COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

INTEGRACIÓN EN VHDL DE LA EXTENSIÓN CHERI EN UN SOFTCORE RISC-V EN FPGA PARA SEGURIDAD DE MEMORIA

Autor: Miguel Herremans Humada

Director: Fermín Zabalegui Sanz

Madrid, junio de 2026

Declaración de originalidad

Declaro bajo mi responsabilidad que el Proyecto presentado con el título **INTEGRACIÓN EN VHDL DE LA EXTENSIÓN CHERI EN UN SOFTCORE RISC-V EN FPGA PARA SEGURIDAD DE MEMORIA** de la ETS de Ingeniería – ICAI de la Universidad Pontificia Comillas en el curso académico 2025/2026 es de mi autoría y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

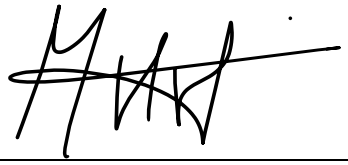
Uso de Inteligencia Artificial[1]

Declaro bajo mi responsabilidad que (indicar la opción correcta):

☐ No he utilizado Inteligencia Artificial en la elaboración del presente documento.

☒ He utilizado Inteligencia Artificial en la elaboración del presente documento y/o del Anexo B siempre en las condiciones permitidas por la Universidad Pontificia Comillas, es decir, aplicando el Nivel 2 de la Escala de Evaluación de Perkins et al. (2024): "*La IA puede utilizarse para actividades previas a la tarea, como la lluvia de ideas, la descripción y la investigación inicial. Este nivel se centra en el uso de la IA para la planificación, las síntesis y la generación de ideas, pero las evaluaciones deben hacer hincapié en la capacidad de desarrollar y refinar estas ideas de forma independiente*". En concreto, las Inteligencia Artificial ha sido empleada para:

Se ha utilizado inteligencia artificial para tareas de búsqueda bibliográfica inicial, revisión de redacción, estructuración de apartados y en búsqueda de ideas para la elaboración de ejemplos. Todo el contenido final, las decisiones de diseño, la implementación en VHDL y la validación experimental han sido desarrollados y verificados por el autor.



Firmado (alumno): Miguel Herremans Humada
Fecha: 24/06/2026

[1] Esta declaración se refiere al uso de la Inteligencia Artificial generativa para realizar los documentos del Proyecto (Anexo B y Memoria). No aplica a Proyectos donde, por su naturaleza, deban emplear inteligencia artificial como parte de los mismos (aplicación de técnicas de aprendizaje automático, redes neuronales, análisis de datos…)

Autorización para la entrega del Proyecto

El Director del Proyecto	El co-Director del Proyecto (si aplica)
 ZABALEGUI SANZ FERMIN - 50202045Z 2026.06.24 13:51:41 +02'00'	
Fdo: Fermín Zabalegui Sanz	Fdo:
Fecha: 24/06/2026	Fecha:

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

INTEGRACIÓN EN VHDL DE LA EXTENSIÓN CHERI EN UN SOFTCORE RISC-V EN FPGA PARA SEGURIDAD DE MEMORIA

Autor: Miguel Herremans Humada

Director: Fermín Zabalegui Sanz

Madrid, junio de 2026

Agradecimientos

Me gustaría agradecer a mi familia por acompañarme durante toda la carrera y el trabajo que han conllevado estos 5 años de estudio. También a todos mis amigos por todas esas horas de biblioteca que hemos pasado estudiando y sufriendo juntos para conseguir llegar a donde estamos ahora. En especial, un abrazo a Juan que ha sido mi compañero de éxitos y fracasos en toda esta etapa. Muchas gracias a todos.

INTEGRACIÓN EN VHDL DE LA EXTENSIÓN CHERI EN UN SOFTCORE RISC-V EN FPGA PARA SEGURIDAD DE MEMORIA

Autor: Herremans Humada, Miguel.
Director: Zabalegui Sanz, Fermín.
Entidad Colaboradora: ICAI– Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

Este trabajo presenta la integración de una versión simplificada de la extensión CHERI (Capability Hardware Enhanced RISC Instructions) en un softcore RISC-V implementado en VHDL. Para ello se han añadido registros de capacidades, memoria etiquetada y nuevas instrucciones orientadas a la protección de memoria. La implementación se ha validado mediante una simulación funcional, verificando la restricción de accesos no autorizados y la mejora de la robustez del sistema frente a vulnerabilidades de memoria habituales.

Palabras clave: RISC-V, CHERI, FPGA, VHDL, Capability

1. Introducción

La seguridad de memoria constituye uno de los principales retos en el diseño de sistemas digitales modernos. Una gran parte de las vulnerabilidades software explotadas actualmente tiene su origen en errores relacionados con la gestión de memoria, como los accesos fuera de rango (out-of-bounds access) o los desbordamientos (buffer overflow). Estas vulnerabilidades pueden provocar desde fallos de funcionamiento hasta la ejecución de código malicioso, comprometiendo la integridad y la seguridad de los sistemas.

La arquitectura RISC-V ha experimentado un crecimiento significativo durante los últimos años gracias a su carácter abierto, modular y extensible [1]. Estas características la convierten en una plataforma especialmente adecuada para la investigación y el desarrollo de nuevas soluciones hardware orientadas a mejorar la seguridad y la eficiencia de los sistemas de computación. Sin embargo, al igual que otras arquitecturas convencionales, el conjunto base de instrucciones RISC-V no incorpora mecanismos específicos para proteger los accesos a memoria frente a este tipo de vulnerabilidades.

Con el objetivo de abordar este problema surge CHERI (Capability Hardware Enhanced RISC Instructions) [2], una extensión que introduce el concepto de *capability*, un tipo especial de puntero que incorpora información sobre límites de acceso, permisos y validez. Mediante estos mecanismos, CHERI permite trasladar parte de la protección de memoria al propio hardware, limitando las operaciones que pueden realizarse sobre cada región de memoria y reduciendo significativamente el impacto de posibles errores o ataques.

En este contexto, este trabajo desarrolla la integración de una versión simplificada de la extensión CHERI en un softcore RISC-V implementado en VHDL. Para ello se incorporan nuevas estructuras hardware y un conjunto de instrucciones específicas orientadas a la gestión de *capabilites*, evaluando posteriormente su funcionamiento mediante una simulación y verificando su contribución a la mejora de la seguridad del sistema.

2. Definición del proyecto

El presente proyecto consiste en la integración de una versión simplificada de la extensión CHERI (*Capability Hardware Enhanced RISC Instructions*) en un procesador RISC-V de tipo RV32I descrito en lenguaje VHDL [1] e implementable sobre FPGA. El objetivo principal es dotar al sistema de mecanismos hardware de protección de memoria mediante el uso de *capabilities*, estructuras que incorporan información de direccionamiento, límites de acceso, permisos y validez. De esta manera se consigue un procesador RV32Y.

Para ello, se ha modificado la arquitectura original del procesador incorporando nuevos elementos hardware, entre los que destacan un banco de registros de capacidades, una memoria específica para almacenar *capabilities* y una unidad de comprobación encargada de verificar los permisos y límites asociados a cada acceso. Asimismo, se han implementado nuevas instrucciones de la especificación CHERI para permitir la creación, modificación y utilización de capacidades durante la ejecución de los programas; así como la inspección de estas con fines de prueba.

La verificación del sistema se realiza mediante simulación funcional en Vivado, empleando programas de prueba desarrollados específicamente para evaluar el comportamiento de las nuevas instrucciones y comprobar la capacidad del sistema para restringir accesos no autorizados a memoria.

Aunque la propuesta inicial del proyecto contemplaba la implementación física de la arquitectura sobre una plataforma FPGA, el alcance final del trabajo se centró en el diseño, integración y validación funcional mediante simulación en Vivado. Esta decisión permitió dedicar un mayor esfuerzo al desarrollo y verificación de los mecanismos de protección inspirados en CHERI incorporados a la arquitectura RV32Y.

3. Descripción del sistema

El sistema desarrollado parte de un procesador RISC-V RV32I multiciclo previamente implementado en VHDL. La arquitectura original está compuesta por los elementos habituales de un procesador RISC-V: contador de programa (PC), memoria de instrucciones, banco de registros, unidad aritmeticológica (ALU), memoria de datos y una unidad de control basada en una máquina de estados finitos (FSM) encargada de coordinar la ejecución de las instrucciones [1].

Sobre esta arquitectura base se ha integrado una versión simplificada de la extensión CHERI (*Capability Hardware Enhanced RISC Instructions*) [2], cuyo objetivo es proporcionar mecanismos hardware de protección de memoria mediante el uso de *capabilities*. Una capability es un puntero mejorado y protegido que implementa métodos de seguridad en el hardware. Además de la dirección numérica en memoria de un puntero clásico, una capability contiene un límite de acceso, permisos y un tag, un bit de validez que marcará su validez [1].

Para soportar estas nuevas funcionalidades se han incorporado diversos bloques hardware adicionales. En primer lugar, se ha añadido un banco de registros de capacidades independiente del banco de registros convencional, permitiendo almacenar y manipular capacidades durante la ejecución del programa. Asimismo, se ha implementado una memoria específica para el almacenamiento de *capabilities*, diferenciada de la memoria de datos tradicional.

Además, se ha desarrollado una unidad de comprobación de capacidades (*CHERI Check Unit*), encargada de verificar que cada acceso a memoria cumple las restricciones definidas por la *capability* utilizada. Esta unidad comprueba la validez de la capacidad mediante el campo *tag*, verifica los permisos asociados a la operación solicitada y evalúa que la dirección de acceso se encuentra dentro de los límites autorizados. En caso de que alguno de estos requisitos no se cumple, esta unidad no permite la ejecución de la mayoría de las instrucciones asociadas a CHERI.

La integración de CHERI también ha requerido modificaciones en la unidad de control del procesador. Se han añadido nuevos estados a la FSM para soportar la ejecución de instrucciones específicas de capacidades, entre las que destacan las operaciones de carga y almacenamiento mediante capacidades (*CLoad* y *CStore*), la carga y almacenamiento de capacidades completas (*CLoadCap* y *CStoreCap*), la restricción de permisos (*CSetPerm*), la modificación de desplazamientos (*CIncOffset*), la consulta de validez (*CGetTag*) y las instrucciones de transferencia de control basadas en capacidades (*CJump* y *CJumpLink*). En la figura 1 se puede observar el diagrama de estados de la nueva unidad de control (*u_control_RV32Y*) implementada.

La implementación y validación del sistema se han realizado utilizando el entorno de desarrollo Vivado de AMD/Xilinx, empleando simulación funcional para verificar el correcto comportamiento tanto del conjunto base RV32I como de las nuevas funcionalidades introducidas por CHERI. Para ello se han desarrollado diferentes programas de prueba y bancos de pruebas (*testbenches*) orientados a comprobar la correcta ejecución de las instrucciones implementadas y la efectividad de los mecanismos de protección de memoria incorporados.

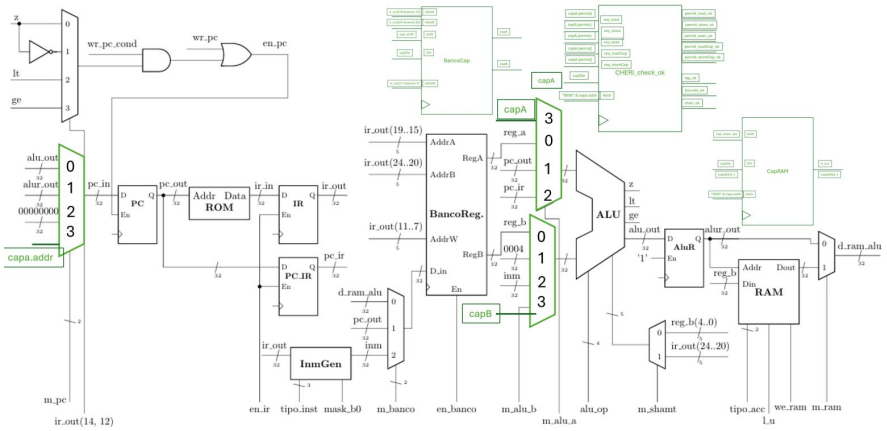


Figura 1.1: Arquitectura del RV32Y

4. Resultados

La integración de la extensión CHERI en el procesador RISC-V permitió incorporar mecanismos de protección de memoria inexistentes en la arquitectura base RV32I. La implementación desarrollada fue validada mediante simulación funcional utilizando Vivado, verificándose tanto el correcto funcionamiento de las instrucciones originales del procesador como de las nuevas instrucciones asociadas a la gestión de *capabilities*.

Los resultados obtenidos demostraron el correcto funcionamiento del banco de registros de capacidades, de la memoria de capacidades y de la unidad de comprobación encargada de validar permisos y límites de acceso. Asimismo, las simulaciones confirmaron la correcta ejecución de las instrucciones implementadas, incluyendo operaciones de carga y almacenamiento mediante capacidades, modificación de permisos, consulta del estado de validez y transferencias de control protegidas.

Los programas de prueba desarrollados permitieron comprobar que los accesos realizados dentro de los límites definidos por una *capability* eran aceptados correctamente, mientras que los intentos de acceso fuera del rango autorizado o con permisos insuficientes eran detectados y bloqueados por la lógica de comprobación implementada. De esta forma, el sistema logró restringir operaciones potencialmente inseguras que en un procesador RV32I convencional podrían ejecutarse sin ningún mecanismo adicional de protección.

Los resultados obtenidos validan la viabilidad de integrar mecanismos inspirados en CHERI en arquitecturas RISC-V implementadas sobre FPGA, demostrando una mejora significativa en la robustez frente a errores de memoria mediante modificaciones hardware. En la Figura 1.2 se observa la simulación de las instrucciones mediante un banco de pruebas.

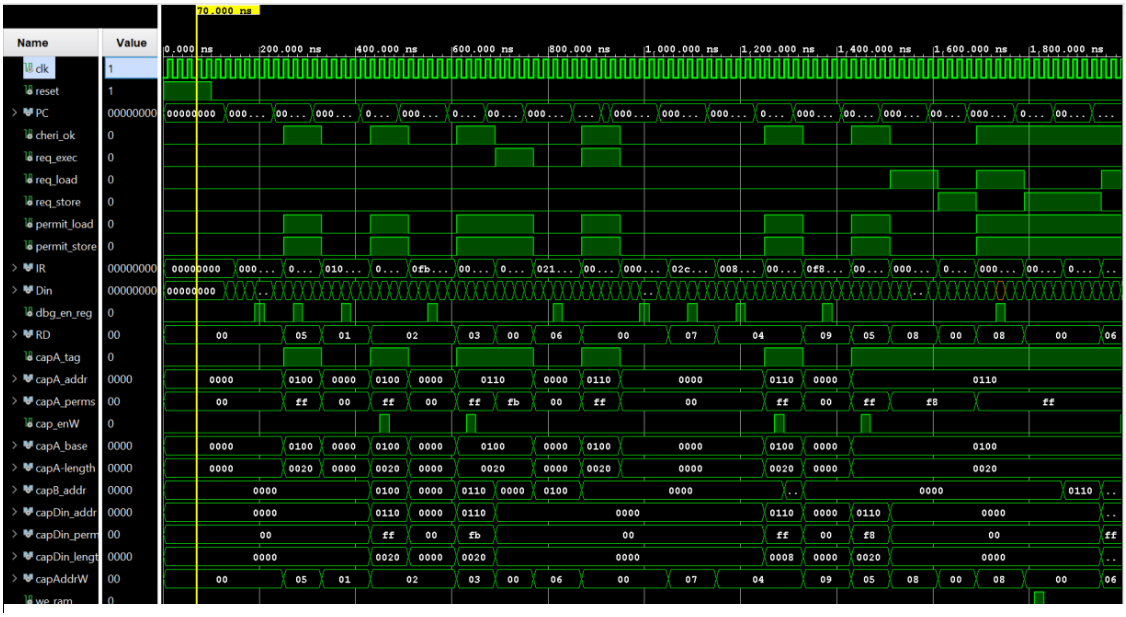


Figura 1.2: Simulación funcional del RV32Y

5. Conclusiones

El desarrollo de este proyecto ha permitido integrar una versión simplificada de la extensión CHERI en un procesador RISC-V RV32I implementado en VHDL. Para ello se han incorporado nuevos elementos hardware destinados a la gestión de *capabilities*, incluyendo un banco de registros específico, una memoria para capacidades y una unidad encargada de verificar permisos y límites de acceso.

La validación mediante simulación funcional ha demostrado el correcto funcionamiento de las instrucciones implementadas y de los mecanismos de protección introducidos. Los resultados obtenidos muestran que es posible restringir accesos no autorizados a memoria y limitar el alcance de posibles errores de programación mediante comprobaciones realizadas directamente en hardware.

Asimismo, el proyecto ha permitido profundizar en el funcionamiento interno de las arquitecturas RISC-V y en las técnicas modernas de protección de memoria, poniendo de manifiesto el potencial de CHERI como solución para mejorar la seguridad de los sistemas de computación sin modificar significativamente el modelo de programación existente.

6. Referencias

[1] Waterman, A.; Asanović, K., *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, RISC-V International.

[2] Watson, R. N. M.; Neumann, P. G.; Woodruff, J.; Roe, M.; Almatary, H.; Anderson, J.; Baldwin, J.; Barnes, G.; Chisnall, D.; Clarke, J.; Davis, B.; Eisen, L.; Filardo, N. W.; Fuchs, F. A.; Grisenthwaite, R.; Joannou, A.; Laurie, B.; Markettos, A. T.; Moore, S. W.; Murdoch, S. J.; Nienhuis, K.; Norton, R.; Richardson, A.; Rugg, P.; Sewell, P.; Son, S.; Xia, H. “Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)”. Technical Report UCAM-CL-TR-987, University of Cambridge Computer Laboratory, Septiembre 2023. Disponible en: https://www.ietfng.org/nwf/_downloads/2e59e293c1afa58f0772fda209e6b54c/2023-cherisav9.pdf.

INTEGRATION IN VHDL OF THE CHERI EXTENSION INTO A RISC-V SOFTCORE ON FPGA FOR MEMORY SECURITY

Author: Herremans Humada, Miguel.
Supervisor: Zabalegui Sanz. Fermín.
Collaborating Entity: ICAI– Universidad Pontificia Comillas

ABSTRACT

This paper presents the integration of a simplified version of the CHERI (Capability Hardware Enhanced RISC Instructions) extension into a RISC-V softcore implemented in VHDL. For this, capability registers, tagged memory, and new instructions aimed at memory protection have been added. The implementation has been validated through functional simulation, checking the restriction of unauthorized access and the improvement of the system's robustness against common memory vulnerabilities.

Keywords: RISC-V, CHERI, FPGA, VHDL, Capability.

1. Introduction

Memory safety is one of the main challenges in designing modern digital systems. A large portion of currently exploited software vulnerabilities originates from errors related to memory management, such as out-of-bounds access or buffer overflows. These vulnerabilities can lead to anything from system crashes to the execution of malicious code, compromising system integrity and security.

The RISC-V architecture has experienced significant growth in recent years thanks to its open, modular, and extensible nature [1]. These characteristics make it an especially suitable platform for research and the development of new hardware solutions aimed at improving the security and efficiency of computing systems. However, just like other conventional architectures, the RISC-V base instruction set does not include specific mechanisms to protect memory accesses against this type of vulnerability.

To tackle this problem, CHERI (Capability Hardware Enhanced RISC Instructions) [2] was developed, an extension that introduces the concept of a capability, a special type of pointer that includes information about access boundaries, permissions, and validity. Through these mechanisms, CHERI allows some of the memory protection to be handled directly by the hardware, limiting the operations that can be performed on each memory region and significantly reducing the impact of possible attacks.

In this context, this work develops the integration of a simplified version of the CHERI extension into a RISC-V softcore implemented in VHDL. To achieve this, new hardware structures and a set of specific instructions aimed at capability management are incorporated, subsequently evaluating their operation through simulation and verifying their contribution to improving system security.

2. System description

The system developed is based on a previously implemented multi-cycle RISC-V RV32I processor in VHDL. The original architecture consists of the usual elements of a RISC-V processor: program counter (PC), instruction memory, register bank, arithmetic logic unit (ALU), data memory, and a control unit based on a finite state machine (FSM) responsible for coordinating instruction execution [1].

On top of this base architecture, a simplified version of the CHERI (Capability Hardware Enhanced RISC Instructions) extension [2] has been integrated, aimed at providing hardware memory protection mechanisms using capabilities. A capability is an enhanced and protected pointer that implements security methods in hardware. In addition to the numeric memory address of a classic pointer, a capability contains an access limit, permissions, and a tag, a validity bit that indicates whether it is valid [1].

To support these new features, several additional hardware blocks have been added. First, an independent capability register bank has been included, separate from the conventional register bank, allowing capabilities to be stored and manipulated during program execution. In addition, specific memory for storing capabilities has been implemented, distinct from traditional data memory.

Furthermore, a Capability Check Unit (CHERI Check Unit) has been developed, responsible for ensuring that each memory access meets the restrictions defined by the capability used. This unit checks the validity of the capability through the tag field, verifies the permissions associated with the requested operation, and ensures that the access address is within authorized boundaries. If any of these requirements are not met, this unit prevents most instructions associated with CHERI from executing.

The integration of CHERI has also required modifications to the processor's control unit. New states have been added to the FSM to support the execution of capability-specific instructions, including load and store operations using capabilities (CLoad and CStore), loading and storing full capabilities (CLoadCap and CStoreCap), permission restriction (CSetPerm), offset modification (CIncOffset), validity checking (CGetTag), and capability-based control transfer instructions (CJump and CJumpLink). Figure 1.3 shows the state diagram of the new control unit (u_control_RV32Y) that was implemented.

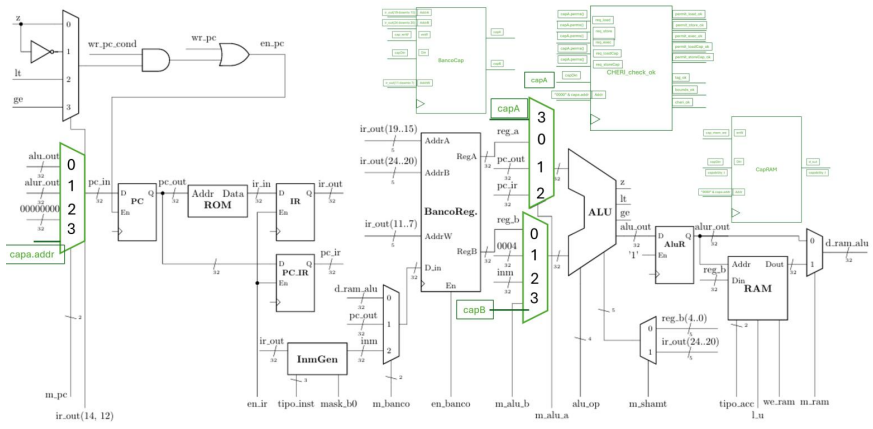


Figura 1.3: RV32Y architecture

The system implementation and validation have been carried out using AMD/Xilinx's Vivado development environment, employing functional simulation to verify the correct behavior of both the base RV32I set and the new features introduced by CHERI. To do this, different test programs and testbenches have been developed aimed at checking the correct execution of the implemented instructions and the effectiveness of the built-in memory protection mechanisms.

Although the project's initial proposal included the physical implementation of the architecture on an FPGA platform, the final scope of the work focused on the design, integration, and functional validation through simulation in Vivado. This decision allowed more effort to be devoted to the development and verification of the CHERI-inspired protection mechanisms incorporated into the RV32Y architecture.

3. Results

The integration of the CHERI extension into the RISC-V processor allowed the incorporation of memory protection mechanisms that didn't exist in the base RV32I architecture. The implementation developed was validated through functional simulation using Vivado, verifying both the correct operation of the processor's original instructions and the new instructions associated with capability management.

The results obtained demonstrated the correct functioning of the capability register file, the capability memory, and the checking unit responsible for validating access permissions and limits. Likewise, the simulations confirmed the correct execution of the implemented

Figura 1.4: Functional simulation of the RV32Y

instructions, including load and store operations via capabilities, permission modification, checking validity status, and protected control transfers.

The test programs that were developed allowed us to verify that accesses made within the limits defined by a capability were accepted correctly, while attempts to access outside the authorized range or with insufficient permissions were detected and blocked by the implemented checking logic. In this way, the system managed to restrict potentially unsafe operations that could run on a conventional RV32I processor without any additional protection mechanism.

The results obtained validate the feasibility of integrating CHERI-inspired mechanisms into RISC-V architectures implemented on FPGA, demonstrating a significant improvement in robustness against memory errors through hardware modifications. In Figure 1.4, you can see the simulation of the instructions through a test bench,

4. Conclusions

The development of this project has made it possible to integrate a simplified version of the CHERI extension into a RISC-V RV32I processor implemented in VHDL. To achieve this, new hardware elements have been added to manage capabilities, including a dedicated register bank, a memory for capabilities, and a unit responsible for verifying permissions and access limits.

Validation through functional simulation has shown the correct operation of the implemented instructions and the protection mechanisms introduced. The results obtained demonstrate that it is possible to restrict unauthorized memory access and limit the scope of potential programming errors through checks performed directly in hardware.

Additionally, the project has allowed for a deeper look into the internal workings of RISC-V architectures and modern memory protection techniques, highlighting the potential of CHERI as a solution to improve the security of computing systems without significantly changing the existing programming model.

5. References

[1] Waterman, A.; Asanović, K., *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, RISC-V International.

[2] Watson, R. N. M.; Neumann, P. G.; Woodruff, J.; Roe, M.; Almatary, H.; Anderson, J.; Baldwin, J.; Barnes, G.; Chisnall, D.; Clarke, J.; Davis, B.; Eisen, L.; Filardo, N. W.; Fuchs, F. A.; Grisenthwaite, R.; Joannou, A.; Laurie, B.; Markettos, A. T.; Moore, S. W.; Murdoch, S. J.; Nienhuis, K.; Norton, R.; Richardson, A.; Rugg, P.; Sewell, P.; Son, S.; Xia, H. "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)". Technical Report UCAM-CL-TR-987, University of Cambridge Computer Laboratory, Septiembre 2023. Disponible en: https://www.ietfng.org/nwf/_downloads/2e59e293c1afa58f0772fda209e6b54c/2023-cherisav9.pdf.

Índice de la memoria

Capítulo 1. Introducción 7

Capítulo 2. Descripción de las Tecnologías 9

2.1 Arquitectura RISC-V 9

2.2 Lenguaje VHDL 11

2.3 Entorno de desarrollo Vivado 12

Capítulo 3. Estado de la Cuestión 13

3.1 Seguridad de memoria en sistemas actuales 13

3.2 Arquitecturas basadas en capabilites 15

3.2.1 Concepto de una capability 15

3.2.2 Arquitectura CHERI 17

3.3 Implementaciones reales de CHERI sobre RISC-V 17

Capítulo 4. Definición del Trabajo 19

4.1 Justificación 19

4.1.1 Problema que resolver 19

4.1.2 Interés Técnico del proyecto 19

4.1.3 Aportación del trabajo 20

4.2 Objetivos 20

4.2.1 Estudio de la arquitectura CHERI 21

4.2.2 Implementación de la extensión CHERI sobre RISC-V 21

4.2.3 Implementación sobre FPGA 21

4.2.4 Desarrollo de un entorno de prueba y validación 22

4.2.5 Evaluación del impacto de la extensión CHERI 22

4.3 Metodología 22

4.3.1 Fase de estudio e investigación 22

4.3.2 Fase de diseño 23

4.3.3 Fase de implementación 23

4.3.4 Fase de validación 23

4.4 Planificación y estimación económica 24

Capítulo 5. Sistema Desarrollado 27

5.1 Análisis del Sistema 27
5.1.1 Arquitectura RISC-V original *27*
5.1.2 Limitaciones respecto a seguridad *28*
5.2 Diseño 30
5.2.1 Arquitectura RV32Y propuesta *30*
5.2.2 Bloques hardware incorporados *33*
5.2.3 Diseño de las capabilities *39*
5.2.4 Capability raíz *40*
5.3 Implementación 41
5.3.1 Modificaciones del datapath *41*
5.3.2 Modificaciones de la FSM *43*
5.3.3 Implementación de las instrucciones CHERI *48*
5.3.4 Instrucción CSetBounds *49*
5.3.5 Instrucción CSetPerm *50*
5.3.6 Instrucción CIncOffset *52*
5.3.7 Instrucciones de inspección de capabilities: CGetTag, CGetPerms, CGetBase, CGetLen y CGetAddr *53*
5.3.8 Instrucción CLoad *55*
5.3.9 Instrucción CStore *56*
5.3.10 Instrucción CLoadCap *57*
5.3.11 Instrucción CStoreCap *59*
5.3.12 Instrucciones CJump y CJumpLink *60*

***Capítulo 6. Análisis de Resultados* 62**

6.1 Verificación funcional del RV32I 62
6.1.1 Validación de instrucciones artimeticológicas *64*
6.1.2 Validación de instrucciones de acceso a memoria *66*
6.1.3 Validación de instrucciones de salto condicional *67*
6.1.4 Validación de instrucciones de salto, comparación y desplazamiento *69*
6.2 Verificación funcional de los bloques hardware 72
6.2.1 Verificación del banco de registros de capabilities *72*
6.2.2 Verificación de la memoria de capabilities (CapRAM) *74*
6.2.3 Verificación de la unidad CHERI Check Unit *75*
6.3 Verificación funcional del RV32Y 79

6.3.1 Validación de instrucciones de inspección y gestión *80*
6.3.2 Validación de instrucciones de transferencia de control *82*
6.3.3 Validación de instrucciones de acceso protegido a memoria *85*
6.3.4 Validación de almacenamiento y recuperación de capabilities *87*
6.4 Casos de uso y análisis de seguridad 89
6.4.1 Restricción de permisos *89*
6.4.2 Protección frente a accesos fuera de límites *90*
6.4.3 Protección de acceso a memoria *91*
6.5 Aplicación en un sistema de telecomunicaciones 91

Capítulo 7. Conclusiones y Trabajos Futuros 94
7.1 Conclusiones 94
7.2 Cumplimiento de objetivos 95
7.3 Implementación de excepciones CHERI 96
7.4 Incorporación de una Unidad de Gestión de Memoria (MMU) 97

Capítulo 8. Bibliografía 99

ANEXO I: ALINEACIÓN DEL PROYECTO CON LOS ODS 101
Anexo I.1 - EDUCACIÓN DE CALIDAD. ODS 4 101
Anexo I.2 - TRABAJO DECENTE Y CRECIMIENTO ECONÓMICO. ODS 8 101
Anexo I.3 - INDUSTRIA, INNOVACIÓN E INFRAESTRUCTURA. ODS 9 101
Anexo I .4 - REDUCCIÓN DE LAS DESIGUALDADES. ODS 10 102
Anexo I .5 - ENERGÍA ASEQUIBLE Y NO CONTAMINANTE. ODS 7 102

ANEXO II: Ficheros nuevos del RV32Y 103

ANEXO III: Bancos de prueba para las simulaciones del RV32Y 125

Índice de figuras

Figura 1.1: Arquitectura del RV32Y 11
Figura 1.2: Simulación funcional del RV32Y 12
Figura 1.3: RV32Y architecture 16
Figura 1.4: Functional simulation of the RV32Y 17
Figura 2.1: Arquitectura del RV32I (Apuntes SSDDII Icai) 9
Figura 2.2: Conjunto de instrucciones del ICAI-RISCV (Apuntes SSDDII Icai) 10
Figura 3.1:Ejemplo Buffer-overflow (https://www.acunetix.com/blog/web-security-zone/what-is-buffer-overflow/) 14
Figura 5.1: Arquitectura ICAI-RISC-V, RV32I (Apuntes SSDDII Icai) 28
Figura 5.2: Arquitectura del RV32Y 32
Figura 5.3: Bloque BancoCap 34
Figura 5.4: Bloque CapRAM 35
Figura 5.5: Bloque CHERI_check_unit 38
Figura 5.6: Definición de una capability en VHDL 40
Figura 5.7: FSM modificada 47
Figura 5.8: Direccionamiento de entrada de la memoria RAM 58
Figura 5.9: Cambio en la entrada del PC 61
Figura 6.1: Simulación de instrucciones aritmeticológicas en Vivado (1) 64
Figura 6.2: Simulación de instrucciones aritmeticológicas en Vivado (2) 65
Figura 6.3: Simulación de instrucciones de almacenamiento y carga en Vivado 66
Figura 6.4: Simulación de la instrucción BEQ en Vivado 67
Figura 6.5: Simulación de la instrucción BLT en Vivado 68
Figura 6.6: Simulación de la instrucción BNE en Vivado 68
Figura 6.7: Simulación de instrucciones de salto, comparación y desplazamiento en Vivado (2) 70
Figura 6.8: Simulación de instrucciones de salto, comparación y desplazamiento en Vivado (1) 70
Figura 6.9: Simulación de BancoCap en Vivado 73

Figura 6.10: Simulación de CapRAM en Vivado 75
Figura 6.11: Simulación de cheri_check_unit en Vivado 77
Figura 6.12: Simulación de instrucciones de inspección y gestión en Vivado (1) 80
Figura 6.13: Simulación de instrucciones de inspección y gestión en Vivado (2) 81
Figura 6.14: Simulación de instrucciones de transferencia de control en Vivado (1) 84
Figura 6.15: Simulación de instrucciones de transferencia de control en Vivado (2) 84
Figura 6.16: Simulación de instrucciones de acceso a memoria en Vivado (1) 86
Figura 6.17: Simulación de instrucciones de acceso a memoria en Vivado (2) 87
Figura 6.18: Simulación de instrucciones de almacenamiento y recuperación de capabilities en Vivado 89

Índice de tablas

Tabla 4.1: Planificación temporal del trabajo desarrollado 25
Tabla 4.2: Estimación económica del trabajo 26
Tabla 5.1: Descripción de los nuevos estados del RV32Y 45
Tabla 5.2:Descripción de las nuevas instrucciones CHERI 49
Tabla 5.3: Codificación de las instrucciones de inspección 54
Tabla 6.1: Programa ejecutado para la simulación de las instrucciones aritmeticológicas . 65
Tabla 6.2: Programa ejecutado para la simulación de instrucciones de acceso a memoria 67
Tabla 6.3: Programa ejecutado para la simulación de instrucciones de salto condicional .. 69
Tabla 6.4:Programa ejecutado para la simulación de instrucciones de salto, comparación y desplazamiento 71
Tabla 6.5: Pruebas comprobadas durante la simulación de BancoCap 74
Tabla 6.6: Pruebas comprobadas durante la simulación de CapRAM 75
Tabla 6.7: Pruebas comprobadas durante la simulación de cheri_check_unit 78
Tabla 6.8: Pruebas comprobadas durante la simulación de las instrucciones de inspección y gestión 82
Tabla 6.9: Pruebas comprobadas durante la simulación de las instrucciones de almacenamiento y recuperación de capabilities 88

Capítulo 1. INTRODUCCIÓN

La arquitectura RISC-V va ganando importancia cada día frente al resto de arquitecturas de microprocesadores. Como miembro de la familia de las arquitecturas RISC (Reduced Instruction Set Computer), se basa en un conjunto de instrucciones reducidos y eficientes. La arquitectura RISC-V destaca sobre el resto debido a ser abierta y libre de licencias, lo que permite la innovación y el desarrollo libre de nuevas extensiones. [1]

Sin embargo, pese a su diseño modular y su flexibilidad, esta arquitectura presenta una serie de carencias relevantes respecto a la seguridad en memoria. Los programas son vulnerables ante diferentes problemas como puede ser el buffer overflow, un error que ocurre cuando un programa escribe más datos de los que la memoria soporta y se sobreescriben datos sobre otros ya existentes. Estos fallos constituyen una las vías de explotación más frecuentes en estos sistemas.[2]

Por ello, en este proyecto abordará la implementación de las instrucciones CHERI (Capability Hardware Enhanced RISC Instructions) para subsanar los posibles fallos en memoria que se puedan producir durante la ejecución de un programa en una FPGA con arquitectura RISC-V, proporcionando una protección de memoria más robusta y reduciendo el riesgo durante la ejecución de dichos programas.

1.1 MOTIVACIÓN DEL PROYECTO

Las arquitecturas RISC están experimentando un crecimiento notable en los sectores tecnológicos actuales, donde la eficiencia energética, la simplicidad del diseño y la capacidad de integración son claves como los sistemas embebidos o dispositivos autónomos. RISC-V en concreto, tiene una posición privilegiada dentro de este auge gracias a su naturaleza abierta y extensible. Permite la implementación de nuevas extensiones sin las restricciones de sus dueños y además de ser más eficiente energéticamente que las arquitecturas CISC (Complex Instructions Set Computer). Por ello dotar a una arquitectura

RISC de protección de memoria, una de sus mayores vulnerabilidades, hace que se convierta en una opción mucho más robusta que su contraparte.

La extensión CHERI ofrece un enfoque perfecto para el problema a resolver al introducir capabilities, punteros enriquecidos con información de límites y permisos que evitan accesos fuera de rango. Por ello, la implementación de CHERI en un softcore RISC V en FPGA presenta una solución viable y coherente con el problema planteado.

Capítulo 2. DESCRIPCIÓN DE LAS TECNOLOGÍAS

2.1 ARQUITECTURA RISC-V

La arquitectura RISC-V es una arquitectura de conjunto de instrucciones (*Instruction Set Architecture*, ISA) abierta y basada en los principios de la filosofía RISC (*Reduced Instruction Set Computer* [1].

A diferencia de otras arquitecturas comerciales ampliamente utilizadas, como x86 o ARM, RISC-V se caracteriza por disponer de una especificación abierta que puede ser implementada, modificada y ampliada sin necesidad de acuerdos de licencia. Esta característica ha impulsado su adopción tanto en entornos académicos como industriales, convirtiéndose en una de las arquitecturas con mayor crecimiento durante los últimos años [2].

La especificación de RISC-V se organiza de forma modular. En lugar de definir una única arquitectura cerrada, se establece un conjunto base de instrucciones al que pueden añadirse extensiones específicas para diferentes aplicaciones. Esta filosofía permite adaptar el procesador a los requisitos de cada sistema, incorporando únicamente aquellas funcionalidades que resultan necesarias, como se desarrolla en este trabajo con la extensión CHERI.

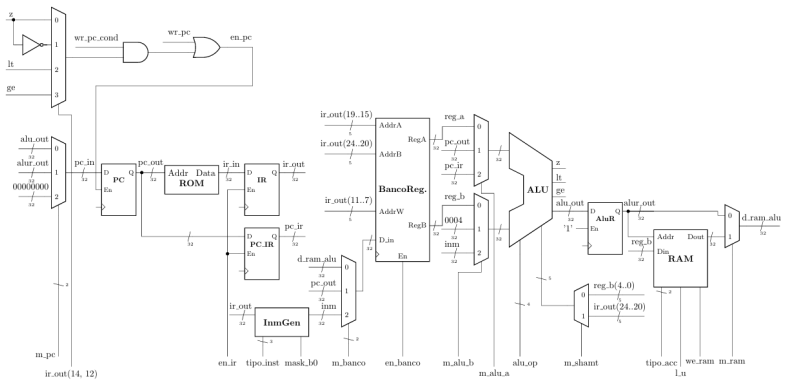


Figura 2.1: Arquitectura del RV32I (Apuntes SSDDII Icai)

En este proyecto se ha utilizado como punto de partida la arquitectura RV32I, correspondiente a la versión de 32 bits del conjunto base de instrucciones enteras de RISC-V. Este conjunto incluye operaciones aritméticas y lógicas, instrucciones de carga y almacenamiento en memoria, instrucciones de salto y bifurcación, así como mecanismos básicos para el control del flujo de ejecución. Gracias a su simplicidad y amplia documentación, RV32I constituye una plataforma adecuada para la experimentación e integración de nuevas extensiones hardware.

La Figura 2.2 muestra el conjunto base de instrucciones RV32I definido por la especificación RISC-V. Este conjunto incluye instrucciones aritmeticológicas, operaciones de carga y almacenamiento, instrucciones de salto y bifurcación, así como operaciones de

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]			rd	0110111	LUI
imm[31:12]			rd	0010111	AUIPC
imm[20|10:1|11|19:12]			rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm	pred succ	rs1	000	rd	0001111	FENCE

Figura 2.2: Conjunto de instrucciones del ICAI-RISCV (Apuntes SSDDII Icai)

control del sistema. La arquitectura desarrollada en este trabajo toma como punto de partida dicho conjunto de instrucciones y lo amplía mediante la incorporación de nuevas instrucciones inspiradas en la arquitectura CHERI, manteniendo la compatibilidad con el funcionamiento original del procesador [1].

2.2 LENGUAJE VHDL

VHDL (*VHSIC Hardware Description Language*) es un lenguaje de descripción hardware. Su principal objetivo es permitir la especificación, modelado, simulación y síntesis de sistemas digitales complejos [3].

A diferencia de los lenguajes de programación tradicionales, VHDL no describe una secuencia de instrucciones ejecutadas por un procesador, sino el comportamiento y la estructura de circuitos electrónicos.

Una de las principales ventajas de VHDL es la posibilidad de realizar simulaciones funcionales antes de la implementación física del diseño. Este proceso permite verificar el comportamiento del sistema, detectar errores y validar la lógica implementada sin necesidad de disponer del hardware final. Por este motivo, VHDL es ampliamente utilizado en entornos académicos e industriales para el desarrollo de circuitos digitales y sistemas embebidos.

En este Trabajo Fin de Grado, VHDL ha sido utilizado para implementar tanto la arquitectura RV32I original como las modificaciones necesarias para incorporar los mecanismos de protección de memoria inspirados en CHERI. Todos los bloques desarrollados, incluyendo el BancoCap, la CapRAM, la unidad CHERI Check Unit y las modificaciones realizadas en la unidad de control, han sido descritos mediante este lenguaje.

Además, VHDL ha permitido desarrollar bancos de pruebas (*testbenches*) específicos para verificar el funcionamiento de cada uno de los bloques implementados y validar posteriormente el comportamiento completo de la arquitectura RV32Y. Gracias a estas

simulaciones se ha podido comprobar el correcto funcionamiento de las instrucciones CHERI antes de su integración definitiva en el sistema.

2.3 ENTORNO DE DESARROLLO VIVADO

Para el desarrollo del proyecto se ha utilizado Vivado, el entorno de diseño hardware de AMD-Xilinx. Esta herramienta permite realizar la edición de código VHDL, la simulación funcional de los diseños y su posterior implementación sobre dispositivos programables [4].

En este trabajo Vivado se ha empleado principalmente para la simulación y verificación de la arquitectura desarrollada. Mediante los bancos de pruebas implementados se ha validado el funcionamiento de los distintos módulos diseñados y de las nuevas instrucciones incorporadas a la arquitectura RV32Y.

Capítulo 3. ESTADO DE LA CUESTIÓN

3.1 SEGURIDAD DE MEMORIA EN SISTEMAS ACTUALES

La seguridad de memoria constituye uno de los principales desafíos en el diseño de sistemas informáticos modernos. Una parte significativa de las vulnerabilidades detectadas en sistemas operativos, aplicaciones y dispositivos embebidos tiene su origen en errores relacionados con el acceso incorrecto a la memoria [5].

Las arquitecturas tradicionales basadas en punteros permiten que los programas accedan directamente a posiciones de memoria mediante direcciones almacenadas en registros o variables. Aunque este mecanismo proporciona una gran flexibilidad, también introduce riesgos importantes cuando los accesos no son controlados adecuadamente. En estos casos, errores de programación simples pueden derivar en vulnerabilidades de seguridad capaces de comprometer completamente el sistema.

Entre los problemas más habituales destacan los accesos fuera de límites (*buffer overflows*), los accesos a memoria liberada (*use-after-free*) y la corrupción de punteros. Estas vulnerabilidades permiten que un proceso acceda a regiones de memoria que no le pertenecen o modifique información crítica utilizada por otros procesos o por el propio sistema operativo [6].

Los desbordamientos de búfer constituyen uno de los ejemplos más conocidos. Este tipo de vulnerabilidad aparece cuando un programa escribe más información de la que puede almacenar una región de memoria determinada. Como consecuencia, los datos sobrantes sobrescriben posiciones adyacentes, pudiendo alterar variables de control, direcciones de retorno o estructuras internas del sistema. Históricamente, este tipo de errores ha sido responsable de numerosas vulnerabilidades críticas utilizadas para ejecutar código malicioso de forma remota [7].

Otro problema especialmente relevante es el denominado *use-after-free*. Esta situación se produce cuando un programa continúa utilizando un puntero asociado a una región de memoria que ya ha sido liberada. Si dicha memoria es posteriormente reutilizada por otro proceso o componente del sistema, el acceso mediante el puntero antiguo puede provocar comportamientos impredecibles, corrupción de datos o vulnerabilidades explotables por un atacante [6].

La corrupción de punteros constituye otra fuente importante de vulnerabilidades. Este problema aparece cuando el valor de un puntero es alterado de forma indebida, ya sea como consecuencia de errores de programación, corrupción de memoria o ataques maliciosos. Como resultado, el programa puede acceder a regiones de memoria no autorizadas, modificar información crítica o perder el control sobre el flujo normal de ejecución. Debido a que los punteros tradicionales únicamente contienen una dirección de memoria, el hardware convencional carece de mecanismos para verificar si dicha referencia sigue siendo válida o si dispone de permisos adecuados para realizar la operación solicitada [6].

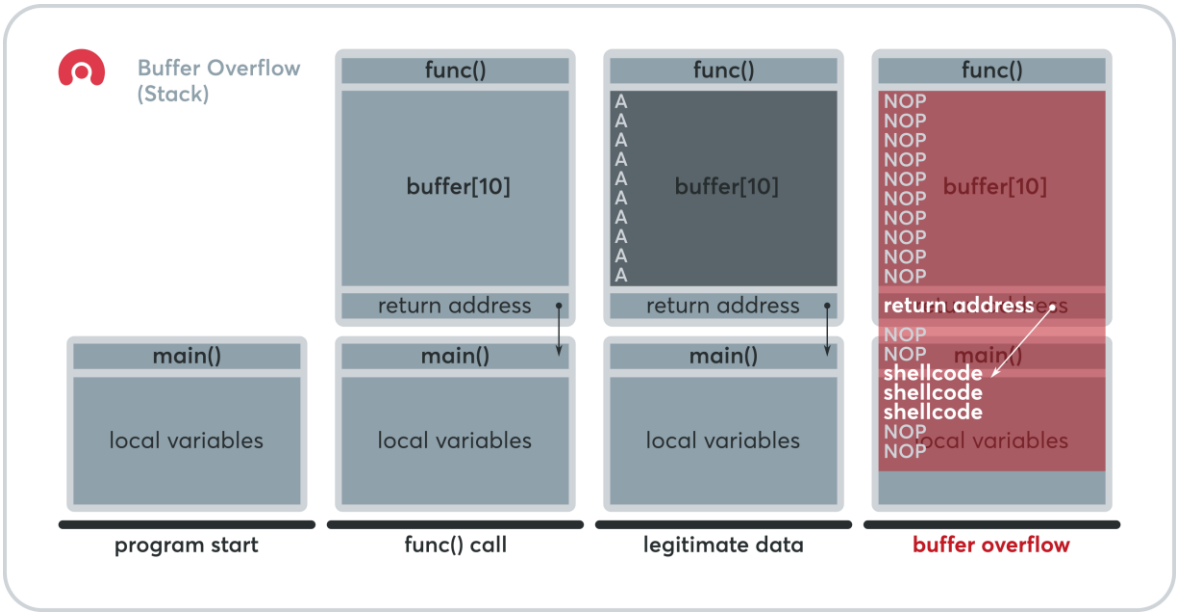


Figura 3.1:Ejemplo Buffer-overflow (https://www.acunetix.com/blog/web-security-zone/what-is-buffer-overflow/)

La Figura 3.1 ilustra de forma simplificada algunos de los principales problemas de seguridad asociados al uso de punteros tradicionales.

Durante los últimos años, diversos estudios han puesto de manifiesto que una parte muy significativa de las vulnerabilidades de software publicadas tiene su origen en errores relacionados con la gestión de memoria [8]. Esta situación ha impulsado la búsqueda de nuevas soluciones capaces de proporcionar mecanismos de protección más robustos directamente desde el hardware.

En este contexto surgen las arquitecturas basadas en capabilities, cuyo objetivo es complementar los punteros tradicionales con información adicional sobre permisos, límites y validez. Entre las propuestas existentes, CHERI se ha consolidado como una de las iniciativas más relevantes en el ámbito de la seguridad de memoria, ofreciendo mecanismos hardware capaces de reducir significativamente el impacto de este tipo de vulnerabilidades [9].

Debe señalarse que las distintas vulnerabilidades descritas requieren mecanismos de protección diferentes. La implementación desarrollada en este trabajo se centra en la protección frente a errores de seguridad espacial mediante la comprobación de límites, permisos y validez de las capabilities. En cambio, la protección frente a vulnerabilidades temporales como use-after-free requiere mecanismos adicionales de revocación de capabilities que quedan fuera del alcance del presente proyecto.

3.2 ARQUITECTURAS BASADAS EN CAPABILITES

3.2.1 CONCEPTO DE UNA CAPABILITY

Las arquitecturas basadas en *capabilities* surgen como una alternativa a los mecanismos tradicionales de protección de memoria. Su objetivo principal es asociar a cada referencia de memoria información adicional que permita controlar de forma precisa qué recursos pueden utilizarse y en qué condiciones [10]. A diferencia de un puntero convencional, una capability no almacena únicamente una dirección de memoria, sino

también información relacionada con los límites de acceso, los permisos disponibles y la validez de la propia referencia.

El concepto de capability puede entenderse como un puntero protegido que incorpora mecanismos de seguridad directamente en hardware. Gracias a esta información adicional, el procesador puede verificar automáticamente si una operación de lectura, escritura o ejecución está autorizada antes de acceder a la memoria. De esta forma, se evita que errores software permitan acceder a regiones no autorizadas o ejecutar operaciones para las que el proceso no dispone de privilegios suficientes.

Una capability suele estar formada por varios elementos fundamentales:

- **Dirección base (Base):** indica el inicio de la región de memoria accesible.
- **Longitud (Length):** determina el tamaño máximo de dicha región.
- **Dirección actual (Address):** dirección utilizada por la operación de acceso.
- **Permisos (Perms):** conjunto de operaciones autorizadas sobre la región.
- **Bit de validez (Tag):** indica si la capability ha sido generada y mantenida correctamente por el hardware.

Cualquier acceso puede ser validado antes de ejecutarse. Si la dirección solicitada se encuentra fuera de los límites determinados o la operación requerida no dispone de los permisos correspondientes, el hardware puede bloquear automáticamente el acceso. Este enfoque permite trasladar parte de la responsabilidad de la protección desde el software hacia el propio procesador.

Las arquitecturas basadas en capabilities no son un concepto reciente. Diversos sistemas experimentales exploraron esta filosofía durante las décadas de 1970 y 1980. Sin embargo, las limitaciones tecnológicas de la época dificultaron su implementación. El incremento actual de las amenazas relacionadas con la seguridad software y la disponibilidad de

hardware más avanzado han favorecido la vuelta de estas soluciones como una vía factible para la protección de memoria [11].

3.2.2 ARQUITECTURA CHERI

CHERI (*Capability Hardware Enhanced RISC Instructions*) es una arquitectura desarrollada por la Universidad de Cambridge cuyo objetivo es incorporar mecanismos de protección de memoria basados en capabilities directamente en el hardware del procesador. Para ello amplía las arquitecturas convencionales mediante nuevos registros, instrucciones y estructuras de datos capaces de almacenar y gestionar capabilities. [10]

Las capabilities de CHERI incluyen información sobre límites de memoria, permisos de acceso y validez, permitiendo que el hardware compruebe automáticamente cada operación antes de ejecutarla. Este enfoque permite reducir el impacto de vulnerabilidades relacionadas con la gestión de memoria, como los accesos fuera de límites o la corrupción de punteros. [10]

La arquitectura CHERI ha sido implementada sobre diferentes plataformas de investigación, como ARM Morello, demostrando la viabilidad de integrar mecanismos avanzados de protección de memoria manteniendo la compatibilidad con arquitecturas existentes. [12]

3.3 IMPLEMENTACIONES REALES DE CHERI SOBRE RISC-V

La evolución reciente de CHERI no se limita ya a prototipos conceptuales o a la plataforma Morello sobre ARM, sino que cuenta con varias implementaciones reales sobre RISC-V, tanto en el ámbito académico como en el industrial. La referencia principal sigue siendo CHERI-RISC-V, mantenida por el grupo CTSRD de la Universidad de Cambridge. La propia página de Cambridge indica que CHERI-RISC-V es ya su plataforma preferida de investigación software y que existe un grupo de trabajo en RISC-V International orientado a su estandarización. [17]

Un segundo eje importante es CHERIoT, impulsado por Microsoft Research. Este proyecto redefine CHERI para microcontroladores y sistemas embebidos de bajo coste, con un ISA y un modelo software orientados a seguridad espacial a nivel de objeto, protección frente a ciertos escenarios de use-after-free (los cuales no entran dentro del alcance de este proyecto) y compartimentación ligera expuesta al modelo de programación en C/C++. El repositorio oficial confirma que sigue tratándose de una plataforma de investigación abierta y no todavía de un producto listo para producción. [18]

Capítulo 4. DEFINICIÓN DEL TRABAJO

4.1 JUSTIFICACIÓN

4.1.1 PROBLEMA QUE RESOLVER

La seguridad de memoria continúa siendo uno de los principales retos en el diseño de sistemas digitales y procesadores modernos. Vulnerabilidades como los accesos fuera de límites, la corrupción de punteros o los desbordamientos de memoria constituyen una fuente habitual de fallos y brechas de seguridad en sistemas embebidos y aplicaciones críticas. Aunque la arquitectura RISC-V ofrece ventajas significativas en términos de simplicidad, modularidad y apertura, su conjunto base de instrucciones no incorpora mecanismos hardware específicos para mitigar este tipo de vulnerabilidades.

En los últimos años han surgido diferentes propuestas para reforzar la seguridad de memoria mediante mecanismos implementados directamente en hardware. Entre ellas destaca CHERI (Capability Hardware Enhanced RISC Instructions), una arquitectura basada en capabilities que permite asociar permisos y límites de acceso a cada referencia de memoria, reduciendo significativamente la superficie de ataque de los sistemas software.

4.1.2 INTERÉS TÉCNICO DEL PROYECTO

La integración de CHERI en arquitecturas RISC-V representa actualmente una de las líneas de investigación más relevantes en el ámbito de la seguridad hardware. Diversos proyectos académicos e industriales han demostrado la viabilidad de este enfoque, siendo el proyecto Morello uno de los ejemplos más representativos de adopción de las tecnologías CHERI [9].

Sin embargo, la mayoría de las implementaciones existentes se han desarrollado sobre procesadores complejos o plataformas de investigación específicas. Este trabajo propone una aproximación diferente, centrada en la integración de mecanismos CHERI en

un softcore RISC-V multiciclo descrito en VHDL, permitiendo estudiar de forma detallada las modificaciones necesarias sobre el datapath, la unidad de control y los subsistemas de memoria.

4.1.3 APORTACIÓN DEL TRABAJO

La principal aportación de este proyecto consiste en el diseño e implementación de una extensión inspirada en CHERI sobre una arquitectura RISC-V funcional, incorporando mecanismos de protección de memoria mediante capabilities.

Para ello se han desarrollado nuevos bloques hardware, entre los que destacan:

- Banco de registros de capabilities (BancoCap).
- Memoria específica para almacenamiento de capabilities (CapRAM).
- Unidad de comprobación de permisos y límites (CHERI Check Unit).
- Nuevas instrucciones de manipulación de capabilities.
- Extensión de la unidad de control para soportar las nuevas operaciones.

Además de la implementación hardware, se ha realizado una validación funcional completa mediante bancos de pruebas desarrollados en VHDL, verificando tanto el funcionamiento original del procesador RV32I como las nuevas funcionalidades introducidas por la extensión CHERI.

4.2 OBJETIVOS

El objetivo principal de este Trabajo Fin de Grado es estudiar la viabilidad de integrar mecanismos de protección de memoria inspirados en la arquitectura CHERI sobre un procesador RISC-V desarrollado en VHDL, analizando las modificaciones necesarias en la arquitectura original y evaluando las mejoras introducidas en términos de robustez y seguridad.

Para alcanzar este objetivo general se definieron los siguientes objetivos específicos:

4.2.1 ESTUDIO DE LA ARQUITECTURA CHERI

Analizar los principios de funcionamiento de la arquitectura CHERI, prestando especial atención al concepto de capability, a los mecanismos de protección de memoria que introduce y a las nuevas instrucciones incorporadas respecto a una arquitectura RISC-V convencional. Este estudio constituye la base teórica necesaria para comprender las modificaciones que posteriormente deben realizarse sobre el procesador.

4.2.2 IMPLEMENTACIÓN DE LA EXTENSIÓN CHERI SOBRE RISC-V

Diseñar e integrar en lenguaje VHDL los elementos necesarios para dotar a la arquitectura RISC-V de funcionalidades inspiradas en CHERI. Para ello será necesario modificar diferentes bloques del procesador original, incorporando nuevas estructuras de datos, registros especializados, mecanismos de validación y soporte para las nuevas instrucciones asociadas a las capabilities.

4.2.3 IMPLEMENTACIÓN SOBRE FPGA

Implementar la arquitectura desarrollada sobre una plataforma FPGA con el objetivo de verificar la viabilidad práctica del diseño hardware y comprobar su correcto funcionamiento en un entorno físico. Este objetivo busca trasladar la implementación más allá del entorno de simulación y evaluar el comportamiento real del sistema.

Debe señalarse que, aunque la propuesta inicial del proyecto contemplaba la implementación física de la arquitectura sobre una plataforma FPGA, el alcance final del trabajo se centró en el diseño, integración y validación funcional mediante simulación en Vivado. Esta decisión permitió dedicar un mayor esfuerzo al desarrollo y verificación de los mecanismos de protección inspirados en CHERI implementados en la arquitectura RV32Y.

Por este motivo, el objetivo relativo a la implementación física sobre FPGA no se considera alcanzado dentro del presente trabajo y se plantea como una posible línea de desarrollo futuro.

4.2.4 DESARROLLO DE UN ENTORNO DE PRUEBA Y VALIDACIÓN

Diseñar un conjunto de programas y escenarios de prueba que permitan verificar el funcionamiento de las nuevas funcionalidades incorporadas. Estos casos de prueba deberán evaluar tanto el comportamiento normal del sistema como situaciones en las que se produzcan accesos fuera de límites, capacidades inválidas o intentos de ejecución sin los permisos adecuados.

4.2.5 EVALUACIÓN DEL IMPACTO DE LA EXTENSIÓN CHERI

Analizar las mejoras introducidas por la incorporación de capabilities en términos de protección de memoria y robustez del sistema. Asimismo, se pretende estudiar el impacto que estas modificaciones tienen sobre la complejidad de la arquitectura y sobre el funcionamiento general del procesador respecto a una implementación RISC-V convencional.

4.3 METODOLOGÍA

El desarrollo de este Trabajo Fin de Grado se ha llevado a cabo siguiendo una metodología basada en el análisis, diseño, implementación y validación progresiva de cada uno de los componentes del sistema. Este enfoque permitió verificar el correcto funcionamiento de cada modificación antes de integrarla con el resto de la arquitectura, facilitando la detección y corrección de errores durante el desarrollo.

4.3.1 FASE DE ESTUDIO E INVESTIGACIÓN

La primera etapa del proyecto estuvo centrada en el estudio de la arquitectura RISC-V y de la extensión CHERI. Para ello se analizó la documentación oficial de ambas arquitecturas, prestando especial atención al funcionamiento de las capabilities, los mecanismos de protección de memoria y las instrucciones asociadas a su gestión.

A la vez, se realizó un estudio detallado del procesador RISC-V proporcionado como punto de partida, identificando los bloques hardware existentes y los elementos que debían modificarse para soportar la nueva funcionalidad.

4.3.2 FASE DE DISEÑO

Una vez comprendidos los fundamentos de ambas arquitecturas, se procedió al diseño de la solución propuesta. En esta fase se definió la estructura de las capabilities y se determinaron los nuevos bloques hardware necesarios para su gestión, incluyendo el banco de registros de capabilities, la memoria específica para almacenamiento de capabilities y la unidad de comprobación encargada de validar permisos, límites y validez.

Por otro lado, se analizaron las modificaciones necesarias sobre el datapath y la unidad de control para permitir la incorporación de nuevas instrucciones inspiradas en CHERI.

4.3.3 FASE DE IMPLEMENTACIÓN

Tras finalizar el diseño, se procedió a la implementación de los distintos bloques utilizando el lenguaje VHDL. El desarrollo se realizó de forma progresiva, comenzando por las estructuras de datos asociadas a las capabilities y continuando posteriormente con los nuevos módulos hardware y las modificaciones realizadas sobre la arquitectura original.

De forma paralela, se implementaron las nuevas instrucciones CHERI seleccionadas para el proyecto, adaptando la máquina de estados de control y el camino de datos para soportar su ejecución.

4.3.4 FASE DE VALIDACIÓN

La validación del sistema se llevó a cabo mediante simulaciones funcionales desarrolladas en el entorno Vivado. Para ello se diseñaron distintos bancos de pruebas destinados a verificar tanto el funcionamiento individual de cada bloque como la correcta integración de todos los componentes dentro de la arquitectura completa.

En primer lugar, se verificó el funcionamiento del procesador RV32I original. Posteriormente se validaron de forma individual los nuevos módulos incorporados, incluyendo el BancoCap, la memoria de capabilities y la unidad CHERI Check Unit. Finalmente, se desarrollaron programas de prueba específicos para comprobar el comportamiento de las nuevas instrucciones CHERI y la capacidad del sistema para detectar accesos no autorizados a memoria.

Este proceso permitió validar progresivamente la arquitectura propuesta y garantizar el correcto funcionamiento de los mecanismos de protección implementados.

4.4 PLANIFICACIÓN Y ESTIMACIÓN ECONÓMICA

Actividad	***Planificación inicial***	***Resultado final***
Estudio de CHERI y análisis bibliográfico	Octubre – Diciembre 2025	Completado
Diseño de la arquitectura RV32Y	Diciembre 2025 – Enero 2026	Completado
Implementación de los bloques hardware	Enero – Abril 2026	Completado
Integración de instrucciones CHERI	Marzo – Mayo 2026	Completado
Desarrollo de testbenches	Abril – Junio 2026	Completado
Validación funcional RV32I	Junio 2026	Completado
Validación funcional RV32Y	Junio 2026	Completado

Implementación física en FPGA	Prevista inicialmente	No realizada
Evaluación mediante programas de prueba	Junio 2026	Completado mediante simulación

Tabla 4.1: Planificación temporal del trabajo desarrollado

La Tabla 4.1 muestra la planificación inicial del proyecto y el cumplimiento alcanzado al finalizar el desarrollo. En términos generales, los objetivos se han completado satisfactoriamente.

Sin embargo, la implementación física sobre FPGA, contemplada inicialmente en la propuesta del proyecto, no se ha llegado a realizar dentro del tiempo disponible. En su lugar, el esfuerzo se centró en la integración de la arquitectura RV32Y y en la validación exhaustiva de sus funcionalidades mediante bancos de pruebas desarrollados en Vivado. Esta decisión permitió verificar de forma detallada el comportamiento del sistema y comprobar el correcto funcionamiento de los mecanismos de protección implementados.

Aunque el objetivo principal de este proyecto ha sido la investigación y el desarrollo de una arquitectura segura basada en CHERI mediante simulación funcional, resulta posible realizar una estimación aproximada de los costes asociados a su desarrollo e implementación.

Desde el punto de vista hardware, la plataforma inicialmente prevista para la implementación física del procesador era una placa FPGA Basys 3. Este tipo de placas tiene un coste aproximado comprendido entre 150 € y 200 €, dependiendo del distribuidor y de la versión adquirida.

Por otra parte, el desarrollo se ha realizado utilizando Vivado Design Suite, herramienta proporcionada por AMD-Xilinx. La versión utilizada dispone de licencia académica gratuita, por lo que no supone un coste adicional para el proyecto.

Sin embargo, el principal recurso consumido durante el desarrollo ha sido el tiempo de ingeniería dedicado al diseño, implementación y validación de la arquitectura. Considerando una dedicación aproximada de 150 horas de trabajo y tomando como referencia un coste de ingeniería de 25 €/hora, se obtiene un coste asociado de aproximadamente 3 750 €.

Concepto	***Coste estimado***
FPGA Basys 3	180 €
Licencia Vivado (académica)	0 €
Horas de ingeniería (150 h · 25 €/h)	3 750 €
Coste total estimado	**3 930 €**

Tabla 4.2: Estimación económica del trabajo

Capítulo 5. SISTEMA DESARROLLADO

5.1 ANÁLISIS DEL SISTEMA

5.1.1 ARQUITECTURA RISC-V ORIGINAL

La base de este trabajo es un procesador RISC-V RV32I multiciclo desarrollado en lenguaje VHDL. Esta arquitectura implementa el conjunto de instrucciones entero de 32 bits definido por la especificación RV32I y está organizada siguiendo una estructura clásica basada en un datapath compartido y una unidad de control secuencial [13].

El procesador está compuesto por los bloques funcionales fundamentales de una arquitectura RISC-V: contador de programa (PC), memoria de instrucciones, banco de registros, unidad aritmeticológica (ALU), memoria de datos y una unidad de control basada en una máquina de estados finitos (FSM). La ejecución de las instrucciones se realiza en varias etapas secuenciales, reutilizando los recursos hardware disponibles en cada ciclo de reloj para reducir la complejidad del diseño.

El contador de programa almacena la dirección de la siguiente instrucción a ejecutar y se actualiza en función del flujo normal del programa o de las instrucciones de salto y bifurcación (Jal…). La memoria de instrucciones contiene el programa ejecutado por el procesador y proporciona la instrucción correspondiente a la dirección indicada por el PC.

El banco de registros está formado por treinta y dos registros de propósito general de 32 bits, permitiendo la lectura simultánea de dos registros y la escritura de un resultado en cada ciclo de ejecución. Por su parte, la ALU realiza las operaciones aritméticas y lógicas necesarias para la ejecución de las instrucciones del procesador, incluyendo sumas, restas, desplazamientos, comparaciones y operaciones lógicas.

La memoria de datos proporciona soporte para las instrucciones de carga y almacenamiento, permitiendo el intercambio de información entre los registros internos y la memoria externa. Finalmente, la unidad de control coordina el funcionamiento de todos los bloques mediante una FSM que genera las señales necesarias para cada etapa de ejecución.

La Figura 5.1: Arquitectura ICAI-RISC-V, RV32I (Apuntes SSDDII Icai)Figura 5.1 muestra la arquitectura original del procesador RV32I utilizada como punto de partida para el desarrollo de este trabajo [13].

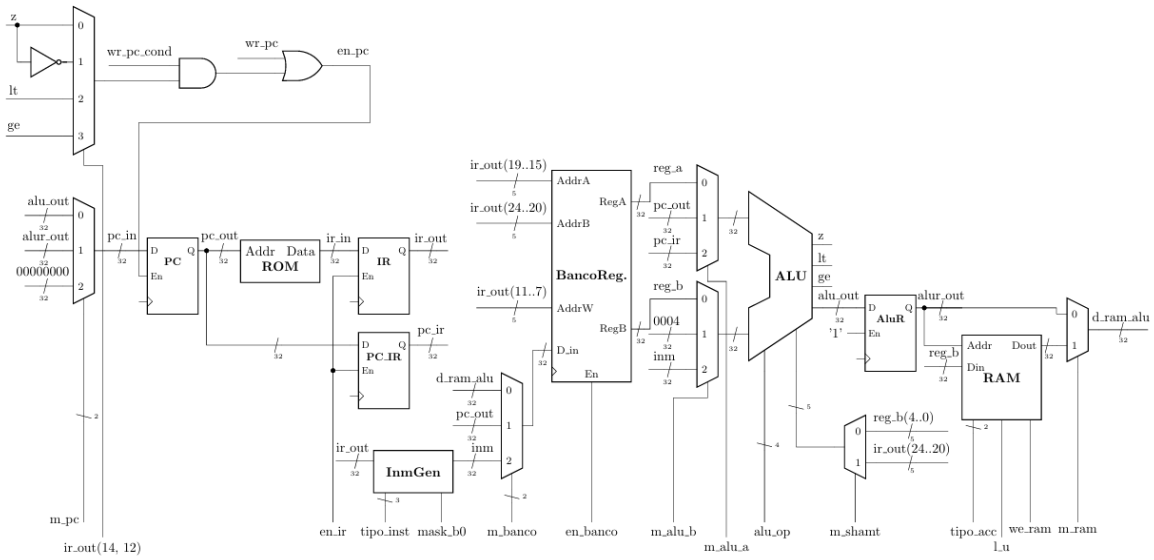


Figura 5.1: Arquitectura ICAI-RISC-V, RV32I (Apuntes SSDDII Icai)

5.1.2 LIMITACIONES RESPECTO A SEGURIDAD

Aunque la arquitectura RISC-V proporciona un diseño eficiente, modular y flexible, el conjunto base de instrucciones RV32I no incorpora mecanismos específicos de protección de memoria. Como consecuencia, la seguridad de los accesos a memoria depende

principalmente del software y del sistema operativo, quedando expuesta a diferentes tipos de vulnerabilidades conocidas [14].

Uno de los problemas más habituales son los accesos fuera de rango (*out-of-bounds accesses*), que ocurren cuando un programa accede a posiciones de memoria situadas fuera del espacio que le corresponde. Este tipo de errores puede provocar la lectura o modificación de datos pertenecientes a otras estructuras del programa, comprometiendo su correcto funcionamiento e incluso permitiendo la ejecución de código malicioso [14].

Otra vulnerabilidad frecuente es la corrupción de punteros (*pointer corruption*). En las arquitecturas convencionales, los punteros son simplemente direcciones almacenadas en memoria y pueden ser modificados accidentalmente por errores de programación o de forma intencionada mediante ataques. Una vez alterado el valor del puntero, el procesador no dispone de mecanismos hardware que permitan verificar si la nueva dirección sigue siendo válida antes de realizar un acceso [10].

Los desbordamientos de búfer (*buffer overflow*) constituyen otro de los problemas clásicos relacionados con la gestión de memoria. Estos errores aparecen cuando un programa escribe más información de la que una región de memoria puede almacenar, sobrescribiendo datos cercanos y pudiendo alterar el comportamiento normal del sistema [14].

La arquitectura RV32I tampoco incorpora mecanismos que permitan asociar permisos específicos a una dirección de memoria. Como consecuencia, una vez conocida una dirección válida, cualquier acceso permitido por el sistema puede realizarse sin comprobar si el programa dispone realmente de autorización para efectuar operaciones de lectura, escritura o ejecución sobre dicha región [10].

Estas limitaciones motivan la incorporación de mecanismos adicionales de protección capaces de trasladar parte de la seguridad al propio hardware. En este contexto surge CHERI, una extensión arquitectónica basada en el uso de capabilities que permite asociar límites, permisos y mecanismos de validación a cada referencia de memoria, reduciendo significativamente el impacto de este tipo de vulnerabilidades [10].

5.2 DISEÑO

5.2.1 ARQUITECTURA RV32Y PROPUESTA

Con el objetivo de incorporar mecanismos de protección de memoria inspirados en la arquitectura CHERI, se desarrolló una nueva versión del procesador denominada RV32Y. Esta arquitectura toma como punto de partida el procesador RV32I descrito anteriormente e incorpora los elementos hardware necesarios para soportar el uso de capabilities durante la ejecución de los programas.

La filosofía seguida durante el diseño consistió en mantener intacto el funcionamiento de las instrucciones originales de RV32I, añadiendo únicamente los bloques necesarios para implementar las nuevas funcionalidades de seguridad. De esta forma, el procesador conserva la compatibilidad con el conjunto base de instrucciones mientras incorpora mecanismos adicionales de validación y control de acceso a memoria. Para futuros programas con el RV32Y, las instrucciones antiguas no se deberían utilizar en un contexto vulnerable a fallos en memoria. Sin embargo, ante programas más simples, mantener estas instrucciones ayuda a conservar una forma más sencilla de ejecutar programas con un coste de complejidad menor.

La principal modificación introducida es la incorporación de un subsistema específico para la gestión de capabilities. Este subsistema está formado por un banco de registros de capacidades (BancoCap), una memoria de capacidades (CapRAM) y una unidad de comprobación (CHERI Check Unit). Estos bloques trabajan conjuntamente para almacenar capabilities, transferirlas entre memoria y registros y validar las operaciones realizadas sobre ellas.

Además de los nuevos bloques hardware, fue necesario modificar diferentes partes del datapath original. Se añadieron nuevas rutas de datos para transportar capabilities entre los distintos módulos del procesador, así como señales de control adicionales para coordinar la ejecución de las nuevas instrucciones CHERI. Asimismo, la unidad de control fue

ampliada mediante nuevos estados encargados de gestionar las operaciones de lectura, escritura, modificación e inspección de capabilities.

La Figura 5.2 muestra la arquitectura completa RV32Y desarrollada en este trabajo. En ella pueden observarse los nuevos bloques incorporados respecto a la arquitectura RV32I original y las conexiones establecidas entre estos elementos y el resto del procesador.

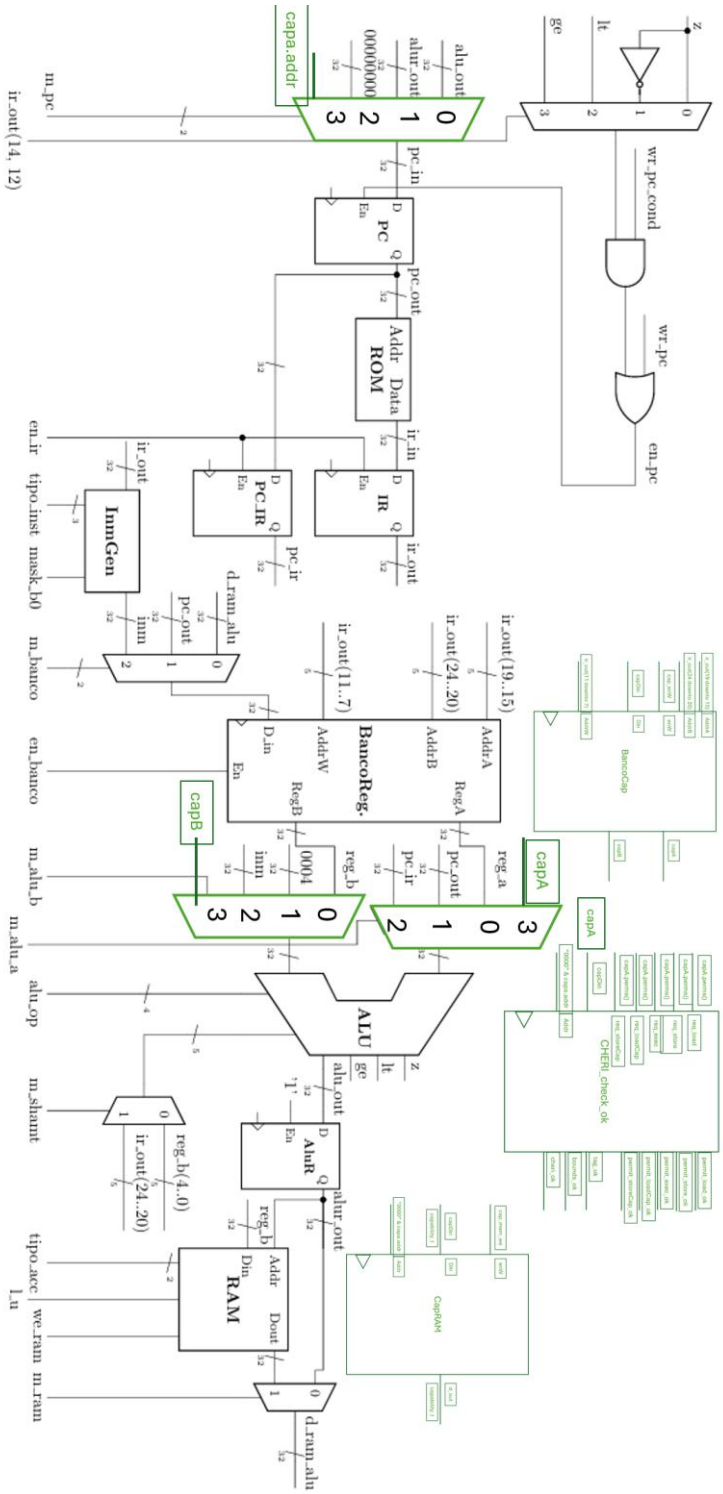


Figura 5.2: Arquitectura del RV32Y propuesta

5.2.2 BLOQUES HARDWARE INCORPORADOS

La integración de la extensión CHERI en la arquitectura RV32I requirió la incorporación de nuevos bloques hardware encargados de gestionar las capabilities y verificar las restricciones de acceso asociadas a ellas. Estos elementos complementan la arquitectura original del procesador y permiten trasladar parte de los mecanismos de protección de memoria al propio hardware.

Los bloques desarrollados pueden agruparse en tres elementos principales: el banco de registros de capabilities (BancoCap), encargado de almacenar las capabilities activas durante la ejecución; la memoria de capabilities (CapRAM), utilizada para guardar y recuperar capabilities desde memoria; y la unidad de comprobación CHERI (CHERI Check Unit), responsable de validar permisos, límites y estado de validez antes de autorizar cualquier acceso protegido.

En los siguientes apartados se describe el funcionamiento y la integración de cada uno de estos módulos dentro de la arquitectura RV32Y.

5.2.2.1 Banco de registros de capabilites (BancoCap)

El banco de registros de capabilities, denominado *BancoCap*, constituye el principal elemento de almacenamiento de capabilities dentro de la arquitectura RV32Y. Su función es equivalente a la realizada por el banco de registros convencional de RISC-V, aunque en este caso cada posición almacena una estructura completa de tipo capability en lugar de una palabra de datos de 32 bits [10].

El bloque implementado dispone de treinta y dos registros direccionables mediante índices de cinco bits, manteniendo la misma organización utilizada por el banco de registros general del procesador. Cada registro contiene todos los campos asociados a una capability: dirección base, longitud del rango accesible, dirección actual, permisos, bits reservados y bit de validez.

Para facilitar la ejecución de las instrucciones CHERI, el BancoCap incorpora dos puertos de lectura y un puerto de escritura. Esta configuración permite leer simultáneamente dos capabilities fuente y almacenar el resultado de una operación sobre una capability destino. Los registros seleccionados se determinan utilizando los mismos campos *rs1*, *rs2* y *rd* presentes en el formato de instrucción empleado por el procesador.

La incorporación de este bloque permite que las instrucciones CHERI operen sobre capabilities de forma independiente al banco de registros convencional, manteniendo separada la información de seguridad del resto de datos procesados por la arquitectura [10]. Además, esta organización simplifica la integración de las nuevas instrucciones y reduce las modificaciones necesarias sobre el diseño original del procesador RV32I.

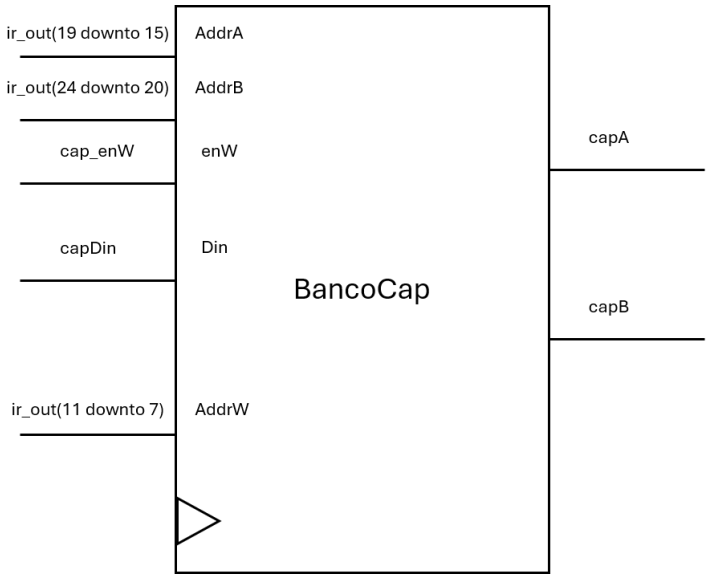


Figura 5.3: Bloque BancoCap

5.2.2.2 Memoria de capabilites (CapRAM)

Además del almacenamiento temporal proporcionado por el BancoCap, la arquitectura RV32Y requiere un mecanismo que permita guardar capabilities de forma permanente en memoria. Para ello se desarrolló una memoria específica denominada CapRAM, encargada de almacenar estructuras completas de tipo capability.

La CapRAM presenta una organización similar a la memoria de datos convencional utilizada por el procesador, aunque cada posición de memoria almacena una capability completa en lugar de una palabra de 32 bits [10]. De esta forma, es posible transferir capabilities entre memoria y registros mediante las instrucciones *CLoadCap* y *CStoreCap*, permitiendo su reutilización durante la ejecución de los programas. Estas instrucciones se explicarán más adelante.

La memoria implementada dispone de 1024 posiciones direccionables. El acceso se realiza utilizando la dirección proporcionada por el procesador, empleando los bits comprendidos entre las posiciones 11 y 2 para seleccionar la entrada correspondiente. Esta organización es equivalente a la utilizada en la memoria de datos y simplifica la integración de ambos sistemas dentro de la ruta de datos.

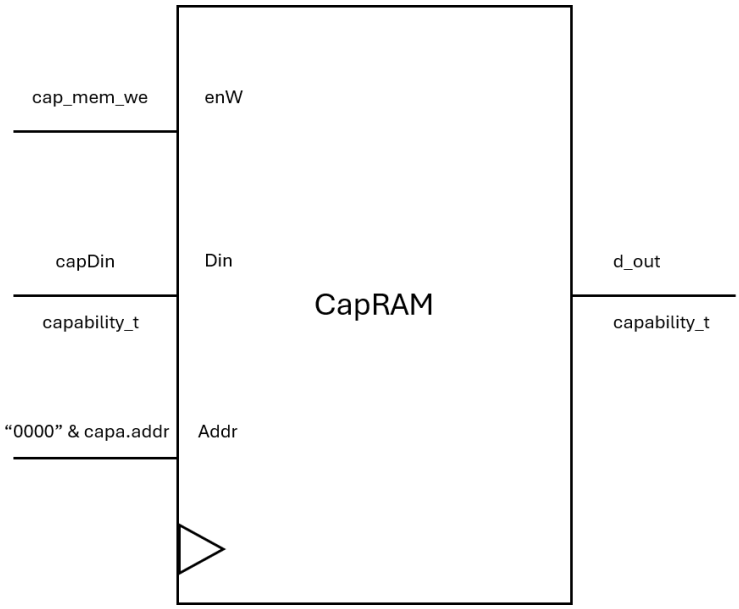


Figura 5.4: Bloque CapRAM

Durante una operación de escritura, la capability presente en la entrada de datos se almacena en la posición seleccionada cuando la señal de *enable* correspondiente se encuentra activa. En las operaciones de lectura, la capability almacenada en la dirección indicada se presenta directamente en la salida del bloque para su posterior utilización por el RV32Y [10].

5.2.2.3 Unidad de Comprobación CHERI (CHERI_check_unit)

La unidad de comprobación CHERI, denominada CHERI Check Unit y visible en la Figura 5.5, es el bloque encargado de verificar que una operación protegida cumple todas las restricciones definidas por la capability utilizada. Este módulo constituye el principal mecanismo de protección de memoria de la arquitectura RV32Y, ya que determina si una operación de acceso puede ejecutarse o debe ser bloqueada.

La unidad recibe como entrada una *capability* completa junto con la dirección de memoria que se desea utilizar durante la operación. Además, dispone de un conjunto de señales de entrada que indican el tipo de operación solicitada: lectura de datos, escritura de datos, ejecución de código, carga de capabilities o almacenamiento de capabilities.

La primera comprobación realizada corresponde al campo *tag* de la *capability*. Este bit indica si la *capability* es válida y puede utilizarse durante la ejecución del programa. Cuando el valor del *tag* es cero, la *capability* se considera inválida y cualquier operación asociada a ella es rechazada automáticamente.

La segunda comprobación evalúa los límites de memoria definidos por la *capability*. Para ello, la unidad calcula la dirección final autorizada mediante la suma de los campos base y length. Posteriormente, la dirección solicitada es comparada con dicho rango para verificar que se encuentra dentro de los límites permitidos. Únicamente se aceptan accesos cuya dirección pertenezca al intervalo definido por la *capability*.

La tercera comprobación analiza los permisos almacenados en el campo *perms*. Cada bit de este campo representa una operación concreta autorizada sobre la capability. En la implementación desarrollada se definieron permisos independientes para lectura de datos, escritura de datos, ejecución de instrucciones, carga de capabilities y almacenamiento de capabilities. Dependiendo del tipo de operación solicitada, la unidad verifica que el permiso correspondiente se encuentre habilitado antes de autorizar el acceso.

Finalmente, los resultados de todas las comprobaciones son combinados para generar la señal *check_ok*. Esta señal únicamente toma el valor lógico uno cuando la capability es válida, la dirección pertenece al rango autorizado y los permisos necesarios para la operación están presentes. En cualquier otro caso, la operación es rechazada por el hardware y el acceso protegido no llega a ejecutarse. En el siguiente código se muestra la comprobación de los tres aspectos por parte de la unidad.

```
   s_tag_ok          <= cap_in.tag;
   s_permit_load_ok  <= cap_in.perms(0);
   s_permit_store_ok <= cap_in.perms(1);
   s_permit_exec_ok  <= cap_in.perms(2);
   s_permit_lcap_ok  <= cap_in.perms(3);
   s_permit_scap_ok  <= cap_in.perms(4);

-- Comprobación de limites
   cap_end_u <= ('0' & cap_base_u) + ('0' & cap_length_u);

   s_bounds_ok <= '1' when
      ('0' & access_u) >= ('0' & cap_base_u) and
      ('0' & access_u) <  cap_end_u
   else '0';

   -- Comprobación de permisos
   op_ok <= ((not req_load)      or s_permit_load_ok)  and
            ((not req_store)     or s_permit_store_ok) and
            ((not req_exec)      or s_permit_exec_ok)  and
            ((not req_load_cap)  or s_permit_lcap_ok)  and
            ((not req_store_cap) or s_permit_scap_ok);

   -- Resultado final
   check_ok <= s_tag_ok and s_bounds_ok and op_ok;
```

La utilización de esta unidad permite trasladar las comprobaciones de seguridad desde el software hacia el hardware, garantizando que todas las instrucciones protegidas por capabilities sean verificadas de forma automática antes de acceder a memoria o modificar el flujo de ejecución del programa.

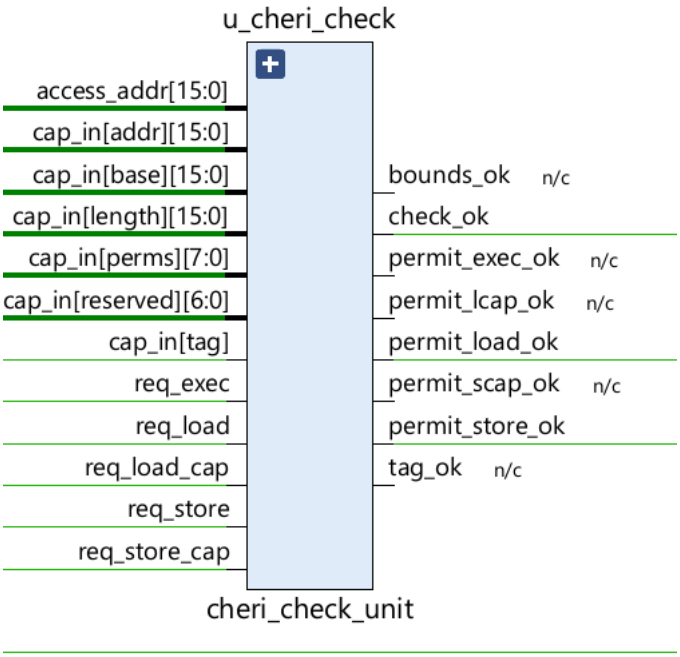


Figura 5.5: Bloque CHERI_check_unit

Debe señalarse que, en la implementación desarrollada, una violación de las restricciones CHERI no genera una excepción ni transfiere el control a un manejador de interrupciones. El mecanismo implementado consiste en bloquear la operación solicitada cuando la señal cheri_check_ok toma el valor lógico cero. De esta forma, las operaciones de escritura en memoria, escritura en registros o actualización del contador de programa asociadas a instrucciones CHERI únicamente se habilitan cuando la comprobación de seguridad resulta satisfactoria.

La incorporación de un mecanismo completo de excepciones o traps CHERI queda fuera del alcance de este trabajo y se plantea como una posible línea de desarrollo futuro en el apartado 7.3.

5.2.3 DISEÑO DE LAS CAPABILITIES

La implementación de la extensión CHERI requiere sustituir los punteros tradicionales de una arquitectura RISC-V por una estructura denominada *capability*. Mientras que un puntero convencional almacena únicamente una dirección de memoria, una capability incorpora información adicional que permite validar en hardware los accesos realizados durante la ejecución del programa. En el sistema desarrollado cada capability está formada por varios campos que definen tanto el rango de memoria accesible como los permisos asociados a dicho acceso. Esta información se almacena en registros especiales creados y es utilizada por las instrucciones CHERI implementadas antes de realizar operaciones de carga y almacenamiento.

En la Figura 5.6 se puede observar los campos que componen cada capability. El campo base es la dirección inicial válida de la región de memoria, el campo length es el tamaño total del rango de memoria accesible, address indica la dirección actual utilizada para el acceso, perms son los permisos asociados a la capability correspondiente, reserved son bits reservados para posibles ampliaciones futuras y el bit de tag es el bit más importante de la capability ya que indica la validez de esta.

A partir de los valores de los campos length y base se calcula el rango válido de acceso para las operaciones de memoria. Antes de ejecutar cualquier acceso a memoria, el sistema verifica que dicha dirección, la indicada en el campo address, se encuentra dentro de los límites calculados. Sí se encuentra dentro de los límites, se evalúan los permisos requeridos para la acción buscada en ese momento y se compararán con los permisos de la capability encontrados en el campo perms. Finalmente, el campo tag actúa con un mecanismo de validación de la capability. Cuando el valor del tag es válido, la capability puede utilizarse en operaciones de memoria. En caso contrario coma el acceso es rechazado por el hardware para evitar el uso de referencias inválidas o corruptas. La utilización de esta estructura permite trasladar parte de la protección de memoria desde el software hacia el

hardware reduciendo la posibilidad de accesos fuera de rango y mejorando la robustez general de sistema frente a vulnerabilidades de memoria.

```
constant CAP_BASE_BITS   : integer := 16;
constant CAP_LENGTH_BITS : integer := 16;
constant CAP_ADDR_BITS   : integer := 16;
constant CAP_PERM_BITS   : integer := 8;
constant CAP_RSVD_BITS   : integer := 7;

type capability_t is record
    base     : std_logic_vector(CAP_BASE_BITS-1 downto 0);
    length   : std_logic_vector(CAP_LENGTH_BITS-1 downto 0);
    addr     : std_logic_vector(CAP_ADDR_BITS-1 downto 0);
    perms    : std_logic_vector(CAP_PERM_BITS-1 downto 0);
    reserved : std_logic_vector(CAP_RSVD_BITS-1 downto 0);
    tag      : std_logic;
```

Figura 5.6: Definición de una capability en VHDL

5.2.4 CAPABILITY RAÍZ

En una arquitectura basada en capabilities surge un problema inicial relacionado con la creación de las primeras capabilities válidas del sistema. Dado que todas las operaciones protegidas requieren una capability previamente existente, es necesario disponer de al menos una capability inicial que permita comenzar la ejecución del programa.

Las implementaciones completas de CHERI suelen resolver este problema mediante mecanismos de inicialización gestionados por firmware o por el sistema operativo, que generan las capacidades raíz del sistema durante el arranque [9].

Sin embargo, debido al carácter simplificado de la arquitectura desarrollada, se optó por una solución más sencilla consistente en precargar una capability válida durante el proceso de reset del procesador. Esta capability inicial se almacena en el registro c1 del BancoCap y dispone de permisos completos sobre una región de memoria predefinida.

La capability raíz implementada posee una dirección base 0x0100, una longitud de 0x0020 bytes, permisos completos y un bit de validez activo. A partir de esta capability

inicial es posible generar nuevas capabilities más restrictivas mediante instrucciones como CSetBounds, CSetPerm o CIncOffset.

Esta decisión de diseño simplifica el proceso de inicialización y permite centrar el estudio en los mecanismos de protección proporcionados por CHERI sin necesidad de desarrollar una infraestructura completa de arranque del sistema.

5.3 IMPLEMENTACIÓN

5.3.1 MODIFICACIONES DEL DATAPATH

La incorporación de la extensión CHERI requirió modificar diferentes elementos del datapath original del procesador RV32I con el objetivo de integrar las nuevas estructuras de datos y soportar la ejecución de instrucciones protegidas. Estas modificaciones afectan principalmente a los caminos de acceso a memoria, a la actualización del contador de programa y a la incorporación de los nuevos bloques hardware descritos anteriormente.

La primera modificación significativa consistió en la integración del banco de registros de capabilities (BancoCap) dentro del datapath. Este bloque se conecta a la unidad de control y permite leer y escribir capabilities de forma similar a como el banco de registros convencional gestiona los registros generales. Durante la ejecución de las instrucciones CHERI, las capabilities almacenadas en BancoCap son utilizadas como operandos de entrada para las distintas operaciones implementadas.

```
capDin <= cheri_inc_cap    when ir_out(6 downto 2) = "10001" and ir_out(14 downto
12) = "001" else
          cheri_perm_cap   when ir_out(6 downto 2) = "10001" and ir_out(14 downto
12) = "010" else
          cheri_bounds_cap when ir_out(6 downto 2) = "10001" and ir_out(14 downto
12) = "011" else
          cap_mem_dout     when ir_out(6 downto 2) = "11111" and ir_out(14 downto
12) = "010" else
          NULL_CAPABILITY;
```

Por otro lado, se incorporó la memoria de capabilities (CapRAM), conectada al sistema de acceso a memoria del procesador. Este bloque permite almacenar y recuperar capabilities completas mediante las instrucciones CLoadCap y CStoreCap, ampliando las posibilidades de gestión de capacidades más allá de los registros internos del procesador.

Otra modificación importante se realizó sobre la lógica de direccionamiento de la memoria. En la arquitectura RV32I original, la dirección de acceso a memoria procedía siempre de la salida de la ALU. Sin embargo, para soportar las instrucciones protegidas de CHERI fue necesario añadir un mecanismo de selección que permite utilizar alternativamente la dirección almacenada en el campo address de una capability, como se puede ver en el código. De esta forma, las instrucciones CLoad, CStore, CLoadCap y CStoreCap pueden acceder a memoria utilizando directamente la información contenida en la capability correspondiente.

```
cheri_mem_addr <= x"0000" & capA.addr;

ram_addr_in <= cheri_mem_addr when ir_out(6 downto 2) = "11111" else
               alur_out;
```

También se modificó la lógica de actualización del contador de programa. En el diseño original, el nuevo valor del PC se obtenía a partir de las instrucciones de salto convencionales. Con la incorporación de CJump y CJumpLink se añadió una nueva ruta de datos que permite cargar en el contador de programa la dirección almacenada en el campo address de una capability, previamente validada por la unidad CHERI Check Unit.

```
cheri_jump_addr <= x"0000" & capA.addr;

pc_in <= alu_out         when m_pc = "00" else
         alur_out        when m_pc = "01" else
         cheri_jump_addr when m_pc = "10" else
         (others => '0');
```

Finalmente, se integró la unidad de comprobación CHERI dentro del camino de ejecución de las instrucciones protegidas. Este bloque recibe la capability utilizada en cada

operación y verifica automáticamente su validez, los límites de memoria autorizados y los permisos asociados antes de permitir el acceso correspondiente. Gracias a esta integración, todas las operaciones protegidas son verificadas directamente por hardware antes de ejecutarse.

5.3.2 MODIFICACIONES DE LA FSM

La unidad de control del procesador RV32Y se implementa mediante una máquina de estados finitos encargada de generar las señales de control necesarias para cada etapa de ejecución. La arquitectura original RV32I ya utilizaba un modelo multiciclo, por lo que la integración de CHERI se realizó ampliando dicha FSM con nuevos estados específicos para las instrucciones de capabilities, como se puede observar en la Figura 5.7.

La primera modificación importante se encuentra en la etapa de decodificación. En este estado, la unidad de control analiza el campo `opcode`, definido como `ir_out(6 downto 2)`, junto con el campo `funct3`, correspondiente a `ir_out(14 downto 12)`. Esta combinación permite distinguir entre las instrucciones RV32I originales y las nuevas instrucciones CHERI añadidas al procesador RV32Y.

En la implementación realizada se utilizaron dos grupos principales de instrucciones CHERI. El primer grupo, identificado mediante el opcode `"10001"`, agrupa instrucciones de gestión, inspección y salto, como `CGetTag`, `CIncOffset`, `CSetPerm`, `CSetBounds`, `CJump` y `CJumpLink`. El segundo grupo, identificado mediante el opcode `"11111"`, incluye las instrucciones de acceso protegido a memoria, como `CLoad`, `CStore`, `CLoadCap` y `CStoreCap`.

El siguiente fragmento muestra la lógica de decodificación incorporada para reconocer las nuevas instrucciones CHERI:

```
elsif opcode = "10001" and ir_out(14 downto 12) = "000" then
   next_state <= s_CGetTag3;
elsif opcode = "10001" and ir_out(14 downto 12) = "001" then
   next_state <= s_CIncOffset3;
elsif opcode = "10001" and ir_out(14 downto 12) = "010" then
```

```
   next_state <= s_CSetPerm3;
elsif opcode = "10001" and ir_out(14 downto 12) = "011" then
   next_state <= s_CSetBounds3;
elsif opcode = "10001" and ir_out(14 downto 12) = "100" then
   next_state <= s_CJump3;
elsif opcode = "10001" and ir_out(14 downto 12) = "101" then
   next_state <= s_CJumpLink3;
elsif opcode = "11111" and ir_out(14 downto 12) = "000" then
   next_state <= s_CLoad3;
elsif opcode = "11111" and ir_out(14 downto 12) = "001" then
   next_state <= s_CStore3;
elsif opcode = "11111" and ir_out(14 downto 12) = "010" then
   next_state <= s_CLoadCap3;
elsif opcode = "11111" and ir_out(14 downto 12) = "011" then
   next_state <= s_CStoreCap3;
```

Una vez reconocida la instrucción, la FSM transita hacia el estado correspondiente. Las instrucciones de gestión de capabilities, como CIncOffset, CSetPerm y CSetBounds, se resuelven en un único estado de ejecución, activando la señal cap_enW para escribir la nueva capability calculada en el BancoCap. En cambio, las instrucciones de carga de datos o capabilities requieren dos estados, separando la preparación del acceso de la escritura del resultado.

La Tabla 5.1 resume los nuevos estados incorporados a la FSM.

Estado	***Instrucción***	***Función principal***
s_CGetTag3	CGetTag	Escribe el valor del tag en el banco de registros general.
s_CIncOffset3	CIncOffset	Actualiza el campo address de una capability.
s_CSetPerm3	CSetPerm	Genera una capability con permisos restringidos.

s_CSetBounds3	CSetBounds	Genera una capability con el campo length modificado.
s_CJump3	CJump	Actualiza el PC con la dirección de una capability si la comprobación es válida.
s_CJumpLink3	CJumpLink	Realiza un salto protegido y guarda la dirección de retorno.
s_CLoad3	CLoad	Prepara una lectura protegida desde memoria.
s_CLoad4	CLoad	Escribe el dato leído si cheri_check_ok = '1'.
s_CStore3	CStore	Escribe en memoria si cheri_check_ok = '1'.
s_CLoadCap3	CLoadCap	Prepara la lectura de una capability desde CapRAM.
s_CLoadCap4	CLoadCap	Escribe la capability leída en BancoCap si la comprobación es válida.
s_CStoreCap3	CStoreCap	Almacena una capability en CapRAM si la comprobación es válida.

Tabla 5.1: Descripción de los nuevos estados del RV32Y

La señal cheri_check_ok se utiliza como condición de seguridad antes de activar operaciones sensibles. En los estados de salto protegido, esta señal controla la actualización del contador de programa. En los estados de acceso a memoria, condiciona la activación de la escritura en memoria, la escritura en el banco de registros general o la escritura en el BancoCap.

El siguiente fragmento muestra cómo la FSM impide que una operación protegida se complete cuando la unidad de comprobación CHERI no autoriza el acceso:

```
when s_CJump3 =>
```

```
   if cheri_check_ok = '1' then
      wr_pc <= '1';
      m_pc  <= conv_std_logic_vector(2, m_pc'length);
   end if;

when s_CLoad4 =>
   if cheri_check_ok = '1' then
      m_banco  <= conv_std_logic_vector(0, m_banco'length);
      en_banco <= '1';
      m_ram    <= '1';
   end if;

when s_CStore3 =>
   if cheri_check_ok = '1' then
      tipo_acc <= ir_out(13 downto 12);
      we_ram   <= '1';
   end if;
```

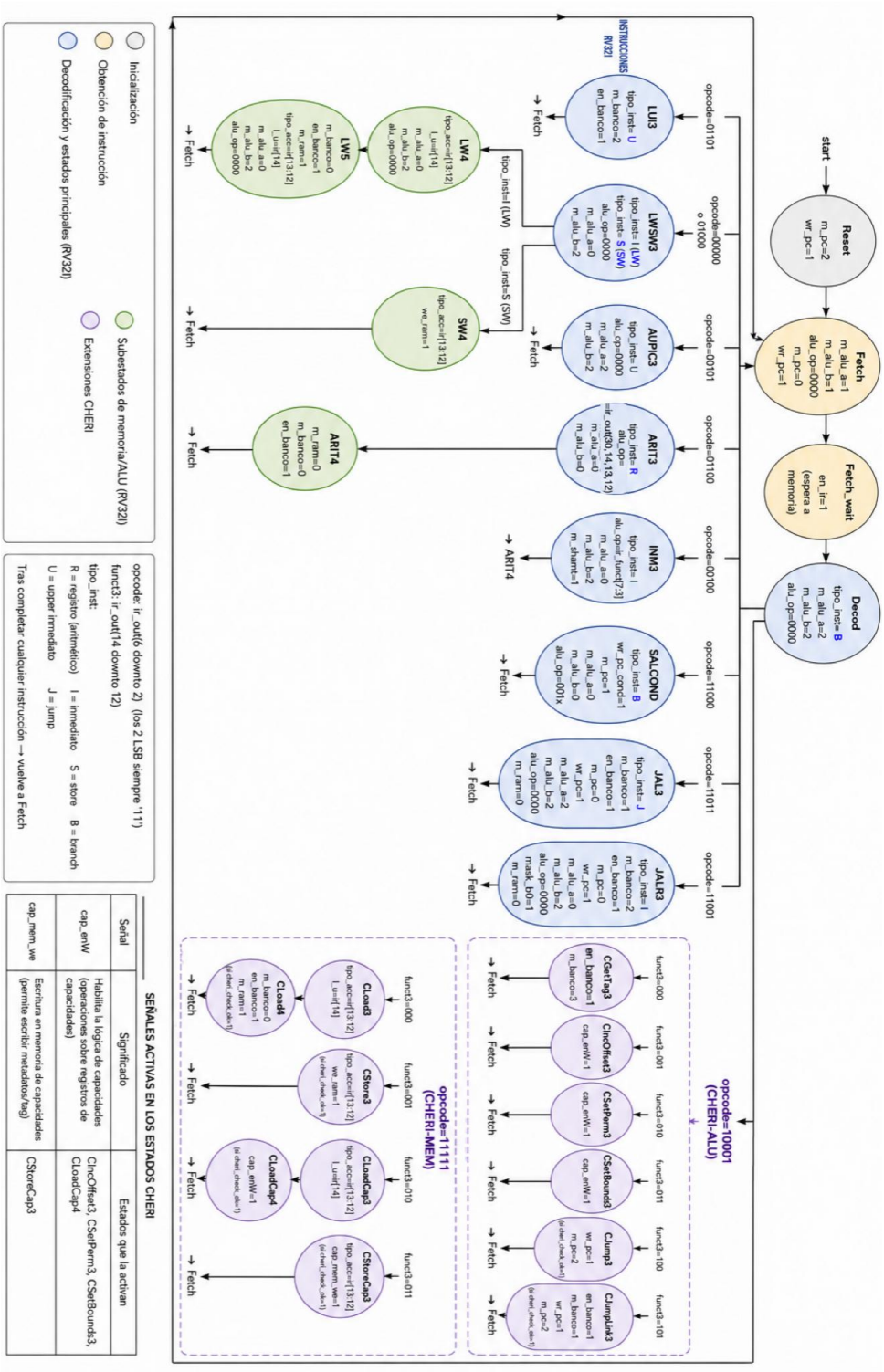


Figura 5.7: FSM modificada

5.3.3 IMPLEMENTACIÓN DE LAS INSTRUCCIONES CHERI

Una vez incorporados los nuevos bloques hardware y ampliada la unidad de control, se procedió a la implementación de las instrucciones CHERI sobre la arquitectura RV32Y. Estas instrucciones permiten crear, modificar, inspeccionar y utilizar capabilities durante la ejecución de los programas.

Las instrucciones desarrolladas pueden agruparse en tres categorías principales. La primera corresponde a las instrucciones de inspección, utilizadas para consultar información contenida en una capability sin modificar su contenido. La segunda engloba las instrucciones de gestión de capabilities, encargadas de generar nuevas capabilities a partir de otras ya existentes mediante la modificación de determinados campos. Finalmente, la tercera categoría incluye las instrucciones de acceso protegido a memoria y transferencia de control, que utilizan la información almacenada en las capabilities para validar operaciones de lectura, escritura y salto antes de ejecutarlas.

En los apartados siguientes se describe detalladamente cada una de las instrucciones implementadas, indicando su objetivo, funcionamiento, integración dentro de la arquitectura y los entornos de prueba utilizados para su validación.

Instrucción	***Categoría***	***Función***
CGetTag	Inspección	Obtiene el valor del bit tag
CGetPerms	Inspección	Obtiene el valor del campo perms
CGetAddr	Inspección	Obtiene el valor del campo addr
CGetBase	Inspección	Obtiene el valor del campo base
CGetlen	Inspección	Obtiene el valor del campo len

CIncOffset	Gestión	Modifica el campo address
CSetPerm	Gestión	Restringe permisos
CSetBounds	Gestión	Modifica los límites de acceso
CLoad	Memoria	Lectura protegida
CStore	Memoria	Escritura protegida
CLoadCap	Memoria	Carga una capability desde memoria
CStoreCap	Memoria	Almacena una capability en memoria
CJump	Control	Salto protegido
CJumpLink	Control	Salto protegido con enlace

Tabla 5.2:Descripción de las nuevas instrucciones CHERI

5.3.4 INSTRUCCIÓN CSETBOUNDS

5.3.4.1 Objetivo de la instrucción

La instrucción *CSetBounds* permite restringir el rango de memoria accesible asociado a una capability. Su función principal es establece unos límites concretos de acceso mediante la modificación del campo *length,* reduciendo así la parte de memoria utilizada para operaciones posteriores.

Esta instrucción representa uno de los mecanismos fundamentales de protección de memoria dentro de CHERI, ya que permite limitar el alcance de una capability para evitar accesos fuera del rango permitido.

5.3.4.2 Funcionamiento de la instrucción

La instrucción *CSeteBounds* recibe una capability de origen y genera una nueva capability cuyos límites de memoria quedarán restringidos al rango especificado. El funcionamiento general de la instrucción puede resumirse con el siguiente código:

```
cheri_bounds_cap.base     <= capA.base;
cheri_bounds_cap.addr     <= capA.addr;
```

```
    cheri_bounds_cap.perms    <= capA.perms;
    cheri_bounds_cap.reserved <= capA.reserved;
    cheri_bounds_cap.tag      <= capA.tag;
    cheri_bounds_cap.length   <= reg_b(15 downto 0);
```

Como se puede observar, todos los campos de la capability se mantienen igual excepto el campo *length.* Este campo se ve modificado a partir de la entrada del registro b.

5.3.4.3 Integración en la arquitectura

La incorporación de *CSetBounds* requirió añadir nuevas señales de control para seleccionar la escritura de campos específicos dentro de los registros de capability. Además, la unidad de control fue modificada para reconocer el nuevo opcode asociado a la instrucción y activar el datapath correspondiente durante la etapa de ejecución. Todas las operaciones de esta instrucción se realizan internamente sobre registros de capability, por ello no requiere acceso a memoria externa.

5.3.5 INSTRUCCIÓN CSETPERM

5.3.5.1 Objetivo de la instrucción

La instrucción *CSetPerm* permite modificar los permisos asociados a una capability, restringiendo las operaciones que pueden realizarse sobre una determinada parte de la memoria.

Su finalidad principal es limitar los privilegios de acceso durante la ejecución de un programa, permitiendo aplicar el principio de mínimo privilegio sobre las operaciones de memoria. Este principio dicta que a los procesos se les deben otorgar los permisos mínimos estrictamente necesarios para realizar sus funciones como limitando el acceso a otros recursos.

Mediante esta instrucción se pueden eliminar diferentes permisos; de lectura escritura o ejecución, por ejemplo; de una capability ya existente, reduciendo el posible impacto de un error de software o un acceso malintencionado.

5.3.5.2 Funcionamiento de la instrucción

Esta instrucción presenta un funcionamiento similar a la instrucción *CSetBounds*, Ya que recibe una capability de entrada y genera una nueva capability con una máscara de permisos modificada. La instrucción mantiene intactos los campos asociados a dirección y límites de memoria alterando únicamente el campo de permisos. El funcionamiento general de la instrucción puede representarse mediante el siguiente código:

```
cheri_perm_cap.base     <= capA.base;
cheri_perm_cap.length   <= capA.length;
cheri_perm_cap.addr     <= capA.addr;
cheri_perm_cap.reserved <= capA.reserved;
cheri_perm_cap.tag      <= capA.tag;
cheri_perm_cap.perms    <= capA.perms and reg_b(7 downto 0);
```

A simple vista se puede apreciar una diferencia respecto a la instrucción *CSetBounds,* ya que presenta la utilización de la operación lógica AND. El uso de esta operación lógica garantiza que únicamente pueden eliminarse permisos existentes, evitando que una capability obtenga privilegios superiores a los originalmente asignados.

5.3.5.3 Integración en la arquitectura

La incorporación de *CSetPerm* requirió añadir nuevas señales de control asociadas a la escritura parcial de los registros de capability. La unidad de control fue modificada para reconocer el opcode correspondiente y activar la lógica encargada de actualizar exclusivamente el campo de permisos. Al igual que *CSetBounds,* esta instrucción no requiere acceso directo a memoria externa, ya que todas las operaciones se realizan internamente sobre los registros de capabilities.

5.3.5.4 Entorno de prueba

Esta instrucción es útil en un entorno de prueba en el que se requiere primero de hacer una determinada instrucción; una carga de datos en memoria, por ejemplo; y luego la capability solo se usa para lectura. De esta manera, una vez finalizada la fase de carga,

CSetPerm restringe los permisos de la capability, manteniendo únicamente la lectura. Así, cualquier intento de escritura quedará bloqueado.

5.3.6 INSTRUCCIÓN CINCOFFSET

5.3.6.1 Objetivo de la instrucción

La instrucción *CIncOffset* permite modificar la dirección actual almacenada en una capability mediante la adición de un desplazamiento. Su principal objetivo es facilitar el recorrido de estructuras de datos o regiones de memoria sin necesidad de crear nuevas capabilities para cada posición de acceso.

Esta instrucción resulta especialmente útil cuando una misma capability debe utilizarse para acceder a diferentes posiciones dentro de una región de memoria previamente autorizada. En lugar de modificar directamente los límites o permisos de la capability, *CIncOffset* únicamente actualiza la dirección utilizada para los accesos posteriores.

5.3.6.2 Funcionamiento de la instrucción

La instrucción recibe una capability de entrada y genera una nueva capability cuya dirección actual se obtiene sumando un desplazamiento procedente de un registro del banco de registros convencional.

Su funcionamiento puede representarse mediante el siguiente fragmento de código:

```
cheri_offset_cap.base     <= capA.base;
cheri_offset_cap.length   <= capA.length;
cheri_offset_cap.addr     <= std_logic_vector(
                               unsigned(capA.addr) +
                               unsigned(reg_b(15 downto 0))
                             );
cheri_offset_cap.perms    <= capA.perms;
cheri_offset_cap.reserved <= capA.reserved;
cheri_offset_cap.tag      <= capA.tag;
```

Como puede observarse, todos los campos de la capability permanecen igual excepto el campo *addr*, que se actualiza mediante la suma del desplazamiento indicado por el registro fuente. De esta forma, la nueva capability conserva los mismos límites, permisos y estado de validez que la capability original.

5.3.6.3 Integración en la arquitectura

La implementación de *CIncOffset* requirió incorporar una nueva ruta de datos capaz de generar una capability modificada a partir de una capability existente y del valor almacenado en un registro del banco de registros convencional.

Además, fue necesario ampliar la unidad de control para reconocer el opcode correspondiente a esta instrucción y habilitar la escritura del resultado en el BancoCap. Dado que la operación se realiza exclusivamente sobre registros internos, no requiere accesos a memoria durante su ejecución.

Esta instrucción desempeña un papel importante dentro de la arquitectura desarrollada, ya que constituye el mecanismo utilizado para desplazar una capability dentro de una región de memoria autorizada sin alterar sus límites ni sus permisos.

5.3.7 INSTRUCCIONES DE INSPECCIÓN DE CAPABILITIES: CGETTAG, CGETPERMS, CGETBASE, CGETLEN Y CGETADDR

5.3.7.1 Objetivo de la instrucción

Todas las instrucciones de inspección comparten el mismo opcode y el mismo funct3, y se diferencian mediante el campo funct7 de la instrucción. Esta decisión permite ampliar el conjunto de instrucciones disponibles sin añadir nuevos estados a la FSM.

La codificación utilizada es la siguiente:

funct7	*Instrucción*	*Campo consultado*	*Resultado*
0000000	CGetTag	tag	rd = tag
0000001	CGetBase	base	rd = base
0000010	CGetLen	length	rd = length
0000011	CGetPerm	perms	rd = perms
0000100	CGetAddr	addr	rd = addr

Tabla 5.3: Codificación de las instrucciones de inspección

Durante la ejecución, la unidad de control transita al estado s_CGetTag3 para todas estas variantes. En dicho estado se habilita la escritura en el banco de registros general y se selecciona como dato de entrada el campo correspondiente de la capability leída desde el BancoCap. La selección concreta del campo se realiza en el datapath a partir del valor de funct7.

De esta forma, se mantiene sin cambios la estructura general de la FSM y se evita añadir un estado independiente para cada instrucción.

5.3.7.2 Funcionamiento de las instrucciones

CGetTag Recibe una capability como entrada y devuelve el valor de su bit de validez. Si el valor obtenido es válido, la capability puede utilizarse en operaciones posteriores. En caso contrario, cualquier acceso asociado a dicha capability deberá ser rechazado.

5.3.7.3 Integración en la arquitectura

La integración de estas instrucciones no requirió modificar el diagrama general de la FSM, ya que todas las operaciones de inspección se resuelven en el mismo estado de control. La ampliación se realizó en la lógica de selección del dato escrito en el banco de registros

general, incorporando un multiplexor que permite elegir entre los distintos campos de la capability.

Esta solución mantiene la complejidad de control prácticamente inalterada y concentra la ampliación en el datapath, donde se selecciona el campo de salida correspondiente.

5.3.8 INSTRUCCIÓN CLOAD

5.3.8.1 Objetivo de la instrucción

La instrucción *CLoad* permite realizar una lectura de memoria utilizando una capability como referencia. A diferencia de una carga convencional de RISC-V, el acceso no depende únicamente de una dirección, sino que antes de leer se comprueban los campos de seguridad de la capability. De esta forma, la instrucción evita excesos fuera de rango y limita el uso indebido de partes protegidas de la memoria.

5.3.8.2 Funcionamiento de la instrucción

La instrucción utiliza una capability almacenada en el BancoCap para determinar la dirección de acceso a memoria. En la implementación desarrollada, la dirección utilizada por CLoad no se obtiene sumando un desplazamiento inmediato de la instrucción, sino que procede directamente del campo addr de la capability seleccionada.

Por tanto, el desplazamiento dentro de una región de memoria autorizada se realiza previamente mediante la instrucción CIncOffset. Una vez actualizada la dirección de la capability, CLoad utiliza dicho campo para acceder a memoria.

Antes de realizar la lectura, el hardware comprueba que la capability sea válida, que el campo addr se encuentre dentro del rango definido por base y length, y que exista permiso de lectura. La operación únicamente se completa cuando todas estas comprobaciones son correctas.

5.3.8.3 Integración en la arquitectura

La implementación de *CLoad* requirió modificar la lógica de acceso a memoria del procesador, incorporando comprobaciones adicionales antes de activar la señal de lectura. La instrucción verifica que dicha dirección pertenece al rango autorizado. En caso de fallo en alguna de las comprobaciones, la operación de lectura es bloqueada y el procesador evita de acceso a memoria.

Esta instrucción requirió modificar tanto la unidad de control como el camino de datos asociados a operaciones de memoria. La unidad de control fue adaptada para reconocer el nuevo opcode correspondiente a la instrucción y habilitar las señales necesarias para leer la capability, calcular la dirección efectiva, ejecutar las comprobaciones de rango y permisos y autorizar o bloquear la lectura de memoria. De esta manera implementamos una protección adicional hardware respecto al funcionamiento de RISC-V

5.3.9 INSTRUCCIÓN CSTORE

5.3.9.1 Objetivo de la instrucción

La instrucción *CStore* permite realizar operaciones de escritura en memoria utilizando una capability como referencia de acceso. Es la instrucción paralela a *CLoad* para almacenamiento en vez de carga. Su objetivo principal es garantizar que las operaciones de almacenamiento puedan ejecutarse sobre regiones de memoria autorizadas y con permisos válidos de escritura.

A diferencia de las instrucciones de almacenamiento Load y Store de RISC-V, *CStore* incorpora comprobaciones hardware sobre los límites y permisos definidos en la capability utilizada. Esto permite evitar escrituras fuera de rango y reducir posibles errores de corrupción en memoria o accesos malintencionados.

5.3.9.2 Funcionamiento de la instrucción

La instrucción utiliza una capability almacenada en el BancoCap para determinar la dirección donde se realizará la escritura. En esta implementación, la dirección de almacenamiento procede directamente del campo addr de la capability seleccionada.

Por tanto, el desplazamiento dentro de una región de memoria autorizada se realiza previamente mediante la instrucción CIncOffset. Una vez actualizada la dirección de la capability, CStore utiliza dicho campo para acceder a memoria.

La operación de almacenamiento únicamente se ejecuta cuando todas las comprobaciones son válidas. En caso contrario, la señal de escritura de memoria permanece desactivada y la operación queda bloqueada.

5.3.9.3 Integración en la arquitectura

La implementación de *CStore* se realizó modificando la lógica encargada de controlar las operaciones escritura de memoria. Durante la ejecución de la instrucción el sistema verifica la dirección efectiva y comprueba los campos de seguridad asociados a la capability antes de activar la señal de almacenamiento de memoria.

Se requirió alterar la unidad de control para reconocer la nueva instrucción de activar las señales necesarias para las comprobaciones de seguridad previas al acceso a memoria. Por otro lado, fue necesario ampliar el datapath para permitir el transporte de los campos de la capability hacia la validación. De esta manera, se logró mejorar la robustez del sistema frente a errores de memoria.

5.3.10 INSTRUCCIÓN CLOADCAP

5.3.10.1 Objetivo de la instrucción

La instrucción *CLoadCap* permite cargar una capability almacenada en memoria hacia un registro del RISC-V. Su objetivo principal es permitir la reutilización y movimiento de capabilities durante la ejecución del programa, haciendo posible almacenar capabilities en memoria y recuperarlas posteriormente manteniendo sus propiedades de seguridad.

A diferencia de una operación de carga convencional cómo *CLoad*, *CLoadCap* hoy no recupera únicamente un dato, sino todos los campos asociados a la capability.

5.3.10.2 Funcionamiento de la instrucción

La instrucción realiza una lectura de memoria utilizando una capability válida y lleva el contenido hacia un registro. Durante la operación, el hardware recupera todos los campos asociados a la capability almacenada: base, length, address, perms, reserved y el bit de validez.

Antes de realizar la lectura, el sistema verifica que la capability utilizada para acceder a memoria sea válida, que la dirección pertenezca al rango autorizado y que existan permisos de lectura. La capability recuperada de memoria puede utilizarse después en otras operaciones protegidas de memoria.

5.3.10.3 Integración en la arquitectura

La implementación de esta instrucción requirió ampliar la lógica de lectura de memoria para permitir llevar múltiples campos simultáneamente hacia los registros de capability.

Para la incorporación de *CLoadCap,* la unidad de control fue adaptada para reconocer la nueva instrucción. Asimismo, el datapath fue ampliado para soportar el movimiento simultáneo de los distintos campos asociados a la capability almacenada en memoria.

En la Figura 5.8 se puede apreciar como cambiamos la dirección de entrada de la RAM. En la arquitectura clásica de RISC-V, la dirección de entrada era siempre la salida de la ALU. Sin embargo, con esta instrucción la dirección de entrada de la RAM cambia cuando el opcode es “11111”, que pertenece a las instrucciones *CLoad, CStore, CLoadCap y CStoreCap.*

```
ram_addr_in <= cheri_mem_addr when ir_out(6 downto 2) = "11111" else
               alur_out;
```

Figura 5.8: Direccionamiento de entrada de la memoria RAM

5.3.11 INSTRUCCIÓN CSTORECAP

5.3.11.1 Objetivo de la instrucción

La instrucción *CStoreCap* permite almacenar una *capability* completa en memoria.

Su principal objetivo es permitir la transferencia de capabilities durante la ejecución del sistema, haciendo posible guardar capabilities para su utilización posterior. A diferencia de una operación de almacenamiento convencional, *CStoreCap* no escribe únicamente un dato, sino todos los campos asociados a una capability.

5.3.11.2 Funcionamiento de la instrucción

La instrucción toma una capability almacenada en un registro de capacidad y escribe todos sus campos en memoria. La estructura almacenada incluye los campos *base*, *length, address, perms, reserved* y el bit de validez.

Antes de realizar la escritura, el sistema verifica que la capability utilizada para acceder a memoria sea válida, que la dirección destino pertenezca al rango autorizado y que existan permisos de escritura.

La capability almacenada puede recuperarse posteriormente mediante la instrucción CLoadCap.

5.3.11.3 Integración en la arquitectura

La implementación de *CStoreCap* requirió ampliar la lógica de escritura en memoria para permitir almacenar estructuras completas de capability. Durante la ejecución de la instrucción, el sistema transfiere todos los campos de la capability hacia la memoria destino.

La incorporación de *CStoreCap* requirió modificar tanto la unidad de control como el sistema de escritura en memoria del procesador. La unidad de control fue adaptada para

reconocer la nueva instrucción y habilitar la transferencia simultánea de múltiples campos asociados a la capability. Por otro lado, el datapath fue ampliado para soportar el movimiento de las capabilities entre los registros y la memoria. Estas modificaciones permiten almacenar capabilities manteniendo intacta toda la información de seguridad asociada a estas.

El cambio en la dirección de entrada de la RAM explicado en la Figura 5.8 también afecta a esta instrucción.

5.3.12 INSTRUCCIONES CJUMP Y CJUMPLINK

5.3.12.1 Objetivo de las instrucciones

Las instrucciones *CJump* y *CJumpLink* permite modificar el flujo de ejecución de programa utilizando una capability, en vez de un puntero, como referencia de salto. Su objetivo principal es extender el mecanismo tradicional de saltos de RISC-V incorporando comprobaciones hardware sobre la dirección de destino, añadiendo una capa extra de seguridad.

En una arquitectura RISC-V convencional, una instrucción de salto modifica directamente el contador de programa, utilizando una dirección inmediata o almacenada en un registro. En cambio, con *CJump y CJumpLink*, la dirección de salto se obtiene a partir del campo *address* de una capability, y antes de actualizar el contador del programa se comprueba que dice capability sea válida y disponga de los permisos adecuados.

5.3.12.2 Funcionamiento de las instrucciones

Ambas instrucciones utilizan una capability como entrada punto esta capability contiene la dirección destino del salto en su campo address, además de los campos necesarios para validar la operación.

Antes de realizar el salto el hardware comprueba que el campo tag de la capability sea válido, que la dirección de destino se encuentre dentro del rango definido por *base* y

length y que la capability disponga de permiso de ejecución. La diferencia entre ambas instrucciones es que *CJumpLink* guarda también la dirección de retorno. Por tanto, *CJump* se puede utilizar para realizar un salto protegido, mientras que *CJumpLink* permite realizar una llamada protegida a una rutina conservando la dirección de retorno.

5.3.12.3 Integración en la arquitectura

La implementación de más instrucciones requirió modificar la lógica de actualización del contador del programa para permitir que el nuevo valor del PC proceda del campo address de una capability. Durante la ejecución de las instrucciones coma el procesador extrae la dirección de destino, comprueba la validez de la capability y verifica que el permiso de ejecución esté habilitado solo si todas las comprobaciones son correctas se actualiza el contador del programa. Como se observa en la Figura 5.9, la entrada del PC ha sido modificada para estas situaciones que corresponden a cuándo *m_pc* tiene el valor de "10".

```
pc_in <= alu_out when m_pc="00" else
   alur_out      when m_pc="01" else
   cheri_jump_addr when m_pc="10" else
   (others => '0');
```

Figura 5.9: Cambio en la entrada del PC

La incorporación de estas instrucciones requirió además adaptar la unidad de control para conocer los nuevos códigos de operación asociados a los saltos protegidos. Además, durante los estados de estas dos instrucciones se da el valor de *m_pc* deseado. En el caso de *CJumpLink* también se habilitó la escritura de la dirección de retorno en el registro de destino correspondiente. Esto permite que el programa puede regresar posteriormente al punto de ejecución deseado. Estas instrucciones amplían el uso de las capability más allá de accesos de lectura y escritura.

Capítulo 6. ANÁLISIS DE RESULTADOS

6.1 VERIFICACIÓN FUNCIONAL DEL RV32I

Antes de integrar la extensión CHERI, se realizó la verificación funcional del procesador RV32I utilizado como arquitectura base del proyecto. Esta validación resulta necesaria para asegurar que el procesador original ejecuta correctamente las instrucciones principales del conjunto base y que la arquitectura de partida es funcional antes de introducir los nuevos mecanismos de protección de memoria CHERI.

Para ello se desarrolló un banco de pruebas específico en el que se ejecuta un programa almacenado en la memoria ROM del procesador. Dicho programa incluye instrucciones aritmeticológicas, accesos a memoria, saltos condicionales, saltos incondicionales e instrucciones de carga de inmediatos.

El banco de pruebas contabiliza cada escritura válida en el banco de registros y compara el resultado obtenido con el valor esperado mediante sentencias `assert`. De esta forma, la simulación se detiene automáticamente si alguna instrucción produce un resultado incorrecto.

En primer lugar, se verificaron instrucciones aritméticas y lógicas básicas. Las instrucciones `ADDI`, `ADD`, `SUB`, `AND`, `OR` y `XOR` permitieron comprobar el correcto funcionamiento del banco de registros, la ALU y la escritura de resultados. Los valores obtenidos coinciden con los esperados, validando la ejecución de operaciones enteras básicas.

Posteriormente se verificaron las instrucciones de acceso a memoria mediante una operación de almacenamiento y una operación de carga. En concreto, el resultado de la suma almacenado en un registro se escribió en memoria mediante `SW` y posteriormente se recuperó

mediante `LW`, comprobándose que el dato leído coincidía con el valor previamente almacenado.

También se comprobaron instrucciones de salto condicional, incluyendo `BEQ`, `BNE` y `BLT`. En estos casos, el programa de prueba incluye instrucciones que no deben ejecutarse si el salto se realiza correctamente. La validación se confirma al observar que los registros asociados a las instrucciones destino contienen los valores esperados, mientras que las instrucciones intermedias quedan anuladas por el cambio del contador de programa.

Finalmente, se verificaron instrucciones adicionales del conjunto RV32I, entre ellas `JAL`, `LUI`, `AUIPC`, `SLT`, `SLTU`, `SLL`, `SRL` y `SRA`. La simulación finaliza correctamente tras validar todas las escrituras esperadas, confirmando el funcionamiento global de la arquitectura RV32I utilizada como base del proyecto. A continuación, se muestra un ejemplo de cómo se han ido validando las instrucciones:

```
IF dbg_en_reg = '1' AND dbg_rd /= "00000" THEN

   n_write := n_write + 1;

   CASE n_write IS
      WHEN 1 =>
         ASSERT dbg_rd = "00001" AND dbg_din_reg = x"00000005"
         REPORT "Fallo ADDI x1,x0,5"
         SEVERITY failure;

      WHEN 3 =>
         ASSERT dbg_rd = "00011" AND dbg_din_reg = x"0000000C"
         REPORT "Fallo ADD x3,x1,x2"
         SEVERITY failure;
   END CASE;

END IF;
```

Debe señalarse que el programa de prueba utilizado durante la validación funcional incorpora una instrucción NOP en la primera posición de memoria. Esta decisión se debe al funcionamiento del mecanismo de captura de instrucciones implementado en la arquitectura

multiciclo, que requiere un ciclo inicial para cargar correctamente la primera instrucción desde la ROM.

Como consecuencia, la primera instrucción útil del programa se sitúa en la dirección siguiente a la posición inicial de memoria. Esta solución simplifica la validación funcional sin afectar al comportamiento de las instrucciones evaluadas.

6.1.1 VALIDACIÓN DE INSTRUCCIONES ARTIMETICOLÓGICAS

El primer grupo de instrucciones verificado corresponde a las operaciones aritméticológicas básicas del conjunto RV32I. Estas instrucciones permiten comprobar el funcionamiento conjunto del banco de registros, la unidad aritmeticológica y la lógica de escritura de resultados.

El programa de prueba comienza cargando los valores 5 y 7 en los registros x1 y x2 mediante instrucciones ADDI. A partir de estas operaciones, se ejecutan diferentes intrucciones de tipo R: suma, resta, AND, OR y XOR. Cada resultado es escrito en un registro destino distinto y posteriormente comprobado mediante sentencias assert en el banco de pruebas.

La Figura 6.1 y la Figura 6.2 muestra la simulación correspondiente a este primer bloque de instrucciones. En ella se observa la activación de la señal dbg_en_reg en cada escritura válida, el registro destino indicado por dbg_rd y el valor escrito en dbg_din_reg.

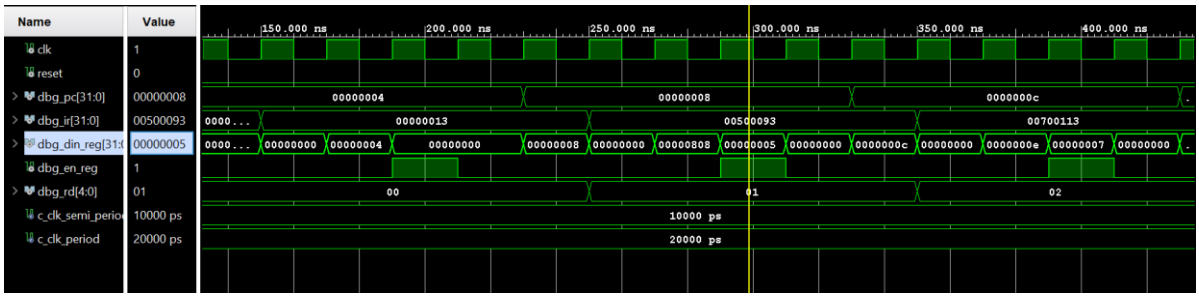


Figura 6.1: Simulación de instrucciones aritmeticológicas en Vivado (1)

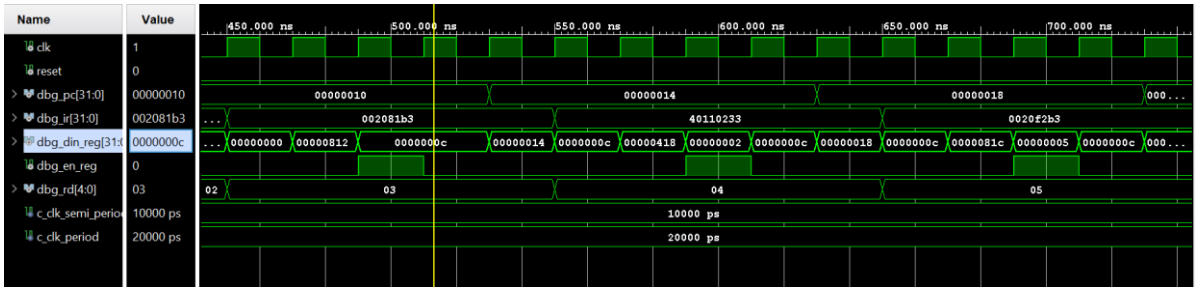


Figura 6.2: Simulación de instrucciones aritmeticológicas en Vivado (2)

Instrucción	***Registro destino***	***Resultado esperado***
ADDI x1, x0, 5	x1	0x00000005
ADDI x2, x0, 7	x2	0x00000007
ADD x3, x1, x2	x3	0x0000000C
SUB x4, x2, x1	x4	0x00000002
AND x5, x1, x2	x5	0x00000005
OR x6, x1, x2	x6	0x00000007
XOR x7, x1, x2	x7	0x00000002

Tabla 6.1: Programa ejecutado para la simulación de las instrucciones aritmeticológicas

Los resultados obtenidos coinciden con los valores esperados para todas las operaciones verificadas. Por tanto, se valida el correcto funcionamiento de la ALU para operaciones aritméticas y lógicas básicas, así como la correcta escritura de los resultados en el banco de registros del procesador.

6.1.2 VALIDACIÓN DE INSTRUCCIONES DE ACCESO A MEMORIA

Las instrucciones de acceso a memoria constituyen otro de los elementos fundamentales de la arquitectura RV32I. Su correcta validación resulta necesaria para garantizar la comunicación entre el procesador y la memoria de datos.

Para verificar este comportamiento se utilizaron las instrucciones `SW` y `LW`. Inicialmente, el resultado de la operación `ADD`, almacenado previamente en el registro `x3`, se escribió en la posición de memoria correspondiente mediante la instrucción `SW`. Posteriormente, dicho valor fue recuperado utilizando una instrucción `LW` y almacenado en el registro `x8`.

La simulación permite comprobar que el dato escrito en memoria coincide exactamente con el valor recuperado durante la operación de lectura. De esta forma se valida tanto el funcionamiento de la RAM como la correcta generación de direcciones y transferencia de datos dentro del datapath del procesador.

La Figura 6.3 muestra la simulación correspondiente a las operaciones de almacenamiento y carga. En ella puede observarse cómo el valor previamente calculado es almacenado en memoria y recuperado posteriormente sin modificaciones.

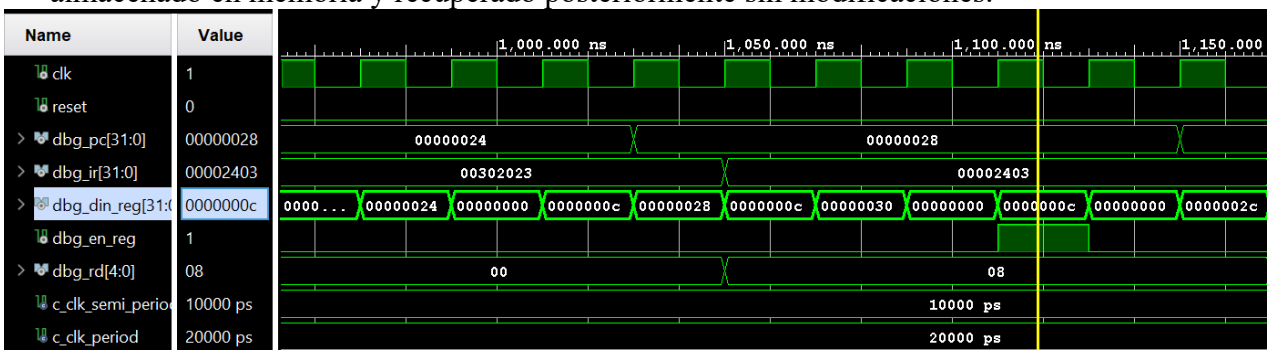


Figura 6.3: Simulación de instrucciones de almacenamiento y carga en Vivado

Instrucción	***Registro destino***	***Resultado esperado***
SW x3, 0(x0)	Memoria[0]	0x0000000C

LW x8, 0(x0)	x8	0x0000000C

Tabla 6.2: Programa ejecutado para la simulación de instrucciones de acceso a memoria

Los resultados obtenidos coinciden con los valores esperados, verificando el correcto funcionamiento de las operaciones de lectura y escritura en memoria dentro de la arquitectura RV32I.

6.1.3 VALIDACIÓN DE INSTRUCCIONES DE SALTO CONDICIONAL

Las instrucciones de salto condicional permiten modificar el flujo de ejecución del programa en función del resultado de una comparación. Su correcto funcionamiento resulta fundamental para implementar estructuras de control como bifurcaciones o bucles.

Para validar este comportamiento se utilizaron las instrucciones `BEQ`, `BNE` y `BLT`. En cada caso se diseñó una secuencia de prueba en la que, si el salto se realizaba correctamente, una instrucción intermedia no debía llegar a ejecutarse. De esta forma, la validación no solo comprueba que la condición se evalúa correctamente, sino también que el contador de programa se actualiza adecuadamente.

En la primera prueba (Figura 6.4) se utilizó una instrucción `BEQ` para comparar los registros `x1` y `x1`. Dado que ambos registros contienen el mismo valor, la condición se cumple y el procesador debe saltar directamente a la instrucción que carga el valor `42` en el registro `x10`. La instrucción intermedia que carga el valor `99` en el registro `x9` no debe ejecutarse.

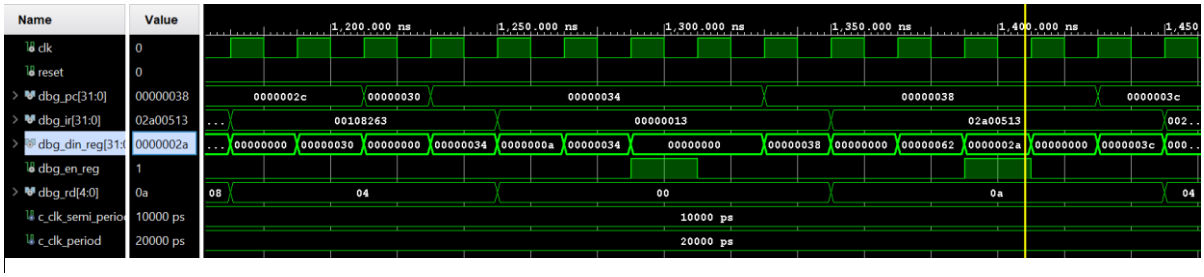


Figura 6.4: Simulación de la instrucción BEQ en Vivado

Posteriormente se verificó la instrucción `BNE` comparando los registros `x1` y `x2` (**¡Error! No se encuentra el origen de la referencia.**). Como ambos contienen valores distintos, la condición resulta verdadera y el salto se realiza correctamente, evitando la ejecución de la instrucción que habría cargado el valor `99` en `x11` y ejecutando directamente la carga del valor `55` en `x12`.

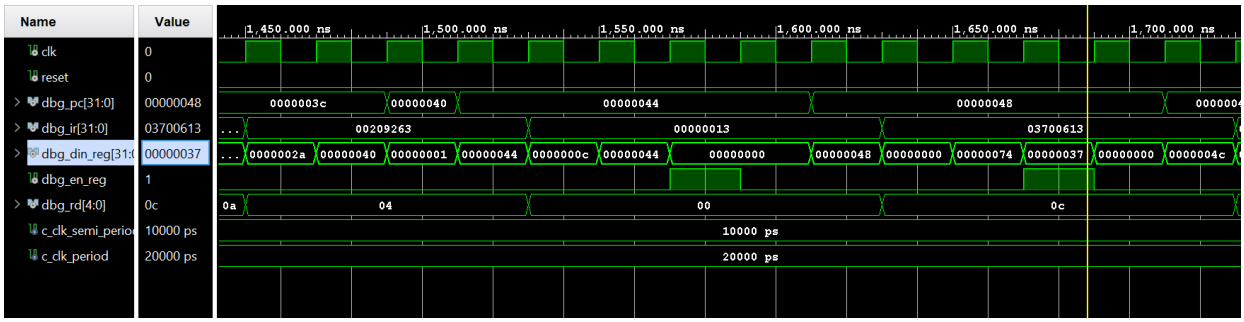


Figura 6.6: Simulación de la instrucción BNE en Vivado

Finalmente se validó la instrucción `BLT`. Dado que el contenido de `x1` es menor que el de `x2`, la condición se satisface y el procesador salta correctamente a la instrucción que escribe el valor `77` en `x14`, evitando nuevamente la ejecución de la instrucción intermedia.

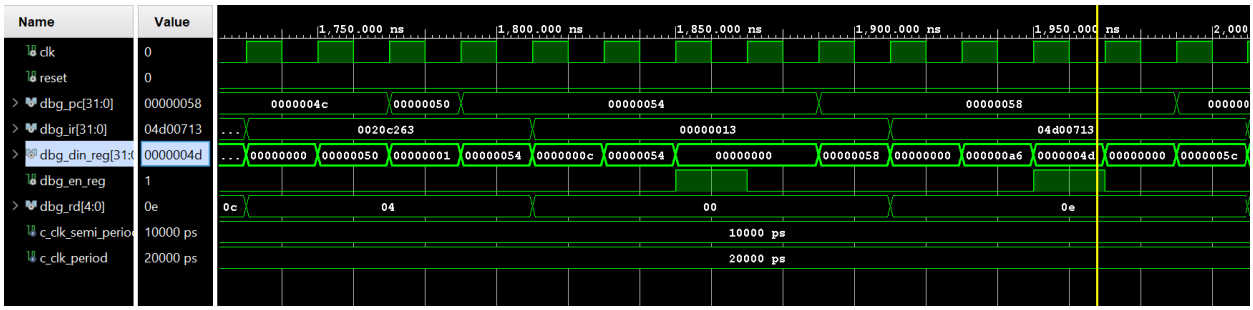


Figura 6.5: Simulación de la instrucción BLT en Vivado

Instrucción	***Resultado esperado***
BEQ x1,x1,+4	x10 = 0x0000002A (42)
BNE x1,x2,+4	x12 = 0x00000037 (55)
BLT x1,x2,+4	x14 = 0x0000004D (77)

Tabla 6.3: Programa ejecutado para la simulación de instrucciones de salto condicional

Los resultados obtenidos confirman que las condiciones de comparación son evaluadas correctamente y que la actualización del contador de programa se realiza de forma adecuada. Asimismo, la no ejecución de las instrucciones intermedias demuestra el correcto funcionamiento de la lógica de control asociada a los saltos condicionales.

6.1.4 VALIDACIÓN DE INSTRUCCIONES DE SALTO, COMPARACIÓN Y DESPLAZAMIENTO

Además de las operaciones aritméticas, de memoria y de salto condicional, se verificaron diversas instrucciones adicionales pertenecientes al conjunto base RV32I. Estas pruebas permiten validar funcionalidades relacionadas con el control del flujo de ejecución, la carga de inmediatos, las comparaciones y los desplazamientos lógicos y aritméticos.

En primer lugar, se comprobó la instrucción `JAL` (*Jump And Link*). Esta instrucción modifica el contador de programa y almacena simultáneamente la dirección de retorno en un registro destino. La simulación demuestra que el salto se realiza correctamente, evitando la ejecución de la instrucción intermedia y alcanzando la instrucción que carga el valor `88` en el registro `x17`. Asimismo, se verifica que la dirección de retorno se almacena correctamente en el registro `x15`.

Posteriormente se validaron las instrucciones `LUI` y `AUIPC`, utilizadas para la generación de constantes y para el cálculo de direcciones relativas al contador de programa. En ambos casos los valores obtenidos coinciden con los resultados esperados.

También se verificaron las instrucciones de comparación `SLT` y `SLTU`. Dado que el contenido del registro `x1` es menor que el de `x2`, ambas instrucciones generan el valor lógico uno como resultado, confirmando el correcto funcionamiento de las comparaciones con y sin signo.

Finalmente se comprobaron las instrucciones de desplazamiento `SLL`, `SRL` y `SRA`. Estas operaciones permiten desplazar los bits de un operando hacia la izquierda o hacia la derecha según el tipo de operación especificado. Los resultados obtenidos coinciden con los valores teóricos esperados para cada desplazamiento.

La Figura 6.8 y la Figura 6.7 muestran la simulación correspondiente a este conjunto de instrucciones. En ella pueden observarse las escrituras producidas en los registros destino y la correcta ejecución de las operaciones verificadas.

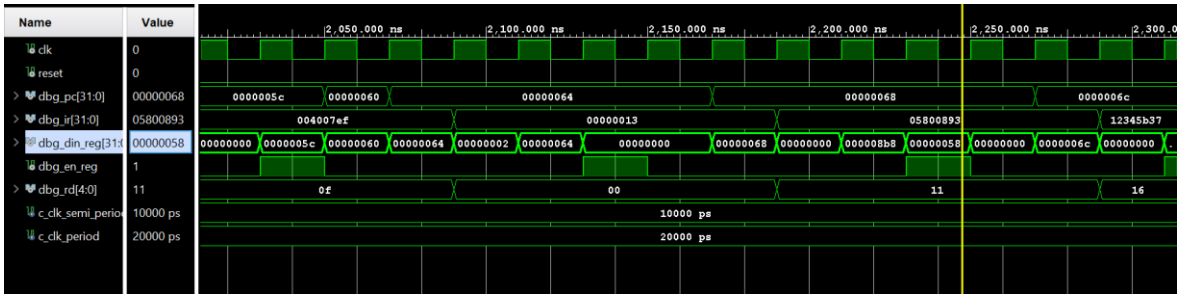


Figura 6.8: Simulación de instrucciones de salto, comparación y desplazamiento en Vivado (1)

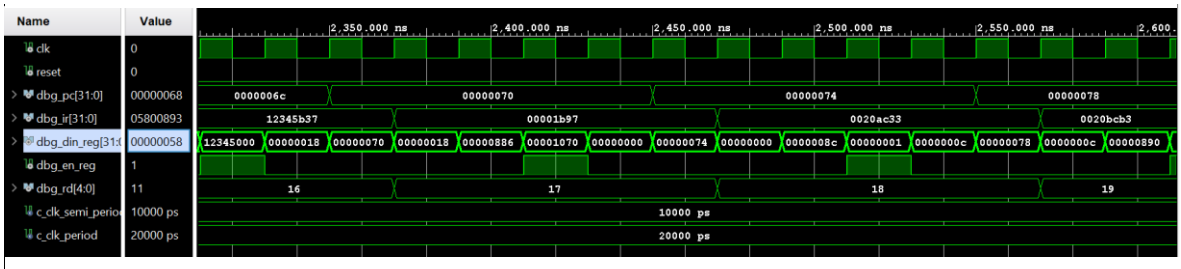


Figura 6.7: Simulación de instrucciones de salto, comparación y desplazamiento en Vivado (2)

Instrucción	***Registro destino***	***Resultado esperado***
JAL x15,+4	x15	Dirección de retorno válida
ADDI x17,x0,88	x17	0x00000058
LUI x22,0x12345	x22	0x12345000
AUIPC x23,0x1	x23	Valor relativo al PC
SLT x24,x1,x2	x24	0x00000001
SLTU x25,x1,x2	x25	0x00000001
SLL x26,x1,x1	x26	0x000000A0
SRL x27,x2,x1	x27	0x00000000
SRA x28,x2,x1	x28	0x00000000

Tabla 6.4:Programa ejecutado para la simulación de instrucciones de salto, comparación y desplazamiento

Los resultados obtenidos demuestran el correcto funcionamiento de las instrucciones restantes del conjunto RV32I implementado. Con ello queda validada la arquitectura base utilizada en el proyecto, proporcionando una plataforma funcional sobre la que posteriormente se integró la extensión CHERI para la incorporación de mecanismos de protección de memoria.

6.2 VERIFICACIÓN FUNCIONAL DE LOS BLOQUES HARDWARE

6.2.1 VERIFICACIÓN DEL BANCO DE REGISTROS DE CAPABILITIES

La primera verificación funcional realizada corresponde al banco de registros de capabilities, denominado BancoCap. El objetivo de esta simulación fue comprobar que el bloque permite almacenar, leer y mantener correctamente estructuras completas de tipo capability.

El banco de pruebas desarrollado genera una señal de reloj de periodo 20 ns y aplica una secuencia de estímulos orientada a verificar los casos principales de funcionamiento: inicialización tras reset, lectura de registros, escritura de una nueva capability, bloqueo de escritura cuando la señal de habilitación está desactivada y lectura simultánea por los dos puertos de salida.

En primer lugar, se comprueba que, tras la activación de reset, el registro c0 contiene una capability nula. Posteriormente se verifica que el registro c1 queda inicializado con una capability válida, con base 0x0100, longitud 0x0020, dirección 0x0100, permisos 0xFF y tag activo. Esta capability raíz, explicada en el apartado 5.2.4, se utiliza como referencia válida dentro del entorno de pruebas.

A continuación, el testbench escribe una nueva capability en el registro c5 activando la señal `enW`. La capability utilizada en esta prueba contiene base 0x0200, longitud 0x0040, dirección 0x0210, permisos 0x0F y tag uno. Tras desactivar la escritura, se lee el registro c5 a través del puerto A y se comprueba que el contenido almacenado coincide exactamente con la capability introducida.

También se verifica que el banco no modifica su contenido cuando la señal `enW` permanece desactivada. Para ello, se intenta escribir una nueva capability en el mismo registro c5 con `enW = 0`. La simulación confirma que el contenido previo no se sobrescribe, manteniéndose la capability almacenada anteriormente.

Finalmente, se comprueba la lectura simultánea de dos capabilities mediante los puertos `capA` y `capB`. En esta prueba se accede al registro c1 a través del puerto A y al registro c5 a través del puerto B, verificando que ambos valores se presentan correctamente en paralelo.

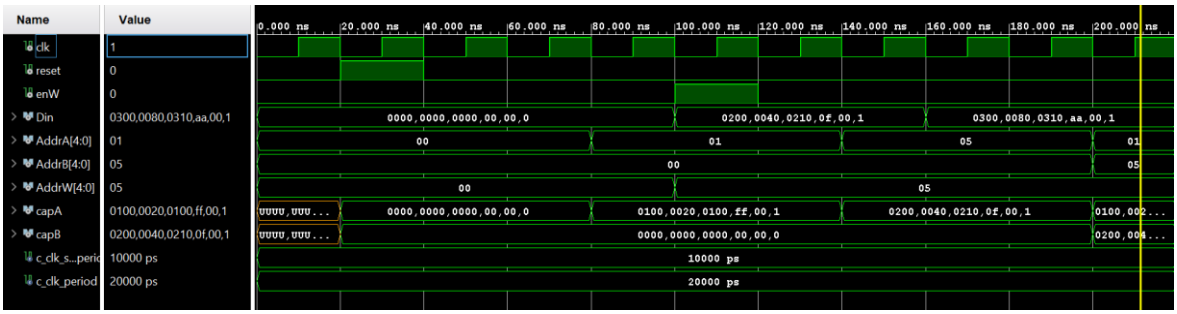


Figura 6.9: Simulación de BancoCap en Vivado

La Figura 6.9 muestra la simulación funcional del BancoCap. En ella puede observarse la activación de la señal de escritura, el direccionamiento de los registros y la actualización de las salidas `capA` y `capB` con las capabilities seleccionadas.

Los resultados obtenidos demuestran el correcto funcionamiento del BancoCap. Las simulaciones verifican tanto las operaciones de lectura y escritura como el comportamiento del bloque ante señales de control deshabilitadas, confirmando que el almacenamiento y recuperación de capabilities se realiza de forma correcta.

Prueba	***Resultado esperado***	***Resultado obtenido***
Reset del banco	c0 = NULL_CAPABILITY	Correcto
Lectura de c1	Capability válida inicial	Correcto
Escritura en c5 con enW = 1	Se almacena la capability	Correcto

Escritura con enW = 0	No se modifica c5	Correcto
Lectura simultánea	capA y capB correctos	Correcto

Tabla 6.5: Pruebas comprobadas durante la simulación de BancoCap

6.2.2 VERIFICACIÓN DE LA MEMORIA DE CAPABILITIES (CAPRAM)

La segunda verificación funcional corresponde a la memoria de capabilities, denominada CapRAM. El objetivo de esta simulación fue comprobar que el bloque permite almacenar y recuperar capabilities completas en distintas posiciones de memoria, manteniendo correctamente la independencia entre direcciones.

El banco de pruebas genera una señal de reloj de periodo 20 ns y aplica una secuencia de estímulos sobre las señales `addr`, `d_in`, `we` y `d_out`. En primer lugar, se comprueba que una posición de memoria inicial contiene una capability nula. Posteriormente, se escribe una capability válida en la dirección `0x00000010` activando la señal `we`.

La capability escrita en esta primera prueba presenta base `0x1000`, longitud `0x0020`, dirección `0x1004`, permisos `0xFF` y tag activo. Tras desactivar la escritura, se verifica que el valor leído en `d_out` coincide con la capability almacenada.

A continuación, se accede a una dirección distinta, `0x00000020`, comprobando que dicha posición no ha sido modificada indebidamente. También se verifica que, cuando la señal `we` permanece desactivada, la memoria no sobrescribe el contenido previamente almacenado.

Finalmente, se escribe una segunda capability en la dirección `0x00000020` y se comprueba que ambas posiciones mantienen sus valores de forma independiente. La Figura 6.10 muestra la simulación funcional de CapRAM, donde se observan las operaciones de escritura, lectura y conservación de datos en distintas direcciones.

Prueba	***Resultado esperado***	***Resultado obtenido***
Memoria inicial	d_out = NULL_CAPABILITY	Correcto
Escritura en 0x00000010	Se almacena cap1	Correcto
Lectura de otra dirección	No se modifica indebidamente	Correcto
Escritura con we = 0	No se sobrescribe cap1	Correcto
Escritura en 0x00000020	Se almacena cap2	Correcto
Relectura de 0x00000010	cap1 permanece intacta	Correcto

Tabla 6.6: Pruebas comprobadas durante la simulación de CapRAM

Los resultados obtenidos demuestran el correcto funcionamiento de la CapRAM. La simulación verifica que la memoria permite almacenar capabilities completas, acceder a ellas mediante direcciones independientes y evitar escrituras no autorizadas cuando la señal de habilitación se encuentra desactivada.

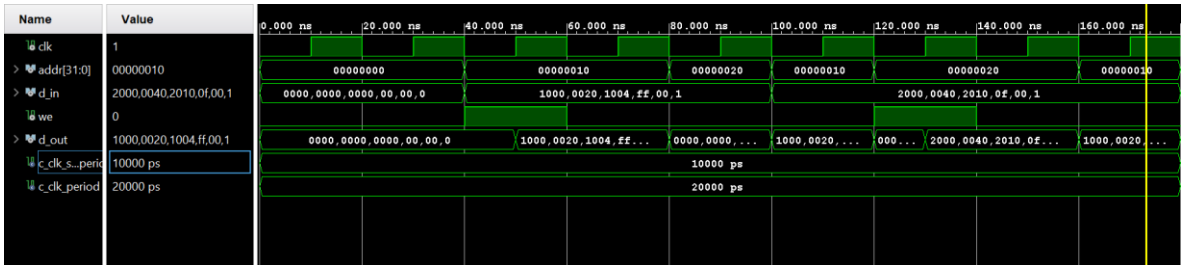


Figura 6.10: Simulación de CapRAM en Vivado

6.2.3 VERIFICACIÓN DE LA UNIDAD CHERI CHECK UNIT

La unidad CHERI Check Unit constituye el principal mecanismo de protección de memoria implementado en la arquitectura RV32Y. Su función es verificar que una capability cumple simultáneamente las condiciones de validez, límites y permisos antes de permitir una

operación protegida. Por este motivo, su correcta validación resulta fundamental para garantizar el funcionamiento del sistema.

Para comprobar el comportamiento de este bloque se desarrolló un banco de pruebas específico que evalúa tanto situaciones válidas como casos en los que el acceso debe ser rechazado. Durante las simulaciones se verificaron las comprobaciones de validez mediante el campo *tag*, la comprobación de límites de memoria mediante los campos *base* y *length* y la comprobación de permisos para todas las operaciones soportadas por la arquitectura: lectura, escritura, ejecución, carga de capabilities y almacenamiento de capabilities.

Inicialmente se definió una capability válida con dirección base `0x0100`, longitud `0x0020`, permisos completos y bit de validez activo. Esta configuración genera un rango de memoria válido comprendido entre las direcciones `0x0100` y `0x011F`, siendo `0x0120` la primera dirección fuera de los límites permitidos.

La simulación verifica en primer lugar operaciones válidas de lectura, escritura, ejecución, carga de capabilities y almacenamiento de capabilities dentro de la región autorizada. En estos casos las señales `tag_ok`, `bounds_ok` y el permiso correspondiente toman el valor lógico uno, provocando que la salida `check_ok` también se active. Este resultado indica que la operación solicitada cumple todas las restricciones impuestas por la capability y puede ejecutarse correctamente.

Posteriormente se analizaron los distintos permisos de forma individual. Para cada tipo de operación se eliminó el bit correspondiente de la máscara de permisos, comprobándose que la señal asociada (`permit_load_ok`, `permit_store_ok`, `permit_exec_ok`, `permit_lcap_ok` o `permit_scap_ok`) pasa a valer cero y que la salida final `check_ok` también se desactiva. De esta forma se verifica que la unidad impide correctamente la ejecución de operaciones no autorizadas.

Asimismo, se estudiaron diversos accesos fuera de límites. En concreto, se comprobó el comportamiento para direcciones inferiores a la dirección base, para la dirección situada

exactamente en el límite superior del rango y para direcciones claramente alejadas de la región autorizada. En todos estos casos la señal `bounds_ok` toma el valor cero, provocando que la salida final `check_ok` también sea cero y bloqueando el acceso solicitado.

Finalmente, se verificó el comportamiento ante capabilities inválidas y regiones de memoria vacías. En el primer caso se desactivó el bit de validez `tag`, comprobándose que cualquier operación es rechazada independientemente de que los límites y permisos sean correctos. En el segundo caso se utilizó una capability con longitud igual a cero, obteniéndose un rango vacío para el que ninguna dirección resulta válida. Ambas situaciones producen correctamente una salida `check_ok` igual a cero.

La Figura 6.11 muestra una parte representativa de la simulación funcional de la unidad CHERI Check Unit. En ella pueden observarse las distintas solicitudes de acceso, la activación y desactivación de los permisos asociados a cada operación y la evolución de las señales de validación utilizadas para generar la salida final `check_ok`.

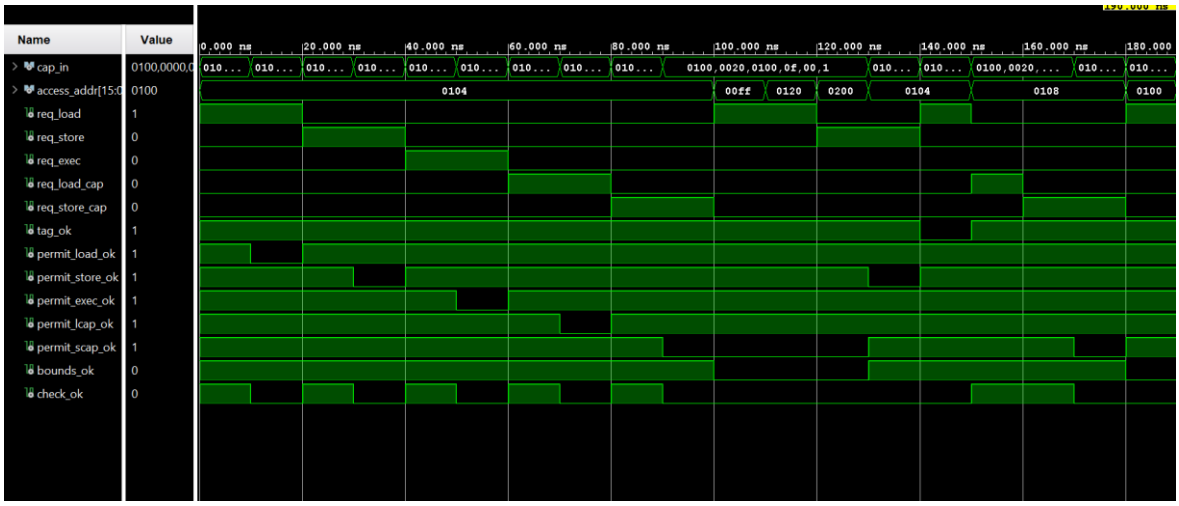


Figura 6.11: Simulación de cheri_check_unit en Vivado

Prueba	***Resultado esperado***	***Resultado obtenido***
LOAD con permiso	check_ok = 1	Correcto
LOAD sin permiso	check_ok = 0	Correcto
STORE con permiso	check_ok = 1	Correcto
STORE sin permiso	check_ok = 0	Correcto
EXEC con permiso	check_ok = 1	Correcto
EXEC sin permiso	check_ok = 0	Correcto
LOAD_CAP con permiso	check_ok = 1	Correcto
LOAD_CAP sin permiso	check_ok = 0	Correcto
STORE_CAP con permiso	check_ok = 1	Correcto
STORE_CAP sin permiso	check_ok = 0	Correcto
Dirección fuera de límites	bounds_ok = 0	Correcto
Tag inválido	check_ok = 0	Correcto
Length = 0	check_ok = 0	Correcto

Tabla 6.7: Pruebas comprobadas durante la simulación de cheri_check_unit

Los resultados obtenidos demuestran el correcto funcionamiento de la unidad CHERI Check Unit. Las simulaciones verifican que el bloque valida adecuadamente la autenticidad de las capabilities, controla los límites de acceso a memoria y comprueba los permisos asociados a cada operación. La señal check_ok únicamente se activa cuando todas las

condiciones de seguridad son cumplidas simultáneamente, confirmando que la unidad implementa correctamente el mecanismo de protección de memoria previsto para la arquitectura RV32Y.

6.3 VERIFICACIÓN FUNCIONAL DEL RV32Y

Una vez validado el correcto funcionamiento de la arquitectura RV32I y de los bloques CHERI individuales, se procedió a verificar la arquitectura RV32Y completa. Esta validación se centró exclusivamente en las instrucciones CHERI implementadas, ya que las instrucciones pertenecientes al conjunto base RV32I habían sido comprobadas previamente.

Para ello se desarrolló un programa de prueba almacenado en la ROM del procesador RV32Y. Este programa ejecuta de forma secuencial instrucciones de inspección, modificación de capabilities, acceso protegido a memoria y transferencia de control.

La validación comprueba tanto operaciones autorizadas como operaciones que deben ser bloqueadas por el hardware. En particular, se verifican saltos protegidos con y sin permiso de ejecución, accesos de carga y almacenamiento con permisos válidos e inválidos, modificación de límites y permisos de capabilities, y almacenamiento y recuperación de capabilities mediante CapRAM.

Al igual que ocurre durante la validación del procesador RV32I, el programa de prueba utilizado para la arquitectura RV32Y incorpora una instrucción NOP en la primera posición de memoria. Esta instrucción actúa como burbuja de arranque y permite que el mecanismo de captura de instrucciones complete correctamente la carga de la primera instrucción útil del programa.

6.3.1 VALIDACIÓN DE INSTRUCCIONES DE INSPECCIÓN Y GESTIÓN

El primer grupo de instrucciones CHERI verificado corresponde a las operaciones de inspección y gestión de capabilities. Estas instrucciones no realizan accesos directos a memoria de datos, sino que permiten consultar o modificar campos concretos de una capability almacenada en el BancoCap.

En primer lugar, se validó la instrucción `CGetTag` (Figura X1), encargada de leer el bit de validez de una capability. En el programa de prueba se ejecuta `CGetTag x5,c1`, utilizando la capability inicial almacenada en `c1`. Dado que dicha capability es válida, el resultado esperado es la escritura del valor `1` en el registro general `x5`. La simulación confirma que `x5` recibe correctamente el valor `0x00000001`.

Posteriormente se comprobó la instrucción `CIncOffset` (Figura 6.12). Para ello se carga previamente el valor `16` en el registro `x1` y se ejecuta `CIncOffset c2,c1,x1`. Esta operación genera una nueva capability en `c2`, manteniendo los campos principales de `c1` y actualizando el campo `addr`. Como la dirección inicial de `c1` es `0x0100`, el resultado esperado tras sumar el desplazamiento es `0x0110`. La simulación verifica que la capability escrita en `c2` contiene efectivamente `addr = 0x0110`.

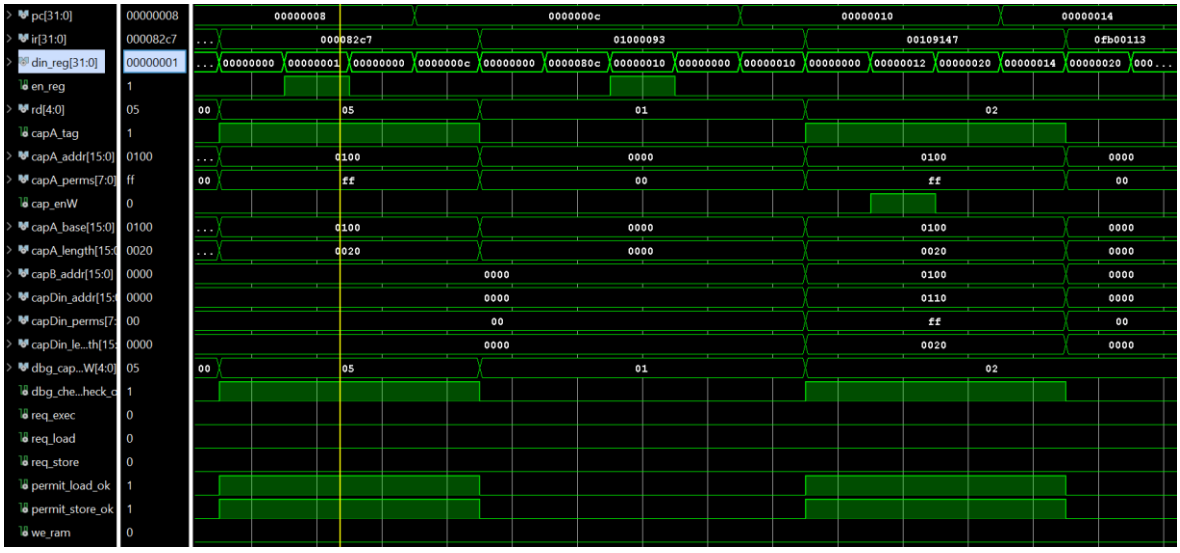


Figura 6.12: Simulación de instrucciones de inspección y gestión en Vivado (1)

A continuación, se validó la instrucción `CSetPerm` (Figura 6.13). En la primera prueba se utiliza la máscara `0xFB`, que elimina el permiso de ejecución de la capability. El resultado se almacena en `c3`, comprobándose que el campo `perms` de la nueva capability toma el valor `0xFB`. Esta prueba resulta especialmente importante porque se utiliza posteriormente para comprobar que un salto protegido mediante `CJump` es bloqueado cuando la capability no dispone de permiso de ejecución.

Finalmente, se comprobó la instrucción `CSetBounds` . Para ello se carga el valor `8` en el registro `x4` y se ejecuta `CSetBounds c4,c2,x4`. En esta implementación, la instrucción modifica el campo `length` de la capability destino, manteniendo el resto de los campos principales. La simulación confirma que la capability escrita en `c4` contiene `length = 0x0008` y conserva la dirección `addr = 0x0110`.

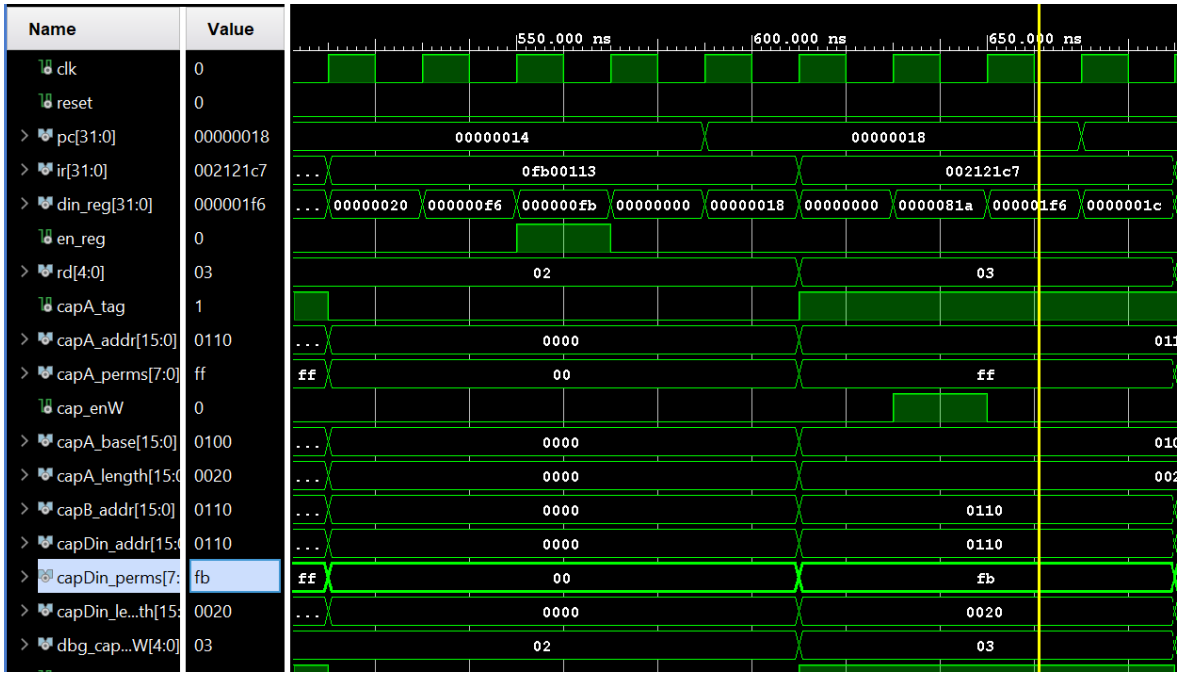


Figura 6.13: Simulación de instrucciones de inspección y gestión en Vivado (2)

Instrucción	***Resultado esperado***	***Resultado obtenido***
CGetTag x5,c1	x5 = 0x00000001	Correcto
CIncOffset c2,c1,x1	c2.addr = 0x0110	Correcto
CSetPerm c3,c2,x2	c3.perms = 0xFB	Correcto
CSetBounds c4,c2,x4	c4.length = 0x0008	Correcto

Tabla 6.8: Pruebas comprobadas durante la simulación de las instrucciones de inspección y gestión

Los resultados obtenidos validan el correcto funcionamiento de las instrucciones de inspección y gestión de capabilities. Estas pruebas confirman que la arquitectura RV32Y es capaz de consultar el estado de validez de una capability, modificar su dirección efectiva, restringir sus permisos y ajustar sus límites de acceso.

6.3.2 VALIDACIÓN DE INSTRUCCIONES DE TRANSFERENCIA DE CONTROL

Las instrucciones de transferencia de control constituyen uno de los mecanismos más relevantes de la arquitectura CHERI, ya que permiten proteger los cambios de flujo de ejecución mediante capabilities. A diferencia de los saltos tradicionales de RISC-V, estas instrucciones verifican que la capability utilizada para realizar el salto sea válida y disponga de permiso de ejecución antes de actualizar el contador de programa.

Para validar este comportamiento se empleó la instrucción `CJump` en dos situaciones diferentes. En la primera prueba se utilizó la capability almacenada en `c3`, obtenida previamente mediante la instrucción `CSetPerm`. Esta capability presenta una máscara de permisos igual a `0xFB`, lo que implica que el permiso de ejecución ha sido eliminado.

Al ejecutarse la instrucción (Figura 6.14) `CJump c3`, la unidad CHERI Check Unit detecta la ausencia del permiso de ejecución y desactiva la señal `check_ok`. Como consecuencia, el salto es bloqueado y el procesador continúa ejecutando secuencialmente la

siguiente instrucción del programa. Este comportamiento se verifica observando que la instrucción `addi x6,x0,33` se ejecuta correctamente, escribiendo el valor decimal 33 en el registro `x6`.

Posteriormente se ejecuta una segunda instrucción (Figura 6.15) `CJump`, esta vez utilizando la capability almacenada en `c2`. Dicha capability conserva el permiso de ejecución y, por tanto, supera correctamente todas las comprobaciones realizadas por la unidad de validación. En este caso la señal `check_ok` toma el valor lógico uno y el contador de programa se actualiza con la dirección almacenada en la capability.

```
6 => x"0001C047", -- CJump c3, sin permiso exec, NO debe saltar
7 => x"02100313", -- addi x6,x0,33, debe ejecutarse
8 => x"00014047", -- CJump c2, con permiso exec, debe saltar
9 => x"00000013", -- nop de relleno
68 => x"00000013", -- nop en destino 0x0110
69 => x"02C00393", -- addi x7,x0,44, debe ejecutarse tras CJump c2
```

La simulación confirma que el salto se realiza correctamente, alcanzando la dirección de destino definida en el campo `addr` de la capability. Como resultado, se ejecuta la instrucción `addi x7,x0,44`, almacenándose el valor decimal 44 en el registro `x7`.

Los resultados obtenidos demuestran que la arquitectura RV32Y implementa correctamente la protección de saltos mediante capabilities. La unidad CHERI Check Unit impide la ejecución de saltos cuando la capability carece de permiso de ejecución y autoriza únicamente aquellos accesos que cumplen simultáneamente las condiciones de validez, límites y permisos establecidas por la arquitectura.

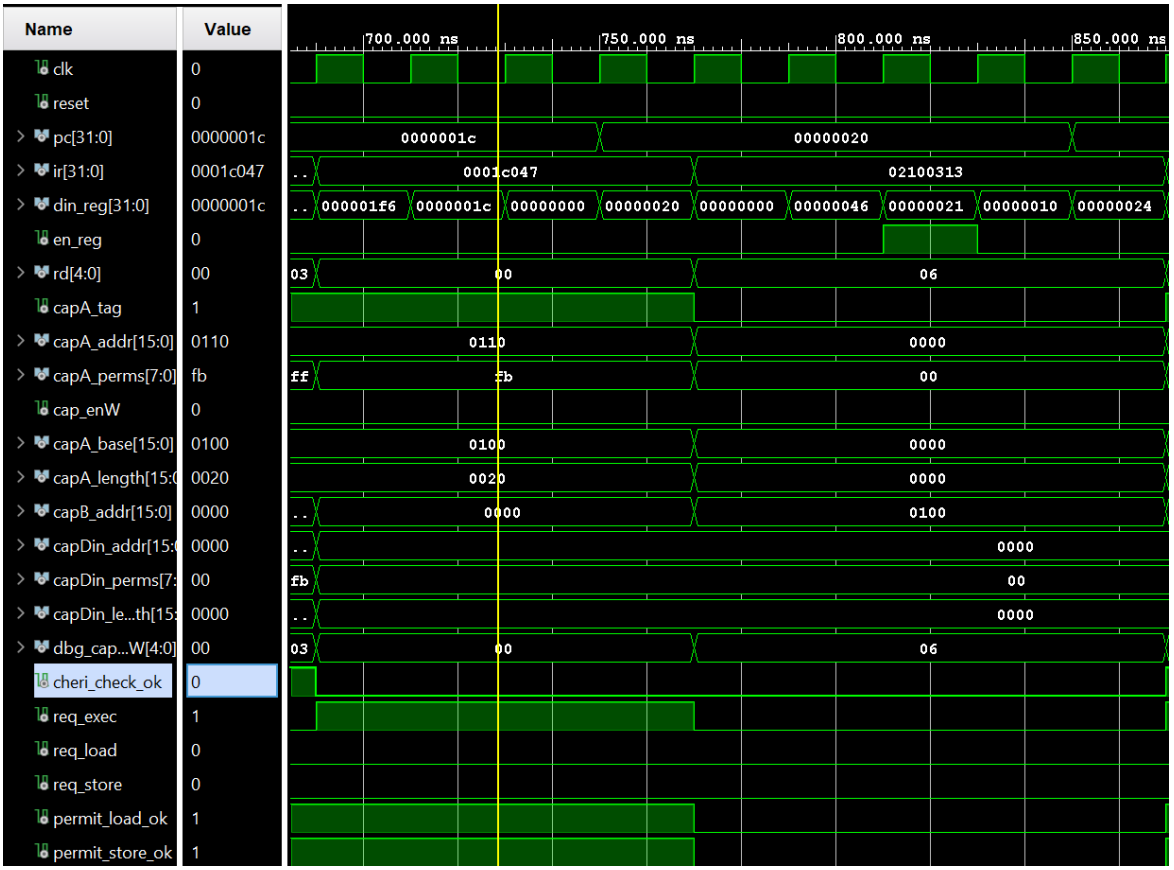


Figura 6.14: Simulación de instrucciones de transferencia de control en Vivado (1)

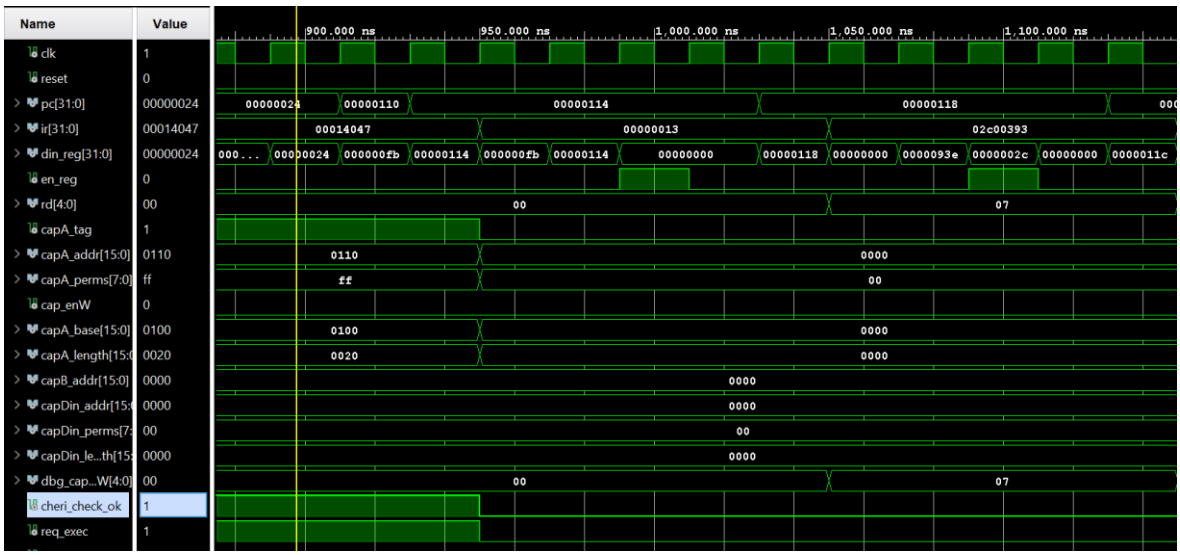


Figura 6.15: Simulación de instrucciones de transferencia de control en Vivado (2)

6.3.3 VALIDACIÓN DE INSTRUCCIONES DE ACCESO PROTEGIDO A MEMORIA

Las instrucciones de acceso protegido a memoria constituyen uno de los principales mecanismos de seguridad introducidos por la extensión CHERI. A diferencia de las instrucciones convencionales de carga y almacenamiento de RISC-V (Load y Store), estas operaciones verifican previamente que la capability utilizada dispone de los permisos necesarios para realizar el acceso solicitado.

Para validar este comportamiento se emplearon las instrucciones `CLoad` y `CStore` utilizando dos capabilities con configuraciones de permisos diferentes. La primera capability, almacenada en `c5`, fue generada mediante la instrucción `CSetPerm`, eliminando los permisos de lectura y escritura mediante una máscara de permisos igual a `0xF8`. La segunda capability, almacenada en `c2`, conserva los permisos originales y permite realizar operaciones de carga y almacenamiento.

En primer lugar, se ejecutó la instrucción `CLoad x8,c5` (Figura 6.16). Durante esta operación, la unidad CHERI Check Unit detecta que la capability utilizada no dispone de permiso de lectura. Como consecuencia, la señal `permit_load_ok` toma el valor lógico cero y la señal `check_ok` también permanece desactivada. Esto provoca que la operación sea bloqueada por el hardware y no se produzca ningún acceso válido a memoria.

Posteriormente se verificó la instrucción `CStore x9,c5` (Figura 6.16). De forma análoga al caso anterior, la capability empleada carece de permiso de escritura. La unidad de comprobación detecta esta restricción y desactiva nuevamente la señal `check_ok`, impidiendo que la operación de almacenamiento llegue a ejecutarse.

```
74 => x"0002847F", -- CLoad  x8,c5    SIN permiso load
75 => x"0092907F", -- CStore x9,c5    SIN permiso store
```

Una vez comprobados los casos de acceso denegado, se procedió a validar las operaciones autorizadas. Para ello se utilizaron las instrucciones `CLoad x8,c2` y `CStore x9,c2` (Figura 6.17). Como la capability almacenada en `c2` mantiene los permisos de lectura y escritura, las señales `permit_load_ok`, `permit_store_ok` y `check_ok` toman el valor lógico uno, permitiendo que las operaciones se ejecuten correctamente.

Los resultados obtenidos demuestran que la arquitectura RV32Y implementa correctamente la protección de accesos a memoria mediante capabilities. Las operaciones de lectura y escritura únicamente son autorizadas cuando la capability dispone explícitamente de los permisos necesarios, bloqueándose cualquier acceso que no cumpla las restricciones definidas por la política de seguridad establecida.

Figura 6.16: Simulación de instrucciones de acceso a memoria en Vivado (1)

Figura 6.17: Simulación de instrucciones de acceso a memoria en Vivado (2)

6.3.4 VALIDACIÓN DE ALMACENAMIENTO Y RECUPERACIÓN DE CAPABILITIES

Las instrucciones `CStoreCap` y `CLoadCap` permiten transferir capabilities completas entre el BancoCap y la memoria de capabilities. La validación de estas instrucciones resulta esencial para garantizar la persistencia y reutilización de capabilities durante la ejecución de programas.

En la primera prueba se ejecutó la instrucción `CStoreCap c4,0(c2)`. Esta operación almacena en memoria la capability contenida en el registro `c4`, incluyendo todos sus campos: dirección base, longitud, dirección actual, permisos y bit de validez. La capability utilizada presenta una longitud reducida de `0x0008`, resultado de la instrucción `CSetBounds` ejecutada previamente.

Posteriormente se ejecutó la instrucción `CLoadCap  c6,0(c2)`, recuperando la capability previamente almacenada y escribiéndola en el registro `c6`. El banco de pruebas verifica individualmente todos los campos de la capability cargada para comprobar que la información recuperada coincide exactamente con la almacenada.

La simulación (Figura 6.18) confirma que los campos `base`, `addr`, `length`, `perms` y `tag` conservan sus valores originales tras el proceso de almacenamiento y recuperación. Esto demuestra el correcto funcionamiento conjunto del BancoCap y la CapRAM.

```
78 => x"0041307F", -- CStoreCap c4,0(c2)
79 => x"0001237F", -- CLoadCap  c6,0(c2)
```

Campo verificado	***Valor esperado***	***Resultado obtenido***
Base	0x0100	Correcto
Address	0x0110	Correcto
Length	0x0008	Correcto
Permisos	0xFF	Correcto
Tag	1	Correcto

Tabla 6.9: Pruebas comprobadas durante la simulación de las instrucciones de almacenamiento y recuperación de capabilities

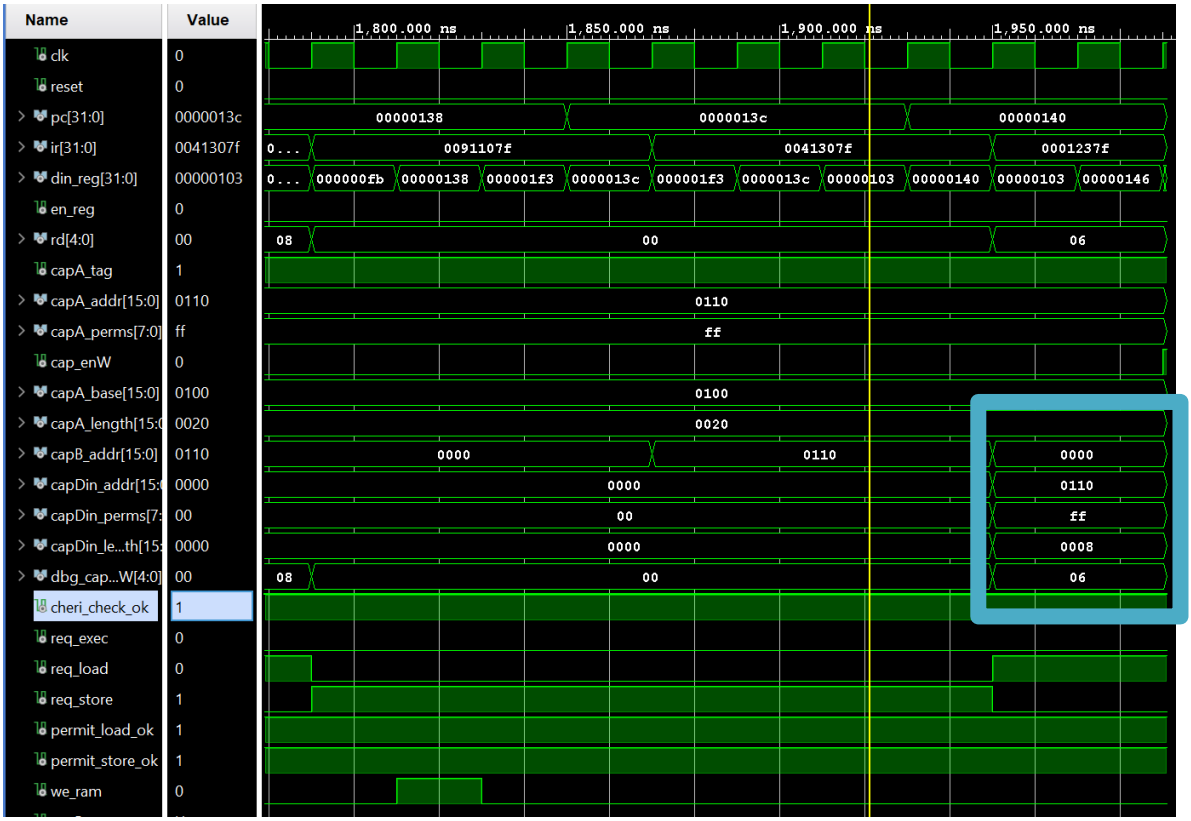


Figura 6.18: Simulación de instrucciones de almacenamiento y recuperación de capabilities en Vivado

Los resultados obtenidos validan el correcto funcionamiento de las instrucciones CStoreCap y CLoadCap. La arquitectura RV32Y es capaz de almacenar y recuperar capabilities completas manteniendo intacta toda la información de seguridad asociada, lo que permite reutilizar capabilities de forma segura durante la ejecución de programas.

6.4 CASOS DE USO Y ANÁLISIS DE SEGURIDAD

6.4.1 RESTRICCIÓN DE PERMISOS

Uno de los principales mecanismos de seguridad proporcionados por CHERI es la posibilidad de restringir dinámicamente los permisos asociados a una capability. Esta

funcionalidad permite aplicar el principio de mínimo privilegio, según el cual cada proceso debe disponer únicamente de los permisos estrictamente necesarios para realizar su tarea [19].

La arquitectura desarrollada permite implementar este mecanismo mediante la instrucción `CSetPerm`. Durante las pruebas realizadas, una capability inicialmente válida y con permisos completos fue modificada para eliminar el permiso de ejecución. Posteriormente se intentó realizar un salto mediante la instrucción `CJump` utilizando dicha capability.

La simulación demuestra que la unidad CHERI Check Unit detecta correctamente la ausencia del permiso de ejecución y bloquea la operación solicitada. Como consecuencia, el procesador continúa ejecutando las instrucciones siguientes sin modificar el contador de programa.

Este comportamiento reduce significativamente el riesgo de ejecución accidental o maliciosa de código situado fuera de las regiones autorizadas, aportando una capa adicional de protección frente a ataques basados en corrupción de punteros o modificación del flujo de ejecución.

6.4.2 PROTECCIÓN FRENTE A ACCESOS FUERA DE LÍMITES

Otro de los mecanismos fundamentales implementados corresponde a la validación de límites de memoria. En una arquitectura convencional basada únicamente en punteros, un error de programación puede provocar accesos a posiciones de memoria situadas fuera de la región asignada a un determinado proceso.

La unidad CHERI Check Unit verifica automáticamente que cualquier dirección utilizada mediante una capability se encuentre dentro del rango definido por los campos `base` y `length`. Durante las pruebas realizadas se utilizaron capabilities con una dirección base de `0x0100` y una longitud de `0x0020`, generando un rango válido comprendido entre las direcciones `0x0100` y `0x011F`.

Las simulaciones muestran que los accesos realizados dentro de esta región son autorizados correctamente, mientras que los accesos dirigidos a posiciones situadas fuera del rango permitido son rechazados por el hardware. Este mecanismo permite prevenir errores de acceso fuera de límites y constituye una de las principales ventajas de CHERI frente a arquitecturas tradicionales.

6.4.3 PROTECCIÓN DE ACCESO A MEMORIA

Las instrucciones `CLoad` y `CStore` permiten realizar accesos protegidos a memoria utilizando capabilities como referencia. Antes de ejecutar la operación solicitada, el sistema comprueba automáticamente la validez de la capability, los límites de acceso y los permisos correspondientes.

Durante las pruebas se generó una capability sin permisos de lectura ni escritura mediante la instrucción `CSetPerm`. Posteriormente se intentaron ejecutar operaciones de carga y almacenamiento utilizando dicha capability. En ambos casos la unidad CHERI Check Unit detectó la ausencia de permisos y bloqueó la operación.

Cuando las mismas instrucciones se ejecutaron utilizando una capability con permisos válidos, las operaciones fueron autorizadas correctamente. Estos resultados demuestran que el sistema es capaz de controlar de forma hardware qué regiones de memoria pueden ser accedidas y qué operaciones pueden realizarse sobre ellas.

6.5 APLICACIÓN EN UN SISTEMA DE TELECOMUNICACIONES

Para ilustrar la utilidad práctica de la extensión CHERI implementada en este proyecto, se plantea un caso de uso basado en un sistema de telecomunicaciones embebido. Este tipo de sistemas son habituales en dispositivos IoT, estaciones de monitorización remota, routers industriales o equipos de adquisición de datos, donde la recepción y procesamiento de información procedente de redes externas constituye una de las principales fuentes de vulnerabilidades de seguridad.

Para el ejemplo se considera una estación meteorológica conectada a una red inalámbrica que recibe periódicamente paquetes de actualización de configuración desde un servidor remoto. El software del sistema dispone de un módulo encargado de recibir los datos de la red y almacenarlos temporalmente en un buffer de memoria para su posterior procesamiento. Además, existe una zona de memoria independiente donde se almacenan parámetros críticos de funcionamiento, como la frecuencia de muestreo de los sensores, los umbrales de alarma o los modos de operación del dispositivo.

En una implementación basada únicamente en RV32I, el módulo de comunicaciones accede a la memoria mediante direcciones convencionales. Si debido a un error de programación o a un paquete malicioso recibido desde la red se intentan escribir más datos de los que admite el buffer reservado, se produciría un desbordamiento de memoria (*buffer overflow*). Como consecuencia, los datos excedentes podrían sobrescribir regiones de memoria adyacentes pertenecientes a la configuración crítica del sistema, alterando su comportamiento o incluso comprometiendo completamente su funcionamiento. La arquitectura RV32I no dispone de mecanismos hardware que permitan detectar este tipo de accesos fuera de rango, por lo que la operación se ejecutaría sin generar ninguna excepción.

La situación cambia con la incorporación de la extensión CHERI. En este caso, el buffer de recepción se asocia a una capability que contiene la dirección base del bloque de memoria, su longitud máxima, los permisos de acceso permitidos y un bit de validez (*tag*). Antes de que el módulo de comunicaciones pueda acceder al buffer, la capability puede restringirse mediante la instrucción `CSetBounds`, limitando explícitamente el rango de direcciones accesibles.

Durante la ejecución de una operación de escritura, la unidad de comprobación de capabilities implementada en este trabajo verifica que la dirección solicitada se encuentra dentro de los límites definidos por la capability y que esta posee los permisos adecuados. Si un paquete recibido intenta provocar una escritura fuera de los límites autorizados, la comprobación de límites falla y la operación es bloqueada antes de llegar a la memoria. De

esta forma, el error queda atrapado en el módulo de comunicaciones y no puede afectar a otras regiones del sistema.

Además, una vez finalizada la fase de configuración del dispositivo, los parámetros críticos pueden protegerse adicionalmente mediante la instrucción `CSetPerm`, eliminando el permiso de escritura de la capability asociada a dicha región de memoria. Como resultado, aunque un atacante consiguiera comprometer alguna parte del software, cualquier intento posterior de modificar la configuración protegida sería rechazado por el hardware.

Este ejemplo demuestra cómo la arquitectura RV32Y desarrollada en el proyecto proporciona mecanismos de protección de memoria inexistentes en RV32I. Mientras que la arquitectura original permite que un error de software pueda propagarse y afectar a cualquier región de memoria accesible, la utilización de capabilities permite aplicar el principio de mínimo privilegio y limitar estrictamente los recursos a los que cada módulo puede acceder [14]. En sistemas de telecomunicaciones, donde la interacción continua con redes externas aumenta la superficie de ataque, las capabilities contribuyen significativamente a mejorar la robustez y la seguridad del sistema frente a errores y vulnerabilidades de memoria.

Capítulo 7. CONCLUSIONES Y TRABAJOS FUTUROS

7.1 CONCLUSIONES

El presente Trabajo Fin de Grado ha abordado la integración de mecanismos de protección de memoria inspirados en la arquitectura CHERI sobre un procesador RISC-V desarrollado en VHDL. El objetivo principal del proyecto consistía en estudiar la viabilidad de incorporar capacidades de control de acceso basadas en capabilities dentro de una arquitectura RISC-V sencilla, evaluando las modificaciones necesarias tanto en el datapath como en la unidad de control.

A lo largo del desarrollo se ha realizado un análisis de la arquitectura CHERI y de los problemas de seguridad de memoria que pretende resolver. A partir de dicho estudio se diseñó una extensión específica para el procesador de partida, incorporando una estructura de capability adaptada a las necesidades del sistema y desarrollando los bloques hardware necesarios para su gestión.

Entre las principales modificaciones realizadas destacan la implementación del banco de registros de capabilities (BancoCap), la memoria específica para almacenamiento de capabilities (CapRAM) y la unidad CHERI Check Unit encargada de validar permisos, límites y estado de validez antes de permitir cualquier operación protegida. Además, se incorporó soporte para un conjunto representativo de instrucciones inspiradas en CHERI, permitiendo manipular capabilities, restringir permisos, limitar regiones de memoria y realizar accesos protegidos.

La validación funcional realizada mediante simulaciones en Vivado ha permitido comprobar el correcto funcionamiento tanto de la arquitectura RV32I original como de la nueva arquitectura RV32Y desarrollada. Los resultados obtenidos muestran que el sistema es capaz de detectar accesos fuera de rango, operaciones realizadas sin permisos suficientes

y el uso de capabilities inválidas, proporcionando un nivel adicional de protección respecto a una implementación RISC-V convencional.

Por tanto, puede concluirse que la incorporación de mecanismos inspirados en CHERI resulta viable incluso en arquitecturas simplificadas, permitiendo mejorar significativamente la robustez frente a errores relacionados con la gestión de memoria mediante modificaciones hardware relativamente acotadas.

7.2 CUMPLIMIENTO DE OBJETIVOS

Los objetivos planteados al inicio del proyecto se han alcanzado de forma satisfactoria en su mayor parte. Se ha realizado el estudio de la arquitectura CHERI, se han implementado las modificaciones necesarias sobre la arquitectura RISC-V de partida y se han desarrollado los mecanismos hardware requeridos para soportar el modelo de seguridad basado en capabilities.

Por otro lado, se han implementado y validado las nuevas instrucciones incorporadas, desarrollándose distintos bancos de pruebas que han permitido verificar el correcto funcionamiento de los componentes diseñados y de la arquitectura completa.

Del mismo modo, el análisis del impacto de las modificaciones introducidas sobre la complejidad hardware de la arquitectura (Objetivo 4.2.5) se ha realizado de forma cualitativa a partir de los bloques, señales y estados adicionales incorporados al diseño. No se ha llevado a cabo una evaluación cuantitativa basada en resultados de síntesis, por lo que aspectos como la utilización de recursos lógicos, la frecuencia máxima de funcionamiento o el consumo energético quedan fuera del alcance del presente trabajo.

No obstante, algunos objetivos inicialmente planteados no han podido abordarse completamente dentro del alcance temporal del proyecto. En particular, la implementación física sobre FPGA y la ejecución de aplicaciones software complejas sobre la arquitectura

resultante quedan fuera del trabajo realizado, constituyendo posibles líneas de desarrollo futuras.

7.3 IMPLEMENTACIÓN DE EXCEPCIONES CHERI

La arquitectura desarrollada durante este trabajo implementa un mecanismo de protección basado en capabilities capaz de detectar accesos no autorizados a memoria y bloquear automáticamente las operaciones que incumplen las restricciones definidas por la capability correspondiente. Sin embargo, cuando se produce una violación de límites, permisos o validez, el sistema simplemente impide la ejecución de la operación solicitada, sin generar ningún tipo de excepción o notificación al software.

Una posible línea de trabajo futuro consiste en incorporar un sistema de excepciones específico para las violaciones CHERI. Este mecanismo permitiría que el procesador identificase la causa concreta del fallo y transfiriese el control a una rutina de tratamiento adecuada. De esta forma, el sistema operativo o el software de supervisión podrían registrar el error, finalizar el proceso responsable o aplicar medidas de recuperación en función del tipo de violación detectada.

La incorporación de excepciones precisas constituye uno de los elementos presentes en las implementaciones completas de CHERI, donde los fallos asociados a capabilities generan información detallada sobre la naturaleza del error producido [9]. Esta capacidad resulta especialmente importante en sistemas complejos, ya que permite gestionar de forma controlada situaciones como accesos fuera de límites, intentos de ejecución sin permiso o utilización de capabilities inválidas.

Desde el punto de vista hardware, esta mejora requeriría ampliar la unidad CHERI Check Unit para generar códigos de error específicos, así como modificar la unidad de control para redirigir la ejecución hacia un vector de excepciones cuando se detectase una violación.

Por tanto, la incorporación de mecanismos de excepción asociados a capabilities representa una evolución natural de la arquitectura propuesta, permitiendo complementar la protección hardware actual con mecanismos de gestión de errores más avanzados y facilitando una futura integración con sistemas operativos de mayor complejidad, como el que se explica en el apartado 7.4.

7.4 INCORPORACIÓN DE UNA UNIDAD DE GESTIÓN DE MEMORIA (MMU)

Una de las principales líneas de evolución de la arquitectura desarrollada consiste en la incorporación de una Unidad de Gestión de Memoria (*Memory Management Unit*, MMU). En la implementación realizada durante este trabajo se ha utilizado direccionamiento físico directo, lo que resulta adecuado para la validación de los mecanismos de protección basados en capabilities, pero limita la posibilidad de ejecutar sistemas operativos complejos y aplicaciones multiproceso.

La MMU es un componente hardware encargado de traducir direcciones virtuales a direcciones físicas y de aplicar mecanismos de protección entre procesos. Gracias a ella, cada aplicación puede disponer de su propio espacio de direcciones aislado, evitando que un proceso acceda accidentalmente a la memoria perteneciente a otro. Este mecanismo constituye la base del funcionamiento de la mayoría de los sistemas operativos modernos, incluyendo Linux, Windows o los sistemas operativos utilizados en dispositivos móviles [15].

La incorporación de una MMU permitiría ampliar significativamente las capacidades de la arquitectura desarrollada, facilitando la ejecución de sistemas operativos más avanzados y proporcionando un primer nivel de aislamiento entre procesos. Sin embargo, la protección proporcionada por una MMU resulta insuficiente para evitar determinados errores relacionados con el uso incorrecto de punteros dentro de un mismo espacio de

direcciones. Es precisamente en este punto donde las arquitecturas basadas en capabilities aportan una capa adicional de seguridad [9].

De hecho, una de las principales motivaciones del proyecto CHERI consiste en complementar los mecanismos tradicionales proporcionados por las MMU mediante capacidades hardware capaces de controlar los límites, permisos y validez de cada referencia de memoria. Mientras que la MMU protege el sistema frente a accesos entre distintos procesos, CHERI permite proteger regiones concretas de memoria dentro de un mismo proceso, reduciendo el impacto de vulnerabilidades como los desbordamientos de búfer o la corrupción de punteros [12].

Por este motivo, la integración conjunta de una MMU y de los mecanismos de protección desarrollados en este trabajo constituye una evolución natural de la arquitectura propuesta. Esta combinación permitiría aproximar el sistema a las implementaciones modernas de CHERI-RISC-V y a plataformas de investigación como Morello, donde la protección basada en capabilities se complementa con los mecanismos tradicionales de memoria virtual [16].

Capítulo 8. BIBLIOGRAFÍA

[1] Waterman, A.; Asanović, K. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA. Document Version 20191213, RISC-V International, Diciembre 2019. Disponible en: https://riscv.org/technical/specifications/

[2] Patterson, D. A.; Waterman, A. The RISC-V Reader: An Open Architecture Atlas. Strawberry Canyon LLC, Berkeley, California, 2017.

[3] Ashenden, P. J. The Designer's Guide to VHDL. 3rd Edition. Morgan Kaufmann Publishers, Burlington, Massachusetts, 2008.

[4] AMD-Xilinx. Vivado Design Suite User Guide. AMD, 2024.

[5] Szekeres, L.; Payer, M.; Wei, T.; Song, D. SoK: Eternal War in Memory. Proceedings of the IEEE Symposium on Security and Privacy, 2013.

[6] MITRE Corporation. Common Weakness Enumeration (CWE): Buffer Errors and Use-After-Free Vulnerabilities. Disponible en: https://cwe.mitre.org

[7] Aleph One. Smashing The Stack For Fun And Profit. Phrack Magazine, Vol. 7, No. 49, 1996.

[8] Google Project Zero. 0days In-The-Wild 2023: A Year in Review. 2024.

[9] Watson, R. N. M.; Woodruff, J.; Neumann, P. G.; Moore, S. W.; Anderson, J.; Chisnall, D.; Davis, B.; Laurie, B.; Murdoch, S. J.; Roe, M.; Sewell, P.; Son, S. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. IEEE Symposium on Security and Privacy, 2015.

[10] Watson, R. N. M.; Neumann, P. G.; Woodruff, J.; Roe, M.; Almatary, H.; Anderson, J.; Baldwin, J.; Barnes, G.; Chisnall, D.; Clarke, J.; Davis, B.; Eisen, L.; Filardo, N. W.; Fuchs, F. A.; Grisenthwaite, R.; Joannou, A.; Laurie, B.; Markettos, A. T.; Moore, S. W.; Murdoch, S. J.; Nienhuis, K.; Norton, R.; Richardson, A.; Rugg, P.; Sewell, P.; Son, S.; Xia, H. “Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)”. Technical Report UCAM-CL-TR-987, University of Cambridge Computer Laboratory, Septiembre 2023. Disponible en: https://www.ietfng.org/nwf/_downloads/2e59e293c1afa58f0772fda209e6b54c/2023-cheri isav9.pdf.

[11] Levy, H. M. Capability-Based Computer Systems. Digital Press, Bedford, Massachusetts, 1984.

[12] Watson, R. N. M. et al. Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the CHERI Design. University of Cambridge Computer Laboratory, Technical Report UCAM-CL-TR-987, 2024.

[13] Muñoz Frías, J. D. "Apuntes de la asignatura Sistemas Digitales II". Enero, 2023.

[14] De, A.; Ghosh, S. HeapSafe: Securing Unprotected Heaps in RISC-V. arXiv:2105.08712, 2021.

[15] Patterson, D. A.; Hennessy, J. L. Computer Organization and Design: The Hardware/Software Interface. RISC-V Edition. Morgan Kaufmann, 2020.

[16] Woodruff, J.; Joannou, A.; Roe, M.; Son, S.; Watson, R. N. M. CHERI Concentrate: Practical Compressed Capabilities. IEEE Transactions on Computers, 2019.

[17] University of Cambridge, CTSRD. *CHERI-RISC-V*. s. f. Disponible en: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-risc-v.html

[18] Microsoft Research. *CHERIoT: Rethinking security for low-cost embedded systems*. 2023. Disponible en : https://www.microsoft.com/en-us/research/publication/cheriot-rethinking-security-for-low-cost-embedded-systems/

[19] Cloudflare, *Principle of Least Privilege (PoLP)*. Cloudflare Learning Center. Disponible en: https://www.cloudflare.com/es-es/learning/access-management/principle-of-least-privilege/

ANEXO I: ALINEACIÓN DEL PROYECTO CON LOS ODS

ANEXO I.1 - EDUCACIÓN DE CALIDAD. ODS 4

Este proyecto contribuye a abordar problemas de tecnologías de arquitectura de microprocesadores dentro de un marco académico abierto, ya que la arquitectura RISC-V es de código abierto y libre de royalties. Esto permite que alumnos e instituciones con recursos limitados puedan estudiar y experimentar con soluciones a los problemas de la arquitectura sin un coste añadido. Asimismo, el proyecto aumenta el conocimiento sobre técnicas emergentes de seguridad de memoria fomentando competencias clave en ciberseguridad y sistemas digitales.

ANEXO I.2 - TRABAJO DECENTE Y CRECIMIENTO ECONÓMICO. ODS 8

La implementación de las instrucciones CHERI en la arquitectura del RISC-V incrementa la fiabilidad y robustez del software embebido, reduciendo los costes de los posibles fallos de seguridad. Estas tecnologías se demandan cada vez más en los sectores industriales, como IoT, por lo que este trabajo contribuye al crecimiento económico a través de la aplicación de las instrucciones y la divulgación de estos. Con ello los profesionales serán capaces de desarrollar sistemas más seguros y eficientes.

ANEXO I.3 - INDUSTRIA, INNOVACIÓN E INFRAESTRUCTURA. ODS 9

Este proyecto se alinea de forma completamente directa con el ODS 9. La integración de las instrucciones CHERI en un softcore RISC-V contribuye a la industria emergente e impulsa la investigación en infraestructuras seguras y abiertas, ya que CHERI representa la vanguardia en tecnología segura.

ANEXO I .4 - REDUCCIÓN DE LAS DESIGUALDADES. ODS 10

El uso de arquitecturas abiertas como RISC-V reduce la brecha entre instituciones de altos recursos y aquellas con unas capacidades más limitadas, permitiendo que laboratorios modestos puedan replicar el trabajo.

ANEXO I .5 - ENERGÍA ASEQUIBLE Y NO CONTAMINANTE. ODS 7

Las arquitecturas RISC tienen una mayor eficiencia energética que las CISC. Con la implementación de la extensión CHERI su uso sería más asequible lo que reduce el uso de arquitecturas CISC y en consecuencia ayuda a limitar el uso de energía.

ANEXO II: FICHEROS NUEVOS DEL RV32Y

RV32Y

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.CHERI_pck.all;

use work.ICAI_RISC_V_pkg.all;


entity RV32Y is
   port (
      -- Ancillary signals
      clk     : in std_logic;
      reset : in std_logic;

   );
end entity RV32Y;


architecture rtl of RV32Y is
   --Señales de la unidad de comprobacion
   signal req_load        : std_logic;
   signal req_store       : std_logic;
   signal req_exec        : std_logic;
   signal req_load_cap    : std_logic;
   signal req_store_cap   : std_logic;

   signal tag_ok          : std_logic;
   signal permit_load_ok  : std_logic;
   signal permit_store_ok : std_logic;
   signal permit_exec_ok  : std_logic;
   signal permit_lcap_ok  : std_logic;
   signal permit_scap_ok  : std_logic;
   signal bounds_ok       : std_logic;
   signal cheri_check_ok  : std_logic;
   -- Señales de CLoad y CStore
   signal cheri_mem_addr : std_logic_vector(31 downto 0);
   signal ram_addr_in    : std_logic_vector(31 downto 0);
   -- Señal para CJump
   signal cheri_jump_addr : std_logic_vector(31 downto 0);
   --Señal para CSetBounds
   signal cheri_bounds_cap : capability_t;
   -- Señal para CSetPerm
   signal cheri_perm_cap : capability_t;
   --Señales para CIncOffSet
```

```
signal cheri_inc_cap : capability_t; --capability increased
signal is_cincoffset : std_logic;
-- Señales para CGetTag
signal cheri_tag_out : std_logic_vector(31 downto 0);
signal is_cgettag    : std_logic;
signal capA, capB : capability_t;
signal capDin     : capability_t;
signal cap_enW    : std_logic;
signal capAddrA   : std_logic_vector(4 downto 0);
signal capAddrB   : std_logic_vector(4 downto 0);
signal capAddrW   : std_logic_vector(4 downto 0);

--Señales para la memoria de cap
signal cap_mem_dout : capability_t;
signal cap_mem_we   : std_logic;

signal alu_in_a   : std_logic_vector(31 downto 0);
signal alu_in_b   : std_logic_vector(31 downto 0);
signal alu_op     : std_logic_vector(3 downto 0);
signal alu_out    : std_logic_vector(31 downto 0);
signal alur_out   : std_logic_vector(31 downto 0);
signal D_in       : std_logic_vector(31 downto 0);
signal d_ram_alu  : std_logic_vector(31 downto 0);
signal en_banco   : std_logic;
signal ge         : std_logic;
signal inm        : std_logic_vector(31 downto 0);
signal ir_in      : std_logic_vector(31 downto 0);
signal ir_out     : std_logic_vector(31 downto 0);
signal l_u        : std_logic;
signal lt         : std_logic;
signal mask_b0    : std_logic;
signal en_pc      : std_logic;
signal pc_ir      : std_logic_vector(31 downto 0);
signal pc_in      : std_logic_vector(31 downto 0);
signal pc_out     : std_logic_vector(31 downto 0);
signal ram_dout   : std_logic_vector(31 downto 0);
signal reg_a      : std_logic_vector(31 downto 0);
signal reg_b      : std_logic_vector(31 downto 0);
signal shmt       : std_logic_vector(4 downto 0);
signal tipo_acc   : std_logic_vector(1 downto 0);
signal tipo_inst  : std_logic_vector(2 downto 0);
signal we_ram     : std_logic;
signal z          : std_logic;
signal m_alu_op   : std_logic_vector(3 downto 0);
signal en_ir      : std_logic;
signal m_alu_a    : std_logic_vector(1 downto 0);
signal m_alu_b    : std_logic_vector(1 downto 0);
signal m_banco    : std_logic_vector(1 downto 0);
signal m_pc       : std_logic_vector(1 downto 0);
signal m_ram      : std_logic;
signal wr_pc      : std_logic;
signal wr_pc_cond : std_logic;
signal m_shamt    : std_logic;
```

```
--Señal de excepciones cheri (1 cuando hay una excepcion, es decir, un error)
signal cheri_fault : std_logic;


component ALU
   generic (
      n : integer := 32
   );
   port (
      a       : in  std_logic_vector(n-1 downto 0);
      b       : in  std_logic_vector(n-1 downto 0);
      alu_op  : in  std_logic_vector(3 downto 0);
      shamt   : in  std_logic_vector(4 downto 0);
      alu_out : out std_logic_vector(n-1 downto 0);
      z       : out std_logic;
      lt      : out std_logic;
      ge      : out std_logic
   );
end component ALU;


component registers_bank
   port (
      -- Ancillary signals
      clk     : in std_logic;
      reset : in std_logic;
      -- Write signals
      En    : in std_logic;
      AddrW : in std_logic_vector(4 downto 0);
      D_in  : in std_logic_vector(31 downto 0);
      -- Register A read signals
      AddrA : in  std_logic_vector(4 downto 0);
      RegA  : out std_logic_vector(31 downto 0);
      -- Register B read signals
      AddrB : in  std_logic_vector(4 downto 0);
      RegB  : out std_logic_vector(31 downto 0)
   );
end component registers_bank;

component control_RV32Y
   port (
      -- Ancillary signals
      clk     : in std_logic;
      reset : in std_logic;
      -- Instruction signal
      ir_out : in std_logic_vector(31 downto 0);
      -- Señales de control
      cheri_check_ok : in std_logic;
      m_alu_op   : out std_logic_vector(3 downto 0);
      en_banco   : out std_logic;
      en_ir      : out std_logic;
      l_u        : out std_logic;
      m_alu_a    : out std_logic_vector(1 downto 0);
```

```
        m_alu_b    : out std_logic_vector(1 downto 0);
        m_banco    : out std_logic_vector(1 downto 0);
        m_pc       : out std_logic_vector(1 downto 0);
        m_ram      : out std_logic;
        mask_b0    : out std_logic;
        tipo_acc   : out std_logic_vector(1 downto 0);
        tipo_inst  : out std_logic_vector(2 downto 0);
        we_ram     : out std_logic;
        wr_pc      : out std_logic;
        wr_pc_cond : out std_logic;
        m_shamt    : out std_logic;
        cap_enW : out std_logic;
        cap_mem_we : out std_logic
    );
  end component control_RV32Y;

  component inm_gen
    port(
        inst      : in  std_logic_vector(31 downto 0);
        tipo_inst : in  std_logic_vector(2 downto 0);
        mask_b0   : in  std_logic;
        inm       : out std_logic_vector(31 downto 0)
    );
  end component inm_gen;

  component ROM_RV32Y
    port(
        clk   : in  std_logic;
        en_pc : in  std_logic;
        addr  : in  std_logic_vector(31 downto 0);
        data  : out std_logic_vector(31 downto 0)
    );
  end component ROM_RV32Y;

  component ram
    port(
        clk : in std_logic;
        addr      : in  std_logic_vector(31 downto 0);
        din       : in  std_logic_vector(31 downto 0);
        dout      : out std_logic_vector(31 downto 0);
        tipo_acc  : in  std_logic_vector(1 downto 0);
        l_u       : in  std_logic;
        we_ram : in  std_logic
    );
  end component ram;
  -- ------------------------------------------------------------------------

begin

  u_BancoCap : entity work.BancoCap
       port map(
           Din     => capDin,
           AddrA   => capAddrA,
```

```
            AddrB   => capAddrB,
            AddrW   => capAddrW,
            enW     => cap_enW,
            clk     => clk,
            reset => reset,
            capA    => capA,
            capB    => capB
);


u_cheri_check : entity work.cheri_check_unit
   port map(
      cap_in          => capA,
      access_addr     => capA.addr,
      req_load        => req_load,
      req_store       => req_store,
      req_exec        => req_exec,
      req_load_cap    => req_load_cap,
      req_store_cap   => req_store_cap,
      tag_ok          => tag_ok,
      permit_load_ok  => permit_load_ok,
      permit_store_ok => permit_store_ok,
      permit_exec_ok  => permit_exec_ok,
      permit_lcap_ok  => permit_lcap_ok,
      permit_scap_ok  => permit_scap_ok,
      bounds_ok       => bounds_ok,
      check_ok        => cheri_check_ok
   );

-- Requiere permisos de carga cuando la instruccion es CLoad o CLoadCap
req_load <= '1' when ir_out(6 downto 2) = "11111" and
                  (ir_out(14 downto 12) = "000" or ir_out(14 downto 12) =
"010")
            else '0';
--Requiere permisos de almacenamiento cuando la instruccion es CStore o CStoreCap
req_store <= '1' when ir_out(6 downto 2) = "11111" and
                  (ir_out(14 downto 12) = "001" or ir_out(14 downto 12) =
"011")
                  else '0';
--Requerir permisos de carag y almacenamiento de capabilities
req_load_cap <= '1' when ir_out(6 downto 2) = "11111" and ir_out(14 downto 12) =
"010" else
                '0';

req_store_cap <= '1' when ir_out(6 downto 2) = "11111" and ir_out(14 downto 12) =
"011" else
                 '0';

--Requiere ejecucion para CJump y CJumpLink
req_exec <= '1' when ir_out(6 downto 2) = "10001" and
                   (ir_out(14 downto 12) = "100" or ir_out(14 downto 12) =
"101")
            else '0';
```

```
cheri_fault <= '1' when
   (req_load='1' or
    req_store='1' or
    req_exec='1' or
    req_load_cap='1' or
    req_store_cap='1')
   and cheri_check_ok='0'
else '0';


capAddrA <= ir_out(19 downto 15);
capAddrB <= ir_out(24 downto 20);
capAddrW <= ir_out(11 downto 7);


   en_pc <= wr_pc or (wr_pc_cond and z)     when ir_out(14 downto 12)="000" else
         wr_pc or (wr_pc_cond and not z) when ir_out(14 downto 12)="001" else
         wr_pc or (wr_pc_cond and lt)    when ir_out(14 downto 12)="100" else
         wr_pc or (wr_pc_cond and ge)    when ir_out(14 downto 12)="101" else
         wr_pc;

   pc_in <= alu_out when m_pc="00" else
      alur_out      when m_pc="01" else
      cheri_jump_addr when m_pc="10" else
      (others => '0');

   process(clk, reset)
   begin
      if reset='1' then
         pc_out <= (others => '0');
         pc_ir  <= (others => '0');

      elsif (clk'event and clk='1') then
         if en_pc='1' then
            pc_out <= pc_in;
         end if;

         if en_ir='1' then
            pc_ir <= pc_out;
         end if;

      end if;
   end process;

   i_rom : ROM_RV32Y
      port map(
         clk   => clk,
         en_pc => en_ir,
         addr  => pc_out,
         data  => ir_in
      );
```

```
   process(clk, reset)
   begin
      if reset='1' then
         ir_out <= (others => '0');
      elsif (clk'event and clk='1') then
         if en_ir='1' then
            ir_out <= ir_in;
         end if;
      end if;
   end process;

   i_inmgen : inm_gen
      port map(
         inst      => ir_out,
         tipo_inst => tipo_inst,
         mask_b0   => mask_b0,
         inm       => inm
      );

   D_in <= d_ram_alu     when m_banco = "00" else
        pc_out         when m_banco = "01" else
        inm            when m_banco = "10" else
        cheri_get_out;

    --Así añado todas las instrucciones de inspección a la vez

    cheri_get_out <= (31 downto 1  => '0') & capA.tag                when
ir_out(31 downto 25) = "0000000" else
                  x"0000" & capA.base                              when ir_out(31
downto 25) = "0000001" else
                  x"0000" & capA.length                            when ir_out(31
downto 25) = "0000010" else
                  x"000000" & capA.perms                           when ir_out(31
downto 25) = "0000011" else
                  x"0000" & capA.addr                              when ir_out(31
downto 25) = "0000100" else
                  (others => '0');
   i_registers : registers_bank
      port map(
         clk     => clk,
         reset => reset,
         En      => en_banco,
         AddrW   => ir_out(11 downto 7),
         D_in    => D_in,
         AddrA   => ir_out(19 downto 15),
         RegA    => reg_a,
         AddrB   => ir_out(24 downto 20),
         RegB    => reg_b
      );

     --CIncOffset
     cheri_tag_out <= (31 downto 1 => '0') & capA.tag;
     cheri_inc_cap.base     <= capA.base;
     cheri_inc_cap.length   <= capA.length;
     cheri_inc_cap.perms    <= capA.perms;
```

```
     cheri_inc_cap.reserved <= capA.reserved;
     cheri_inc_cap.tag      <= capA.tag;
     cheri_inc_cap.addr     <= std_logic_vector(unsigned(capA.addr) +
unsigned(reg_b(15 downto 0)));

     --CSetPerm
     cheri_perm_cap.base     <= capA.base;
     cheri_perm_cap.length   <= capA.length;
     cheri_perm_cap.addr     <= capA.addr;
     cheri_perm_cap.reserved <= capA.reserved;
     cheri_perm_cap.tag      <= capA.tag;
     cheri_perm_cap.perms    <= capA.perms and reg_b(7 downto 0);

     --CsetBounds
     cheri_bounds_cap.base     <= capA.base;
     cheri_bounds_cap.addr     <= capA.addr;
     cheri_bounds_cap.perms    <= capA.perms;
     cheri_bounds_cap.reserved <= capA.reserved;
     cheri_bounds_cap.tag      <= capA.tag;
     cheri_bounds_cap.length   <= reg_b(15 downto 0);

     capDin <= cheri_inc_cap    when ir_out(6 downto 2) = "10001" and ir_out(14
downto 12) = "001" else
          cheri_perm_cap   when ir_out(6 downto 2) = "10001" and ir_out(14 downto
12) = "010" else
          cheri_bounds_cap when ir_out(6 downto 2) = "10001" and ir_out(14 downto
12) = "011" else
          cap_mem_dout     when ir_out(6 downto 2) = "11111" and ir_out(14 downto
12) = "010" else
          NULL_CAPABILITY;

     -- CJump
     cheri_jump_addr <= x"0000" & capA.addr;

   alu_in_a <= reg_a  when m_alu_a="00" else
            pc_out when m_alu_a="01" else
            pc_ir;
   alu_in_b <= reg_b when m_alu_b="00" else
      x"00000004"    when m_alu_b="01" else
      inm;
   shmt <= reg_b(4 downto 0) when m_shamt='0' else
      ir_out(24 downto 20);

   i_alu : alu
      port map(
         a       => alu_in_a,
         b       => alu_in_b,
         alu_op  => alu_op,
         shamt   => shmt,
         alu_out => alu_out,
         z       => z,
         lt      => lt,
         ge      => ge
```

```
   );

p_alu_out_register : process(clk, reset)
begin
   if reset='1' then
      alur_out <= (others => '0');
   elsif (clk'event and clk='1') then
      alur_out <= alu_out;
   end if;
end process p_alu_out_register;

--CStore y CLoad
cheri_mem_addr <= x"0000" & capA.addr;
ram_addr_in <= cheri_mem_addr when ir_out(6 downto 2) = "11111" else
            alur_out;

 u_cap_ram : entity work.cap_ram
    port map(
       clk => clk,
       addr  => cheri_mem_addr,
       d_in  => capB,
       we    => cap_mem_we,
       d_out => cap_mem_dout
    );
i_ram : ram
   port map(
      clk   => clk,
      addr      => ram_addr_in,
      din       => reg_b,
      dout      => ram_dout,
      tipo_acc  => tipo_acc,
      l_u       => l_u,
      we_ram => we_ram
   );


d_ram_alu <= ram_dout when m_ram='1' else
          alur_out;
-- ------------------------------------------------------------------------

alu_op <= m_alu_op;
-- ------------------------------------------------------------------------
-- Control
-- ------------------------------------------------------------------------
i_control : control_RV32Y
   port map(
      clk        => clk,
      reset    => reset,
      ir_out     => ir_out,
      m_alu_op   => m_alu_op,
      cheri_check_ok => cheri_check_ok,
      en_banco   => en_banco,
      en_ir      => en_ir,
```

```
        l_u         => l_u,
        m_alu_a     => m_alu_a,
        m_alu_b     => m_alu_b,
        m_banco     => m_banco,
        m_pc        => m_pc,
        m_ram       => m_ram,
        mask_b0     => mask_b0,
        tipo_acc    => tipo_acc,
        tipo_inst   => tipo_inst,
        we_ram      => we_ram,
        wr_pc       => wr_pc,
        wr_pc_cond => wr_pc_cond,
        m_shamt     => m_shamt,
        cap_enW => cap_enW,
        cap_mem_we => cap_mem_we
    );
end architecture rtl;
```

CHERI_check_unit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.CHERI_pck.all;

entity cheri_check_unit is
    port(
      cap_in          : in  capability_t;
      --Comprobación de bounds
      access_addr     : in  std_logic_vector(15 downto 0);
      req_load        : in  std_logic;
      req_store       : in  std_logic;
      req_exec        : in  std_logic;
      req_load_cap    : in  std_logic;
      req_store_cap   : in  std_logic;
      tag_ok          : out std_logic;
      permit_load_ok  : out std_logic;
      permit_store_ok : out std_logic;
      permit_exec_ok  : out std_logic;
      permit_lcap_ok  : out std_logic;
      permit_scap_ok  : out std_logic;
      bounds_ok       : out std_logic;
      check_ok        : out std_logic
      );
end cheri_check_unit;

architecture rtl of cheri_check_unit is
   signal op_ok : std_logic;
   signal s_permit_load_ok  : std_logic;
```

```
   signal s_permit_store_ok : std_logic;
   signal s_permit_exec_ok  : std_logic;
   signal s_permit_lcap_ok  : std_logic;
   signal s_permit_scap_ok  : std_logic;
   signal s_tag_ok          : std_logic;
   signal s_bounds_ok       : std_logic;
   --Señales para comprobación de limites/bounds
   signal cap_base_u   : unsigned(15 downto 0);
   signal cap_length_u : unsigned(15 downto 0);
   signal access_u     : unsigned(15 downto 0);
   signal cap_end_u    : unsigned(16 downto 0);
begin

   s_tag_ok          <= cap_in.tag;
   s_permit_load_ok  <= cap_in.perms(0);
   s_permit_store_ok <= cap_in.perms(1);
   s_permit_exec_ok  <= cap_in.perms(2);
   s_permit_lcap_ok  <= cap_in.perms(3);
   s_permit_scap_ok  <= cap_in.perms(4);
   cap_base_u   <= unsigned(cap_in.base);
   cap_length_u <= unsigned(cap_in.length);
   access_u     <= unsigned(access_addr);

   -- Comprobación de limites
   cap_end_u <= ('0' & cap_base_u) + ('0' & cap_length_u);

   s_bounds_ok <= '1' when
      ('0' & access_u) >= ('0' & cap_base_u) and
      ('0' & access_u) <  cap_end_u
   else
      '0';
   -- Comprobación de permisos
   op_ok <= ((not req_load)      or s_permit_load_ok)  and
            ((not req_store)     or s_permit_store_ok) and
            ((not req_exec)      or s_permit_exec_ok)  and
            ((not req_load_cap)  or s_permit_lcap_ok)  and
            ((not req_store_cap) or s_permit_scap_ok);

   check_ok <= s_tag_ok and s_bounds_ok and op_ok;

   tag_ok          <= s_tag_ok;
   permit_load_ok  <= s_permit_load_ok;
   permit_store_ok <= s_permit_store_ok;
   permit_exec_ok  <= s_permit_exec_ok;
   permit_lcap_ok  <= s_permit_lcap_ok;
   permit_scap_ok  <= s_permit_scap_ok;
   bounds_ok       <= s_bounds_ok;

end architecture rtl;
```

BancoCap

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.CHERI_pck.all;

entity BancoCap is
port(
    Din                  : in  capability_t;
    AddrA, AddrB, AddrW  : in  std_logic_vector(4 downto 0);
    enW, clk, reset      : in  std_logic;
    capA, capB           : out capability_t
);
end BancoCap;

architecture behavioural of BancoCap is

    signal br_cap : cap_regfile_t;

begin

    Escritura : process(clk, reset)
    begin
        if reset = '1' then
            br_cap <= (others => NULL_CAPABILITY);
             br_cap(1).base     <= x"0100";
             br_cap(1).length   <= x"0020";
             br_cap(1).addr     <= x"0100";
             br_cap(1).perms    <= x"FF";
             br_cap(1).reserved <= (others => '0');
             br_cap(1).tag      <= '1';
        elsif (clk'event and clk = '1') then
            if (enW = '1') then
                br_cap(to_integer(unsigned(AddrW))) <= Din;
            end if;
        end if;
    end process;

    Lectura : process(AddrA, AddrB, br_cap)
```

```
    begin
        capA <= br_cap(to_integer(unsigned(AddrA)));
        capB <= br_cap(to_integer(unsigned(AddrB)));
    end process;

end behavioural;
```

CapRAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.CHERI_pck.all;

entity cap_ram is
   port(
      clk   : in  std_logic;
      addr  : in  std_logic_vector(31 downto 0);
      d_in  : in  capability_t;
      we    : in  std_logic;
      d_out : out capability_t
   );
end cap_ram;

architecture rtl of cap_ram is

   type cap_memory_t is array (1023 downto 0) of capability_t;
   signal memory : cap_memory_t := (others => NULL_CAPABILITY);

begin

   p_write : process(clk)
    begin
       if rising_edge(clk) then
          if we = '1' then
             memory(to_integer(unsigned(addr(11 downto 2)))) <= d_in;
          end if;
       end if;
    end process p_write;

   d_out <= memory(to_integer(unsigned(addr(11 downto 2))));

end rtl;
```

Control_RV32Y

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.conv_std_logic_vector;

library ICAI_RISC_V;
use work.ICAI_RISC_V_pkg.all;


entity control_RV32Y is
   port (
      -- Ancillary signals
      clk     : in std_logic;
      reset : in std_logic;
      -- Instruction signal
      ir_out : in std_logic_vector(31 downto 0);
      -- Control signals
      cheri_check_ok : in std_logic;
      m_alu_op    : out std_logic_vector(3 downto 0);
      en_banco    : out std_logic;
      en_ir       : out std_logic;
      l_u         : out std_logic;
      m_alu_a     : out std_logic_vector(1 downto 0);
      m_alu_b     : out std_logic_vector(1 downto 0);
      m_banco     : out std_logic_vector(1 downto 0);
      m_pc        : out std_logic_vector(1 downto 0);
      m_ram       : out std_logic;
      mask_b0     : out std_logic;
      tipo_acc    : out std_logic_vector(1 downto 0);
      tipo_inst   : out std_logic_vector(2 downto 0);
      we_ram      : out std_logic;
      wr_pc       : out std_logic;
      wr_pc_cond  : out std_logic;
      m_shamt     : out std_logic;
      cap_enW : out std_logic;
      cap_mem_we : out std_logic
   );
end entity control_RV32Y;


architecture fsm of control_RV32Y is
   -- ------------------------------------------------------------------------
   -- Internal signals declaration
   -- ------------------------------------------------------------------------
   signal opcode : std_logic_vector(4 downto 0); -- Instruction OP Code

   type state_type is (
         s_reset,
         s_fetch,
         s_fetch_wait,
         s_decod,
```

```
         s_Inm3,
         s_Jal3,
         s_Jalr3,
         s_lui3,
         s_lwsw3,
         s_aupic3,
         s_Arit3,
         s_SalCond,
         s_lw4,
         s_sw4,
         s_Arit4,
         s_lw5,
         s_CIncOffset3,
         s_CGetTag3,
         s_CSetPerm3,
         s_CSetBounds3,
         s_CJump3,
         s_CJumpLink3,
         s_CLoad3,
         s_CLoad4,
         s_CStore3,
         s_CLoadCap3,
         s_CLoadCap4,
         s_CStoreCap3
      );
   attribute state_vector        : string;
   attribute state_vector of fsm : architecture is "current_state";


   signal current_state : state_type;
   signal next_state    : state_type;


   attribute syn_preserve                   : BOOLEAN;
   attribute syn_preserve of current_state : signal is TRUE;

begin

   opcode <= ir_out(6 downto 2);

   p_clocked : process(clk, reset)
   begin
      if (reset = '1') then
         current_state <= s_reset;
      elsif (clk'event and clk = '1') then
         current_state <= next_state;
      end if;
   end process p_clocked;

   p_nextstate : process (current_state, opcode, ir_out)
   begin
      case current_state is
         when s_reset =>
```

```
            next_state <= s_fetch;

         when s_fetch =>
            next_state <= s_fetch_wait;

         when s_fetch_wait =>
            next_state <= s_decod;

         when s_decod =>
            if opcode="01101" then
               next_state <= s_lui3;
            elsif opcode="00000" or opcode="01000" then
               next_state <= s_lwsw3;
            elsif opcode="00101" then
               next_state <= s_aupic3;
            elsif opcode="01100" then
               next_state <= s_Arit3;
            elsif opcode="11000" then
               next_state <= s_SalCond;
            elsif opcode="00100" then
               next_state <= s_Inm3;
            elsif opcode="11011" then
               next_state <= s_Jal3;
            elsif opcode="11001" then
               next_state <= s_Jalr3;
            elsif opcode = "10001" and ir_out(14 downto 12) = "000" then
               next_state <= s_CGetTag3;
            elsif opcode = "10001" and ir_out(14 downto 12) = "001" then
               next_state <= s_CIncOffset3;
            elsif opcode = "10001" and ir_out(14 downto 12) = "010" then
               next_state <= s_CSetPerm3;
            elsif opcode = "10001" and ir_out(14 downto 12) = "011" then
               next_state <= s_CSetBounds3;
            elsif opcode = "10001" and ir_out(14 downto 12) = "100" then
               next_state <= s_CJump3;
            elsif opcode = "10001" and ir_out(14 downto 12) = "101" then
               next_state <= s_CJumpLink3;
            elsif opcode = "11111" and ir_out(14 downto 12) = "000" then
               next_state <= s_CLoad3;
            elsif opcode = "11111" and ir_out(14 downto 12) = "001" then
               next_state <= s_CStore3;
            elsif opcode = "11111" and ir_out(14 downto 12) = "010" then
               next_state <= s_CLoadCap3;
            elsif opcode = "11111" and ir_out(14 downto 12) = "011" then
               next_state <= s_CStoreCap3;
            else
               next_state <= s_fetch;
            end if;

         when s_Inm3 =>
            next_state <= s_Arit4;

         when s_Jal3 =>
```

```
            next_state <= s_Fetch;

         when s_Jalr3 =>
            next_state <= s_Fetch;

         when s_lui3 =>
            next_state <= s_fetch;

         when s_lwsw3 =>
            if opcode="00000" then
               next_state <= s_lw4;
            elsif opcode="01000" then
               next_state <= s_sw4;
            else
               next_state <= s_fetch;
            end if;

         when s_aupic3 =>
            next_state <= s_Arit4;

         when s_Arit3 =>
            next_state <= s_Arit4;

         when s_SalCond =>
            next_state <= s_fetch;

         when s_lw4 =>
            next_state <= s_lw5;

         when s_sw4 =>
            next_state <= s_fetch;

         when s_Arit4 =>
            next_state <= s_fetch;

         when s_lw5 =>
            next_state <= s_fetch;

         when s_CGetTag3 =>
            next_state <= s_fetch;

         when s_CIncOffset3 =>
            next_state <= s_fetch;

         when s_CSetPerm3 =>
            next_state <= s_fetch;

         when s_CSetBounds3 =>
            next_state <= s_fetch;

         when s_CJump3 =>
            next_state <= s_fetch;
```

```
        when s_CJumpLink3 =>
           next_state <= s_fetch;

        when s_CLoad3 =>
           next_state <= s_CLoad4;

        when s_CLoad4 =>
           next_state <= s_fetch;

        when s_CStore3 =>
           next_state <= s_fetch;

        when s_CLoadCap3 =>
           next_state <= s_CLoadCap4;

        when s_CLoadCap4 =>
           next_state <= s_fetch;

        when s_CStoreCap3 =>
           next_state <= s_fetch;

        when others =>
           next_state <= s_reset;
     end case;
  end process p_nextstate;

  p_output : process (current_state, ir_out, cheri_check_ok, opcode)
  begin
     m_alu_op   <= (others => '0');
     en_banco   <= '0';
     en_ir      <= '0';
     l_u        <= '0';
     m_alu_a    <= (others => '0');
     m_alu_b    <= (others => '0');
     m_banco    <= (others => '0');
     m_pc       <= (others => '0');
     m_ram      <= '0';
     tipo_acc   <= (others => '0');
     tipo_inst  <= (others => '0');
     we_ram     <= '0';
     wr_pc      <= '0';
     wr_pc_cond <= '0';
     m_shamt    <= '0';
     mask_b0    <= '0';
     cap_enW <= '0';
     cap_mem_we <= '0';

     case current_state is
        when s_reset =>
           m_pc  <= conv_std_logic_vector(2, m_pc'length);
           wr_pc <= '1';

        when s_fetch =>
```

```
            m_alu_a  <= conv_std_logic_vector(1, m_alu_a'length);
            m_alu_b  <= conv_std_logic_vector(1, m_alu_b'length);
            m_alu_op <= conv_std_logic_vector(0, m_alu_op'length);
            m_pc     <= conv_std_logic_vector(0, m_pc'length);
            wr_pc    <= '1';

         when s_fetch_wait =>
            en_ir    <= '1';

         when s_decod =>
            tipo_inst <= TYPE_B;
            m_alu_a   <= conv_std_logic_vector(2, m_alu_a'length);
            m_alu_b   <= conv_std_logic_vector(2, m_alu_b'length);
            m_alu_op  <= conv_std_logic_vector(0, m_alu_op'length);

         when s_Inm3 =>
            tipo_inst <= TYPE_I;
            if ir_out(14 downto 12) = "101" then
               m_alu_op <= ir_out(30) & ir_out(14 downto 12);
            else
               m_alu_op <= '0' & ir_out(14 downto 12);
            end if;
            m_alu_a <= conv_std_logic_vector(0, m_alu_a'length);
            m_alu_b <= conv_std_logic_vector(2, m_alu_b'length);
            m_shamt <= '1';

         when s_Jal3 =>
            tipo_inst <= TYPE_J;
            wr_pc     <= '1';
            m_pc      <= conv_std_logic_vector(0, m_pc'length);
            en_banco  <= '1';
            m_banco   <= conv_std_logic_vector(1, m_banco'length);
            m_ram     <= '0';
            m_alu_a   <= conv_std_logic_vector(2, m_alu_a'length);
            m_alu_b   <= conv_std_logic_vector(2, m_alu_b'length);
            m_alu_op  <= conv_std_logic_vector(0, m_alu_op'length);

         when s_Jalr3 =>
            tipo_inst <= TYPE_I;
            wr_pc     <= '1';
            m_pc      <= conv_std_logic_vector(0, m_pc'length);
            en_banco  <= '1';
            m_banco   <= conv_std_logic_vector(1, m_banco'length);
            m_ram     <= '0';
            m_alu_a   <= conv_std_logic_vector(0, m_alu_a'length);
            m_alu_b   <= conv_std_logic_vector(2, m_alu_b'length);
            m_alu_op  <= conv_std_logic_vector(0, m_alu_op'length);
            mask_b0   <= '1';

         when s_lui3 =>
            tipo_inst <= TYPE_U;
            m_banco   <= conv_std_logic_vector(2, m_banco'length);
            en_banco  <= '1';
```

```
when s_lwsw3 =>
   if opcode="00000" then
      tipo_inst <= TYPE_I;
   else
      tipo_inst <= TYPE_S;
   end if;
   m_alu_op <= conv_std_logic_vector(0, m_alu_op'length);
   m_alu_a  <= conv_std_logic_vector(0, m_alu_a'length);
   m_alu_b  <= conv_std_logic_vector(2, m_alu_b'length);

when s_aupic3 =>
   tipo_inst <= TYPE_U;
   m_alu_op  <= conv_std_logic_vector(0, m_alu_op'length);
   m_alu_a   <= conv_std_logic_vector(2, m_alu_a'length);
   m_alu_b   <= conv_std_logic_vector(2, m_alu_b'length);

when s_Arit3 =>
   m_alu_op <= ir_out(30) & ir_out(14 downto 12);
   m_alu_a  <= conv_std_logic_vector(0, m_alu_a'length);
   m_alu_b  <= conv_std_logic_vector(0, m_alu_b'length);

when s_SalCond =>
   wr_pc_cond <= '1';
   m_pc       <= conv_std_logic_vector(1, m_pc'length);
   m_alu_a    <= conv_std_logic_vector(0, m_alu_a'length);
   m_alu_b    <= conv_std_logic_vector(0, m_alu_b'length);
   m_alu_op <= "001" & ir_out(13);

when s_lw4 =>
   tipo_acc <= ir_out(13 downto 12);
   l_u      <= ir_out(14);

   m_alu_a  <= conv_std_logic_vector(0, m_alu_a'length);
   m_alu_b  <= conv_std_logic_vector(2, m_alu_b'length);
   m_alu_op <= conv_std_logic_vector(0, m_alu_op'length);

when s_sw4 =>
   tipo_acc <= ir_out(13 downto 12);
   we_ram   <= '1';

when s_Arit4 =>
   m_ram    <= '0';
   m_banco  <= conv_std_logic_vector(0, m_banco'length);
   en_banco <= '1';

when s_lw5 =>
   m_banco  <= conv_std_logic_vector(0, m_banco'length);
   en_banco <= '1';
   m_ram    <= '1';

   tipo_acc <= ir_out(13 downto 12);
   l_u      <= ir_out(14);
```

```
   m_alu_a  <= conv_std_logic_vector(0, m_alu_a'length);
   m_alu_b  <= conv_std_logic_vector(2, m_alu_b'length);
   m_alu_op <= conv_std_logic_vector(0, m_alu_op'length);

when s_CGetTag3 =>
   en_banco <= '1';
   m_banco  <= conv_std_logic_vector(3, m_banco'length);

when s_CIncOffset3 =>
   cap_enW <= '1'
   ;

when s_CSetPerm3 =>
   cap_enW <= '1'
;

when s_CSetBounds3 =>
   cap_enW <= '1'
;

when s_CJump3 =>
   if cheri_check_ok = '1' then
       wr_pc <= '1';
       m_pc  <= conv_std_logic_vector(2, m_pc'length);
   end if;

when s_CJumpLink3 =>
   if cheri_check_ok = '1' then
       wr_pc    <= '1';
       m_pc     <= conv_std_logic_vector(2, m_pc'length);
       en_banco <= '1';
       m_banco  <= conv_std_logic_vector(1, m_banco'length);
   end if;

when s_CLoad3 =>
   tipo_acc <= ir_out(13 downto 12);
   l_u  <= ir_out(14)
;

when s_CLoad4 =>
   if cheri_check_ok = '1' then
       m_banco  <= conv_std_logic_vector(0, m_banco'length);
       en_banco <= '1';
       m_ram    <= '1';
   end if;

when s_CStore3 =>
   if cheri_check_ok = '1' then
        tipo_acc <= ir_out(13 downto 12);
        we_ram   <= '1';
   end if;
```

```
          when s_CLoadCap3 =>
            tipo_acc <= ir_out(13 downto 12);
            l_u      <= ir_out(14)
            ;

          when s_CLoadCap4 =>
            if cheri_check_ok = '1' then
                cap_enW <= '1';
            end if;

          when s_CStoreCap3 =>
             if cheri_check_ok = '1' then
                tipo_acc  <= ir_out(13 downto 12);
                cap_mem_we <= '1';
             end if;

         when others =>
            null;
      end case;
   end process p_output;

end architecture fsm;
```

ANEXO III: BANCOS DE PRUEBA PARA LAS SIMULACIONES DEL RV32Y

Tb_RV32Y_instr

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY tb_RV32Y_instr IS
END tb_RV32Y_instr;

ARCHITECTURE testbench OF tb_RV32Y_instr IS

   CONSTANT c_clk_semi_period : TIME := 10 ns;
   CONSTANT c_clk_period      : TIME := 2 * c_clk_semi_period;

   SIGNAL clk   : std_logic := '0';
   SIGNAL reset : std_logic;

   SIGNAL dbg_pc      : std_logic_vector(31 DOWNTO 0);
   SIGNAL dbg_ir      : std_logic_vector(31 DOWNTO 0);
   SIGNAL dbg_din_reg : std_logic_vector(31 DOWNTO 0);
   SIGNAL dbg_en_reg  : std_logic;
   SIGNAL dbg_rd      : std_logic_vector(4 DOWNTO 0);

   SIGNAL dbg_capA_tag      : std_logic;
   SIGNAL dbg_capA_addr     : std_logic_vector(15 DOWNTO 0);
   SIGNAL dbg_capA_perms    : std_logic_vector(7 DOWNTO 0);
   SIGNAL dbg_cap_enW       : std_logic;
   SIGNAL dbg_capA_base     : std_logic_vector(15 DOWNTO 0);
   SIGNAL dbg_capA_length   : std_logic_vector(15 DOWNTO 0);
   SIGNAL dbg_capB_addr     : std_logic_vector(15 DOWNTO 0);
   SIGNAL dbg_capDin_addr   : std_logic_vector(15 DOWNTO 0);
   SIGNAL dbg_capDin_perms  : std_logic_vector(7 DOWNTO 0);
   SIGNAL dbg_capDin_length : std_logic_vector(15 DOWNTO 0);
   SIGNAL dbg_capAddrW      : std_logic_vector(4 DOWNTO 0);

   SIGNAL dbg_cheri_check_ok  : std_logic;
   SIGNAL dbg_req_exec        : std_logic;
   SIGNAL dbg_req_load        : std_logic;
   SIGNAL dbg_req_store       : std_logic;
   SIGNAL dbg_permit_load_ok  : std_logic;
   SIGNAL dbg_permit_store_ok : std_logic;
   SIGNAL dbg_we_ram          : std_logic;
```

```
   SIGNAL dbg_capB_tag    : std_logic;
   SIGNAL dbg_capB_perms  : std_logic_vector(7 downto 0);
   SIGNAL dbg_capDin_tag  : std_logic;
   SIGNAL dbg_capDin_base : std_logic_vector(15 downto 0);



BEGIN

   clk <= NOT clk AFTER c_clk_semi_period;

   i1 : entity work.RV32Y
      PORT MAP(
         clk               => clk,
         reset             => reset,

         dbg_pc            => dbg_pc,
         dbg_ir            => dbg_ir,
         dbg_din_reg       => dbg_din_reg,
         dbg_en_reg        => dbg_en_reg,
         dbg_rd            => dbg_rd,

         dbg_capA_tag      => dbg_capA_tag,
         dbg_capA_addr     => dbg_capA_addr,
         dbg_capA_perms    => dbg_capA_perms,
         dbg_cap_enW       => dbg_cap_enW,

         dbg_capA_base      => dbg_capA_base,
         dbg_capA_length    => dbg_capA_length,
         dbg_capB_addr      => dbg_capB_addr,
         dbg_capDin_addr    => dbg_capDin_addr,
         dbg_capDin_perms   => dbg_capDin_perms,
         dbg_capDin_length => dbg_capDin_length,
         dbg_capAddrW       => dbg_capAddrW,

         dbg_cheri_check_ok  => dbg_cheri_check_ok,
         dbg_req_exec        => dbg_req_exec,
         dbg_req_load        => dbg_req_load,
         dbg_req_store       => dbg_req_store,
         dbg_permit_load_ok  => dbg_permit_load_ok,
         dbg_permit_store_ok => dbg_permit_store_ok,
         dbg_we_ram          => dbg_we_ram,
         dbg_capDin_tag   => dbg_capDin_tag,
         dbg_capDin_base => dbg_capDin_base
      );

   wave : PROCESS
      VARIABLE n_reg_write : integer := 0;
      VARIABLE n_cap_write : integer := 0;
      VARIABLE ciclos      : integer := 0;

      VARIABLE seen_cjump_c3  : boolean := false;
      VARIABLE seen_cjump_c2  : boolean := false;
```

```
      VARIABLE seen_cload_c5  : boolean := false;
      VARIABLE seen_cstore_c5 : boolean := false;
      VARIABLE seen_cload_c2  : boolean := false;
      VARIABLE seen_cstore_c2 : boolean := false;
   BEGIN

      reset <= '1';
      WAIT FOR 5 * c_clk_period;
      reset <= '0';

      LOOP
         WAIT UNTIL rising_edge(clk);
         WAIT FOR 1 ns;

         ciclos := ciclos + 1;

         ASSERT ciclos < 700
         REPORT "Timeout: no se han comprobado todas las instrucciones CHERI"
         SEVERITY failure;

         ----------------------------------------------------------------
         -- Comprobacion de permisos en CJump
         ----------------------------------------------------------------
         IF dbg_req_exec = '1' AND dbg_ir = x"0001C047" AND seen_cjump_c3 = false
THEN
            seen_cjump_c3 := true;

            ASSERT dbg_cheri_check_ok = '0'
            REPORT "Fallo CJump c3: deberia bloquearse por falta de permiso exec"
            SEVERITY failure;
         END IF;

         IF dbg_req_exec = '1' AND dbg_ir = x"00014047" AND seen_cjump_c2 = false
THEN
            seen_cjump_c2 := true;

            ASSERT dbg_cheri_check_ok = '1'
            REPORT "Fallo CJump c2: deberia permitirse con permiso exec"
            SEVERITY failure;
         END IF;

         ----------------------------------------------------------------
         -- Comprobacion de permisos en CLoad / CStore
         ----------------------------------------------------------------
         IF dbg_ir = x"0002847F" AND dbg_req_load = '1' AND seen_cload_c5 = false
THEN
            seen_cload_c5 := true;

            ASSERT dbg_permit_load_ok = '0' AND dbg_cheri_check_ok = '0'
            REPORT "Fallo CLoad c5: deberia bloquearse por falta de permiso load"
            SEVERITY failure;
         END IF;
```

```
         IF dbg_ir = x"0092907F" AND dbg_req_store = '1' AND seen_cstore_c5 =
false THEN
            seen_cstore_c5 := true;

            ASSERT dbg_permit_store_ok = '0' AND dbg_cheri_check_ok = '0'
            REPORT "Fallo CStore c5: deberia bloquearse por falta de permiso
store"
            SEVERITY failure;
         END IF;

         IF dbg_ir = x"0001047F" AND dbg_req_load = '1' AND seen_cload_c2 = false
THEN
            seen_cload_c2 := true;

            ASSERT dbg_permit_load_ok = '1' AND dbg_cheri_check_ok = '1'
            REPORT "Fallo CLoad c2: deberia permitirse con permiso load"
            SEVERITY failure;
         END IF;

         IF dbg_ir = x"0091107F" AND dbg_req_store = '1' THEN
           seen_cstore_c2 := true;

           ASSERT dbg_permit_store_ok = '1' AND dbg_cheri_check_ok = '1'
           REPORT "Fallo CStore c2: deberia permitirse con permiso store"
           SEVERITY failure;

        END IF;

         ---------------------------------------------------------------------
         -- Banco de registros normal
         ---------------------------------------------------------------------
         IF dbg_en_reg = '1' AND dbg_rd /= "00000" THEN

            n_reg_write := n_reg_write + 1;

            CASE n_reg_write IS

               WHEN 1 =>
                  ASSERT dbg_rd = "00101" AND dbg_din_reg = x"00000001"
                  REPORT "Fallo CGetTag: x5 no recibe tag=1"
                  SEVERITY failure;

               WHEN 2 =>
                  ASSERT dbg_rd = "00001" AND dbg_din_reg = x"00000010"
                  REPORT "Fallo ADDI x1,x0,16"
                  SEVERITY failure;

               WHEN 3 =>
                  ASSERT dbg_rd = "00010" AND dbg_din_reg = x"000000FB"
                  REPORT "Fallo ADDI x2,x0,0xFB"
                  SEVERITY failure;

               WHEN 4 =>
```

```
                    ASSERT dbg_rd = "00110" AND dbg_din_reg = x"00000021"
                    REPORT "Fallo CJump c3: x6 no recibe 33, puede que haya saltado
indebidamente"
                    SEVERITY failure;

                WHEN 5 =>
                    ASSERT dbg_rd = "00111" AND dbg_din_reg = x"0000002C"
                    REPORT "Fallo CJump c2: no se ha saltado correctamente al
destino"
                    SEVERITY failure;

                    ASSERT seen_cjump_c3 = true
                    REPORT "Fallo: no se ha comprobado CJump c3"
                    SEVERITY failure;

                    ASSERT seen_cjump_c2 = true
                    REPORT "Fallo: no se ha comprobado CJump c2"
                    SEVERITY failure;

                WHEN 6 =>
                    ASSERT dbg_rd = "00100" AND dbg_din_reg = x"00000008"
                    REPORT "Fallo ADDI x4,x0,8"
                    SEVERITY failure;

                WHEN 7 =>
                    ASSERT dbg_rd = "01001" AND dbg_din_reg = x"000000F8"
                    REPORT "Fallo ADDI x9,x0,0xF8"
                    SEVERITY failure;
                WHEN 8 =>
                    ASSERT dbg_rd = "01000"
                    REPORT "Fallo CLoad c2: no escribe en x8"
                    SEVERITY failure;

                WHEN OTHERS =>
                    ASSERT false
                    REPORT "Escritura inesperada en banco de registros normal"
                    SEVERITY failure;

            END CASE;

        END IF;

        ---------------------------------------------------------------------
        -- Banco de capabilities
        ---------------------------------------------------------------------
        IF dbg_cap_enW = '1' THEN

            n_cap_write := n_cap_write + 1;

            CASE n_cap_write IS

                WHEN 1 =>
                    ASSERT dbg_capAddrW = "00010"
```

```
            REPORT "Fallo CIncOffset: no escribe en c2"
            SEVERITY failure;

            ASSERT dbg_capDin_addr = x"0110"
            REPORT "Fallo CIncOffset: c2.addr incorrecto"
            SEVERITY failure;

         WHEN 2 =>
            ASSERT dbg_capAddrW = "00011"
            REPORT "Fallo CSetPerm: no escribe en c3"
            SEVERITY failure;

            ASSERT dbg_capDin_perms = x"FB"
            REPORT "Fallo CSetPerm: c3.perms no vale 0xFB"
            SEVERITY failure;

         WHEN 3 =>
            ASSERT dbg_capAddrW = "00100"
            REPORT "Fallo CSetBounds: no escribe en c4"
            SEVERITY failure;

            ASSERT dbg_capDin_length = x"0008"
            REPORT "Fallo CSetBounds: c4.length no vale 0x0008"
            SEVERITY failure;

            ASSERT dbg_capDin_addr = x"0110"
            REPORT "Fallo CSetBounds: c4.addr no conserva 0x0110"
            SEVERITY failure;

         WHEN 4 =>
            ASSERT dbg_capAddrW = "00101"
            REPORT "Fallo CSetPerm: no escribe en c5"
            SEVERITY failure;

            ASSERT dbg_capDin_perms = x"F8"
            REPORT "Fallo CSetPerm: c5.perms no vale 0xF8"
            SEVERITY failure;

        WHEN 5 =>
             ASSERT dbg_capAddrW = "00110"
             REPORT "Fallo CLoadCap: no escribe en c6"
             SEVERITY failure;

             ASSERT dbg_capDin_base = x"0100"
             REPORT "Fallo CLoadCap: base incorrecta"
             SEVERITY failure;

             ASSERT dbg_capDin_addr = x"0110"
             REPORT "Fallo CLoadCap: addr incorrecta"
             SEVERITY failure;

             ASSERT dbg_capDin_length = x"0008"
             REPORT "Fallo CLoadCap: length incorrecto"
```

```
                    SEVERITY failure;

                    ASSERT dbg_capDin_perms = x"FF"
                    REPORT "Fallo CLoadCap: permisos incorrectos"
                    SEVERITY failure;

                    ASSERT dbg_capDin_tag = '1'
                    REPORT "Fallo CLoadCap: tag incorrecto"
                    SEVERITY failure;

                    ASSERT seen_cload_c5 = true
                    REPORT "Fallo: no se ha ejecutado CLoad sobre c5"
                    SEVERITY failure;

                    ASSERT seen_cstore_c5 = true
                    REPORT "Fallo: no se ha ejecutado CStore sobre c5"
                    SEVERITY failure;

                    ASSERT seen_cload_c2 = true
                    REPORT "Fallo: no se ha ejecutado CLoad sobre c2"
                    SEVERITY failure;

                    ASSERT seen_cstore_c2 = true
                    REPORT "Fallo: no se ha ejecutado CStore sobre c2"
                    SEVERITY failure;

                    ASSERT false
                    REPORT "Fin simulacion: Instrucciones CHERI correctas"
                    SEVERITY failure;

                WHEN OTHERS =>
                   ASSERT false
                   REPORT "Escritura inesperada en banco de capabilities"
                   SEVERITY failure;

            END CASE;

         END IF;

      END LOOP;

   END PROCESS;

END ARCHITECTURE;
```

Cheri_check_unit_tb

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.CHERI_pck.all;

entity cheri_check_unit_tb is
end cheri_check_unit_tb;

architecture testbench of cheri_check_unit_tb is

   signal cap_in        : capability_t;
   signal access_addr   : std_logic_vector(15 downto 0);

   signal req_load      : std_logic;
   signal req_store     : std_logic;
   signal req_exec      : std_logic;
   signal req_load_cap  : std_logic;
   signal req_store_cap : std_logic;

   signal tag_ok          : std_logic;
   signal permit_load_ok  : std_logic;
   signal permit_store_ok : std_logic;
   signal permit_exec_ok  : std_logic;
   signal permit_lcap_ok  : std_logic;
   signal permit_scap_ok  : std_logic;
   signal bounds_ok       : std_logic;
   signal check_ok        : std_logic;

begin

   i_cheri_check_unit : entity work.cheri_check_unit
      port map(
         cap_in          => cap_in,
         access_addr     => access_addr,
         req_load        => req_load,
         req_store       => req_store,
         req_exec        => req_exec,
         req_load_cap    => req_load_cap,
         req_store_cap   => req_store_cap,
         tag_ok          => tag_ok,
         permit_load_ok  => permit_load_ok,
         permit_store_ok => permit_store_ok,
         permit_exec_ok  => permit_exec_ok,
         permit_lcap_ok  => permit_lcap_ok,
         permit_scap_ok  => permit_scap_ok,
         bounds_ok       => bounds_ok,
         check_ok        => check_ok
      );

   wave : process
```

```
    variable cap_test : capability_t;

begin

    ---------------------------------------------------------------
    -- Capability valida:
    -- base   = 0x0100
    -- length = 0x0020
    --
    -- Rango valido:
    -- 0x0100 <= access_addr < 0x0120
    ---------------------------------------------------------------
         ---------------------------------------------------------------
    -- Capability valida de referencia
    -- base   = 0x0100
    -- length = 0x0020
    -- rango valido: 0x0100 <= access_addr < 0x0120
    ---------------------------------------------------------------
    cap_test.base     := x"0100";
    cap_test.length   := x"0020";
    cap_test.addr     := x"0100";
    cap_test.reserved := (others => '0');
    cap_test.tag      := '1';

    req_load      <= '0';
    req_store     <= '0';
    req_exec      <= '0';
    req_load_cap  <= '0';
    req_store_cap <= '0';

    access_addr <= x"0104";

    ---------------------------------------------------------------
    -- LOAD: con permiso funciona
    ---------------------------------------------------------------
    cap_test.perms := "00011111";
    cap_in <= cap_test;

    req_load <= '1';
    wait for 10 ns;

    assert permit_load_ok = '1'
    report "Fallo: LOAD deberia tener permiso"
    severity failure;

    assert bounds_ok = '1'
    report "Fallo: LOAD esta dentro de bounds"
    severity failure;

    assert check_ok = '1'
    report "Fallo: LOAD con permiso deberia funcionar"
    severity failure;
```

```
req_load <= '0';

-------------------------------------------------------------------
-- LOAD: sin permiso NO funciona
-- perms(0) = 0
-------------------------------------------------------------------
cap_test.perms := "00011110";
cap_in <= cap_test;

req_load <= '1';
wait for 10 ns;

assert permit_load_ok = '0'
report "Fallo: LOAD no deberia tener permiso"
severity failure;

assert bounds_ok = '1'
report "Fallo: LOAD sin permiso sigue estando dentro de bounds"
severity failure;

assert check_ok = '0'
report "Fallo: LOAD sin permiso no deberia funcionar"
severity failure;

req_load <= '0';

-------------------------------------------------------------------
-- STORE: con permiso funciona
-------------------------------------------------------------------
cap_test.perms := "00011111";
cap_in <= cap_test;

req_store <= '1';
wait for 10 ns;

assert permit_store_ok = '1'
report "Fallo: STORE deberia tener permiso"
severity failure;

assert bounds_ok = '1'
report "Fallo: STORE esta dentro de bounds"
severity failure;

assert check_ok = '1'
report "Fallo: STORE con permiso deberia funcionar"
severity failure;

req_store <= '0';

-------------------------------------------------------------------
-- STORE: sin permiso NO funciona
-- perms(1) = 0
```

```
    ---------------------------------------------------------------------
    cap_test.perms := "00011101";
    cap_in <= cap_test;

    req_store <= '1';
    wait for 10 ns;

    assert permit_store_ok = '0'
    report "Fallo: STORE no deberia tener permiso"
    severity failure;

    assert bounds_ok = '1'
    report "Fallo: STORE sin permiso sigue estando dentro de bounds"
    severity failure;

    assert check_ok = '0'
    report "Fallo: STORE sin permiso no deberia funcionar"
    severity failure;

    req_store <= '0';

    ---------------------------------------------------------------------
    -- EXEC: con permiso funciona
    ---------------------------------------------------------------------
    cap_test.perms := "00011111";
    cap_in <= cap_test;

    req_exec <= '1';
    wait for 10 ns;

    assert permit_exec_ok = '1'
    report "Fallo: EXEC deberia tener permiso"
    severity failure;

    assert bounds_ok = '1'
    report "Fallo: EXEC esta dentro de bounds"
    severity failure;

    assert check_ok = '1'
    report "Fallo: EXEC con permiso deberia funcionar"
    severity failure;

    req_exec <= '0';

    ---------------------------------------------------------------------
    -- EXEC: sin permiso NO funciona
    -- perms(2) = 0
    ---------------------------------------------------------------------
    cap_test.perms := "00011011";
    cap_in <= cap_test;

    req_exec <= '1';
    wait for 10 ns;
```

```
assert permit_exec_ok = '0'
report "Fallo: EXEC no deberia tener permiso"
severity failure;

assert bounds_ok = '1'
report "Fallo: EXEC sin permiso sigue estando dentro de bounds"
severity failure;

assert check_ok = '0'
report "Fallo: EXEC sin permiso no deberia funcionar"
severity failure;

req_exec <= '0';

-----------------------------------------------------------------
-- LOAD_CAP: con permiso funciona
-----------------------------------------------------------------
cap_test.perms := "00011111";
cap_in <= cap_test;

req_load_cap <= '1';
wait for 10 ns;

assert permit_lcap_ok = '1'
report "Fallo: LOAD_CAP deberia tener permiso"
severity failure;

assert bounds_ok = '1'
report "Fallo: LOAD_CAP esta dentro de bounds"
severity failure;

assert check_ok = '1'
report "Fallo: LOAD_CAP con permiso deberia funcionar"
severity failure;

req_load_cap <= '0';

-----------------------------------------------------------------
-- LOAD_CAP: sin permiso NO funciona
-- perms(3) = 0
-----------------------------------------------------------------
cap_test.perms := "00010111";
cap_in <= cap_test;

req_load_cap <= '1';
wait for 10 ns;

assert permit_lcap_ok = '0'
report "Fallo: LOAD_CAP no deberia tener permiso"
severity failure;

assert bounds_ok = '1'
```

```
        report "Fallo: LOAD_CAP sin permiso sigue estando dentro de bounds"
        severity failure;

        assert check_ok = '0'
        report "Fallo: LOAD_CAP sin permiso no deberia funcionar"
        severity failure;

        req_load_cap <= '0';

        ---------------------------------------------------------------------
        -- STORE_CAP: con permiso funciona
        ---------------------------------------------------------------------
        cap_test.perms := "00011111";
        cap_in <= cap_test;

        req_store_cap <= '1';
        wait for 10 ns;

        assert permit_scap_ok = '1'
        report "Fallo: STORE_CAP deberia tener permiso"
        severity failure;

        assert bounds_ok = '1'
        report "Fallo: STORE_CAP esta dentro de bounds"
        severity failure;

        assert check_ok = '1'
        report "Fallo: STORE_CAP con permiso deberia funcionar"
        severity failure;

        req_store_cap <= '0';

        ---------------------------------------------------------------------
        -- STORE_CAP: sin permiso NO funciona
        -- perms(4) = 0
        ---------------------------------------------------------------------
        cap_test.perms := "00001111";
        cap_in <= cap_test;

        req_store_cap <= '1';
        wait for 10 ns;

        assert permit_scap_ok = '0'
        report "Fallo: STORE_CAP no deberia tener permiso"
        severity failure;

        assert bounds_ok = '1'
        report "Fallo: STORE_CAP sin permiso sigue estando dentro de bounds"
        severity failure;

        assert check_ok = '0'
        report "Fallo: STORE_CAP sin permiso no deberia funcionar"
        severity failure;
```

```
      req_store_cap <= '0';
      ---------------------------------------------------------------
      -- Primer caso fuera de bounds: access_addr < base
      ---------------------------------------------------------------
      req_load <= '1';
      access_addr <= x"00FF";
      wait for 10 ns;

      assert bounds_ok = '0'
      report "Fallo cheri_check_unit: direccion menor que base deberia dar
bounds_ok = 0"
      severity failure;

      assert check_ok = '0'
      report "Fallo cheri_check_unit: direccion menor que base deberia dar
check_ok = 0"
      severity failure;

      req_load <= '0';

      ---------------------------------------------------------------
      -- Segundo caso fuera de bounds:
      -- access_addr = base + length
      -- Como el limite superior es exclusivo, 0x0120 NO es valido
      ---------------------------------------------------------------
      req_load <= '1';
      access_addr <= x"0120";
      wait for 10 ns;

      assert bounds_ok = '0'
      report "Fallo cheri_check_unit: base + length no deberia estar dentro de
bounds"
      severity failure;

      assert check_ok = '0'
      report "Fallo cheri_check_unit: base + length deberia dar check_ok = 0"
      severity failure;

      req_load <= '0';

      ---------------------------------------------------------------
      -- Caso muy fuera de bounds
      ---------------------------------------------------------------
      req_store <= '1';
      access_addr <= x"0200";
      wait for 10 ns;

      assert bounds_ok = '0'
      report "Fallo cheri_check_unit: direccion muy fuera de bounds deberia dar
bounds_ok = 0"
      severity failure;
```

```
      assert check_ok = '0'
      report "Fallo cheri_check_unit: direccion muy fuera de bounds deberia dar
check_ok = 0"
      severity failure;

      req_store <= '0';

      ----------------------------------------------------------------------
      -- Permiso denegado: quitamos permiso STORE
      -- perms(1) = 0
      ----------------------------------------------------------------------
      cap_test.perms := "00011101";
      cap_in <= cap_test;

      req_store <= '1';
      access_addr <= x"0104";
      wait for 10 ns;

      assert bounds_ok = '1'
      report "Fallo cheri_check_unit: bounds deberia ser correcto aunque falle
permiso"
      severity failure;

      assert permit_store_ok = '0'
      report "Fallo cheri_check_unit: STORE deberia estar denegado"
      severity failure;

      assert check_ok = '0'
      report "Fallo cheri_check_unit: STORE sin permiso deberia dar check_ok = 0"
      severity failure;

      req_store <= '0';

      ----------------------------------------------------------------------
      -- Tag invalido
      ----------------------------------------------------------------------
      cap_test.perms := "00011111";
      cap_test.tag   := '0';
      cap_in <= cap_test;

      req_load <= '1';
      access_addr <= x"0104";
      wait for 10 ns;

      assert tag_ok = '0'
      report "Fallo cheri_check_unit: tag_ok deberia ser 0"
      severity failure;

      assert bounds_ok = '1'
      report "Fallo cheri_check_unit: bounds puede ser correcto aunque el tag
falle"
      severity failure;
```

```
      assert permit_load_ok = '1'
      report "Fallo cheri_check_unit: permiso LOAD deberia seguir siendo 1"
      severity failure;

      assert check_ok = '0'
      report "Fallo cheri_check_unit: tag invalido deberia dar check_ok = 0"
      severity failure;

      req_load <= '0';

      ---------------------------------------------------------------
      -- LOAD_CAP permitido
      ---------------------------------------------------------------
      cap_test.tag := '1';
      cap_test.perms := "00011111";
      cap_in <= cap_test;

      req_load_cap <= '1';
      access_addr <= x"0108";
      wait for 10 ns;

      assert permit_lcap_ok = '1'
      report "Fallo cheri_check_unit: permiso LOAD_CAP incorrecto"
      severity failure;

      assert check_ok = '1'
      report "Fallo cheri_check_unit: LOAD_CAP permitido deberia dar check_ok =
1"
      severity failure;

      req_load_cap <= '0';

      ---------------------------------------------------------------
      -- STORE_CAP permitido
      ---------------------------------------------------------------
      req_store_cap <= '1';
      access_addr <= x"0108";
      wait for 10 ns;

      assert permit_scap_ok = '1'
      report "Fallo cheri_check_unit: permiso STORE_CAP incorrecto"
      severity failure;

      assert check_ok = '1'
      report "Fallo cheri_check_unit: STORE_CAP permitido deberia dar check_ok =
1"
      severity failure;

      req_store_cap <= '0';

      ---------------------------------------------------------------
      -- STORE_CAP denegado
      -- perms(4) = 0
```

```
      ------------------------------------------------------------------
      cap_test.perms := "00001111";
      cap_in <= cap_test;

      req_store_cap <= '1';
      access_addr <= x"0108";
      wait for 10 ns;

      assert permit_scap_ok = '0'
      report "Fallo cheri_check_unit: STORE_CAP deberia estar denegado"
      severity failure;

      assert bounds_ok = '1'
      report "Fallo cheri_check_unit: bounds deberia seguir siendo correcto"
      severity failure;

      assert check_ok = '0'
      report "Fallo cheri_check_unit: STORE_CAP sin permiso deberia dar check_ok
= 0"
      severity failure;

      req_store_cap <= '0';

      ------------------------------------------------------------------
      -- Caso length = 0
      -- Rango vacio: ninguna direccion debe ser valida
      ------------------------------------------------------------------
      cap_test.base     := x"0100";
      cap_test.length   := x"0000";
      cap_test.addr     := x"0100";
      cap_test.perms    := "00011111";
      cap_test.reserved := (others => '0');
      cap_test.tag      := '1';

      cap_in <= cap_test;

      req_load <= '1';
      access_addr <= x"0100";
      wait for 10 ns;

      assert bounds_ok = '0'
      report "Fallo cheri_check_unit: length = 0 deberia dar rango vacio"
      severity failure;

      assert check_ok = '0'
      report "Fallo cheri_check_unit: length = 0 deberia dar check_ok = 0"
      severity failure;

      req_load <= '0';

      ------------------------------------------------------------------
      -- Fin
      ------------------------------------------------------------------
```

```
      assert false
      report "Fin de la simulacion: cheri_check_unit correcta"
      severity failure;

   end process;

end architecture;
```

Tb_BancoCap

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.CHERI_pck.all;

entity tb_BancoCap is
end tb_BancoCap;

architecture testbench of tb_BancoCap is

   constant c_clk_semi_period : time := 10 ns;
   constant c_clk_period      : time := 2 * c_clk_semi_period;

   signal clk   : std_logic := '0';
   signal reset : std_logic := '0';
   signal enW   : std_logic := '0';

   signal Din   : capability_t;
   signal AddrA : std_logic_vector(4 downto 0) := (others => '0');
   signal AddrB : std_logic_vector(4 downto 0) := (others => '0');
   signal AddrW : std_logic_vector(4 downto 0) := (others => '0');

   signal capA  : capability_t;
   signal capB  : capability_t;

   component BancoCap is
      port(
         Din                  : in  capability_t;
         AddrA, AddrB, AddrW  : in  std_logic_vector(4 downto 0);
         enW, clk, reset      : in  std_logic;
         capA, capB           : out capability_t
      );
   end component;

begin

   clk <= not clk after c_clk_semi_period;

   i_BancoCap : BancoCap
      port map(
         Din   => Din,
         AddrA => AddrA,
         AddrB => AddrB,
         AddrW => AddrW,
         enW   => enW,
         clk   => clk,
         reset => reset,
         capA  => capA,
         capB  => capB
      );
```

```
wave : process

   variable cap_test : capability_t;

begin

   Din   <= NULL_CAPABILITY;
   AddrA <= (others => '0');
   AddrB <= (others => '0');
   AddrW <= (others => '0');
   enW   <= '0';
   reset <= '0';

   wait for c_clk_period;

   ---------------------------------------------------------------------
   -- reset
   ---------------------------------------------------------------------
   reset <= '1';
   wait for c_clk_period;
   reset <= '0';
   wait for c_clk_period;

   ---------------------------------------------------------------------
   -- comprobacion registro 0 nulo
   ---------------------------------------------------------------------
   AddrA <= "00000";
   wait for c_clk_period;

   assert capA = NULL_CAPABILITY
   report "Fallo BancoCap: c0 no es NULL_CAPABILITY tras reset"
   severity failure;

   ---------------------------------------------------------------------
   -- comprobacion registro 1 inicializado
   ---------------------------------------------------------------------
   AddrA <= "00001";
   wait for c_clk_period;

   assert capA.base = x"0100"
   report "Fallo BancoCap: base inicial de c1 incorrecta"
   severity failure;

   assert capA.length = x"0020"
   report "Fallo BancoCap: length inicial de c1 incorrecta"
   severity failure;

   assert capA.addr = x"0100"
   report "Fallo BancoCap: addr inicial de c1 incorrecta"
   severity failure;

   assert capA.perms = x"FF"
```

```
    report "Fallo BancoCap: perms inicial de c1 incorrectos"
    severity failure;

    assert capA.tag = '1'
    report "Fallo BancoCap: tag inicial de c1 incorrecto"
    severity failure;

    ---------------------------------------------------------------------
    -- escritura en c5
    ---------------------------------------------------------------------
    cap_test.base     := x"0200";
    cap_test.length   := x"0040";
    cap_test.addr     := x"0210";
    cap_test.perms    := x"0F";
    cap_test.reserved := (others => '0');
    cap_test.tag      := '1';

    Din   <= cap_test;
    AddrW <= "00101";
    enW   <= '1';

    wait for c_clk_period;

    enW <= '0';
    wait for c_clk_period;

    AddrA <= "00101";
    wait for c_clk_period;

    assert capA = cap_test
    report "Fallo BancoCap: escritura/lectura en c5 incorrecta"
    severity failure;

    ---------------------------------------------------------------------
    -- comprobar que con enW = 0 no se escribe
    ---------------------------------------------------------------------
    cap_test.base     := x"0300";
    cap_test.length   := x"0080";
    cap_test.addr     := x"0310";
    cap_test.perms    := x"AA";
    cap_test.reserved := (others => '0');
    cap_test.tag      := '1';

    Din   <= cap_test;
    AddrW <= "00101";
    enW   <= '0';

    wait for c_clk_period;

    AddrA <= "00101";
    wait for c_clk_period;

    assert capA.base = x"0200"
```

```
      report "Fallo BancoCap: se ha escrito aunque enW = 0"
      severity failure;

      ---------------------------------------------------------------------
      -- lectura simultanea capA y capB
      ---------------------------------------------------------------------
      AddrA <= "00001";
      AddrB <= "00101";
      wait for c_clk_period;

      assert capA.base = x"0100"
      report "Fallo BancoCap: lectura simultanea capA incorrecta"
      severity failure;

      assert capB.base = x"0200"
      report "Fallo BancoCap: lectura simultanea capB incorrecta"
      severity failure;

      ---------------------------------------------------------------------
      -- fin
      ---------------------------------------------------------------------
      assert false
      report "Fin de la simulacion: BancoCap correcto"
      severity failure;

   end process;

end architecture;
```

Tb_cap_ram

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.CHERI_pck.all;

entity tb_cap_ram is
end tb_cap_ram;

architecture testbench of tb_cap_ram is

   constant c_clk_semi_period : time := 10 ns;
   constant c_clk_period      : time := 2 * c_clk_semi_period;

   signal clk   : std_logic := '0';
   signal addr  : std_logic_vector(31 downto 0) := (others => '0');
   signal d_in  : capability_t;
   signal we    : std_logic := '0';
   signal d_out : capability_t;

   component cap_ram is
      port(
         clk   : in  std_logic;
         addr  : in  std_logic_vector(31 downto 0);
         d_in  : in  capability_t;
         we    : in  std_logic;
         d_out : out capability_t
      );
   end component;

begin

   clk <= not clk after c_clk_semi_period;

   i_cap_ram : cap_ram
      port map(
         clk   => clk,
         addr  => addr,
         d_in  => d_in,
         we    => we,
         d_out => d_out
      );

   wave : process

      variable cap1 : capability_t;
      variable cap2 : capability_t;

   begin

      d_in <= NULL_CAPABILITY;
```

```
        addr <= (others => '0');
        we   <= '0';

        wait for c_clk_period;

        ---------------------------------------------------------------------
        -- memoria inicial a NULL_CAPABILITY
        ---------------------------------------------------------------------
        addr <= x"00000000";
        wait for c_clk_period;

        assert d_out = NULL_CAPABILITY
        report "Fallo cap_ram: posicion inicial no nula"
        severity failure;

        ---------------------------------------------------------------------
        -- escritura en direccion 0x00000010
        -- indice interno = addr(11 downto 2)
        ---------------------------------------------------------------------
        cap1.base     := x"1000";
        cap1.length   := x"0020";
        cap1.addr     := x"1004";
        cap1.perms    := x"FF";
        cap1.reserved := (others => '0');
        cap1.tag      := '1';

        addr <= x"00000010";
        d_in <= cap1;
        we   <= '1';

        wait for c_clk_period;

        we <= '0';
        wait for c_clk_period;

        assert d_out = cap1
        report "Fallo cap_ram: escritura/lectura de cap1 incorrecta"
        severity failure;

        ---------------------------------------------------------------------
        -- comprobar otra direccion distinta
        ---------------------------------------------------------------------
        addr <= x"00000020";
        wait for c_clk_period;

        assert d_out = NULL_CAPABILITY
        report "Fallo cap_ram: otra posicion se ha modificado indebidamente"
        severity failure;

        ---------------------------------------------------------------------
        -- comprobar que con we = 0 no escribe
        ---------------------------------------------------------------------
        cap2.base     := x"2000";
```

```
      cap2.length   := x"0040";
      cap2.addr     := x"2010";
      cap2.perms    := x"0F";
      cap2.reserved := (others => '0');
      cap2.tag      := '1';

      addr <= x"00000010";
      d_in <= cap2;
      we   <= '0';

      wait for c_clk_period;

      assert d_out = cap1
      report "Fallo cap_ram: se ha escrito aunque we = 0"
      severity failure;

      ------------------------------------------------------------------
      -- escritura real de cap2 en otra direccion
      ------------------------------------------------------------------
      addr <= x"00000020";
      d_in <= cap2;
      we   <= '1';

      wait for c_clk_period;

      we <= '0';
      wait for c_clk_period;

      assert d_out = cap2
      report "Fallo cap_ram: escritura/lectura de cap2 incorrecta"
      severity failure;

      ------------------------------------------------------------------
      -- comprobar que cap1 sigue en su sitio
      ------------------------------------------------------------------
      addr <= x"00000010";
      wait for c_clk_period;

      assert d_out = cap1
      report "Fallo cap_ram: cap1 ha sido modificada incorrectamente"
      severity failure;

      ------------------------------------------------------------------
      -- fin
      ------------------------------------------------------------------
      assert false
      report "Fin de la simulacion: cap_ram correcta"
      severity failure;

   end process;

end architecture;
```