tighter bandwidth constraints are available, although they would increase the hardware cost in area and power.

The results from the synthesis of the encoder show that it is associated with considerable hardware costs. The cost of 8% of the available look-up tables (LUTs) would have a significant impact on the total design, and it could prove vital to develop the design further in order to decrease this metric. A second result from the synthesis is the timing data. Although passing the 100 MHz mark, which is stated as the highest attainable clock frequency for NOEL-V with the XCKU040-2FFVA1156E FPGA, another FPGA or an ASIC implementation of NOEL-V might be able to reach higher clock frequencies. This could potentially make the encoder the most critical path in the design, highlighting the need for further optimization of the encoder design if intended for higher clock frequencies.

6.2 Sources of error

Even though the encoder was successfully integrated into NOEL-V, we were never able to verify the design on hardware. The source of this error was localized to the hart-to-encoder interface, which is beyond the scope of this project. The verification of the encoder was therefore limited to simulation in Spike. A consequence of this error is that not much use was found in pursuing verification in hardware, as the trace would never be decoded correctly anyways. Consequently, no analysis on potential discrepancies between simulated and real results could be made.

Part of the goal of the thesis was to achieve the highest possible theoretical compression rate and if this could not be met, explain why. This was not met, as no optional compression rate enhancing functionalities were implemented. The main reason for this is that the project had limited time, and that the basics had to be implemented and verified before optional extensions could be pursued. Another reason for this is that the provided hart-to-encoder interface did not provide the necessary optional signals in order to implement these functions. A third reason is that the used open-source decoder did not have the ability to decode the trace resulting from these functions.

When calculating the number of cores that can be traced, two potential sources of error have been identified. The first one is parameter X, representing the worst observed compression rate, as discussed in chapter 5.3. It is entirely possible that certain programs will result in a higher bpi, which would decrease the number of cores that could be traced. This could potentially push the result for 1 GHz to below one core, and decrease the number of cores that could be traced for lower frequencies. However, since there is a decent margin for the compression rate, where the bpi would have to increase by 25 % from our worst found case for us to not trace a single core at 1 GHZ, and all parameters are chosen for a worst case scenario, it is also plausible

that other parameters would have a decreasing impact on bpi. This could possibly mitigate this potential increase of bpi from a worsened compression rate of a specific program. The second source of error when calculating the number of cores is that every core to be traced requires an individual encoder. This results in the need of a trace funnel in between the individual encoder buffers and the transmission via the highspeed link. The impact of the trace funnel on the system is not accounted for, as it was never designed due to limited time. As it should introduce new overhead costs to the system, it is reasonable to assume that this would impact the system performance in a negative way. The extent of this negative impact is unknown.

6.3 Future work

The most prioritized suggestion for future work is verification on hardware, using two FPGAs connected with a Gaisler Research high-speed serial link (GRHSSL). Upon a successful verification, an analysis could be made on the differences between these results and the ones presented in this report.

After this, a trace funnel could be designed and implemented. This should enable verification of multi-core tracing, and should yield useful results that could be contrasted to the predicted ones presented in this report. The goal of this work should be to verify the results presented in this report in hardware.

Another suggestion for future work is alteration of the hart-to-encoder interface so that it can send optional signals. This should enable the implementation of the rest of the optional functionalities available through the E-trace standard.

Next, compression enhancing functionalities, data trace, and timestamping could be implemented. However, none of the compression enhancing functionalities, nor timestamping, is implemented in the utilized open-source decoder. Therefore, a verification of the complete system would require added abilities to the decoder, in order to decode this trace. As the decoder is written in Python, this suggestion is more software-oriented work.

6.4 Societal, ethical and ecological aspects

The encoder is intended to be part of systems that will be sent into space, which brings up some ethical aspects such as space debris and arguments of environmental impact. As this is a project in the space industry, and ultimately aims to enable an increase in effectiveness of space mission processors, such arguments are relevant to consider in this report.

Space debris

One ecological aspect is the amount of debris that is already in our orbit [31][32][33]. Sending a system into space has the risk of becoming debris, either by colliding with something else or if it stops working. The debris is traveling at a very high speed and can in turn collide with other things and cause damage [33], potentially even be life

threatening to humans [34][32]. In recent years, ways of removing these debris are being proposed. One idea of removing some debris is by retrieving malfunctioning satellites and reusing some parts of the satellite [31]. Another idea, proposed by [33] is to use a one launch satellite system that will de-orbit five large pieces of debris. This chaser satellite is meant to have batteries that will be charged by solar panels.

Environmental and societal discussion on space exploration

As the global warming increases [35][36], and the ecological crisis deepens [37], there is a case to be made that further investment in space exploration may cause more harm than benefit to the human race. The green house gas emissions and ozone depleting impact of solid rocket propellants are widely known [38]. There are voices being raised for an assertion of the environmental impact of the, nowadays more common, liquid propellant before continuing to increase the number of launches [39]. In her essay published in 2019, National Aeronautics and Space Administration (NASA) consultant Linda Billings suggests that the colonization of other planets, and any space exploration by extension, should be halted in order to focus more societal resources on preserving "spaceship Earth" and to find means and solutions to peaceful co-existence for its inhabitants [40]. There exist several counter arguments to these, one is the ever-persistent risk of environmental disasters not dependant on the global warming or the ecological crisis, such as asteroid collisions which may render Earth inhabitable. The argument here is that space exploration, and especially space colonization, lowers the risk of human extinction and that the benefits outweigh the consequences [41]. Other counter arguments made are that space exploration has a proven track record of developing spin-off technologies that can prove to have major benefits to the human race (such as global communication systems), that increased knowledge and refined tools increase the accuracy and impact of counter measures to the environmental crisis, and that pollution heavy industry could be moved off the planet [42][43].

7

Conclusion

The problem stated is the limited bandwidth for extreme-long-range applications of computers, primarily in space, creating a problem to debug software running on these machines, as the full instruction trace cannot be transmitted. The presented solution is a hardware implementation of a trace data encoder standard, resulting in compressed instruction branch trace.

The results show that the E-trace is the preferred standard to solve to the problem. Furthermore, the results indicate that an actual implementation of an E-trace compliant encoder is a viable solution to the problem. The encoded trace of 36 benchmark tests from the implemented encoder has been successfully decoded by a third-party software decoder. The full address mode results in an average compression rate of 0.5110 bpi, and a worst observed case of 4.0330 bpi. The default (and preferred) delta address mode results in an average compression rate of 0.3710 bpi, and a worst observed case of 2.0930 bpi. When transmitted through a high speed link with a bandwidth of 6.25 GHz, it is possible to trace at least one core up to 1 Gigabits per second (Gbps), and a full eight-core system up to 150 MHz clock frequency. Even though there are still several uncertainties, mainly concerning the lack of hardware verification, we conclude that the proposed solution to the problem is viable.

It is possible to further increase the compression rate(though associated with increased hardware costs), as several outlined compression rate enhancing extensions are available within the E-trace standard.

Bibliography

[1] "RISC-V Foundation - Efficient Trace for RISC-V," https://github.com/riscv-non-isa/riscv-trace-spec/blob/master/riscv-trace-spec.pdf, Accessed: 2024-01-29.

[2] "Nexus 5001 debug tool standard," https://nexus5001.org/wp-content/uploads/2015/01/nexus-wp-200408.pdf, Accessed: 2024-01-03.

[3] I. Gamino del Río, A. Martínez Hellín, R. Polo, M. Jiménez Arribas, P. Parra, A. da Silva, J. Sánchez, and S. Sánchez, "A RISC-V processor design for transparent tracing," *Electronics*, vol. 9, no. 11, 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/11/1873

[4] H. Kükner, G. Kaplayan, A. Efe, and M. A. Gülden, "RISC-V processor trace encoder with multiple instructions retirement support," in *Proc. of 2022 IFIP/IEEE 30th Int. Conf. on Very Large Scale Integration (VLSI-SoC)*, 2022, pp. 1–6.

[5] "Nexus 5001 Forum Dissolves After 20 Years," https://nexus5001.org/nexus-5001-forum-dissolves-after-20-years/, Accessed: 2024-01-29.

[6] "RISC-V Standardizes E-Trace and Binary Interface," https://www.electronicdesign.com/technologies/embedded/video/21246665/electronic-design-risc-v-standardizes-e-trace-and-binary-interface, Accessed: 2024-01-29.

[7] Britannica, The Editors of Encyclopaedia. "RISC". Encyclopedia Britannica, 12 Jan. 2024, https://www.britannica.com/technology/RISC. Accessed 31 January 2024.

[8] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, December 2019.

[9] V. A. Frolov, V. A. Galaktionov, and V. V. Sanzharov, "Investigation of RISC-V," *Programming and Computer Software*, vol. 47, no. 7, pp. 493–504, Dec 2021. [Online]. Available: https://link.springer.com/article/10.1134/S0361768821070045

[10] S. Di Mascio, A. Menicucci, G. Furano, C. Monteleone, and M. Ottavi, "The case for RISC-V in space," in *Applications in Electronics Pervading Industry, Environment and Society*, S. Saponara and A. De Gloria, Eds. Cham: Springer International Publishing, 2019, pp. 319–325.

[11] "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203", Editors Andrew Waterman, Krste Asanovic and John Hauser, RISC-V International, December 2021.

[12] https://github.com/riscv-software-src/riscv-isa-sim, , Accessed 2024-05-21.

[13] "Frontgrade Gaisler - NOEL-V Processor," https://www.gaisler.com/index.php/products/processors/noel-v, Accessed: 2024-01-30.

[14] "RISC-V Community - NOEL-V: A RISC-V Processor for High-Performance Space Applications," https://riscv.org/blog/2023/05/noel-v-a-risc-v-processor-for-high-performance-space-applications/, Accessed: 2024-01-30.

[15] "IEEE The Nexus 5001 Forum - Standard for a Global Embedded Processor Debug Interface," https://github.com/riscv-non-isa/tg-nexus-trace/blob/master/docs/nexus-standard/IEEE-ISTO-5001-2012-v3.0.1-Nexus-Standard.pdf, Accessed: 2024-01-26.

[16] S. P. Reiss, "Trace-based debugging," in *Automated and Algorithmic Debugging*, P. A. Fritzson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 305–314.

[17] E. Johnson, J. Ha, and M. Baqar Zaidi, "Lossless trace compression," *IEEE Transactions on Computers*, vol. 50, no. 2, pp. 158–173, 2001.

[18] C.-F. Kao, S.-M. Huang, and I.-J. Huang, "A hardware approach to real-time program trace compression for embedded processors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 3, pp. 530–543, 2007.

[19] "RISC-V Debug and Trace Working Group - RISC-V Trace-Control-Interface, howpublished = https://github.com/riscv-non-isa/tg-nexus-trace/blob/master/docs/RISC-V-Trace-Control-Interface.adoc," Accessed: 2024-02-14.

[20] "RISC-V Summit - Iain Robertson: Leveraging the RISC-V Efficient Trace (E-trace) Standard (video)," https://www.youtube.com/watch?v=rJUij2vH8mY, Accessed: 2024-03-27.

[21] "RISC-V Summit - Iain Robertson: Leveraging the RISC-V Efficient Trace (E-trace) Standard (slides)," https://static.sched.com/hosted_files/riscvsummit2023/6e/UL-005442-PT-B-Leveraging%20the%20RISC-V%20Efficient%20Trace%20%28E-Trace%29%20Standard.pdf, Accessed: 2024-03-27.

[22] https://github.com/riscv-non-isa/tg-nexus-trace/blob/master/docs/RISC-V-N-Trace.adoc, "RISC-V N-Trace (Nexus-based Trace) Specification", RISC-V International, Accessed 2024-02-16.

[23] "ARM LTD - AMBA APB Protocol Specification, howpublished = https://developer.arm.com/documentation/ihi0024/latest/," Accessed: 2024-05-07.

[24] "Lauterbach - Trace32 Powerview Software," https://www.lauterbach.com/products/software, Accessed: 2024-02-16.

[25] "Siemens - Tessent Embedded Analytics," https://blogs.sw.siemens.com/tessent/2023/11/01/debugging-risc-v-processors-using-e-trace/, Accessed: 2024-02-16.

[26] "IAR - Open source C-based software decoder for Nexus 5001 standard," https://github.com/IARSystems/trace-riscv, Accessed: 2024-02-16.

[27] "Ashling - RiscFree embedded debugging software," https://www.ashling.com/ashling-riscv/, Accessed: 2024-02-16.

[28] "ESA - VHDL," https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Microelectronics/VHDL, Accessed: 2024-03-25.

[29] "Jiri Gaisler - A structured VHDL design method," https://www.gaisler.com/doc/vhdl2proc.pdf, Accessed: 2024-03-25.

[30] "Frontgrade Gaisler - GRHSSL Space Fibre, howpublished = https://www.gaisler.com/index.php/products/ipcores/high-speed-serial-links," Accessed: 2024-05-22.

[31] T.-Y. Fung, S. S. Roy, Q. Shi, and D. A. DeLaurentis, "Space junk aggregation, neutralization, in-situ transformation, and orbital recycling," in *2022 17th Annual System of Systems Engineering Conference (SOSE)*, 2022, pp. 239–245.

[32] "What is space junk?." *McKinsey Insights*, p. N.PAG, 2023. [Online]. Available: https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-space-junk

[33] M. Dutta, M. Ashiqul Islam, A. Mohajon, S. Dev, D. Tripura, and I. Ahmed, "Extraction of space debris approach: Diminish the threats from outer space," in *2022 IEEE International Conference on Nanoelectronics, Nanophotonics, Nanomaterials, Nanobioscience Nanotechnology (5NANO)*, 2022, pp. 1–6.

[34] L. Liu, P. Jia, Y. Huang, J. Han, and E. Lichtfouse, "Space industrialization," *Environmental Chemestry Letters*, vol. 21, pp. 1–7, 2023.

[35] "The Global Climate 2011-2020," https://library.wmo.int/idurl/4/68585, Accessed: 2024-01-18.

[36] "Provisional State of the Global Climate 2023," https://wmo.int/files/provisional-state-of-global-climate-2023, Accessed: 2024-01-18.

[37] J. W. Moore, "The Capitalocene, Part I: on the nature and origins of our ecological crisis," *The Journal of Peasant Studies*, vol. 44, no. 3, pp. 594–630, 2017. [Online]. Available: https://www.researchgate.net/publication/263276994_The_Capitalocene_Part_I_On_the_Nature_Origins_of_Our_Ecological_Crisis

[38] A. I. Kuzin, "Ecological problems of space exploration," *AIP Conference Proceedings*, vol. 387, no. 1, pp. 1341–1346, Jan 1997. [Online]. Available: https://pubs.aip.org/aip/acp/article-abstract/387/1/1341/811835/Ecological-problems-of-space-exploration?redirectedFrom=fulltext

[39] J. Dallas, S. Raval, J. Alvarez Gaitan, S. Saydam, and A. Dempster, "The environmental impact of emissions from space launches: A comprehensive review,"

Journal of Cleaner Production, vol. 255, p. 120209, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0959652620302560

[40] L. Billings, "Colonizing other planets is a bad idea," *Futures*, vol. 110, pp. 44–46, 2019, human Colonization of Other Worlds. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0016328718303136

[41] G. Munévar, "Space exploration and human survival," *Space Policy*, vol. 30, no. 4, pp. 197–201, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0265964614000836

[42] W. S. Bainbridge, "Motivations for space exploration," *Futures*, vol. 41, no. 8, pp. 514–522, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0016328709000676

[43] "William K. Hartmann - Space Exploration and Environmental Issues," https://www.pdcnet.org/enviroethics/content/enviroethics_1984_0006_0003_0227_0239, Accessed: 2024-01-18.

A

Appendix 1

Table A.1: Encoder statistics full address.

Test	Inst	PKG	PL [B]	bpi [b]	PL(w/o 0s)[B]	bpi(w/o 0s)[b]
meadian	15015	277	1600	0.8520	1536	0.8180
qsort	235015	2351	13312	0.4530	12800	0.4350
embench-sglib-combined	2269624	34910	273088	0.9620	258752	0.9120
dhrystone	215015	8960	63040	2.3450	60288	2.2430
vvadd	10016	164	896	0.7150	832	0.6640
embench-nbody	6394547	56604	441088	0.5510	420032	0.5250
embench-minver	2620520	42615	338624	1.0330	319872	0.9760
embench-aha-mont64	4541666	17662	112000	0.1970	106624	0.1870
mt-matmul	41454	522	2816	0.5430	2752	0.5310
embench-statemate	1038140	13282	106240	0.8180	101120	0.7790
embench-wikisort	2346534	44813	309696	1.0550	292800	0.9980
embench-nettle-aes	4523974	2765	17856	0.0310	17344	0.0300
mm	297038	909	4800	0.1290	4672	0.1250
multiply	55016	862	4352	0.6320	4160	0.6040
embench-huffbench	2461122	18714	1136000	0.3690	108032	0.3510
xrle	164959	546	2624	0.1270	2496	0.1210
br_j_asm	10027	801	5376	4.2890	5056	4.0330
pmp	1110463	11047	70336	0.5060	67072	0.4830
embench-picojpeg	4012853	17255	93440	0.1860	89472	0.1780
coremark	33399232	256974	1821504	0.4360	1733248	0.4150
hello_world	358353	14340	95424	2.1300	93120	2.0780
embench-nettle-sha256	3874839	3224	22912	0.0470	21696	0.0440
spmv	70015	331	2176	0.2480	2048	0.2340
embench-crc32	4028862	197016	1260928	2.5030	1182272	2.3470
embench-matmult-int	2891811	14837	78400	0.2160	75520	0.2080
rsort	375016	812	3520	0.0750	3392	0.0720
embench-cubic	7724342	57852	449216	0.4650	426688	0.4410
embench-qrduino	3426829	20216	131008	0.3050	124480	0.2900
towers	15016	396	2752	1.4660	2688	1.4320
new_hw	356128	14285	95296	2.1400	92928	2.0870
embench-st	4412662	49692	389056	0.7050	370496	0.0710
embench-nsichneu	2241146	27899	176256	0.6290	166336	0.5930
embench-ud	1277151	5012	32064	0.2000	31552	0.1970
mt-vvadd	61072	1126	5888	0.7710	5760	0.7540
embench-slre	2622482	44636	345856	1.0550	329728	1.0050
embench-edn	3492782	12943	54464	0.1240	52672	0.1200
Average	2860881	27684	192819	0.5390	183064	0.5110

Table A.2: Encoder statistics delta address.

Test	Inst	PKG	PL [B]	bpi [b]	PL(w/o 0s)[B]	bpi(w/o 0s)[b]
meadian	15015	277	1600	0.8520	1536	0.8180
qsort	235015	2351	13120	0.4460	12544	0.4270
embench-sglib-combined	2269624	34910	209472	0.7380	199552	0.7030
dhrystone	215015	8960	51008	1.8970	48832	1.8160
vvadd	10016	164	896	0.7150	832	0.6640
embench-nbody	6394547	56604	333888	0.4170	320000	0.4000
embench-minver	2620520	42615	242816	0.7410	234432	0.7150
embench-aha-mont64	4541666	17662	106880	0.1880	102144	0.1790
mt-matmul	41454	522	2944	0.5680	2752	0.5310
embench-statemate	1038140	13282	84992	0.6540	81920	0.6310
embench-wikisort	2346534	44813	239936	0.8180	227648	0.7760
embench-nettle-aes	4523974	2765	17216	0.0300	16640	0.0290
mm	297038	909	4608	0.1240	4416	0.1180
multiply	55016	862	3520	0.5110	3392	0.4930
embench-huffbench	2461122	18714	109696	0.3560	105664	0.3430
xrle	164959	546	2560	0.1240	2496	0.1210
br_j_asm	10027	801	2944	2.3480	2624	2.0930
pmp	1110463	11047	68608	0.4940	65920	0.4740
embench-picojpeg	4012853	17255	79104	0.1570	76224	0.1510
coremark	33399232	256974	1472768	0.3520	1411456	0.3380
hello_world	358353	14340	52032	1.1610	50176	1.1200
embench-nettle-sha256	3874839	3224	16064	0.0330	15488	0.0310
spmv	70015	331	2112	0.2410	1984	0.2260
embench-crc32	4028862	197016	467072	0.9270	450944	0.8950
embench-matmult-int	2891811	14837	75392	0.2080	73280	0.2020
rsort	375016	812	3392	0.0720	3264	0.0690
embench-cubic	7724342	57853	337536	0.3490	325312	0.3360
embench-qrduino	3426829	20216	118592	0.2760	114048	0.2660
towers	15016	396	2880	1.5340	2588	1.4320
new_hw	356128	14285	52096	1.1700	50176	1.1270
embench-st	4412662	49692	287872	0.5210	277568	0.5030
embench-nsichneu	2241146	27899	172992	0.6170	165824	0.5910
embench-ud	1277151	5012	26752	0.1670	25984	0.1620
mt-vvadd	61072	1126	5952	0.7790	5696	0.7460
embench-slre	2622482	44636	264064	0.8050	252608	0.7700
embench-edn	3492782	12943	52288	0.1190	50688	0.1160
Average	2850881	27684	138490	0.3870	132965	0.3710