CHALMERS
UNIVERSITY OF TECHNOLOGY

Image owned by Frontgrade Gaisler AB

Enhancing NOEL3 RISC-V processor with CHERI memory protection

Master's thesis in Embedded Electronic System Design

Alfonso Carballo Boullosa
Elías Vázquez Maceiras

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

Master's thesis 2026

Enhancing NOEL3 RISC-V processor with CHERI memory protection

Alfonso Carballo Boullosa
Elías Vázquez Maceiras

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2026

Enhancing NOEL3 RISC-V processor with CHERI memory protection

Alfonso Carballo Boullosa
Elías Vázquez Maceiras

Supervisor: Ioannis Sourdis, Computer Science and Engineering
Company advisor: Nils Wessman, RISC-V Group Sec. Head, Frontgrade Gaisler
Examiner: Per Larsson-Edefors, Director of Master's Programme

Master's Thesis 2026
Department of Microtechnology and Nanoscience
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Enhancing NOEL3 RISC-V processor with CHERI memory protection
Alfonso Carballo Boullosa
Elías Vázquez Maceiras
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

Memory safety vulnerabilities are a concern in modern computing systems. CHERI (Capability Hardware Enhanced RISC Instructions) is an architectural extension that replaces traditional pointers with capabilities that include permissions and bounds, among other metadata. This allows fine-grained memory protection and compartmentalization that provide guarantees against a number of these safety issues. This thesis presents the addition of CHERI into NOEL3, a 32-bit RISC-V processor characterized by its deterministic barrel architecture, multiple extensions and emphasis on fault tolerance and reliability. The baseline processor has been extended with the microarchitectural structures needed to support CHERI, such as capability registers, tags and enforcement logic. The implementation, targeting the KCU105 FPGA evaluation board, was verified and debugged by means of specific instruction tests through comparison with a golden signature to ensure compliance with the CHERI specification. With respect to the standard NOEL3 with Physical Memory Protection, results indicate an average clock rate degradation of 37 %, while resource utilization (heavily influenced by CLB LUTs) is increased by 38 %.

Keywords: CHERI, capability, tag, metadata, address, memory

Acknowledgments

We would like to thank Ioannis Sourdis (our supervisor), Nils Wessman (our company advisor) and Gaisler employees that helped us to make this project possible: Jan Andersson, Asier Fernández de Lecea Navarro, Shushruth Holla, Jonathan Jonsson and Johan Klockars. We would also like to thank Frontgrade Gaisler for allowing us to undertake this project and giving us the freedom and flexibility that lead to engineering excellence and innovation.

Alfonso Carballo Boullosa and Elías Vázquez Maceiras, Gothenburg, June 2026

Contents

List of Figures

List of Tables

List of Codes

1

Introduction

Modern computing systems are susceptible to a wide range of memory safety vulnerabilities [1]. These classes of defects constitute a substantial fraction of severe security failures observed in production hardware and software platforms. Microsoft has determined that 70% of vulnerabilities treated as CVEs (Common Vulnerabilities and Exposures) are memory-related [2], while Google's Chromium project has revealed that around 70% of "serious security bugs are memory safety problems" [3].

Four well-known types of these vulnerabilities are the following:

- Buffer overflows [4]: a program writes more data into a buffer than this can store, therefore overwriting adjacent memory locations.
- Use-after-free errors [4]: a program continues to use a pointer which should have been cleared (dangling pointer) after its corresponding memory region has been deallocated.
- Pointer corruption: the value held by a pointer is accidentally or maliciously modified, allowing it to access a different memory region.
- Privilege escalation: design flaws or weak protection measures result in an unintended obtention of higher or broader permissions.

Virtual memory, processor privilege levels, software strategies like stack canaries and coarse-grained hardware isolation primitives like RISC-V Physical Memory Protection (PMP) [5] are conventional mechanisms deemed insufficient to guarantee memory safety at a fine-grained, per-object level [6].

CHERI [7, 8] (Capability Hardware Enhanced RISC Instructions) introduces architectural capabilities (pointers with hardware-enforced bounds, permissions, and unforgeability) to mitigate such vulnerabilities at the hardware layer, building on the idea of capability-based computing systems [9] and the concept of fat pointers [10, 11]. CHERI has been integrated into experimental RISC-V prototypes and select commercial cores, but remains an emerging technology with limited architectural diversity, incomplete toolchain and software ecosystem support, and lacking evaluation across different microarchitectural designs.

1.1 Related work

Developed by the University of Cambridge and SRI International, CHERI is extensively documented in a technical report [12] produced by these two entities. It has been implemented on MIPS and ARM (Morello [13, 14]). Besides that, several implementations of CHERI on RISC-V have been made:

- CHERI-CVA6 [15], an open-source core extended for research purposes.
- Codasip X730 [16], the first commercial CHERI-RISC-V CPU.
- CHERIoT [17], an implementation for small devices initially developed by Microsoft.

These efforts demonstrate that CHERI provides spatial and temporal memory safety [4, 18, 19]: the former prevents out-of-bounds access and the latter prevents access to freed memory. It also supports compartmentalization, enables fine-grained privilege reduction and can be applied to execute C programs with the memory safety that this programming language lacks [20, 21]; it has also been applied to embedded systems [22]. However, existing implementations are either research prototypes (FPGA-based, not optimized for industrial requirements) or commercial implementations with limited public insight. Furthermore, the RISC-V CHERI spec states that capability permissions are enforced alongside privilege-based mechanisms such as PMP, a relationship that has not yet been deeply evaluated. Therefore, the literature strongly motivates further investigation into microarchitecture-specific CHERI implementations. As of now, there have been no known efforts in implementing CHERI in the frame of fault tolerance or multithreading, giving us a solid ground and justification to undertake our proposed project.

1.2 Purpose and goal

This thesis proposes to add CHERI to Frontgrade Gaisler's NOEL3 [23], a recently released RISC-V processor with unique microarchitectural features and designed for use in space applications (RISC-V space-grade systems have only recently begun to gain popularity [24]). We intend to evaluate its functional correctness and quantify the architectural overheads; the results will expand the understanding of CHERI applicability beyond existing prototypes, providing insight into the applicability of CHERI across diverse RISC-V microarchitectures and contributing to the development of future secure processors.

Although CHERI's effectiveness has been demonstrated, the integration effort is complex and implementing it requires modifying certain structures within the processor. Existing CHERI-RISC-V implementations leave open the question of how CHERI interacts with different microarchitectural organizations. The problem addressed in this thesis is therefore:

> *How can CHERI be integrated into NOEL3, and what performance and hardware implications arise from the integration?*

Given these statements, we defined the scope to be the following:

- Researching CHERI specification to understand the purpose, operation, encoding and decoding of capabilities.
- Translating the theoretical knowledge of CHERI it into a series of required changes to the standard NOEL3.
- Applying those modifications to the standard processor in question, incorporating the necessary microarchitectural structures and logical changes required for the operation and, if necessary, verification of the enhanced core.
- Designing a workflow that effectively and efficiently verifies the functional correctness of the design, resulting in a series of tests that allow us to debug our implementation.
- Comparing the verified, CHERI-enhanced NOEL3 with its baseline counterpart, analyzing the overhead incurred in terms of performance and resource utilization to incorporate the capability architectural model.

1.3 Thesis outline

Following this **Chapter 1**, **Chapter 2** offers an insight into RISC-V and why its PMP mechanism lacks appropriate security. After this, it moves on to a detailed explanation of CHERI and why it offers strong memory protection guarantees; said section delves into the concept of capabilities and how these are handled at the register and memory level. The chapter concludes with a description of NOEL3 and its main features.

Chapter 3 describes the overall workflow followed in the project. It connects the study of CHERI and NOEL3, the implementation process, the successful directed verification flow, the attempted TestRIG flow and the final FPGA evaluation. It also explains how the main tools were used, including the role of GHDL and Verilator in the attempted RVFI-DII/TestRIG integration.

Chapter 4 moves on to a deeper technical level to break down specific microarchitectural changes, listing modifications made to specific pipeline stages, the memory structure and certain aspects of instruction flow control. It then briefly describes the additional logic required to communicate with a random testing environment.

Chapter 5 covers the testing method and the debugging process. It first explains the aforementioned random testing environment and the issues that prevented us from successfully using it to verify our implementation. What follows is a thorough description of our specific testing process, including methodology and underlying principles, the actions performed by the tests themselves and how these are executed and their results compared with expected values.

Afterwards, **Chapter 6** states the finished design's functional correctness before focusing on comparing the baseline processor with the modified version, breaking down the setup used and analyzing and discussing implementation results and the consequent overheads in performance and resource utilization.

Finally, **Chapter 7** summarizes the achievements and goals fulfilled in this project, also highlighting possible future opportunities.

2

Technical Background

There are several concepts which are essential for this project; they have been mentioned in **Chapter 1** and will be explained in this chapter in further detail. Firstly, RISC-V will be introduced, leading to PMP and why it fails to provide appropriate memory protection. An overview of NOEL3 will follow, explaining the processor's main features and why it is an interesting base onto which implement CHERI. Lastly, a technical breakdown of CHERI will close this chapter, stating why it provides memory security and how it is designed to deliver it. For greater detail than is provided about CHERI in this chapter, please refer to the official specification [7].

2.1 RISC-V and PMP

RISC-V [25] (Reduced Instruction Set Computer Five) is an open source ISA (Instruction Set Architecture) that is becoming increasingly widespread in industry. The architecture is based on 32 registers (*x0* to *x31*, each with its own alias) of length *XLEN*, which can be 32 or 64 bits. A thread (or hart) has an address space of 2^{XLEN} bytes; a word is defined as 4 bytes or 32 bits, the same size and alignment as base RISC-V instructions; there are also a Program Counter (PC) register and CSRs (Control and Status Registers). This project makes use of the 32-bit architecture.

RISC-V includes a standard feature called Physical Memory Protection (PMP) [26], which lets software partition physical memory into regions and assign access permissions (read, write, execute) to each of these regions. The RISC-V PMP unit consists of a small number of configuration registers (typically up to 16 or 32 entries). Each entry defines a physical address range (using one or two registers for start and end) and a set of access permissions for that range. The permissions indicate whether loads, stores, or fetches from that region are allowed in the current privilege mode. On each memory access, the processor checks the address against the PMP entries to determine if the access should be permitted.

The weakness of this mechanism is its enforcement of protection at the granularity of memory regions (via fixed address ranges) rather than individual objects or pointers. This means that if a pointer is accidentally or maliciously misused (e.g., through an out-of-bounds computation), PMP cannot prevent illegal access unless region protections happen to catch it. Furthermore, PMP entries are limited in number and must be configured in software; dynamic use (e.g., per-pointer permissions or a large number of fine grained objects) would exhaust the hardware. In safety-critical domains like aerospace, these limitations motivate use of a finer-grained scheme like CHERI, which provides per-object

capabilities to enforce bounds and permissions.

2.2 CHERI

CHERI offers its own memory access control by substituting integer pointers with object called capabilities. Two guiding principles dictate the way CHERI works:

- Principle of least privilege: running software shall have the minimum and lowest essential number of privileges [27].
- Principle of intentional use: if several privileges are available, the software must explicitly state the one being used.

Furthermore, three properties are derived from CHERI's protection mechanism:

1. Provenance: valid capabilities shall only be derived from other valid capabilities.
2. Integrity: corrupted capabilities shall not be eligible for access (also called dereference).
3. Monotonicity: when modifying a capability, its bounds and rights shall be equal or reduced, never increased [28].

2.2.1 Capability definition, tags and storage

If the baseline system's architecture uses a certain number of bits (*XLEN*), capabilities will be stored in registers of length *2*XLEN = YLEN*. The address to where the capability points is stored in the lower *XLEN* bits(this field is also used to store regular values when the register is not used as a capability); the metadata (essential information for CHERI's operation) is stored in the higher *XLEN* bits.

A tag is a hardware-managed bit associated to a capability, as seen in **Figure 2.1**; if it is set, the capability has been correctly created and is valid, and can therefore be dereferenced. The tag will remain set if the capability is modified properly according to CHERI's three properties of protection:

1. Provenance: one capability tag is needed as input to write another capability.
2. Integrity: a capability must not have any corrupted values.
3. Monotonicity: capability writes must not increase bounds or rights and must set the output capability's tag.

When failing any of these checks, either the tag is cleared or an exception is raised; the latter will also happen when attempting to use an invalid capability as a valid one.

The register file is doubled in length to accommodate capabilities (*x0* receives empty metadata and a cleared tag, a null capability). For capabilities stored in memory [29], every aligned *YLEN*-bit wide region also has a tag, which will be cleared if any aligned *YLEN/8* region is written with an instruction that is not a capability store which is allowed to set the tag: a capability must be modified as a whole. The PC is also extended

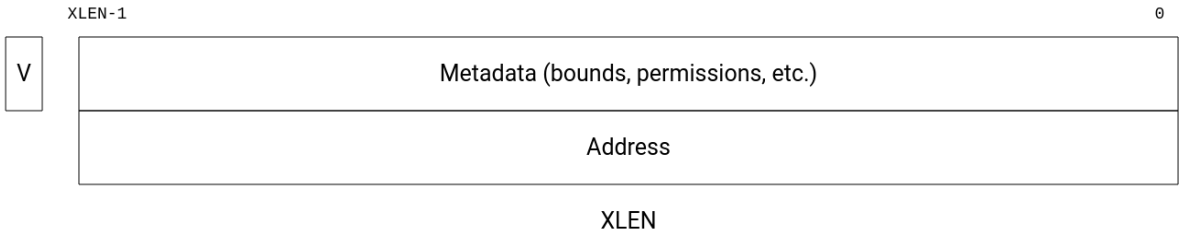


Figure 2.1: Capability storage and tag (marked V). Image extracted from the CHERI spec document.

and becomes a capability, adding security to code flow control; its address field is the same as in standard RISC-V so that the hardware can update it. This capability is checked on every executed instruction; it also includes the M-bit, which allows to change the processor's operation mode between CHERI and standard RISC-V; this is enabled by the *Zyhybrid* extension. Several CSRs are also extended to support the new functionality. On startup, bounds and permissions must be set so that boot instructions can be executed; at the same time, root capabilities with unlimited bounds and permissions are created.

2.2.2 Capability bounds and addresses

The capability's metadata includes the bounds (range of memory addresses accessible), from the base (included) to the top (not included). These are encoded in a manner alike floating point. This leads to an essential concept: representability.

Suppose a capability with a certain address a and metadata that results in certain decoded bounds (the address pointer participates in bounds encoding and decoding, as explained later). Now suppose that a new capability is derived from it, with address a' and the same metadata. If the new capability shares base and top decoded bounds with the previous capability, a' is considered to be inside the source capability's representable range. Otherwise, the derived capability will have its tag set to zero.

It is important to note that the representable region is wider than the dereferenceable region. If the pointer is within the former and outside of the latter, a hardware exception should be raised due to an invalid memory access; the capability, however, will remain valid. This is done to support certain semantics such as a loop where the pointer value may be just beyond the array's limit; comparison is allowed, but data retrieval is not. These concepts are graphically explained in **Figure 2.2**. The representable region may wrap around zero in a circular manner, but the decoded top address does not; this prevents a possible vulnerability that attackers might exploit to escalate their privileges.

2.2.3 Architectural permissions

The metadata contains a field that stores its permissions: Read (R) from memory, Write (W) to memory, Execute (X) instruction in memory, Capability (C), Access System Registers (ASR) and Load Mutable (LM).

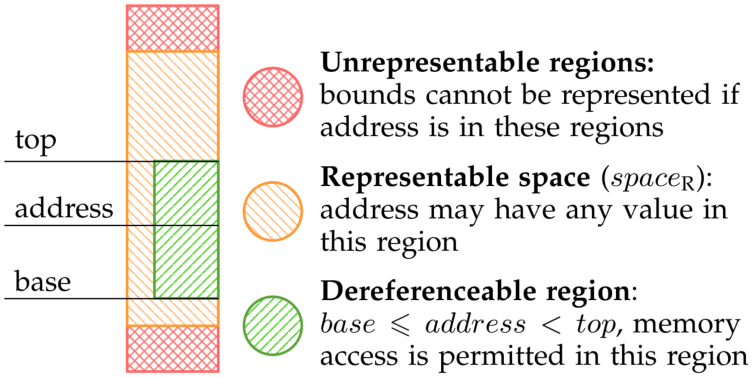


Figure 2.2: Capability bounds and regions. Image extracted from the CHERI Concentrate.

C allows loading (from memory) or storing (to memory) a capability with its tag, propagating its authority to the destination capability. Loading requires Reading permissions and storing requires Writing permissions. If the *C* permission is missing, loaded and stored capabilities will have their tag cleared. *ASR* allows access (both read and write) to Control and Status Registers (CSRs).

LM allows propagation of the *W* permission to the target (or destination) capability. Suppose the following instruction: *ly rd, 0(rs1)*, which loads into *rd* the capability held in the memory address pointed to by *rs1*. If *rs1* has *LM* set, then *rd* will preserve *W* and *LM* from the capability pointed to in memory. Otherwise, *W* and *LM* will be cleared in *rd*; this is a key security feature, since it effectively allows the creation of read-only copies of capabilities. This has two main advantages:

- Preventing privilege escalation: if *LM* were ignored, any loaded capability would have writing access.
- Creating immutable references without duplicating memory content; the target register can point to a specified memory region while only being able to read it.

Some permissions are dependent on others; for our choice of encoding (see **Subsection 2.2.4**), each property will be available if its corresponding logical condition is true:

- *R*: independent of other permissions
- *W*: `not(C) or LM`
- *X*: `R and (W or C) and ((C and LM) or not(C or LM))`
- *C*: `R`
- *ASR*: `W and C and X`
- *LM*: `C`

As mentioned in **Subsection 2.2.1**, the PC capability is checked on every instruction: its tag must be set, it must have the *X* permission and all bytes in the instruction (RISC-V fetches multiple bytes on every instruction) must be within bounds.

2.2.4 CHERI encoding formats

In this project, we will use the *RV32_LYmw10rc1pc* encoding. The meaning of this character string is the following:

- *RV32*: RISC-V, 32-bit architecture.
- *L*: indicator that the following parameters define the encoding; intended for standardization and simplicity.
- *Y*: CHERI.
- *mw10*: mantissa width is 10 bits.
- *rc1*: one additional bit is used in encoding, the out-of-bounds representable region is centered and guaranteed to be at least one quarter of the capability's length.
- *pc*: permissions are not encoded using one bit for each; rather, a compressed format takes advantage of dependencies between permissions.

2.2.5 Metadata description

The format in which capabilities are stored is shown in **Figure 2.3**. The parameters defined in the metadata are the following:

- *SDP*: software-defined permissions.
- *AP*, *M*: architectural permissions and operation mode.
- *GL*: global flag.
- *CT*: seal state of the capability (optional, not used in our CHERI implementation).
- *EF*: internal exponent format.
- L_8: top address reconstruction bit.
- *T*: top address bits.
- *TE*: top address exponent bits.
- *B*: bottom address bits.
- *BE*: bottom address exponent bits.

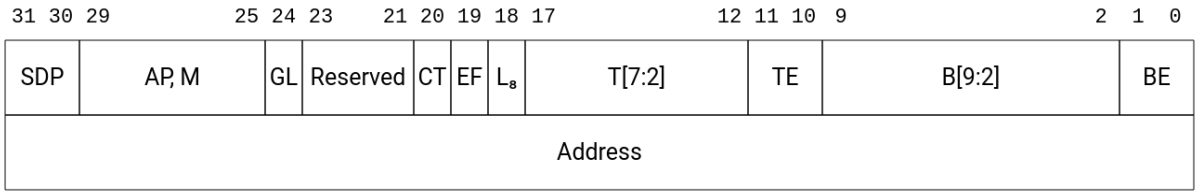


Figure 2.3: *RV32Y* capability register format. Image extracted from the CHERI spec document.

2.2.6 Bounds encoding and decoding

Capability bounds, like in floating-point, are encoded with a scheme where the larger the number to be encoded, the lower the precision. The exact bounds can be decoded using the stored pointer and the encoded bounds. If the pointer moves past the representable region, a different set of bounds will be decoded, and the capability will have its tag cleared, as explained in **Subsection 2.2.2**. The algorithm is very similar to the one proposed in the CHERI Concentrate [30].

The internal exponent, first value needed to encode bounds, will be defined as

$$E = size\_of(length[31:9]) - count\_leading\_zeros(length[31:9])$$

This means that if the length is smaller than 2^9, E will be zero. In that case, EF can be set to one and the fields TE and BE will be used to store more top and bottom bits (adding precision), while L_8 stores bit 8 of the length, i.e. $L[MW-2]$ for $MW = 10$. Otherwise, EF will be set to zero, with TE, BE and L_8 storing the exponent and trading precision for a larger dereferenceable region. Considering top (t) and bottom (b) actual bounds, **Table 2.1** shows how fields T and B are encoded into the capability's metadata.

E=0 ⇒ EF=1	*E>0 ⇒ EF=0*; *TE* and *BE* used to store exponent
(T & TE) = t[7:0]	*T[7:2] = t[E+7:E+2] + round* *round = 1 if t[E+1:0]!=0, else 0*
	L_8 = length's MSB
(B & BE) = b[9:0]	*B[9:2] = b[E+9:E+2]*
	(L_8 & TE & BE) = 24 - E

Table 2.1: Capability bounds encoding. Notations: *A[X:Y]* means bits X down to Y in vector A; *(A & B)* means the concatenation of vectors A and B.

The inverse process is to be described now. Considering the values in **Table 2.2**, **Code 2.1** and **Figure 2.4** show how to calculate different decoding parameters and obtain the capability's complete bounds. The first E bits of the decoded bounds are zero, which means that the dereferenceable region will be aligned to 2^E bytes.

Parameter	**Value**
XLEN	32
Exponent Width *EW*	5
Mantissa Width *MW*	10
CAP_MAX_E	$XLEN - MW + 2 = 24$

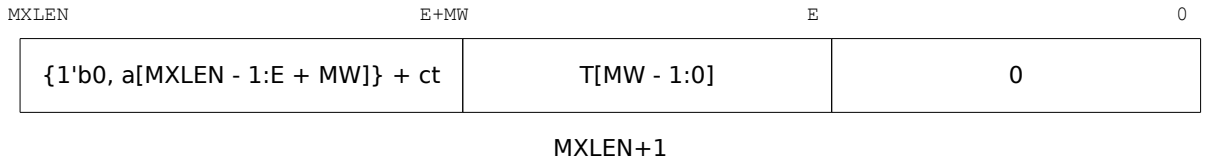
Table 2.2: *RV32Y* parameters for bounds encoding.

2.2.7 CHERI instructions

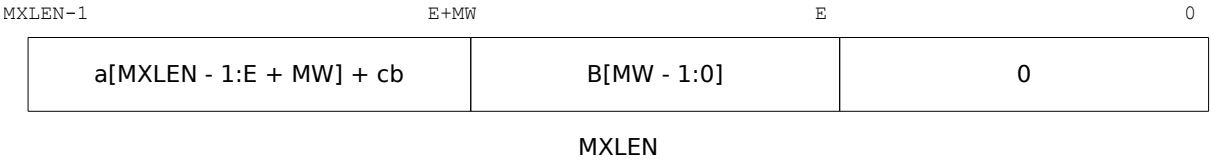
The new functionalities added by CHERI mean there is a list of new instructions related to capabilities. For greater detail about their encoding, please refer to the CHERI specification document.

```
if EF = 1{
    E = 0;
    T[1:0] = TE; //[1:0] means bits 1 downto 0 in the vector
    B[1:0] = BE;
    LCout = T[7:0] < B[7:0]; //1 if true, 0 if false
    LMSB = L8;
}else{
    E = 24 - (L8 & TE & BE); //& means vector concatenation
    T[1:0] = 0;
    B[1:0] = 0;
    LCout = T[7:2] < B[7:2]; //1 if true, 0 if false
    LMSB = 1;
}
T[9:8] = B[9:8] + LCout + LMSB;
A[9:0] = a[E+9:E]; //a is the current pointer
R[9:0] = B - 2^8;
ct = (T < R) - (A < R); //Each comparison returns 1 if true, 0 if false
cb = (B < R) - (A < R);
```

Code 2.1: CHERI parameter decoding pseudocode; *a* is the capability's current pointer. Values from **Table 2.2** are implicitly used.



(2.4a) Top bound decoding; note that *1'b0* is a single zero bit, concatenated to the following expression within braces.



(2.4b) Base bound decoding.

Figure 2.4: Capability bounds decoding. Images extracted from the CHERI spec document.

- Address set instructions:
  - *addy rd, rs1, rs2* - Copies the capability in *rs1* to *rd*, increments *rd*'s address by the value in *rs2*.
  - *addiy rd, rs1, imm* - Copies the capability in *rs1* to *rd*, increments *rd*'s address by the 12-bit immediate value *imm*.
  - *yaddrw rd, rs1, rs2* - Copies the capability in *rs1* to *rd*, sets *rd*'s address to the value in *rs2*.
- Metadata manipulation instructions:
  - *ypermc rd, rs1, rs2* - Converts metadata permissions values into a 32-bit bitmap. Any high bit in *rs2* will clear the corresponding bit in the bitmap. The capability in *rs1* is copied to *rd* and *rd*'s permissions are given the newly calculated values.
  - *ymv rd, rs1* - Copies the capability in *rs1* to *rd* bit by bit; *rd*'s tag will be the same as *rs1*'s tag.
  - *packy rd, rs1, rs2* - Moves the lower *XLEN* bits of *rs1* to the lower *XLEN* bits of *rd* and the lower *XLEN* bits of *rs2* to the higher *XLEN* bits of *rd*; sets *rd*'s tag to zero.
  - *yhiw rd, rs1, rs2* - Copies the capability in *rs1* to *rd*, replaces *rd*'s metadata with the value in *rs2*. It is a pseudoinstruction for *packy*.
  - *ybndsw rd, rs1, rs2* - Copies the capability in *rs1* to *rd*, sets *rd*'s base to the value in *rs1* and the length of the bounds to the value in *rs2*.
  - *ybndswi rd, rs1, imm* - Copies the capability in *rs1* to *rd*, sets *rd*'s base to the value in *rs1* and the length of the bounds to the 10-bit immediate value *imm*. Only *rd* = *rs1* is supported.
  - *ybndsrw rd, rs1, rs2* - Copies the capability in *rs1* to *rd*, sets *rd*'s base to the value in *rs1* and the length of the bounds to the value in *rs2*. The base is rounded down and the top is rounded up by the smallest margins possible to accommodate for the requested capability.
  - *yamask rd, rs1* - Sets *rd* to a mask that can be used to round addresses to set exact bounds for the nearest representable length to the value in *rs1*.
- Metadata reading instructions:
  - *ybaser rd, rs1* - Obtains *rs1*'s base bound and stores it in *rd*.
  - *ylenr rd, rs1* - Obtains *rs1*'s bounds' length and stores it in *rd*.
  - *ytagr rd, rs1* - Obtains *rs1*'s tag, zero-extends it and stores it in *rd*.
  - *ypermr rd, rs1* - Converts metadata permissions values into the same 32-bit bitmap as in *ypermc*. The bitmap is stored in *rd*.
  - *ytyper rd, rs1* - Obtains *rs1*'s capability type and stores it in *rd*.

  - *srliy rd, rs1, shamt* - Performs a logical right shift of *shamt* bits to *rs1*, filling the upper bits with zeros; the result is written to *rd*.
  - *yhir rd, rs1* - Obtains *rs1*'s metadata and stores it in *rd*'s address field. It is a pseudoinstruction for *srliy*.
- Capability comparison instructions:
  - *syeq rd, rs1, rs2* - Compares *rs1* and *rs2* and sets *rd* to 1 if these capabilities are exactly equal.
  - *ylt rd, rs1, rs2* - Sets *rd* to 1 if the tags of *rs1* and *rs2* are equal and the bounds and permissions of *rs2* are a subset of those of *rs1*.
- Capability load and store instructions:
  - *ly rd, offset(rs1)* - Calculates the address of the memory access by adding an immediate 12-bit offset to the value in *rs1* and uses this capability to authorize memory access. It then loads an aligned *YLEN*-bit value from memory and stores it as a capability in *rd*. If *rs1*'s tag is zero or if it does not have C permission, *rd*'s tag will be set to zero. If *rs1* does not have LM permission, an implicit *ypermc* is performed to clear W and LM permissions from *rd*.
  - *sy rs2, offset(rs1)* - Calculates the address of the memory access by adding an immediate 12-bit offset to the value in *rs1* and uses this capability to authorize memory access. It then stores the *YLEN*-bit capability in *rs2* in the calculated memory access. The stored capability's tag is set to zero if *rs2*'s tag is zero or if *rs1* does not have C permission.

2.2.8 Changes to base RISC-V instructions

Whenever a register is used as an address in load/store operations, all *YLEN* bits of the capability are used; the lower *XLEN* bits are used as the address, while the metadata is used to authorize memory access. Instructions that write back an address (such as *auipc*, explained below) write the full capability; for all others, the metadata is zeroed. If a capability with a new address is returned, the result is created using the semantics of *yaddrw*.

Load and store instructions authorized by a capability stored in a register *rs1* will raise an exception if *rs1* is *x0*, if its tag is not set, if a load is attempted without R permission or if a store is attempted without W permission. Load instructions, except *ly*, will clear the capability tag and metadata of the destination register. Store instructions, except *sy*, will clear tags from the memory locations that are being written.

The PC is also updated using the semantics of *yaddrw*. The *auipc* instruction, which stores the PC's address plus a 32-bit offset (formed by a 20-bit immediate that has been left shifted 12 bits) is unaltered in its functioning and jump instructions such as *jal* and *jalr* are unaffected.

For the branch instructions *beq* and *bne*, if *rs1* is in a higher physical location than *rs2*

(for instance, *rs1* being *x5* and *rs2* being *x4*), the encoding is reserved for future changes to the CHERI specification.

2.3 NOEL3

NOEL3 [23] is a 32-bit RISC-V processor core developed by Frontgrade Gaisler for space and safety-critical applications. It is described in VHDL and meant to be synthesized into FPGAs or ASICs. It is optimized for fault tolerance [31] and deterministic execution time, making it well suited for real-time systems in aerospace where timing predictability and reliability are paramount. NOEL3's features, which are explained below, make it a "clean" RISC-V core for deterministic, mission-critical systems, and its straightforward design greatly simplifies timing analysis, which is valuable for real-time certification; all of this makes NOEL3 a unique option for CHERI implementation.

2.3.1 Barrel architecture

Unlike superscalar or out-of-order CPUs, NOEL3 uses a "barrel" architecture; it schedules independent hardware threads in a round-robin fashion each clock cycle. Conceptually, if the core has N threads, the fetch stage gets an instruction from thread 0 in cycle 0, from thread 1 cycle 1, and so on. By cycle N, the fetch stage returns to thread 0, while the second pipeline stage is processing an instruction from thread *N-1*. This means that at every pipeline stage there is work being done for a different thread. A diagram showing this architecture is shown in **Figure 2.5**.

This design eliminates data hazards and the need for complex forwarding logic, since no two stages ever handle the same thread simultaneously; branch prediction is not needed either. Each thread's instructions progress through the pipeline in a time-sliced fashion, and one thread's instructions are retired each cycle (rotating among the threads). The result is fully predictable timing: the execution time of a thread's program does not depend on the particular mix of instructions in other threads, since pipeline stalls never occur. However, it also means that if some threads are idle, the processor's throughput is lower (since it still cycles through all contexts). In NOEL3, the number of threads is configurable (typically from 2 to 7).

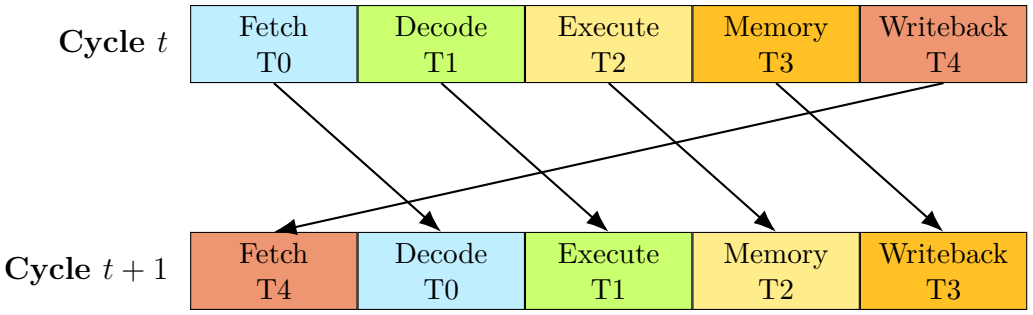


Figure 2.5: Barrel architecture diagram.

2.3.2 Pipeline design

Figure 2.6 shows a simplified view of the processor's pipeline. The figure highlights the main processing stages that an instruction goes through inside the core, from the early decode steps up to execution, memory access, and register write back. Although the actual implementation includes additional details and configurable options that are not covered here, this simplified representation is useful to understand the basic instruction data flow. For a complete description, the original Gaisler documentation should be consulted, in particular the NOEL3 user manual [32].

The main pipeline stages shown in the figure can be described as follows:

- *DEC1*: decodes the source register fields of the instruction, so the processor can determine which operands must be read.
- *DEC2*: performs the actual instruction decoding and generates the control signals required by the following stages.
- *EX0*: does forwarding from the write to read port in the register file.
- *EX1*: generates the next program counter and prepares load and store operations towards the DTCM.
- *EX2*: executes the operation in the ALU.
- *EX3*: handles bus access and writes results back to the register file.

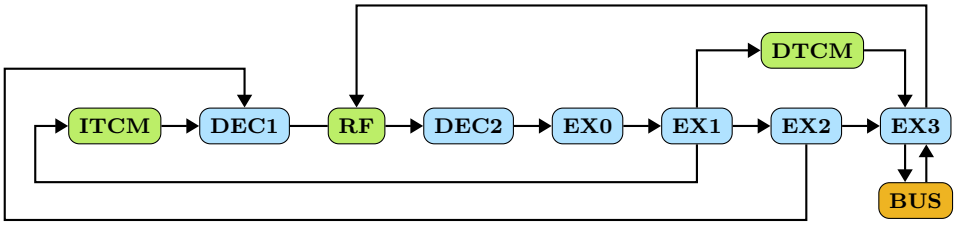


Figure 2.6: Simplified NOEL3 pipeline diagram.

The connection from *EX1* back to the ITCM represents the path used to provide the next program counter to the fetch stage. This allows the processor to continue fetching the correct instruction stream after sequential execution or a control flow change. The connection from *EX3* back to *RF* represents the write back path, through which the result produced by an instruction is written into the register file and becomes available for later instructions.

2.3.3 Tightly coupled memories

To achieve deterministic memory access, NOEL3 uses on-chip tightly coupled memories (TCMs) for instructions and data, rather than traditional caches. There is one instruction TCM (ITCM) and one data TCM (DTCM), each implemented as simple scratchpad RAM directly connected to the pipeline. Accessing an address in TCM always takes the same fixed number of cycles (a single cycle in the current configuration), with no chance of a

cache miss. This makes timing fully deterministic.

Using TCMs means lower and predictable access latency [33, 34], since there are no cache tags or lookup tables and access is guaranteed for real-time code or data (for example, critical routines can be placed in the ITCM so they never stall). The trade-off is that TCMs typically have smaller total capacity than a cache hierarchy and must be managed explicitly by software (data and instructions must be placed in them by linker configuration or code). NOEL3 also provides a separate off-chip (AHB) bus for accessing larger memory or peripherals, but those accesses are not time critical. Additionally, since NOEL3 has been designed for small microcontrollers, the entire software can be expected to fit in the TCMs in most scenarios.

2.3.4 Extensions and other relevant features

NOEL3 implements the *RV32I* base integer instruction set. In addition, it includes several standard RISC-V extensions, each of which is optional and can be turned on or off at design time. By selecting a subset of these extensions, NOEL3 can be tailored to the target application: for example, omitting floating-point functionality to save area, or including only necessary atomic operations for concurrency.

- *M*: integer multiplication and division.
- *A*: atomic instructions for concurrency.
- *F*: single-precision floating-point arithmetic.
- *C*: compressed instructions to reduce code size.
- *B*: bit manipulation instructions.
- *Zfh*: half-precision floating-point operations.
- *Zfa*: additional floating-point instructions.
- Debug module

In addition to the above, NOEL3 offers other features which are important for embedded systems and space applications. It adheres to the AMBA AHB bus protocol [35], simplifying integration with other peripherals or memory systems that use AHB. Memory blocks (in FPGA or ASIC) can use error-correcting codes (ECC), and register files can be triplicated with majority voting (Triple Modular Redundancy, TMR) if very high reliability is needed; these fault-tolerant features help NOEL3 operate correctly in the radiation-rich environment of space. The processor is also compliant with the GrLIB IP library conventions, which means it has a parameterized configuration (clock gate enable, pipeline flush on branch misprediction, etc.) and integrates with standard bus masters/slaves.

2.4 Concluding remarks

We have provided an explanation that includes the flaws present in PMP, the conventional RISC-V memory protection mechanism. This scenery suggests that an architectural ex-

tension such as CHERI is a plausible solution for these safety issues; its capability-based approach provides bounds and permissions enforcement at the object level, greatly minimizing unintended or malicious memory accesses during instruction flow. Understanding these concepts allows us to devise a design that integrates these features into the specific microarchitecture of NOEL3.

3

Method

This chapter describes the overall workflow followed in the project, from the initial study of CHERI and NOEL3 to implementation, verification and evaluation. The purpose is to connect the design details, verification work and evaluation results presented in the following chapters.

3.1 Overall workflow

The work was carried out as an iterative hardware design and verification process. First, the CHERI RISC-V specification was studied to identify the architectural state, instruction behavior, tag handling, exception behavior and memory access rules required by the capability model. In parallel, the baseline NOEL3 implementation was analyzed to identify where these requirements affected the existing pipeline, register file, CSR logic, tightly coupled memories, bus accesses and exception handling.

The information gathered during this analysis was then translated into a set of concrete microarchitectural changes. These changes were implemented incrementally in the processor, with small simulation tests used throughout development to debug instruction decoding, capability propagation, memory accesses, tag handling and exception generation. The implementation was not treated as a single monolithic change, but as a sequence of modifications that were repeatedly simulated and corrected before moving to larger tests.

Once the core CHERI functionality was implemented, the verification work followed two paths. The successful path used directed assembly tests, a Python preprocessing script and the existing NOEL3 simulation environment to compare the processor output against expected signatures. The attempted path explored random instruction testing through TestRIG and the RVFI-DII interface, but this flow was not completed successfully within the project. Both paths are part of the method because the second one influenced the verification-related logic added to the design, even though it was not used as the final source of functional correctness.

Finally, the verified design was evaluated by implementing different baseline and CHERI configurations on an FPGA target. These configurations were used to separate the overhead caused by CHERI from the overhead caused by other features such as memory size, PMP and the FPU.

3.2 Tool flow

Different tools were used for different parts of the workflow. Questa was used as the main simulation and debugging environment. It was the tool used for the directed verification flow, where assembly tests were loaded into the processor simulation, executed, and compared against expected signatures. This was the successful verification path used to support the functional correctness claims in this thesis.

GHDL and Verilator were used in the attempted TestRIG integration flow. The reason for using both tools was that TestRIG communicates with implementations through a C++ socket-based interface and expects a model that can interact with the RVFI-DII infrastructure [36, 37, 38]. Since NOEL3 is written in VHDL, it could not be connected to this C++ environment directly. GHDL was therefore used to translate the VHDL design into an intermediate Verilog representation. Verilator was then used to compile this generated Verilog into a C++ simulation model. Around this model, additional C++ logic was intended to handle the socket communication with TestRIG, inject DII instructions into the processor and return RVFI execution traces.

This GHDL and Verilator based flow was exploratory and was not the final verification method. It exposed the practical difficulty of connecting the modified VHDL processor to TestRIG, including version differences in the CHERI encoding, possible bugs in the RVFI-DII interface and the effort required to interpret failing random traces. For that reason, the project moved to the directed testing flow described in **Section 5.1**, while the TestRIG work remained an attempted verification approach described in **Section 5.2**.

Vivado was used for synthesis, implementation, area extraction and timing analysis on the FPGA target. The reports produced by Vivado were used to compare the baseline and CHERI configurations in terms of resource utilization and maximum achievable frequency.

3.3 Connection between method and thesis structure

The following chapters follow the same structure as the workflow. **Chapter 4** describes the processor modifications introduced to implement CHERI support. **Chapter 5** describes both verification paths, separating the successful directed tests from the attempted TestRIG flow. **Chapter 6** then uses the verified implementation to quantify the timing and hardware cost of adding CHERI to NOEL3.

4

Design

This chapter presents the design changes introduced in NOEL3 to support CHERI, together with the additional logic added for verification. The first part of the chapter describes the architectural and microarchitectural modifications required to handle capabilities in the processor. This includes changes to the datapath, register file, memory system, control and status registers, instruction decoding, execution stages, exception handling and tag storage (**Section 4.1**).

The second part of the chapter describes the logic added to expose the internal processor state through the RVFI-DII verification interface. This interface is used to observe instruction execution and compare the behavior of the modified processor against the expected architectural model (**Section 4.3**).

4.1 Modifications in NOEL3

Microarchitectural changes were introduced in NOEL3 to support the functionality required by CHERI. The original processor operates on 32-bit integer values and uses a conventional 32-bit program counter, register file and memory path. In the CHERI extended version, pointer values must carry additional architectural information. In this implementation, a capability is represented as a 64-bit value, composed of a 32-bit address and 32 bits of metadata, together with a separate validity tag. Therefore, supporting CHERI required structural modifications across the pipeline, the register file, the memory system, the control and status registers, the exception logic, and the memory access paths.

The added logic allows the processor to create, derive, manipulate and dereference capabilities. It also enforces the required protection checks when a capability is used to access memory or to update the program counter. These checks include validating the tag, checking that the requested address is within the bounds encoded in the capability, and verifying that the requested operation is allowed by the permission bits. If one of these checks fails, the processor raises a CHERI fault. In this implementation, all CHERI access failures are reported using a single exception cause, with *mcause* set to `0x1c`. The faulting address is stored in *mtval*.

CHERI support is integrated into the existing NOEL3 pipeline because the capability state affects the normal execution path of the processor. Capabilities are not handled as values processed by an external block. Instead, the address field, metadata field and tag move through decode, execution, memory access and write back together with the rest

of the instruction state. This affects explicit CHERI instructions, memory operations, control flow instructions, traps, CSR accesses and forwarding paths.

The instruction set was extended with operations for reading, writing, deriving and inspecting capabilities. These instructions update address fields, modify bounds, change permissions, inspect tags and move capability values between architectural locations; all of this is done according to the *RV32_LYmw10rc1pc* encoding (see **Subsection 2.2.4**). Some instructions produce ordinary integer results, while others produce capability results. This distinction is important because integer operations must not create valid capabilities. Only instructions with capability semantics may propagate or generate valid tags.

To support these changes, the datapath was extended in the pipeline stages marked in red in **Figure 4.1**. The instruction fetch path is highlighted separately in blue, since it is easy to confuse with the normal left-to-right pipeline flow. The numbered markers in the figure provide a direct mapping between the diagram and the subsections that follow.

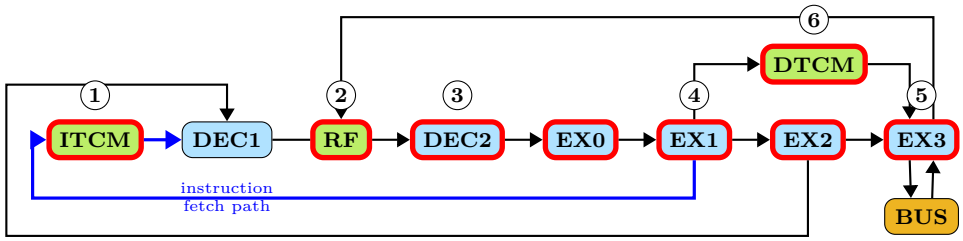


Figure 4.1: Simplified NOEL3 pipeline diagram with modified stages marked in red. The instruction fetch path is highlighted in blue, and the numbered markers map the diagram to **Table 4.1**.

Table 4.1 explains how the numbered markers in **Figure 4.1** correspond to the subsections in this chapter.

The register file, program counter, instruction and data memories, CSR logic, forwarding paths and bus interface were adapted to transport either ordinary 32-bit values or complete capability values, depending on the instruction being executed. The main differences between the original and modified design are summarized in **Table 4.2**.

4.1.1 Changes to instruction fetch

The first major modification affects the instruction fetch path. In the original NOEL3 design, the program counter is a 32-bit value used directly to fetch instructions from the instruction memory. With CHERI, the program counter is represented as a capability. Therefore, instruction fetch is no longer only an address generation operation. The processor must also preserve and check the metadata and tag associated with the program counter capability.

The execute permission check for instruction fetch is performed in EX0. At this point, the processor verifies that the program counter capability is valid and that execution from the selected address is allowed. If the program counter capability is not valid, if the execute

Marker	Related subsection	Meaning in the diagram
1	**Subsection 4.1.1**	Instruction fetch path. This includes the ITCM, the program counter capability path, the redirection path back to instruction fetch, and the execute permission check performed before the fetched instruction is allowed to proceed.
2	**Subsection 4.1.2**	Register file changes. The RF stores widened 64-bit capability values together with validity tags.
3	**Subsection 4.1.3**	Decode changes. DEC2 identifies CHERI instructions and generates the control information needed by the capability datapath.
4	**Subsection 4.1.4**	Execution-stage changes. The EX stages perform capability manipulation, bounds checks, permission checks and tag propagation.
5	**Subsection 4.1.5**	Exception and trap handling. Fault information is produced in the later execution stages and written to the trap-related CSRs.
6	**Subsection 4.1.6**	Memory and tag storage. The ITCM and DTCM are extended with tag storage, and memory operations preserve or invalidate tags according to the CHERI rules.

Table 4.1: Mapping between **Figure 4.1** and the design subsections.

Component	Original NOEL3	CHERI extended NOEL3
Register file	32 GPRs, 32-bit each	32 GPRs, 64-bit each, with tag storage
Program Counter	32-bit	64-bit, with one tag bit
CSR storage	32-bit	64-bit, with tag support where required
mtvec	32-bit trap vector	64-bit trap vector capability, with tag
mepc	32-bit return PC	64-bit return capability, with tag
mcause	32-bit trap code	Widened storage; `0x1c` used for CHERI faults
mtval	32-bit trap value	Widened storage; reports the faulting address
TCMs	32-bit words	32-bit words with parallel tag storage
Tag memory	None	One tag bit per 32-bit word, implemented using BRAM
External bus	32-bit accesses	Checked by CHERI logic, without tag preservation

Table 4.2: Key datapath structures before and after adding CHERI support to NOEL3.

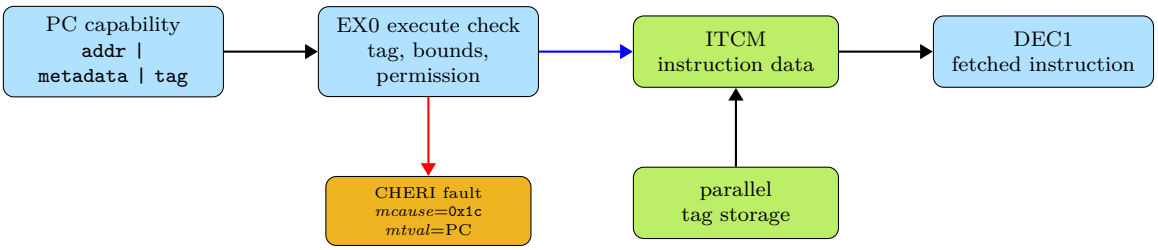


Figure 4.2: Instruction fetch with a program-counter capability. The fetch address is only used after the execute permission, bounds and tag checks have succeeded; failed checks generate a CHERI fault.

permission is not present, or if the address is outside the bounds allowed by the capability, the processor raises a CHERI fault. In this case, *mcause* is set to `0x1c`, and *mtval* receives the failing program counter address. This ensures that instruction fetch is subject to the same capability protection model as data memory accesses.

The ITCM was extended with parallel tag storage. Since the memory remains organized around 32-bit words, the implementation stores one tag bit per 32-bit word. A full capability spans two 32-bit words. When a capability is read from memory, the resulting capability tag is set only if the tags of both words are set. If either tag is clear, the loaded capability is invalid.

This approach forces capability storage to be aligned to two 32-bit words, which is inefficient in terms of flexibility because a valid capability must occupy the expected 64-bit aligned pair. However, BRAM storage is cheap enough for this design, and the approach simplifies the memory path. It also reduces the amount of logic required to reconstruct and validate a loaded capability. In the future, if support for less restrictive capability placement is required, this enforcement could be relaxed by modifying the memory wrapper and tag reconstruction logic.

The same principle is used for the DTCM. Capability data and tag information are stored separately, and tag updates are controlled by the hardware. The capability store instruction *sy* writes both the data and the corresponding tag information. In contrast, an ordinary store does not create a valid capability tag. If ordinary data overwrites one half of a previously stored capability, the pair of tags is no longer valid for reconstructing a tagged capability.

Control flow instructions also required changes to the fetch path. Instructions such as *JALR*, trap entry and trap return may update not only the address component of the program counter, but also the associated capability metadata. The fetch path must therefore use the updated hart context after these operations. This guarantees that the next instruction is fetched using the correct program counter capability, including its bounds, permissions and tag.

4.1.2 Changes to the register file

The register file was widened to store complete capability values. Each architectural register now contains a 64-bit value, formed by the 32-bit address and 32-bit metadata fields, together with separate tag storage. The tag is not part of the integer value itself. It is architectural validity information used to determine whether the associated bits represent an authorized capability. This separation is important because the processor must be able to store arbitrary integer values without allowing them to become valid capabilities.

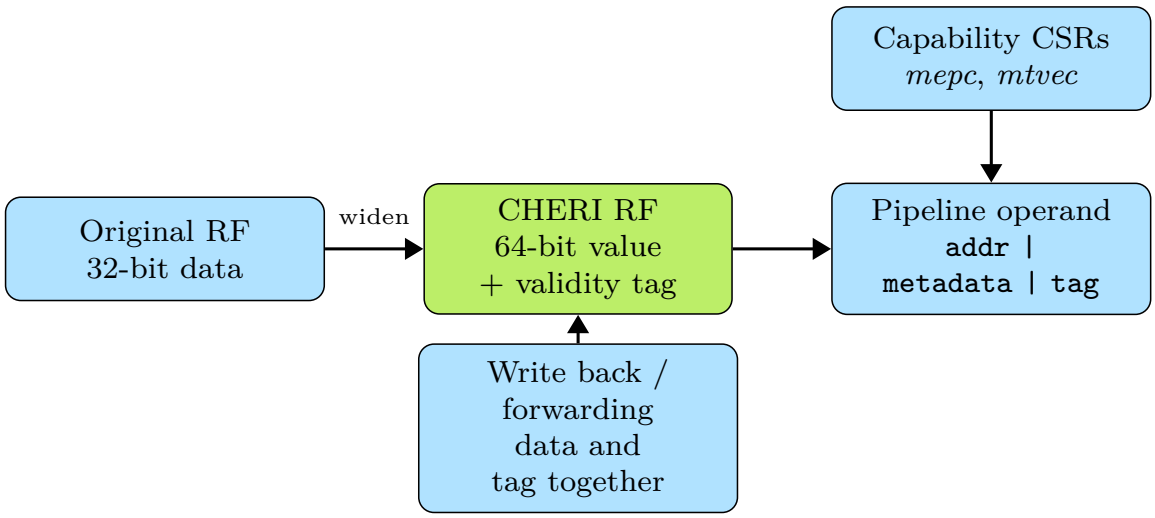


Figure 4.3: Register file and CSR capability datapath after the CHERI extension. Capability data and validity tags are moved together through operand read, forwarding and write back.

The register file storage follows the same general structure used in NOEL3, where two port RAM blocks are used as the basis for storage. In the CHERI extended implementation, the register datapath was doubled from 32 bits to 64 bits, while the tag storage was implemented using BRAM. The same base memory IP is reused for the register file, ITCM and DTCM. Each structure then has wrapper logic that implements the specific behavior required for register accesses, instruction memory accesses or data memory accesses.

This change has two important consequences. First, ordinary integer operations must still behave as expected for existing RISC-V code. Integer results may update the data portion of a register, but they must not create valid capabilities. Second, CHERI instructions must be able to consume and produce complete capability values. The design therefore distinguishes between instructions that operate on ordinary integer data and instructions that produce capability results. When an operation invalidates a capability, the destination tag is cleared even if the data bits are still written.

The forwarding and write back paths were also affected by this change. A forwarded capability must include the address field, the metadata field and the tag. Forwarding only the 64-bit data value would not be sufficient, since the consumer instruction must

know whether the value is still a valid capability. Similarly, write back must update the data and tag consistently. Otherwise, a register could contain mismatched data and validity information, which would break the capability protection model.

The same concept was applied to control and status registers. The relevant machine mode CSRs were extended to 64 bits plus a tag. For CSRs implemented as flip-flops, this extension is direct because the storage is explicit in logic. For CSRs stored in BRAM, the extension follows the same approach as the GPR register file, since they share the same widened storage structure.

The *mtvec* CSR stores the trap vector as a capability. Its address field provides the trap handler entry address, while its metadata and tag provide the authority used when control is transferred to the handler. The *mepc* CSR stores the return program counter as a capability. On trap entry, the interrupted program counter capability is saved into *mepc*. On *mret*, the program counter capability is restored from *mepc*. This is required because trap entry and trap return are control flow operations and must preserve capability metadata.

The *mscratch* and *mtval* CSRs were also extended to hold capability sized values. In the case of *mtval*, the implementation stores the failing address when a CHERI fault occurs. CSRs whose architectural meaning is an integer cause or status field still use their data bits to encode the corresponding value, even though the underlying CSR storage has been widened.

4.1.3 Changes to the DEC2 stage

The DEC2 stage was extended to recognize the new CHERI instructions and generate the control signals required by the added datapath logic. In the original design, decode mainly identifies the instruction class, selects operands, and configures the existing ALU, memory and CSR paths. With CHERI, decode must additionally determine whether an instruction consumes capabilities, produces capabilities, reads tags, writes tags, modifies metadata, or performs a checked memory access.

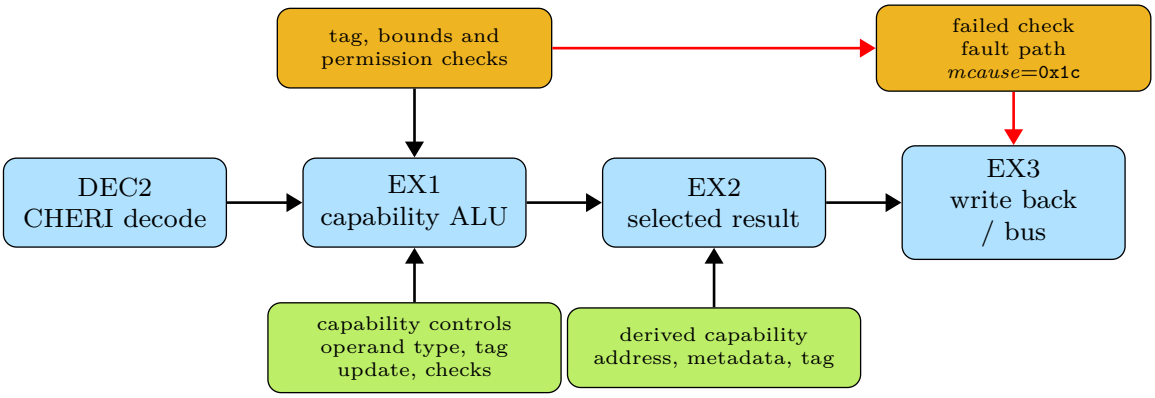


Figure 4.4: Decode and execution flow for CHERI operations. DEC2 marks the required capability behaviour, while the execution stages perform the checks, derive the result and route either the capability result or the fault information.

The CHERI decode logic identifies operations that manipulate bounds, permissions, tags and address fields. It also selects whether the result should be treated as an integer value or as a capability value. This distinction is important because the destination tag must be updated according to the semantics of the instruction. For example, an instruction that derives a valid capability from another valid capability may preserve the tag, while an instruction that produces a plain integer must not create a valid capability tag.

The decode stage also generates control information for exception handling. Instructions that dereference capabilities, update the program counter, or access capability metadata require additional checks in later stages. DEC2 therefore marks which operands must be interpreted as capabilities and which checks must be performed before the instruction can commit its result. This allows the new functionality to be integrated into the existing pipeline without changing the general flow of instruction processing.

Another important responsibility of decode is to prevent ambiguity between integer and capability interpretations of the same register contents. Because the register file stores 64-bit values and tags, the same physical storage can hold either ordinary integer data or a valid capability. The instruction semantics determine how the operands are interpreted. The decode signals therefore control whether the later stages use only the integer portion of the operand or the full capability, including metadata and tag.

4.1.4 Changes to the execution stages

The execution path required changes across several pipeline stages. The main purpose of these changes is to perform capability checks, derive new capability values, and route complete capability objects through the pipeline. This affects arithmetic and logical operations, memory accesses, CSR accesses, jumps, traps and write back.

As described in **Subsection 4.1.1**, EX0 performs the execute check for the program counter capability. This stage verifies that the processor is allowed to fetch and execute the instruction at the selected address. If the check fails, the instruction fetch is treated as a CHERI fault, *mcause* is set to `0x1c`, and *mtval* is updated with the failing program counter address.

In EX1, the operand selection multiplexers were extended so that they can select complete capability values rather than only 32-bit operands. Load and store operations were also modified to propagate address, metadata and tag information. Before a memory access is issued, the source capability is checked. If the source register tag is clear, the access is invalid because the operand does not represent a valid capability. If the tag is set, the processor verifies that the requested address is within bounds and that the permission bits allow the operation. If one of these checks fails, the processor raises a CHERI fault and stores the failing memory address in *mtval*.

EX1 also contains the execution logic used for CHERI capability operations, including the bounds encoding logic. This logic derives updated capabilities from source capabilities, modifies address fields, updates permissions, computes representable bounds, and generates the metadata associated with the result. The capability logic must enforce the

rule that derived capabilities cannot increase authority. A derived capability may reduce authority, for example by narrowing bounds or removing permissions, but it must not create authority that was not present in the source capability.

The result tag is generated from the operation semantics and from the validity of the source operands. A capability derived from an invalid source cannot produce a valid result. Similarly, an operation that attempts to generate an unrepresentable or unauthorized capability clears the result tag. This keeps the validity of the result tied to the same checks that generate the data and metadata fields.

The memory interface required special handling because the connection between the pipeline and the memories remains 32 bits wide. A complete capability is 64 bits plus tag information, so capability load and store operations require two iterations. In the first iteration, the metadata half is read or written. In the second iteration, the address half is read or written. The tag information is handled in parallel through the tag storage associated with the corresponding memory words.

The final execution stage, EX3, handles bus access and register file write back. This stage was extended so that complete capabilities can be written back to the register file, including their tag. It also handles accesses to the external bus when an operation is not served by the local memories. External accesses are still checked using the CHERI permission, bounds and tag logic, so the processor can prevent invalid reads, writes or instruction accesses to external addresses.

The external bus does not preserve capability tags. Therefore, capability load and store instructions such as *ly* and *sy* cannot rely on external memory or peripherals to store and return valid tags. The access itself is still protected by the capability used to address the external region, but the external target is treated as not having tag storage.

4.1.5 Exception and trap handling changes

CHERI affects exception and trap handling because the program counter is no longer only a 32-bit address. On trap entry, the processor must preserve enough information to return to the interrupted context as a valid capability. Therefore, *mepc* was extended to store the return capability and its tag. Similarly, *mtvec* was extended to hold the trap vector capability used to enter the trap handler.

This change is required for correctness. If the trap handler were entered using only a raw address, the processor would lose the capability metadata associated with the control flow authority. Likewise, if *mret* restored only an address, execution after the trap would continue without the correct bounds, permissions or tag state. By representing both the trap vector and return PC as capabilities, the processor preserves the CHERI execution model across exceptions.

The exception logic was updated to report CHERI access failures using a single CHERI fault cause. In this implementation, *mcause* is set to `0x1c` for invalid execute, read and write attempts caused by failed tag, bounds or permission checks. The faulting address is reported through *mtval*. Using a single CHERI fault cause simplifies the exception logic

and the software visible fault model. The drawback is that software cannot distinguish the exact reason for the failure directly from *mcause*. For example, tag, bounds and permission failures all use the same cause. However, this approach is sufficient for detecting illegal capability use and for reporting the address that caused the failure.

Trap entry and trap return also required changes in the CSR datapath. When a trap is taken, the current program counter capability must be saved into *mepc*. When the processor executes *mret*, the program counter capability is restored from *mepc*. Similarly, the trap target is obtained from the capability stored in *mtvec*. This keeps exception handling compatible with the capability control flow model and prevents traps from becoming a path through which capability metadata is lost.

4.1.6 Memory and tag storage

The memory system was extended to store tags alongside data [29]. Since tags are not ordinary software visible bits, they must be protected from direct manipulation by regular stores. Software can store the data portion of a capability, but it cannot forge the validity tag. The tag is set only when the hardware performs a capability operation that stores a valid capability. Ordinary stores do not preserve the tag information required to reload a valid capability.

The final implementation stores tags in BRAM instead of flip-flops. This is important because tag storage scales with memory size. In this design, one tag bit is stored per 32-bit memory word. Since a full capability spans two 32-bit words, a capability read produces a valid tag only if both corresponding memory tags are set. If either tag is clear, the loaded capability is invalid. Modifying only one half of a capability is therefore enough to invalidate the capability when it is later loaded.

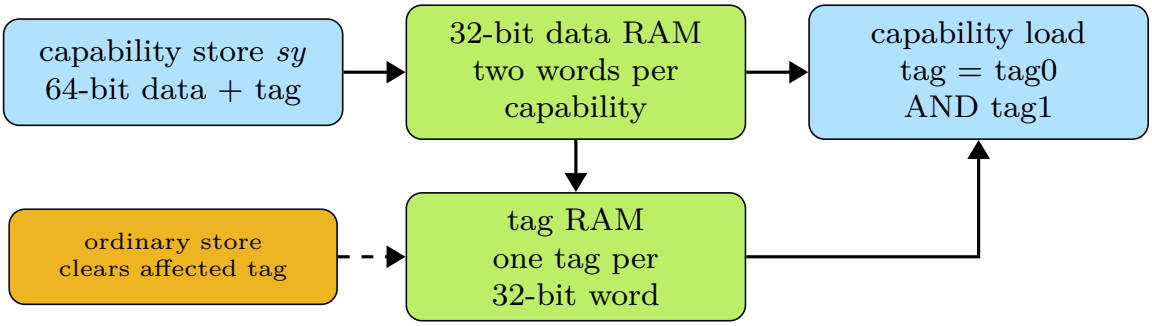


Figure 4.5: Memory and tag storage for capabilities. Capability data is stored as two 32-bit words, while the corresponding tag bits are stored in parallel tag memory and combined on capability loads.

This approach is more conservative than storing only one tag per complete 64-bit capability word. It makes invalidation straightforward when only part of a capability is overwritten. Since each 32-bit half has its own tag, the hardware can detect whether both halves were produced by capability stores. If only one half was preserved, the reconstructed capability is not valid. This avoids relying on additional bookkeeping to detect

partial capability writes.

In some implementations, storing one tag per 32-bit word may also help synthesis. If the target FPGA supports efficient memories with 33-bit wide entries, the synthesis tool may be able to map the 32 data bits and the associated tag bit into the same physical memory structure. This kind of fusion is less likely with a layout that stores one tag for every two memory positions, because the tag no longer naturally belongs to a single memory word. Therefore, even though the selected approach uses more tag bits, it can be simpler and may be more efficient on some FPGA targets.

The base memory IP is reused for the register file, ITCM and DTCM. The wrappers around this memory implement the different access rules required by each structure. For the ITCM and DTCM, this means reading and writing tags in parallel with memory data. For the register file, it means preserving one tag per architectural register. Since all these memories are built from the same two-port memory design, they also inherit the same EDAC protection. This keeps tag handling consistent without duplicating separate storage designs for each block.

External memory and peripheral accesses are different because the processor cannot assume that external targets implement CHERI tag storage. The processor still checks whether the access is allowed by the source capability, but tag preserving capability storage is only guaranteed for memories that implement the tag mechanism. Capability values that must remain valid after being stored and reloaded should therefore be placed in memories with tag support.

4.2 Optimizations

The first synthesis of the CHERI-extended processor usedaround three times the resources of the original NOEL3 design. Most of the increase came from the CHERI tag storage and from the capability bounds encoding logic in EX1.

The first issue was the initial implementation of tag storage. In the early version, CHERI tags were stored using flip-flops. Although each tag is only one bit, the number of tags scales linearly with memory size. Since the design stores one tag per 32-bit word, large memories require a correspondingly large number of tag bits. Implementing those bits as discrete flip-flops generated a large amount of sequential logic and significantly increased the resrouce utilization of the design.

This was corrected by implementing the tag storage using BRAM. Tag storage is naturally memory like because tags are indexed by address, read together with data, and updated on stores. Using BRAM substantially reduces the resource cost compared with a flip-flop based implementation and makes the design scale better with memory size. The same reasoning applies to the register file tag storage, where memory based storage is a better match for the existing NOEL3 structure than adding large amounts of explicit register logic.

The second major resource utilization issue was located in EX1, where the capability

execution logic performs bounds encoding. The original implementation encoded bounds into the CHERI metadata format using an iterative loop. Although this was compact in the VHDL source code, it synthesized into a large amount of combinational hardware. A loop in VHDL does not imply sequential execution unless it is explicitly placed in sequential logic. In this case, the loop was unrolled into combinational logic, producing a large circuit with high logic cost.

The bounds encoding function was rewritten to avoid the iterative implementation. The new version computes the required encoding more directly, reducing the amount of generated combinational logic. This significantly reduced the size of the CHERI execution logic in EX1 and made the design more suitable for synthesis. This optimization is especially important because the bounds encoding logic is used by capability manipulation instructions and is therefore part of a timing sensitive execution path.

No meaningful timing comparison is available for the unoptimized implementation because the implementation setup was not working before these changes were applied. However, the synthesis results were enough to identify the original implementation as impractical. The EX1 logic alone was approximately 1.5 times the size of the complete baseline NOEL3 processor. Even without final timing data, such a large amount of combinational logic would be expected to increase routing complexity and negatively affect the maximum achievable clock frequency.

These optimisations show that adding CHERI support is not only a matter of extending architectural state. The implementation of capability metadata and tag storage has a direct impact on resource utilization and timing. In particular, tag storage should be implemented as memory rather than as distributed flip-flops, and bounds encoding must be written carefully to avoid excessive combinational logic. Without these optimisations, the resulting design would be significantly less practical for FPGA implementation.

4.3 Microarchitectural changes for verification

Our verification strategy initially involved using TestRIG [36], a RISC-V framework which is compatible with CHERI. In order to support its usage, NOEL3 was extended with the RVFI-DII interface, which consists of two complementary components: the Direct Instruction Injection (DII) mechanism provides a test input, while the RISC-V Formal Interface (RVFI) produces an execution trace output.

When a DII instruction trace is received, the corresponding 32-bit word is inserted directly into the fetch queue on each cycle. This mechanism bypasses the conventional program counter and branch selection logic, meaning that, during injection mode, NOEL3 does not compute the next program counter internally but instead consumes instructions sequentially from the injected stream. This approach guarantees that the core can correctly process externally supplied instruction sequences as required by TestRIG.

On the retire side, the existing NOEL3 trace infrastructure is extended to conform to the RVFI format. For each retired instruction, the processor generates a set of RVFI signals (known collectively as an execution trace), including relevant architectural values; these

are redirected to the external interface.

Unfortunately, we were not able to complete the communication between our core and TestRIG, which will be explained in **Section 5.2**.

4.4 Concluding remarks

This chapter has described the design changes required to integrate CHERI support into NOEL3. The processor datapath, register file, CSR storage, memory system, exception logic and execution stages were extended to handle capability values, metadata and tags. The chapter has also explained tag storage optimizations and RVFI-DII related changes added for verification. Together, these modifications define the microarchitectural basis of the CHERI-extended NOEL3 implementation evaluated in the following chapters.

5

Verification

In parallel to the design of the modified processor, a verification strategy was defined using two approaches. Firstly, we attempted to use TestRIG (introduced in **Section 4.3**) for random instruction testing. Due to several difficulties, specific directed tests were executed instead to check that the implementation correctly performs the CHERI instructions listed in **Subsection 2.2.7**.

5.1 Specific testing (successful approach)

We created directed tests to verify individual CHERI instructions and selected corner cases. This manual approach is slower than automated test generators (see **Section 5.2**), but it gave direct control over the executed instruction sequences, the expected architectural state and the memory locations used to store observable results.

Each test is written as a RISC-V assembly file. During execution, the test stores selected results to memory. These values are later extracted from the simulated processor and compared against a golden signature generated from markers in the assembly source. This makes the tests self checking at the signature level, while still allowing the compiled code and the simulation waveforms to be inspected when a mismatch appears.

Because this environment does not use a CHERI aware assembler, capability instructions cannot be assembled directly from their mnemonics and operands. Instead, they are translated into raw instruction words before compilation. This translation is performed by a Python script, as shown in **Code 5.1**. The process also generates the expected signature file from special markers written in the assembly source, as illustrated in **Code 5.2**. The overall specific testing process is shown in **Figure 5.1**.

In order to automate the process, the Python script takes a RISC-V assembly input file (`.S`) and translates CHERI instructions into their opcode while leaving standard RISC-V instructions unaltered. It also creates a signature file (`.sig`) with the expected values, provided that these are contained in the input file after a special comment marker (`//`). Other comments, starting with a hashtag (`#`), are discarded, as well as empty lines.

Tests were divided into three different categories: positive, negative [39] and stress. Due to the fact that some instructions' validity can only be checked by executing others, it is necessary to assume that all helper instructions used by the test work correctly. For example, in order to check that a capability is correctly created, the tag must be checked, which means assuming that the instruction used to read the tag is correctly executed.

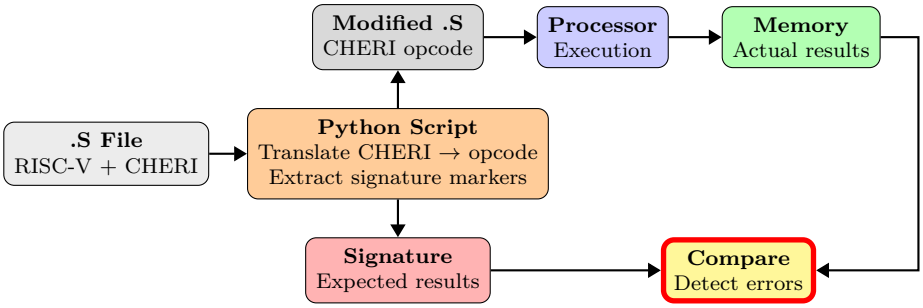


Figure 5.1: Specific testing process diagram.

Specific registers from the RISC-V naming convention [40] are used in these tests. Register *a0* is a root capability (see **Subsection 2.2.1**) that holds the address used to sequentially store data in memory. Register *t0* stores the loop iteration variable. Register *t1* stores the capability under test. Registers *t2*, *t3* and *t4* store auxiliary values, which may be capabilities or ordinary integers depending on the test. Register *t5* stores integer values and *t6* stores read capability tags. For greater detail about CHERI instructions, please refer to **Subsection 2.2.7**.

```
in_loop: //steploop 0 96 4
    addiy a0, a0, 4
    yaddrw t1, t1, a0
    sw t0, 0(t1) #To memory
    addi t0, t0, 4
    bne t0, t5, in_loop
```

```
in_loop:
    .4byte 0x0045251b
    .4byte 0x0ca31333
    sw t0, 0(t1)
    addi t0, t0, 4
    bne t0, t5, in_loop
```

Code 5.1: RISC-V assembly code before (left) and after (right) the Python script. CHERI instructions are translated into their raw opcode.

5.1.1 Positive tests

Positive tests are carried out to ensure that the implementation correctly performs CHERI instructions. These tests execute valid instruction sequences, without attempting malicious actions, and check that capabilities are correctly derived, encoded, decoded, manipulated and used for dereferencing.

Address manipulation

The pointer in *a0* is manipulated with *yaddrw*, *addiy* and *addy*, storing the address after each operation. Validity is checked using *ytagr*, and the resulting tag is stored in memory. Afterwards, *ymv* is executed and the copied capability address and tag are stored. The following step checks metadata manipulation. Two 32-bit integer values are written into *t1* using *yhiw*. The address of *t1* is stored, followed by the tag, which should have been cleared. The metadata in *t1* is then read using *yhir* and stored.

```
sw t1, 0(t1) // 0x80000084
in_loop: // steploop 0 96 4
    addiy a0, a0, 4
    yaddrw t1, t1, a0
    sw t0, 0(t1) #To memory
    addi t0, t0, 4
    bne t0, t5, in_loop
```

```
80000084
0
4
8
...
96
```

```
80000084
0
4
8
...
80000090
```

Code 5.2: Signature file generation and comparison example. Markers in the code on the left lead to the signature file in the middle, which is compared to values read from memory (listed in the code at the right).

Bounds modification and reading

The two instructions that set bounds (*ybndsw* and *ybndswi*) are used to set *t1*'s pointer as its own base and to use a register-stored value or an immediate as the length of its dereferenceable region. The new values for base and length are then stored. The correct execution of *ybndsrw* is then checked, comparing the expected rounded values with the actual ones.

Architectural permission clearing

Each permission is removed individually in *t1* by means of the *ypermc* instruction; the new set of permissions is read with *ypermr* and compared with the expected value, taking into account that clearing certain permissions might clear others (see **Subsection 2.2.3**). 6 tests are executed here, one for the clearing of each permission.

Rounding mask calculation, type reading, subset and equality checks

This test involves executing *yamask* and comparing results with expected values, testing different exponent alignments. Following this, *ytyper* is executed to check that capabilities are unsealed: this is expected since we are not implementing the sealing extension of CHERI. Another tests carried out are those of capability subset (*ylt*) and equality (*syeq*); for these, *t1* is compared with *a0* in two scenarios: the former being a copy of the latter and the former having its permissions cleared.

Capability load and store

Finally, in order to ensure *ly* and *sy* work correctly, *a0* is loaded to memory and then stored in *t2*, both operations being authorized by *a0* itself. Equality between the two capabilities is then examined with *syeq*.

5.1.2 Negative tests

Negative tests attempt to exploit possible flaws in the implementation. Taking the role of a hypothetical attacker, the aim is to increase capability bounds and permissions, forge a capability, dereference outside decoded bounds, use an invalid capability and alter data within a capability.

Any instruction that violates CHERI principles should raise an exception that triggers a custom trap handler (for simplicity, we will say that an instruction traps whenever this takes place). This handler retrieves the faulty instruction's address (using *csrr* to read the *mepc* CSR), stores it into memory as an expected result, and resumes test execution at the following instruction.

Dereferencing outside bounds but within representable region

A copy of *a0* is made in *t1*, after which *ybndswi* sets a new length and *addiy* moves the pointer one byte past the top bound; this is done to test whether bounds are established exactly as expected. A standard RISC-V store instruction (*sw*) is then attempted using *t1* as the authorizing capability, which should trap. A common vulnerability associated to this test is that of buffer overflows.

Moving the pointer past the representable region

The pointer is moved beyond the representable region. This would result in different decoded bounds in the new capability, which causes the tag to be cleared. This new value of the tag is stored in memory.

Architectural permission clearing

This test removes permissions individually in the same manner as the positive tests. After clearing each permission, instructions are executed attempting to make use of it, as well as others that have been cleared as a result of dependencies; all instances should trap.

- *R* and *W* are checked using *lw* and *sw* respectively
- *C* is checked using *ly* and *sy*
- *X* is checked by trying a jump to another instruction using *jalr*
- *ASR* cannot be checked in our implementation due to the absence of the user-thread *utidc* CSR
- *LM* is checked using the following steps: a capability with *W* permission is stored in memory and then loaded by an authorizing capability that lacks *LM*. This loaded capability should have *W* cleared, which will cause an *sw* instruction to trap.

Invalid address modification

The address in *t1* is modified using a non-CHERI instruction, such as *addi*. This clears the tag, which should render the capability unable to authorize memory access; *lw*, *sw*, *ly* and *sy* are attempted, all of which should trap. This test verifies whether the implementation can detect cases of pointer corruption.

Reading metadata or packing data and attempting to use the destination register as a capability

Memory accesses with *lw*, *sw*, *ly* and *sy* are attempted with *t1* as the authorizing capability, after it has been used as the destination register of *packy* in one test and *yhir* in another; all attempts should trap. This test, as well as the previous one, can be linked to use-after-free errors; clearing the tag essentially means to mark the pointer as invalid.

Attempting to escalate bounds

Two instructions such as *ybndsw* or *ybndswi* are used consecutively, the second one establishing a greater length. This request to increase bounds violates CHERI's monotonicity principle, which leads to the tag being cleared; this tag is read and stored to memory. This test highlights CHERI's per-object protection against privilege escalation, a property that gives it a substantial advantage over mechanisms such as PMP.

Invalid metadata modification

Encoded metadata bounds are manually modified in *t1*. In order to achieve this, the capability is stored to memory, where an *sb* instruction stores a byte that changes the region containing the top bound. This action should invalidate the capability, which is then loaded and use as the authorizing capability in memory accesses, all of which should trap. The new capability is also compared with the starting one using *syeq*, expecting them to be different.

5.1.3 Stress tests

Stress tests are applied to evaluate the robustness of the processor and to find less obvious weaknesses. They rely on combinations of positive and negative tests, performing successive executions of instructions with different values. The goal is to find corner cases where the implementation might behave incorrectly.

Capability derivation

Being a valid capability, *t1* is copied into *t2*, then *t2* into *t3* and successively. The equality of these capabilities is then checked with *syeq* to test that derivation remains correct after continuous repetition.

Pointer movement

The address of *t1* is moved within its representable region after the capability has been given specific bounds. Starting below the base and ending above the top, memory access is attempted for each instruction word. In the dereferenceable region, an integer value representing the offset from the base should be stored to memory; outside of these bounds, the instruction should trap.

Architectural permission clearing

The test in this case is the same as in **Subsection 5.1.2**, but illegal accesses are performed multiple times to verify our implementation's robustness.

Attempting to escalate bounds

An iteration is made across different bounds length values for *t1*, ranging between 2^2 and 2^{30} bytes; on each iteration, the length is doubled. Four cases are considered for each length:

- Lowering base while keeping length constant (therefore moving the top down)
- Raising top while keeping length constant (therefore moving the base up)

- Lowering base while keeping top constant (therefore increasing length)
- Raising top while keeping base constant (therefore increasing length)

Each of these cases consists of another loop, this time through different offsets by which bounds are modified. These offsets range between 1 byte and the capability's current length, also doubling every iteration. For clarification, the code for one of the cases is shown in **Code 5.3**.

```
#Attempt to decrease bottom bound
li t4, 1 #Offset from bound
bounds_len_bottom:
    ymv t1, t2 # Restore t1's validity
    sub t3, t1, t4 # Calculate new base
    yaddrw t1, t1, t3
    ybndsw t1, t1, t5 # Tag should be cleared
    ytagr t6, t1 # Read tag
    addiy a0, a0, 4
    sw t6, 0(a0) # Store in memory
    slli t4, t4, 1 # Multiply offset by 2
    bgt t4, t5, bounds_len_bottom
```

Code 5.3: Excerpt from bounds escalation stress test.

5.1.4 Test coverage

Our specific tests cover the execution of instructions in the base CHERI set, as well as the enforcement of CHERI properties (using said instructions correctly or maliciously). They have been designed with the aim of achieving a functional, CHERI-aware implementation. Due to the tests being manually produced, we were not able to verify all possible corner cases; this could be achieved with a successful use of random-instruction testing environments (see **Section 5.2**).

5.1.5 Test execution on the processor

The tests were executed using a simulation environment derived from an existing Gaisler verification environment for NOEL3. This already provided the infrastructure needed to load programs through the debug module, control processor execution and observe the final state of the test. The original environment was adapted so that the CHERI tests could be compiled, loaded into the local memories, executed and checked against the generated signatures.

After the Python preprocessing step, the processed assembly file is compiled using the provided `link.ld` linker file. This linker file separates the instruction and data sections so that they are placed in the ITCM and DTCM respectively. It also defines the *tohost* section and symbol, and therefore the concrete *tohost* address is determined by the linker.

The hardware block that implements the *tohost* functionality is called *htif*, and it is connected to the deterministic bus. When the program writes to the selected *tohost* address, the testbench detects that the program has finished.

Once the test is compiled, two auxiliary files are generated. The first one is a `.dump` file, which is used to inspect the compiled program and relate executed instructions to their addresses. This is especially useful when debugging traps, jumps or unexpected signature mismatches. The second one is a `.srec` file, which is used by the testbench to load the processor memories.

The `.srec` file is loaded by the testbench through the debug module. The debug module uses the housekeeping thread to write the program and data into the corresponding memories. After the memory image has been loaded, the testbench initializes all integer GPRs, all floating point registers, and the CSRs that are stored in BRAM. These CSRs are *mtvec*, *mscratch*, *mepc*, *mcause*, *dpc* and *mtval*. This explicit initialization is needed because these structures are implemented in BRAM and are not initialized by reset. The testbench then sets the processor entry point to the start address of the program and resumes execution.

The program executes until it writes to the *tohost* address. At that point, the testbench halts the processor and extracts the signature range from memory. The signature range starts at the *begin_signature* symbol. This symbol is aligned and reserves the memory space used for the signature. The *sig_end_canary* symbol is placed after this reserved region and stores a known value, as shown in **Code 5.4**.

```
.align 4
.global begin_signature
begin_signature:
  .space 0x3C000

.global sig_end_canary
sig_end_canary:
  .word 0xa3096f5c
```

Code 5.4: Signature memory range markers.

The extracted memory contents are written to a signature output file, which is then compared against the golden signature generated by the Python script. If both files match, the test passes. If they differ, the mismatch identifies the memory position where the observed execution diverged from the expected result, allowing us to locate and fix errors. This process is graphically explained in **Figure 5.2**.

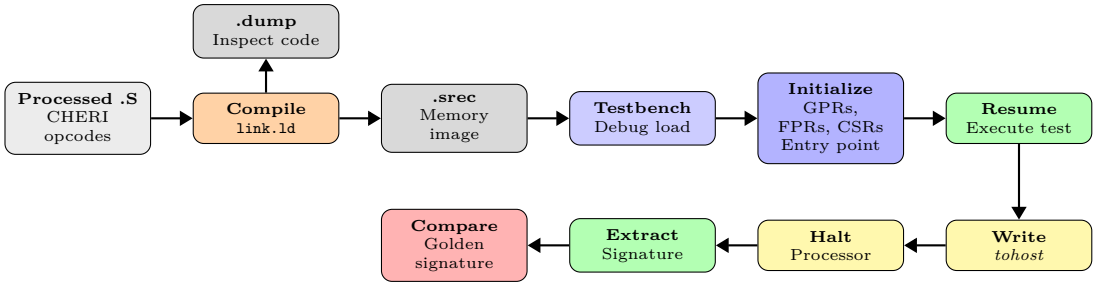


Figure 5.2: Test execution flow in the NOEL3 simulation environment.

5.2 Random testing (attempted approach)

Automated test generation [41, 42, 43], including random instruction generation [44, 45, 46], is an efficient method to carry out exhaustive processor verification. An example of such environments is TestRIG.

TestRIG operates by interacting with two types of entities: a *VEngine* (verification engine) and implementations. The *VEngine* feeds random DII instruction traces (format in **Subsection 5.2.1**) directly into each implementation's pipeline, which returns an RVFI execution trace (format in **Subsection 5.2.2**) and sends it to the *VEngine* for examination. Execution traces are then compared and differences are listed, which is useful when debugging. The entire process is shown graphically in **Figure 5.3**.

Interaction between the *VEngine* and the implementations is carried out by means of a bidirectional C++ TCP socket. TestRIG requires 8 MiB of memory accessible at address *0x80000000*, with all other addresses returning an access fault, and must support resetting to a known state upon injection of a reset DII trace. This known state includes zeroed registers, known default CSR values and zeroed memory [37].

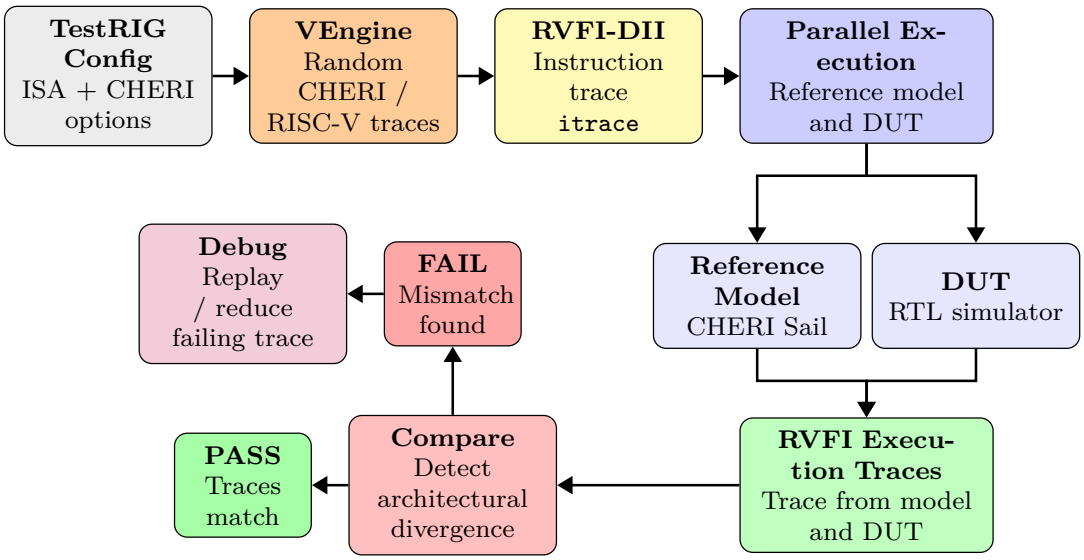


Figure 5.3: TestRIG workflow diagram.

5.2.1 DII trace

- *rvfi_cmd* (1 byte): trace command. Currently, two values are defined:
  - *EndOfTrace = 0*: reset the implementation, including registers, memory and PC, to *0x80000000*.
  - *Instruction = 1*: execute the instruction in *rvfi_insn*.
- *rvfi_time* (2 bytes): time at which the instruction is to be injected. The difference between this value and the previous instruction time gives a delay before injecting this instruction. This can be ignored for models, but gives repeatability for implementations while shortening counterexamples.
- *rvfi_insn* (4 bytes): instruction word. The lower 16 bits may decode to a 16 bit compressed instruction.

5.2.2 RVFI trace

1 byte information elements:

- *rvfi_intr*: trap handler, set for first instruction in trap handler.
- *rvfi_halt*: halt indicator, marks last instruction retired before execution halt.
- *rvfi_trap*: trap indicator: invalid decode, misaligned access or jump to misaligned address.
- *rvfi_rd_addr*: write register address; 0 if not used, otherwise set as decoded.
- *rvfi_rs2_addr*: read register address, can be arbitrary when not used.
- *rvfi_rs1_addr*: read register address, can be arbitrary when not used.
- *rvfi_mem_wmask*: write mask; indicates valid bytes written, 0 if unused.
- *rvfi_mem_rmask*: read mask; indicates valid bytes read, 0 if unused.

8 byte information elements:

- *rvfi_mem_wdata*: write data; data written to memory by this command.
- *rvfi_mem_rdata*: read data; data read from *mem_addr*, before write.
- *rvfi_mem_addr*: memory access address; points to byte address, or 0 if unused.
- *rvfi_rd_wdata*: write register value; must be 0 if *rd_addr* is 0.
- *rvfi_rs2_data*: read register value.
- *rvfi_rs1_data*: read register value.
- *rvfi_insn*: instruction word, 32-bit command value.
- *rvfi_pc_wdata*: PC after instruction; following PC, either PC + 4 or jump/trap target.
- *rvfi_pc_rdata*: PC before instruction, value for current instruction.

- *rvfi_order*: instruction number, *INSTRET* value after completion.

5.2.3 Issues with TestRIG

We attempted to use TestRIG with a *VEngine* named *QCVEngine* (QuickCheck Verification Engine) on two implementations: the modified processor and a golden reference model, which in this case is Sail. However, we were not able to make the implementation pass these tests. This failure can be narrowed down to three main reasons.

Firstly, we decided to implement version 0.9.6 of the CHERI RISC-V specification, which was the latest version when the project started. This decision followed information given by a researcher at the University of Cambridge whose main focus is implementing CHERI on RISC-V. He stated that, although TestRIG targets version 0.9.3, most differences with 0.9.6 are related to instruction mnemonics. These differences are subtle, but they may still have led to incorrect instruction encoding in some cases.

Secondly, the RVFI-DII interface and the C++ socket logic may contain design errors. Since both parts had to be adapted to the modified NOEL3 implementation, bugs in this interface could prevent TestRIG from working even if the processor itself had no flaws.

Finally, TestRIG is cumbersome to use without following a workflow that translates Verilog code into C++. This introduced an additional problem, since NOEL3 is written in VHDL and therefore had to be translated. Interpretation of trace comparison results also turned out to be exceedingly complicated. Given the project's deadlines, it was not feasible to complete the work required to format these traces and extract useful debugging information from them.

5.3 Concluding remarks

The process of verifying the CHERI-enhanced NOEL3 has involved specific testing, ensuring that our design correctly executes all base CHERI instructions according to the *RV32_LYmw10rc1pc* encoding; a Python script translates the latter into their opcode, as well as generating a golden signature file with expected values. Tests are compiled and loaded into memory through the debug module; after execution, results are read from memory and compared against expected values. Random instruction testing using TestRIG was attempted; however, version differences, possible interface issues and time-consuming workflow prevented us from achieving successful results with this environment. Specific testing, however, returned satisfactory results, which will be explained in the following chapter, beginning with functional correctness in **Section 6.1**.

6

Evaluation

After verifying our design, the next step was to evaluate the implementation cost of adding CHERI to NOEL3. This chapter studies several processor configurations and compares them in terms of resource usage and timing. The goal is to separate the overhead introduced by CHERI from the effects caused by memory size, PMP and the FPU.

The chapter is divided into four parts. First, **Section 6.1** is reserved for the discussion of functional correctness. Then, **Section 6.2** describes the experimental setup and the tools used during development, simulation and implementation. Afterwards, **Section 6.3** presents the raw implementation results. Finally, **Section 6.4** analyses the timing and resource overheads introduced by CHERI.

6.1 Functional correctness

The successful completion of the verification process indicates that the proposed design achieves an operational bare-metal implementation of the *RV32_LYmw10rc1pc* CHERI-RISC-V 32-bit architecture. The implementation includes support for the base CHERI instruction set and capability semantics, including tagged memory, bounded capabilities, permission enforcement, and secure capability propagation. Consequently, the processor is able to execute capability-aware software while enforcing hardware-level memory safety guarantees. The most typical corner cases are covered.

6.2 Experimental setup

Git was used for version control during the development of the CHERI modifications, the verification logic and the evaluation configurations. This made it possible to track the different design versions used during simulation, synthesis and implementation, and to keep the implementation runs associated with the corresponding processor configurations.

The general project workflow and the role of each tool are described in **Section 3.2**. For the evaluation results reported in this chapter, Questa was used as the main simulation and debugging environment before implementation. This was useful when validating changes to the pipeline, CSR logic, memory accesses, exception handling and tag propagation, since the internal processor state could be inspected directly during execution.

GHDL and Verilator were not used to produce the final timing or area results. They were used during the attempted TestRIG integration flow, where GHDL translated the VHDL

implementation into Verilog and Verilator compiled the generated Verilog into a C++ simulation model that could be connected to the RVFI-DII socket infrastructure. Since this flow was not completed successfully, the functional correctness claims are based on the directed verification flow described in **Section 5.1**.

Synthesis and implementation were performed using Vivado, targeting the KCU105 FPGA evaluation board [47]. Resource utilization results reported in this chapter were extracted from the Vivado reports after placement. The timing results were extracted from the timing summary reports after implementation and physical optimization. These reports were used to compare the different NOEL3 and CHERI configurations described in **Section 6.3**.

Several configurations were implemented to separate the cost of CHERI from the cost of other processor features. The baseline processor is referred to as NOEL3. Some NOEL3 configurations include PMP, since PMP is the standard RISC-V mechanism available in the baseline processor for memory protection. The CHERI configurations do not include PMP, because memory protection is provided through capabilities instead.

The implemented configurations are summarized in **Table 6.1**.

Configuration	**FPU**	**PMP**	**Memory**	**Target freq. (MHz)**
`NOEL3_FPU_PMP_512KB_54MHz`	Yes	Yes	512 KB	54
`CHERI_FPU_noPMP_512KB_54MHz`	Yes	No	512 KB	54
`NOEL3_FPU_PMP_16KB_54MHz`	Yes	Yes	16 KB	54
`CHERI_FPU_noPMP_16KB_54MHz`	Yes	No	16 KB	54
`NOEL3_noFPU_noPMP_512KB_120MHz`	No	No	512 KB	120
`NOEL3_noFPU_PMP_512KB_120MHz`	No	Yes	512 KB	120
`CHERI_noFPU_noPMP_512KB_75MHz`	No	No	512 KB	75
`NOEL3_noFPU_PMP_512KB_20MHz`	No	Yes	512 KB	20
`CHERI_noFPU_noPMP_512KB_20MHz`	No	No	512 KB	20

Table 6.1: Implemented configurations used for evaluation.

6.3 Implementation results

The raw implementation results are shown in **Table 6.2**. These values are used in the following sections to quantify resource and timing overheads introduced by CHERI. The table reports CLB LUT usage, combined BRAM block usage and the reported frequency for each implemented configuration.

The FPU has a strong impact on the timing results. When the FPU is enabled, both NOEL3 and CHERI designs are limited to approximately 54 MHz in the KCU105 FPGA evaluation board [47]. Therefore, these configurations are useful for comparing resource utilization, but they do not isolate the timing cost of CHERI. To evaluate the timing impact of CHERI, the comparison must be made using the configurations without FPU.

Configuration	CLB LUTs	BRAM	Freq. (MHz)	LUT overhead	BRAM overhead
NOEL3_FPU_PMP_512KB_54MHz	12840	290	54.567	–	–
CHERI_FPU_noPMP_512KB_54MHz	16988	314	54.567	32.3%	8.3%
NOEL3_FPU_PMP_16KB_54MHz	12016	20	54.567	–	–
CHERI_FPU_noPMP_16KB_54MHz	16035	38	54.567	33.4%	90.0%
NOEL3_noFPU_noPMP_512KB_120MHz	8614	290	120.048	–	–
NOEL3_noFPU_PMP_512KB_120MHz	10274	290	120.048	–	–
CHERI_noFPU_noPMP_512KB_75MHz	14167	314	75.030	37.9%	8.3%
NOEL3_noFPU_PMP_512KB_20MHz	10240	290	20.008	–	–
CHERI_noFPU_noPMP_512KB_20MHz	14093	314	20.008	37.6%	8.3%

Table 6.2: Raw implementation results after implementation.

In the no FPU configurations, the NOEL3 design with PMP reaches 120.048 MHz, while the CHERI design reaches 75.030 MHz. This comparison is used in **Subsection 6.4.1** to estimate the timing cost of the added capability checks and metadata manipulation logic.

The BRAM overhead corresponds mainly to the additional memory structures required by CHERI, especially the tag memories. Each capability stored in memory has an associated validity tag, which must be stored separately from normal software-visible data. Therefore, CHERI configurations require extra BRAM blocks in addition to the ones used for the instruction and data memories. BRAM overhead is also affected by FPGA BRAM granularity, since even narrow tag memories may consume complete BRAM primitives.

6.4 Overheads

The overhead analysis compares configurations that differ in only one relevant aspect whenever possible. The memory size comparison uses `NOEL3_FPU_PMP_512KB_54MHz`, `NOEL3_FPU_PMP_16KB_54MHz`, `CHERI_FPU_noPMP_512KB_54MHz` and `CHERI_FPU_noPMP_16KB_54MHz`. This allows the impact of changing the local memory size from 16 KB to 512 KB to be separated from the mostly constant cost of CHERI.

The main resource utilization comparison uses `NOEL3_noFPU_PMP_512KB_120MHz`, `NOEL3_noFPU_noPMP_512KB_120MHz` and `CHERI_noFPU_noPMP_512KB_75MHz`. The `NOEL3_noFPU_PMP_512KB_120MHz` configuration is the most relevant NOEL3 baseline because it includes PMP, while `CHERI_noFPU_noPMP_512KB_75MHz` is the corresponding CHERI design without FPU. The `NOEL3_noFPU_PMP_512KB_20MHz` and `CHERI_noFPU_noPMP_512KB_20MHz` configurations are also compared to check whether the same difference appears under a relaxed timing constraint.

6.4.1 Performance

Performance was evaluated in two complementary ways. First, the maximum reported frequency after implementation was compared between the NOEL3 and CHERI configu-

rations. Second, two microbenchmarks were executed to compare the number of cycles required by representative memory access patterns.

For a fixed workload, execution time depends on both the number of cycles and the clock frequency:

$$T = \frac{C}{f},$$

where C is the cycle count and f is the clock frequency. Therefore, the final performance overhead is not given by frequency alone or by cycle count alone. It is the combination of the cycle-count overhead and the clock rate reduction. In slowdown form, this can be written as:

$$\frac{T_{\text{CHERI}}}{T_{\text{NOEL3}}} = \frac{C_{\text{CHERI}}}{C_{\text{NOEL3}}} \cdot \frac{f_{\text{NOEL3}}}{f_{\text{CHERI}}}$$

Frequency comparisons must be made carefully because the FPU dominates timing when it is enabled. The `NOEL3_FPU_PMP_512KB_54MHz`, `NOEL3_FPU_PMP_16KB_54MHz`, `CHERI_FPU_noPMP_512KB_54MHz` and `CHERI_FPU_noPMP_16KB_54MHz` configurations all reach the same frequency of 54.567 MHz. Therefore, the FPU enabled configurations are useful for resource utilization comparisons, but they do not isolate the timing cost of CHERI.

The effect of memory size on frequency is not visible in these results. For the NOEL3 designs with FPU and PMP, both the 512 KB configuration and the 16 KB configuration reach 54.567 MHz. The same happens for the CHERI designs with FPU, where both memory sizes also reach 54.567 MHz. Therefore, within these configurations, changing the local memory size does not affect the reported maximum frequency. The limiting factor is the FPU path, not the memory size.

To observe the timing impact of CHERI, the FPU must be removed. In this case, the NOEL3 design with PMP reaches 120.048 MHz, while the CHERI design reaches 75.030 MHz (seen in **Figure 6.1**). This corresponds to a frequency reduction of approximately 37.5 %.

The reduction in frequency is expected because CHERI adds capability checks and capability metadata manipulation to the execution path. Bounds checking, permission checking, tag propagation and bounds encoding increase the amount of logic that must be evaluated. This effect is hidden when the FPU is present, but becomes visible once the FPU is removed.

The timing report for `CHERI_noFPU_noPMP_512KB_75MHz` supports this interpretation. The worst setup path ends in the `alu_out_cap_tag` register, with a data path delay of 13.190 ns over a 13.328 ns clock period and 40 logic levels. This path is in the CHERI capability ALU logic in the execute stage. It computes the resulting capability metadata and output tag after capability operations, bounds-related transformations, permission checks and tag-validity propagation. The tag calculation is especially timing-sensitive because it depends on the generated capability metadata. A likely optimization would be to pipeline this capability ALU more aggressively, spreading part of the computation between EX1 and EX2 instead of performing it entirely in EX1.

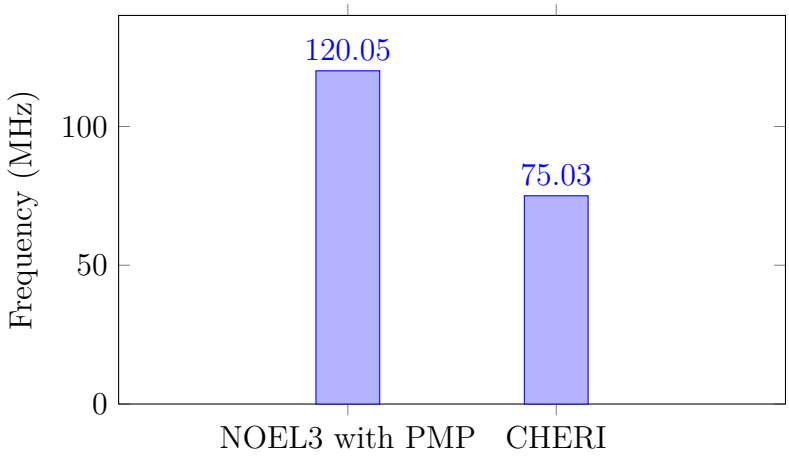


Figure 6.1: Frequency comparison without FPU. The NOEL3 design corresponds to `NOEL3_noFPU_PMP_512KB_120MHz`, while the CHERI design corresponds to `CHERI_noFPU_noPMP_512KB_75MHz`.

The low frequency configurations `NOEL3_noFPU_PMP_512KB_20MHz` and `CHERI_noFPU_noPMP_512KB_20MHz` are not used to measure maximum frequency, because both were implemented with a relaxed 20 MHz constraint. However, they are useful for resource utilization analysis because they show whether this overhead remains visible even when the implementation is not under strong timing pressure.

The second performance metric is the number of cycles required to execute small microbenchmarks. Two access patterns were evaluated. The first one traverses an array of integers. The second one traverses a linked list. These benchmarks may appear too short and simple, but this is acceptable for this processor. The memory model in NOEL3 is simple and deterministic, and memory accesses always have the same behavior. Therefore, increasing the benchmark size would mainly repeat the same access pattern more times instead of exposing cache, predictor or memory hierarchy effects.

The results are shown in **Figure 6.2**. The integer array traversal takes 324 cycles in both processors. Therefore, this benchmark does not show an additional cycle cost from using the CHERI processor. This is expected because the access pattern is sequential and does not require repeatedly loading new capabilities from memory. Instead, the traversal can reuse and update the capability used to access the array.

The linked list traversal shows a different result. The NOEL3 processor completes the benchmark in 324 cycles, while the CHERI version requires 452 cycles. This is an increase of 128 cycles, or approximately 39.5 %. The main reason for this increase is that the CHERI linked list stores the next pointer as a capability. Therefore, each step of the traversal must load a capability from memory using *ly*. In this implementation, *ly* takes two cycles because a capability is wider than the normal 32-bit memory path. One access retrieves one half of the capability, and the other access retrieves the remaining half together with the corresponding tag information. As a result, following a linked list has a direct cycle overhead when the links are represented as capabilities.

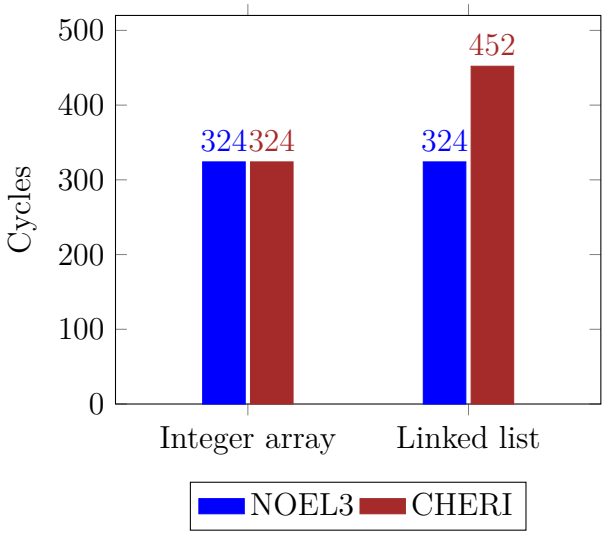


Figure 6.2: Microbenchmark cycle comparison between NOEL3 and CHERI.

These microbenchmarks show that the cycle overhead of CHERI is workload dependent. Sequential integer array traversal shows no cycle increase in this test, while linked list traversal shows a clear overhead because it repeatedly loads capabilities from memory. Therefore, the performance cost of CHERI should not be described as a fixed per instruction penalty. It depends on how frequently the program manipulates, loads, stores and dereferences capabilities.

If the cycle counts are combined with the post-implementation frequencies, the total execution time difference becomes larger. For the integer array benchmark, the cycle count is the same, so the expected execution time increase comes only from the lower maximum frequency of the CHERI design. Since NOEL3 reaches 120.048 MHz and CHERI reaches 75.030 MHz, this corresponds to a slowdown of approximately $120.048/75.030 = 1.6$. For the linked list benchmark, both effects apply: the CHERI design executes more cycles and also has a lower maximum frequency. The cycle-count ratio is $452/324 \approx 1.40$, and the frequency-related slowdown is approximately 1.6, giving a combined slowdown of approximately $1.40 \cdot 1.6 = 2.24$. Therefore, pointer-intensive code is the case where the performance cost of CHERI is expected to be most visible.

6.4.2 Resource utilization

The first resource utilization comparison studies the impact of memory size. The `NOEL3_FPU_PMP_512KB_54MHz` and `NOEL3_FPU_PMP_16KB_54MHz` configurations compare NOEL3 with 512 KB and 16 KB memories, both with FPU and PMP enabled. These results are shown in **Figure 6.3** and **Figure 6.4**. Reducing the memory from 512 KB to 16 KB reduces the number of BRAM blocks from 290 to 20. The LUT count is also reduced, from 12840 to 12016 LUTs. Therefore, most of the memory size impact appears in BRAM block usage, while the LUT count changes by only 824 LUTs.

The same trend appears in the CHERI configurations. The `CHERI_FPU_noPMP_512KB_54MHz` and `CHERI_FPU_noPMP_16KB_54MHz` configurations compare CHERI with 512 KB

and 16 KB memories, both with FPU enabled. Reducing the memory size from 512 KB to 16 KB reduces BRAM block usage from 314 to 38, while the LUT count changes from 16988 to 16035 LUTs. This is a reduction of 953 LUTs. The LUT differences are visible in **Figure 6.3**, while the much larger change in BRAM block usage is shown in **Figure 6.4**.

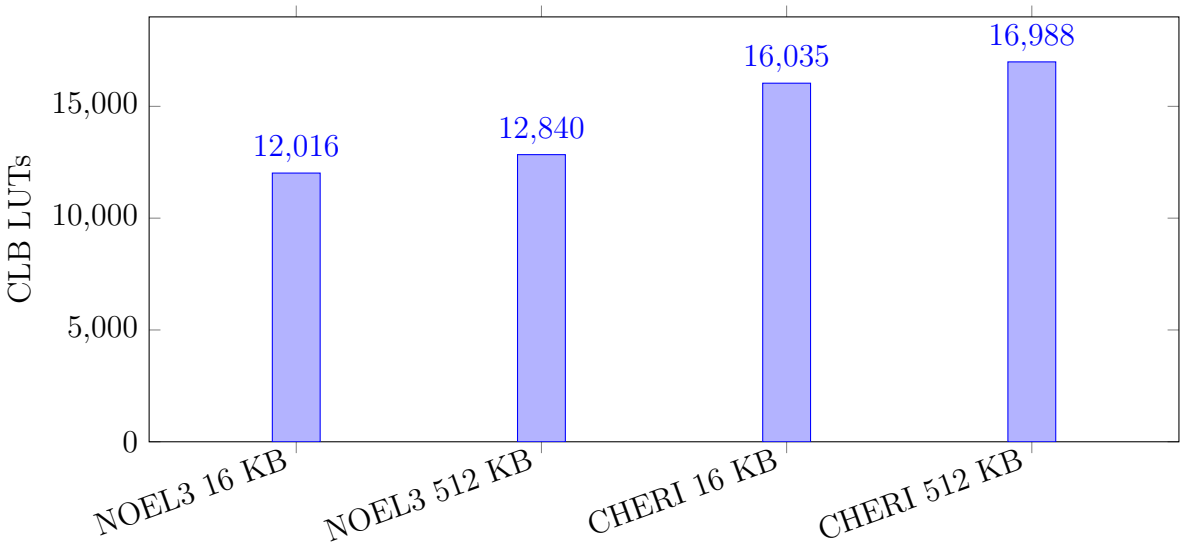


Figure 6.3: CLB LUT usage for 16 KB and 512 KB memories. The `NOEL3_FPU_PMP_512KB_54MHz`, `NOEL3_FPU_PMP_16KB_54MHz`, `CHERI_FPU_noPMP_512KB_54MHz` and `CHERI_FPU_noPMP_16KB_54MHz` configurations are compared.

Comparing CHERI against NOEL3 for each memory size isolates the resource cost of capability support. With 512 KB memories, CHERI uses 16988 LUTs while NOEL3 uses 12840 LUTs. This is an increase of 4148 LUTs, or approximately 32.3 %. In BRAM block usage, CHERI uses 24 additional BRAM blocks.

With 16 KB memories, CHERI uses 16035 LUTs while NOEL3 uses 12016 LUTs. This is an increase of 4019 LUTs, or approximately 33.4 %. In BRAM block usage, CHERI uses 18 additional BRAM blocks. The LUT increase is therefore almost the same for both memory sizes. This shows that most of the LUT overhead introduced by CHERI is constant and does not scale strongly with the configured memory size.

In this implementation, memory tag bits are stored in dedicated tag memories inferred as FPGA BRAMs, rather than as individual flip-flop registers. This is done because tag storage scales with memory size. Implementing all memory tags as registers would be inefficient, especially for the 512 KB configurations. The tags are also kept separate from normal data so that ordinary stores cannot directly modify or forge capability validity. Tags may be carried together with capabilities inside the processor pipeline and register file, but persistent memory tags are stored in the dedicated tag memory.

The next comparison removes the FPU to focus on the cost of memory protection mechanisms. The `NOEL3_noFPU_noPMP_512KB_120MHz` configuration is NOEL3 without FPU and without PMP. The `NOEL3_noFPU_PMP_512KB_120MHz` configuration adds PMP to this design. The `CHERI_noFPU_noPMP_512KB_75MHz` configuration replaces PMP-based

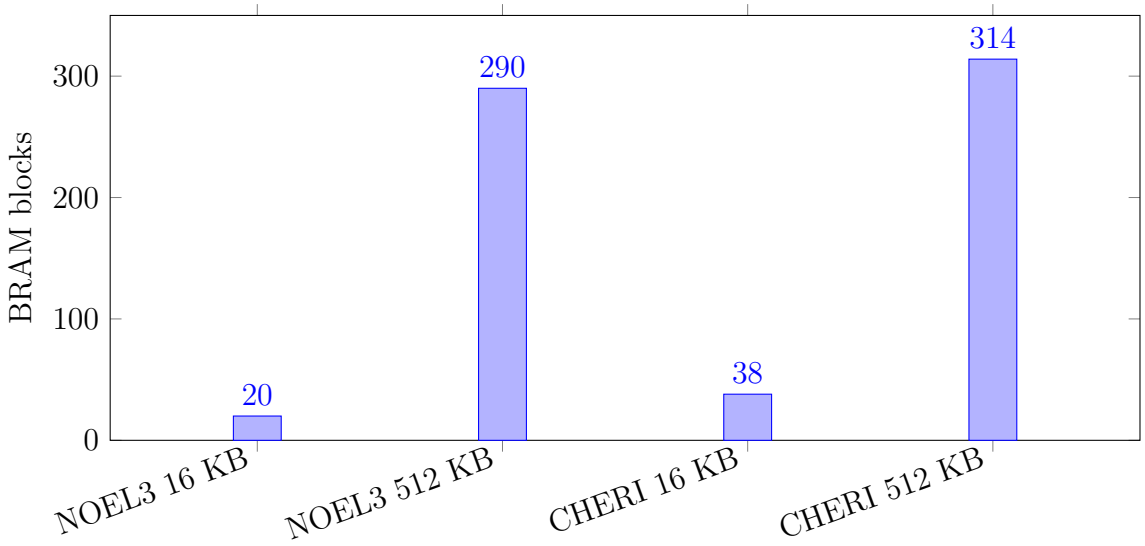


Figure 6.4: BRAM block usage for 16 KB and 512 KB memories. The `NOEL3_FPU_PMP_512KB_54MHz`, `NOEL3_FPU_PMP_16KB_54MHz`, `CHERI_FPU_noPMP_512KB_54MHz` and `CHERI_FPU_noPMP_16KB_54MHz` configurations are compared.

protection with CHERI capability support. These results are shown in **Figure 6.5**. Removing PMP from NOEL3 reduces the LUT count from 10274 to 8614, meaning that PMP accounts for 1660 LUTs in this configuration. The CHERI design uses 14167 LUTs. Compared with NOEL3 with PMP, this is an increase of 3893, or approximately 37.9 %.

The comparison between `NOEL3_noFPU_PMP_512KB_120MHz` and `CHERI_noFPU_noPMP_512KB_75MHz` is the most relevant one. Comparing CHERI only against NOEL3 without PMP would exaggerate the apparent cost of CHERI, because the standard NOEL3 design would not include a memory protection mechanism. Therefore, NOEL3 with PMP is the fairest baseline for estimating the additional resource cost of replacing PMP-based protection with CHERI capabilities.

The `NOEL3_noFPU_PMP_512KB_20MHz` and `CHERI_noFPU_noPMP_512KB_20MHz` configurations repeat the comparison under a relaxed 20 MHz timing constraint, as shown in **Figure 6.6**. The `NOEL3_noFPU_PMP_512KB_20MHz` configuration uses 10240 LUTs. The `CHERI_noFPU_noPMP_512KB_20MHz` configuration uses 14093 LUTs. The difference is 3853 LUTs, which is very close to the 3893 LUT difference observed between `NOEL3_noFPU_PMP_512KB_120MHz` and `CHERI_noFPU_noPMP_512KB_75MHz`. This confirms that the additional resource utilization is not an artifact of the higher timing target. CHERI logic requires more resources even when timing pressure is relaxed.

Overall, these results show that CHERI introduces a significant resource overhead in the evaluated configurations. For the main LUT comparisons, the CHERI designs are approximately one third larger than the corresponding NOEL3 baselines. This is visible both in the memory size comparison, where CHERI adds around 32 to 33 % more LUTs, and in the no-FPU comparison against NOEL3 with PMP, where the increase is around 38 %. Therefore, as a general estimate, adding CHERI increases LUT usage by roughly

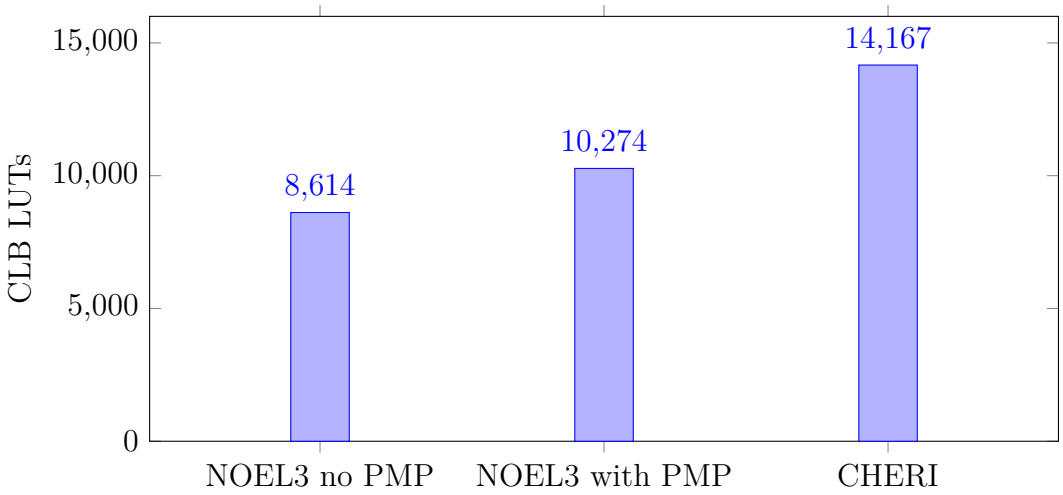


Figure 6.5: CLB LUT usage comparison without FPU. The `NOEL3_noFPU_noPMP_512KB_120MHz` configuration is NOEL3 without PMP, `NOEL3_noFPU_PMP_512KB_120MHz` is NOEL3 with PMP, and `CHERI_noFPU_noPMP_512KB_75MHz` is CHERI.

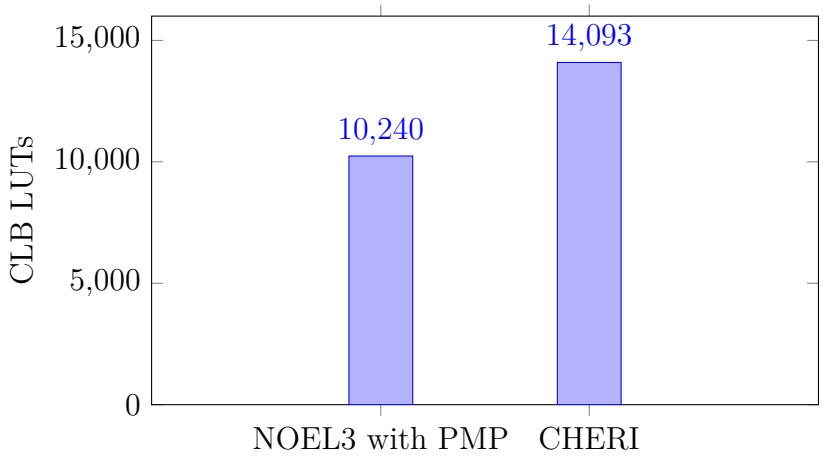


Figure 6.6: CLB LUT usage under a relaxed 20 MHz timing constraint. The `NOEL3_noFPU_PMP_512KB_20MHz` configuration is NOEL3 with PMP and `CHERI_noFPU_noPMP_512KB_20MHz` is CHERI.

33 % in these configurations. Memory size primarily affects BRAM block usage, while the CHERI LUT overhead remains relatively stable across the tested memory sizes.

7

Conclusion

The work carried out in this thesis shows that CHERI can be integrated into NOEL3, a deterministic barrel RISC-V processor. This is relevant because NOEL3 is not a conventional single thread pipeline. Its barrel architecture executes several hardware threads in a deterministic round robin manner, which means that architectural state, pipeline control, memory accesses and exception handling are organized around multiple hardware contexts. Adding CHERI to this type of processor therefore required more than extending a standard RISC-V datapath. Capability state had to be integrated into a multithreaded microarchitecture while preserving deterministic execution and compatibility with the existing NOEL3 structure.

The implementation required structural changes across the processor. Capabilities were added as architectural objects containing an address, metadata and a tag. The register file, program counter, relevant CSRs, ITCM, DTCM, instruction decoding and execution stages were extended to propagate and check this information. Capability checks were added for instruction fetch, memory reads and memory writes, and invalid uses of capabilities are reported as CHERI faults. Trap entry and trap return were also adapted so that control flow can preserve capability metadata through *mtvec* and *mepc*.

The design confirms that CHERI cannot be added as a small isolated unit. Its checks and metadata affect the normal execution path of the processor, especially when capabilities are used to authorize memory accesses or update the program counter. This was particularly visible in the memory system, where tag storage had to be added, and in the execution logic, where bounds encoding and permission checks introduced additional hardware cost.

The evaluation shows that this cost is significant but bounded. In the evaluated configurations, CHERI increased LUT usage by roughly one third compared with the corresponding NOEL3 baselines. Most of this LUT overhead was approximately constant across the tested memory sizes, while memory size mainly affected BRAM usage due to the additional tag storage. Timing was also affected. When the FPU was removed, the baseline NOEL3 design with PMP reached approximately 120 MHz, while the CHERI design reached approximately 75 MHz. This shows that the capability checks and metadata operations introduce additional timing pressure when they are exposed as the limiting path. Since execution time depends on both cycle count and clock frequency, this frequency reduction must be combined with any cycle-count overhead when evaluating the final performance impact.

These results are higher than the overheads reported for other CHERI implementations

that can be used as reference points for CHERI overhead minimization. Two relevant implementations are described in papers authored by Jonathan Woodruff, one of the developers of CHERI [48]. Both works add CHERI support to BERI, the Bluespec Extensible RISC Implementation, which shares some high-level characteristics with NOEL3, such as in-order execution and interleaved multithreading. One implementation reports a resource overhead of 32 % and a frequency reduction of 8.1 % [49], while the other reports 27 % and 11.2 %, respectively [50].

These values are lower than those obtained in our implementation, where the main no-FPU comparison showed a resource overhead of approximately 38 % and a frequency reduction of approximately 37.5 %. Leaving aside the technical differences between the baseline processors, this suggests that there is still room for optimization, especially in the timing-critical CHERI execution logic. However, the obtained results are still considered satisfactory for this project, given the implementation time constraints, the limited initial CHERI implementation experience, and the lack of mature CHERI toolchain support. These limitations motivate the future work described in **Section 7.2**.

The microbenchmarks show that the cycle overhead depends on the workload. Sequential integer array traversal took the same number of cycles in the baseline and CHERI processors. In contrast, linked list traversal was slower with CHERI because each link is represented as a capability and must be loaded using *ly*. Since capability loads require two accesses in this implementation, pointer-intensive workloads expose a clearer cycle cost than simple sequential array traversal. Combined with the lower maximum frequency of the CHERI configuration, this makes pointer-intensive workloads the most affected case in the evaluated benchmarks.

Verification was performed using directed tests after the attempted TestRIG based random testing flow could not be completed within the project. The successful flow used assembly tests, opcode preprocessing, signature generation and simulation in a modified Gaisler environment. This allowed CHERI instructions, invalid capability uses, stress cases and microbenchmarks to be checked against expected signatures.

Overall, the project demonstrates that CHERI can be adapted to a deterministic multithreaded barrel processor such as NOEL3. The resulting design provides hardware-enforced bounds, permission and tag checks, at the cost of additional logic, extra tag storage and reduced maximum frequency. These results provide a concrete reference for future work on capability based memory protection in deterministic and fault tolerant RISC-V processors.

7.1 Achievements

The main achievement of this thesis is the implementation of CHERI support in NOEL3. This required extending the processor so that capabilities could be represented, stored, propagated and checked throughout a barrel microarchitecture. Since NOEL3 maintains several hardware contexts, the changes had to preserve capability state consistently across the processor structures used by each thread.

The register file was widened to store 64-bit capability values and tag information. The program counter was also extended so that instruction fetch could be authorized through a capability. Relevant CSRs were extended to support capability sized values, including the CSRs used for trap entry and trap return. The ITCM and DTCM were modified to include dedicated memory tag storage, implemented using FPGA BRAM. These tags are stored separately from normal software-visible data so that ordinary stores cannot directly modify capability validity.

The instruction decoding and execution logic were extended to support the implemented CHERI instructions. This includes address manipulation, metadata manipulation, bounds updates, permission updates, tag reading, capability comparison and capability load and store instructions. The execution path was also extended to enforce tag, bounds and permission checks when capabilities are used for instruction fetch or data memory accesses.

Exception handling was adapted to the capability model. CHERI access failures are reported using a single CHERI fault cause, and the faulting address is stored for software inspection. Trap entry and return were modified so that the processor can preserve capability metadata through *mtvec* and *mepc*, instead of treating the trap target and return address as plain 32-bit addresses.

The memory tag implementation was optimized. The initial flip-flop based memory tag storage was not scalable because the number of tag bits grows with memory size. Moving memory tag storage to BRAM reduced the implementation cost and made the design better aligned with the existing memory organization of NOEL3. The bounds encoding logic was also rewritten to reduce the amount of combinational hardware generated during synthesis.

A directed verification flow was developed and used successfully. The flow translates CHERI instructions into raw opcodes, generates golden signatures, loads compiled tests into the simulated processor, executes them and compares the resulting signature against the expected one. This allowed positive tests, negative tests, stress tests and microbenchmarks to be executed on the modified processor.

The design was evaluated after synthesis and implementation. Resource utilization results showed that CHERI adds approximately 4 k LUTs in the main evaluated configurations, corresponding to roughly one third more LUT usage than the corresponding NOEL3 baselines. The timing results showed a reduction in maximum frequency when the FPU was removed, exposing the cost of the added capability logic. The microbenchmarks showed that the cycle overhead depends on the memory access pattern, with linked list traversal being affected more strongly than sequential array traversal.

In summary, the main achievements are:

- Integration of CHERI capability support into the NOEL3 barrel microarchitecture.
- Extension of the register file, program counter, CSRs and memories to support capabilities and tags.
- Implementation of CHERI instruction decoding and execution logic.

- Enforcement of capability checks for instruction fetch and memory accesses.
- Adaptation of trap entry and trap return to preserve capability state.
- BRAM-based memory tag storage for local memories, with tag support in the register file.
- Optimization of the bounds encoding logic to reduce synthesis overhead.
- Development of a directed verification flow based on assembly tests and signature comparison.
- Quantification of CHERI overheads in resource utilization, timing and microbenchmark cycle counts.

7.2 Future work

The first direction for future work is to complete the TestRIG integration. The RVFI-DII interface was partially developed, but the complete random testing flow was not made operational within the project timeline. Completing this work would provide stronger verification coverage and would make it possible to compare the modified processor against a reference model using randomly generated instruction traces.

The verification suite should also be expanded. The directed tests developed in this thesis cover positive cases, negative cases, stress cases and microbenchmarks, but more coverage is needed for corner cases. Future tests should exercise trap behavior, representability limits in bounds encoding, partial capability overwrites, external bus accesses, CSR corner cases and interactions with different NOEL3 configuration options; the goal in this case would be to achieve formal verification [51, 52].

A second direction is to improve toolchain support. In this thesis, CHERI instructions were translated into raw opcodes by a preprocessing script because the environment did not use a CHERI aware assembler. A complete CHERI toolchain would simplify test development, reduce the risk of encoding mistakes and allow larger, higher-level programs to be evaluated [53].

The memory interface is another important area for improvement. In the current implementation, capability loads and stores require two accesses because capabilities are wider than the 32-bit memory path. This directly affects workloads that frequently load capabilities from memory, such as linked list traversal. A wider internal memory path or a more optimized mechanism for capability loads and stores could reduce this overhead.

Future work should also investigate tag support outside the local memories. The current design checks CHERI permissions for external accesses, but does not preserve capability tags on the external bus. Adding a tag preserving external memory interface would make it possible to store valid capabilities outside the ITCM and DTCM, which would be important for larger systems [54].

Further timing optimization is also needed. The evaluation shows that CHERI reduces the maximum frequency when the FPU is not the limiting path. Future work should

focus on the critical paths introduced by the capability execution logic, especially bounds checking, permission checking, tag propagation and bounds encoding. Possible improvements include restructuring the capability ALU, pipelining selected checks, or moving some metadata computations away from timing-critical paths.

More realistic workloads should also be evaluated. The microbenchmarks used in this thesis are intentionally small and deterministic, which is useful for isolating specific overheads. However, larger embedded workloads would provide better insight into the practical cost of CHERI in real applications. These workloads should include different pointer access patterns, memory layouts and ratios between integer operations and capability operations. For this purpose, the use of already existing benchmarks [55] might further prove functional correctness, and more exhaustive performance evaluations [56] should be carried out.

Finally, an ASIC implementation should be considered as future work. The current evaluation is based on FPGA implementation results, which are useful for studying resource utilization trends and timing bottlenecks, but are not ideal for drawing strong conclusions about power consumption. FPGA power estimates are heavily affected by the target device, routing resources and FPGA fabric overheads, and therefore do not directly represent the cost of the processor logic itself. An ASIC implementation would provide a more meaningful platform for evaluating the power impact of CHERI, as well as its resource and timing cost in a technology closer to a final integrated circuit.

7.3 Disclaimers

7.3.1 Ethical disclaimer

There are no specific ethical concerns identified for this project. The work is centered on integrating CHERI into NOEL3 to enhance memory safety and overall robustness. These changes are aimed purely at improving technical reliability in space and research contexts and do not create additional ethical issues or obligations beyond those already inherent in processor development for commercial or scientific satellite systems.

7.3.2 Generative AI disclaimer

Generative artificial intelligence models have been used in the making of this thesis. Their use has been limited to the following:

- Technical help for solving issues related to tool usage, coding and ambiguous or unclear documentation.
- Thesis report phrasing, wording and styling, as well as bibliography suggestions.

The authors of this thesis wish to stress that generative AI has been used as a tool and not as a substitute for their work and effort.

Bibliography

[1] "Common Weakness Enumeration's Top 25 Most Dangerous Software Weaknesses." [Online]. Available: https://cwe.mitre.org/top25/ *(Referenced in page 1).*

[2] "A proactive approach to more secure code." [Online]. Available: https://www.microsoft.com/en-us/msrc/blog/2019/07/a-proactive-approach-to-more-secure-code *(Referenced in page 1).*

[3] "The Chromium Porjects - Memory Safety." [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/ *(Referenced in page 1).*

[4] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62. *(Referenced in pages 1, 1, and 2).*

[5] "Ibex Reference Guide - Physical Memory Protection (PMP)." [Online]. Available: https://ibex-core.readthedocs.io/en/latest/03_reference/pmp.html *(Referenced in page 1).*

[6] O. Oleksenko, P. Felber, D. Kuvaiskii, C. Fetzer, and P. Bhatotia, "Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, pp. 1–30, 2018. *(Referenced in page 1).*

[7] "RISC-V specification for CHERI Extensions, version 0.9.6.1." [Online]. Available: https://github.com/riscv/riscv-cheri/releases/tag/v0.9.6.1 *(Referenced in pages 1 and 5).*

[8] J. Woodruff *et al.*, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201505-oakland2015-cheri-compartmentalization.pdf, accessed: 2025-11-19. *(Referenced in page 1).*

[9] H. M. Levy, *Capability-Based Computer Systems*. Digital Press, 1984. [Online]. Available: https://homes.cs.washington.edu/~levy/capabook/ *(Referenced in page 1).*

[10] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 245–258, 2009. *(Referenced in page 1).*

[11] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the C programming language," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 103–114, 2008. *(Referenced in page 1).*

[12] R. N. Watson, P. G. Neumann, and J. Woodruff, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (version 9)," Sep 2023. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf *(Referenced in page 2).*

[13] *Morello Program*, Arm Ltd., 2022, part of the UKRI Digital Security by Design (DSbD) Morello program. [Online]. Available: https://www.arm.com/architecture/cpu/morello *(Referenced in page 2).*

[14] Arm Ltd., *Arm Architecture Reference Manual Supplement Morello for A-profile Architecture (DDI0606A.k)*, Arm Ltd., 2022. [Online]. Available: https://developer.arm.com/documentation/ddi0606/latest *(Referenced in page 2).*

[15] "CHERI RISC-V: A Case Study on the CVA6." [Online]. Available: https://riscv-europe.org/summit/2024/media/proceedings/plenary/Thu-16-30-Bruno-Sa.pdf *(Referenced in page 2).*

[16] *Codasip X730 CHERI application processor documentation*, Codasip GmbH. [Online]. Available: https://codasip.com/solutions/riscv-processor-safety-security/cheri/x730-risc-v-application-processor/ *(Referenced in page 2).*

[17] "CHERIoT Platform." [Online]. Available: https://github.com/CHERIoT-Platform *(Referenced in page 2).*

[18] N. W. Filardo *et al.*, "Cornucopia: Temporal Safety for CHERI Heaps," in *2020 IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 608–625. *(Referenced in page 2).*

[19] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and T. M. Jones, "CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*. ACM, 2019, pp. 545–557. *(Referenced in page 2).*

[20] D. Chisnall, C. Rothwell, B. Davis, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, P. G. Neumann, and M. Roe, "Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, 2015, pp. 117–130. *(Referenced in page 2).*

[21] B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo *et al.*, "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*, 2019. *(Referenced in page 2).*

[22] H. Almatary, M. Dodson, D. Chisnall, H. Xia, A. Joannou *et al.*, "CompartOS: CHERI Compartmentalization for Embedded Systems," *arXiv preprint arXiv:2206.02852*, 2022. *(Referenced in page 2).*

[23] "NOEL3 - Frontgrade Gaisler." [Online]. Available: https://www.gaisler.com/products/noel3 *(Referenced in pages 2 and 14).*

[24] N. J. Wessman *et al.*, "A Complete RISC-V Based Space-Grade Platform," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2022)*, 2022. [Online]. Available: https://past.date-conference.com/proceedings-archive/2022/pdf/3004.pdf *(Referenced in page 2).*

[25] "The RISC-V Instruction Set Manual Volume I - Unprivileged Architecture." [Online]. Available: https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf *(Referenced in page 5).*

[26] K. Cheang *et al.*, "Verifying RISC-V Physical Memory Protection," *arXiv preprint arXiv:2211.02179*, 2022. *(Referenced in page 5).*

[27] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975. *(Referenced in page 6).*

[28] T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. M. Watson, and P. Sewell, "Verified Security for the Morello Capability-Enhanced Prototype Arm Architecture," in *European Symposium on Programming (ESOP 2022)*. Springer, 2022, pp. 174–203. *(Referenced in page 6).*

[29] A. Joannou, J. Woodruff, R. N. M. Watson, S. W. Moore *et al.*, "Efficient Tagged Memory," in *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE, 2017. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201711-iccd2017-efficient-tags.pdf *(Referenced in pages 6 and 28).*

[30] J. Woodruff, A. Joannou *et al.*, "CHERI Concentrate: practical compressed capabilities," *IEEE Transactions on Computers*, vol. 68, pp. 1455–1469, 10 2019. *(Referenced in page 10).*

[31] A. Dörflinger *et al.*, "ECC Memory for Fault Tolerant RISC-V Processors," *Electronics*, vol. 9, no. 8, 2020. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC7343426/ *(Referenced in page 14).*

[32] "NOEL3 - User's Manual." [Online]. Available: https://download.gaisler.com/products/noel3-examples/doc/noel3-um-2025-12-19.pdf *(Referenced in page 15).*

[33] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad Memory: A Design Alternative for Cache On-Chip Memory in Embedded Systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES+ISSS 2002)*, 2002, pp. 73–78. *(Referenced in page 16).*

[34] S. Udayakumaran and R. Barua, "Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems," in *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2003)*, 2003. [Online]. Available: https://user.eng.umd.edu/~barua/udayakumaran-CASES-2003.pdf *(Referenced in page 16).*

[35] Arm Ltd., *AMBA AHB Protocol Specification*, Arm Ltd., 2006. [Online]. Available: https://developer.arm.com/documentation/ihi0033/c/?lang=en *(Referenced in page 16).*

[36] "CHERI TestRIG Github repository." [Online]. Available: https://github.com/CTSRD-CHERI/TestRIG *(Referenced in pages 19 and 30).*

[37] A. Joannou, P. Rugg, J. Woodruff *et al.*, "Randomized testing of RISC-V CPUs using Direct Instruction Injection," *IEEE Design & Test*, vol. 41, no. 1, pp. 40–49, 2024. *(Referenced in pages 19 and 39).*

[38] "RISC-V Formal Interface - Direct Instruction Injection Specification." [Online]. Available: https://github.com/CTSRD-CHERI/TestRIG/blob/master/RVFI-DII.md *(Referenced in page 19).*

[39] V. Herdt, D. Große, and R. Drechsler, "Closing the RISC-V Compliance Gap: Looking from the Negative Testing Side," in *Proceedings of the Design Automation Conference (DAC)*, 2020. *(Referenced in page 32).*

[40] "RISC-V ABIs Specification." [Online]. Available: https://riscv-non-isa.github.io/riscv-elf-psabi-doc/ *(Referenced in page 33).*

[41] V. Herdt, D. Große, and R. Drechsler, "Towards Specification and Testing of RISC-V ISA Compliance," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2020. *(Referenced in page 39).*

[42] V. Herdt, D. Große, E. Jentzsch, and R. Drechsler, "Efficient Cross-Level Testing for Processor Verification: A RISC-V Case-Study," in *Forum on Specification and Design Languages (FDL)*, 2020. *(Referenced in page 39).*

[43] R.-V. A. T. SIG, "RISC-V Architectural Certification Tests," https://github.com/riscv/riscv-arch-test, n.d., software test suite. *(Referenced in page 39).*

[44] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin, "X-Gen: A random test-case generator for systems and SoCs," in *Seventh IEEE International High-Level Design Validation and Test Workshop, 2002.* IEEE, 2002, pp. 145–150. *(Referenced in page 39).*

[45] B. Campbell and I. Stark, "Randomised testing of a microprocessor model using SMT-solver state generation," *Science of Computer Programming*, vol. 118, pp. 60–76, 2016. *(Referenced in page 39).*

[46] D. D. Tran, T. G. Truong, T. G. Do, and T. D. Do, "Risc-V Random Test Generator," in *2021 15th International Conference on Advanced Computing and Applications*, 2021, pp. 150–155. [Online]. Available: https://dblp.org/rec/conf/acomp/TranTDD21 *(Referenced in page 39).*

[47] "AMD Kintex™ UltraScale™ FPGA KCU105 Evaluation Kit." [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/kcu105.html *(Referenced in pages 43 and 43).*

[48] R. N. Watson, P. G. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. W. Moore, S. J. Murdoch, P. Paeps *et al.*, "Cheri: a research platform deconflating hardware virtualisation and protection," in *RESoLVE'12*. Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE), 2012, unpublished. *(Referenced in page 53).*

[49] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 457–468, 2014. *(Referenced in page 53).*

[50] J. D. Woodruff, "CHERI: A RISC capability machine for practical memory safety," University of Cambridge, Computer Laboratory, Tech. Rep., 2014. *(Referenced in page 53).*

[51] D. Gao and T. Melham, "End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers," in *Proceedings of FMCAD 2021*, 2021, pp. 24–33. *(Referenced in page 55).*

[52] L.-E. Ploix, A. Armstrong, T. Melham, R. Lin, H. Wang, and A. Courtney, "Comprehensive Formal Verification of Observational Correctness for the CHERIoT-Ibex Processor," *arXiv preprint arXiv:2502.04738*, 2025. [Online]. Available: https://arxiv.org/pdf/2502.04738 *(Referenced in page 55).*

[53] R. N. M. Watson, A. Richardson, B. Davis, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, S. W. Moore, E. Napierala, P. Sewell, and P. G. Neumann, "CHERI C/C++ Programming Guide," University of Cambridge, Tech. Rep. UCAM-CL-TR-947, 2020. *(Referenced in page 55).*

[54] E. Ackermann, N. Mauthe, and S. Bugiel, "Work-in-Progress: Northcape: Embedded Real-Time Capability-Based Addressing," in *2024 IEEE European Symposium on Security and Privacy Workshops*. IEEE, 2024, pp. 683–690. *(Referenced in page 55).*

[55] S. Gal-On, "CoreMark." [Online]. Available: https://www.eembc.org/coremark/ *(Referenced in page 56).*

[56] X. Sun, J. Singer, and Z. Wang, "Sweet or Sour CHERI: Performance Characterization of the Arm Morello Platform," 2025. [Online]. Available: https://eprints.gla.ac.uk/362548/ *(Referenced in page 56).*