ULPGC Universidad de Las Palmas de Gran Canaria | eite ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

# ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

## TRABAJO FIN DE GRADO

# Estudio e Implementación de extensiones dedicadas del juego de instrucciones de RISC-V

**Titulación:** Grado en Ingeniería en Tecnologías de la Telecomunicación


**Mención:** SISTEMAS ELECTRÓNICOS

**Autor:** RENÉ RODRÍGUEZ PAREDES

**Tutor:** Dr. PEDRO PÉREZ CARBALLO


**Fecha:** 18 de julio de 2025

# Resumen

Linux ha logrado cambiar por completo el paradigma tanto de los sistemas operativos como el *software*, revolucionando la forma en la que interactuamos con la computación en su totalidad.

RISC-V representa un hito en la misma dirección, listo para heredar los avances en acceso abierto implantados por Linux. Este juego de instrucciones abierto (ISA) se aplica en el plano del *hardware* por medio de un conjunto de instrucciones definidas por la comunidad, abierta y libre de *royalties*, concediendo la posibilidad a cualquier equipo de pasar de idea a realidad sin las barreras de los conjuntos de instrucciones tradicionales. Por tanto, la consecuencia prevista es la posibilidad de avanzar en la introducción en nueva funcionalidades a medida en los sistemas microprocesadores del futuro, hasta ahora limitados por las ISAs propietarias, ya sea en términos de costes, de facilidad y libertad de modificación. La nueva propuesta permite establecer las bases de la computación moderna, y por tanto, de tendrá un impacto directo en nuestras vidas.

Este trabajo explora cómo, mediante la expansión de RISC-V se puede acelerar la ejecución de las aplicaciones presentes en potencialmente todos los ámbitos de la computación. Se ha seguido una metodología de trabajo basada en un proceso iterativo basado en fijar objetivos, adquisición de conocimientos, realización de prototipado y la obtención de resultados y realización de la documentación.

En primer lugar, se ha estudiado el conjunto de instrucciones base de RISC-V y su extensión. Posteriormente, se ha hecho lo propio con los aceleradores de forma general, hasta llegar al estado del arte en relación con los aceleradores basados en RISC-V. Además, se ha llevado a cabo el prototipado con diversas implementaciones RISC-V y aceleradores. Finalmente, después de determinar el acelerador que es capaz de cumplir con los objetivos, se ha realizado tanto la implementación del mismo como la obtención de los resultados de uso de área, consumo y rendimiento sobre la plataforma *Field-Programmable Gate Array* (FPGA) VCU128 [1].

El siguiente paso lógico tras tener una implementación sobre FPGA es llevar el sistema hacia *Application-Specific Integrated Circuit* (ASIC), por tanto, se ha efectuado un estudio de las implementaciones y resultados de área, consumo y rendimiento de implementaciones ya existentes, marcando así, un camino claro a seguir.

En definitiva, este trabajo busca definir un camino claro desde un concepto completamente abstracto como lo es el conjunto de instrucciones de RISC-V hacia una forma de acelerar la ejecución de las aplicaciones existentes en los diferentes ámbitos de la computación.

# Abstract

Linux has achieved a paradigm shift, from operating systems to everyday software, revolutionizing the way people interact with computing as a whole.

RISC-V represents a high point towards the same direction; it is ready to continue the heritage of openness that Linux started. This open instruction set architecture (ISA) is community-driven, open, and royalty-free, enabling every team to bring any idea to life without the hurdles of traditional instruction set architectures.

The expected outcome is the development of future microprocessor systems with custom functionality, development that has been hampered by the limitations of current instruction set architectures in terms of cost, ease of use, and freedom to modify. This new proposition lays the groundwork for the future of computation and thus will have a great and direct impact on our lives.

This work explores how expanding RISC-V is possible, enabling the acceleration of applications present in potentially all computing domains.

The methodology followed in this work is rooted in an iterative process that involves setting objectives, acquiring knowledge, prototyping, extracting results, and elaborating documentation.

First, a study on RISC-V's base instruction set and how it is extended was carried out. Afterward a similar study on accelerators and their state of the art was performed.

Furthermore, prototyping with different RISC-V implementations was conducted, and accelerators were used. Finally, once the accelerator that fulfilled the objectives that were previously set was found, its implementation as well as the gathering of the results related to power, performance, and area for the VCU128 FPGA were conducted.

The next logical step, once an FPGA implementation was accomplished, is to bring the system to an ASIC; thus, a study on the already present implementations and results on power, performance, and area was analyzed, setting a clear path to follow.

Ultimately, the present work aims to define a direct line of action, starting from an abstract concept like the RISC-V instruction set architecture, towards a means of accelerating the execution of existing applications across various computing domains.

# Índice general

# Índice de figuras

# Índice de tablas

# Códigos fuente

# Acrónimos

**A2A** *All-two-All*

**ABB** *Adaptive Body Biasing*

**ABI** *Application Binary Interface*

**ALU** Unidad Aritmético-lógica
Arithmetic logic unit

**ASIC** *Application-Specific Integrated Circuit*
Circuito integrado de aplicación específica

**ASIP** *Application-specific Instruction Set Processor*
Procesador con juegos de instrucciones específicas

**AXI** Advanced eXtensible Interface

**BSD** *Berkeley Software Distribution*

**CCIX** *Cache Coherent Interconnect for Accelerators*

**CFU** *Custom Functions Unit*

**CGRA** *Coarse-Grained Reconfigurable Array*

**CMOS** Complementary Metal-Oxide-Semiconductor

**CNN** Convolutional Neural Network
Red Neruronal Convolucional

**CPU** Unidad Central de Procesamiento
Central Processing Unit

**CSR** Registros de Control y Estado

**CISC** Complex Instruction Set Computing
Computación con un conjunto de instrucciones complejas

**CV-X-IF** *Core-V eXtension interface*

**D2D** Die-to-Die

**DFG** *DataFlow Graph*

**DIMM** módulo de memoria de dos líneas
Dual In-line Memory Module

**DMA** *Direct Memory Access*

**DNN** *Deep Neural Networks*

**DRAM** Memoria Dinámica de Acceso Aleatorio

**DSA** Domain-specific accelerator

**DSL** Domain-specific language

**DSP** *Digital Signal Processing*
Procesamiento digital de señales

**FD-SOI** Fully Depleted Silicon-on-Insulator

**FIFO** *First-In First-Out*

**FPGA** *Field-Programmable Gate Array*
Matriz de puerta programable en campo

**FPU** Floating-Point Unit

**GDB** *GNU's Not Unix (GNU) Debugger*

**GF** Global Foundries

**GNU** *GNU's Not Unix*

**GPU** Unidad de procesamiento gráfico
Unidad de Procesamiento Gráfico

**HDL** *Hardware Description Language*

**HMC** *Hybrid Memory Cube*

**HPC** *High Performance Computing*
Computación de alto rendimiento

**IA** Inteligencia Artificial

**I/O** *Input/Output*
entrada/salida

**IoT** *Internet of Things*
Internet de las cosas

**IP** *Intellectual property*
Propiedad intelectual

**ISA** juego de instrucciones
Instruction set architecture

**JTAG** *Joint Test Action Group*

**LLC** *Last-Level Cache*

**LVT** Low Voltage Threshold
Baja Tensión Umbral

**ML** *Machine Learning*

**MMU** *Memory Management Unit*
Unidad de gestión de memoria

**NFA** Autómata finito no determinista

**NoC** *Network on a Chip*

**NRE** *Non-Recurring Engineering*
Ingeniería no recurrente

**OAM** *Open Core Protocol (OCP) Accelerator Module*

**OCM** *On-Chip Monitoring*

**OCP** *Open Core Protocol*

**OpenOCD** *Open On-Chip Debugger*

**PC** Contador de Programa
Program counter

**PCIe** *Peripheral Component Interconnect Express*

**PE** *Processing Engine*
Motor de Procesamiento

**PIM** *Processing in Memory*

**PMP** Protección Física de Memoria
Physical Memory Protection

**PoD** *Point of Delivery*

**PPA** Performance, Power and Area
Prestaciones, Potencia y Área

**PQC** Criptografía Post Cuántica
Post-quantum cryptography

**PTW** *Page Table Walker*

**RA** Realidad Aumentada

**RBE** *Reconfigurable Binary Engine*

**ReRAM** Memoria de Acceso Aleatorio Resistiva
Resistive Random-access Memory

**RISC** Reduced Instruction Set Computing
Computación con un conjunto de instrucciones reducido

**RTL** *Register Transfer Level*

**RVV** RISC-V Vector Extension

**SATA** *Serial AT Attachment*

**SIMD** Single Instruction Multiple Data

**SIMT** Single Instruction Multiple Threads

**SLDU** Slide Unit

**SLVT** Super Low Voltage Threshold
Super Baja Tensión Umbral

**SMP** Symmetric Multiprocessing

**SoC** *System-on-Chip*

**SO** Sistema Operativo

**SRAM** Memoria Estática de Acceso Aleatorio
Static Random-access Memory

**STTMRAM** Memoria Magnética de Torque de Transferencia de Espín
Spin-Transfer Torque Magnetic Random-access Memory

**SVD** *System View Description*

**SWD** *Serial Wire Debug*

**TLB** *Translation Lookaside Buffer*

**UART** *Universal Asynchronous Receiver-Transmitter*

**USB** *Universal Serial Bus*

**VLIW** Very Large Instruction Words

**XTEA** Extended Tiny Encryption Algorithm

**TEA** Tiny Encryption Algorithm

**FMA** Fused Multiply Add

**DWT** Discrete Wavelet Transform

**FWT** Fast Wavelet Transform

**LUT** Look-Up Table

**RAM** Random-access memory

**BRAM** Block Random-access memory (RAM)

**BBL** Berkeley boot loader

**DHCP** Dynamic Host Configuration Protocol

**FFT** Fast Fourier Transform

# Capítulo 1

# Introducción

> RISC-V representa en el *hardware*, lo que ha supuesto Linux en el desarrollo *software*
> *Mateo Valero*

En este capítulo se introducen los aspectos generales del trabajo, abordando los antecedentes del proyecto, su contexto de desarrollo, sus objetivos y la visión general del proyecto.

## 1.1. Antecedentes

Las tecnologías de fabricación de circuitos integrados han avanzado de tal forma que es posible la implementación de un sistema de computación completo, y personalizado según el dominio de aplicación, en un único SoC [2]. La evolución está guiada por la Ley de Moore [3] y por aproximaciones More than Moore [4]. Por ello, los productos electrónicos contienen núcleos microprocesadores y memorias de primer nivel (caché) integrados. Existen numerosos ejemplos de productos en el mercado actual. La mayoría de ellos utilizan arquitecturas con juego de instrucciones (ISA) propietarias, como es el caso de Intel, AMD o ARM. Estas ISAs están protegidas por las leyes de propiedad intelectual.

En arquitectura de computadores, a nivel de ISA, es necesario distinguir claramente una ISA Complex Instruction Set Computing (CISC) de una arquitectura Reduced Instruction Set Computing (RISC). Una de las principales diferencias está en la sencillez de las instrucciones, y un factor de distinción es el número de instrucciones. Una ISA CISC contiene un número considerable de instrucciones que realizan cálculos complejos, mientras que una arquitectura RISC posee un conjunto de instrucciones reducido y mucho más simples. A modo de ejemplo, la ISA base de RISC-V contiene 40 instrucciones frente a las más de 150 instrucciones del Intel 80386.

RISC-V ofrece un conjunto de instrucciones de código abierto que facilita la personalización del SoC y, por tanto, aportará soluciones eficientes para el contexto de computación en que se utiliza. RISC-V incluye distintos formatos de instrucciones (R-Type, I-Type, S-Type y U-Type) cuya característica principal es la regularidad de la posición de los registros, los inmediatos se tratan como valores con signo e incluye otros aspectos que simplifican la implementación *hardware*. RISC-V incluye una especificación base de enteros de 32 bits (RV32I) y distintas

extensiones para dar soporte a versiones de 64 bits, operaciones de multiplicación y división, operaciones atómicas, operaciones con punto flotante (32 y 64 bits), instrucciones comprimidas, manipulación de bits, operaciones sobre vectores y otro tipo de operaciones especiales. La especificación completa del juego de instrucciones está disponible en [5].

A diferencia de ISA propietarias, RISC-V permite extender las instrucciones para dar soporte a nuevos problemas. Por ejemplo, en [6] se explica la extensión del juego de instrucciones para dar soporte a Criptografía Post Cuántica (PQC). Implementa algunas instrucciones que permiten acelerar los algoritmos PQC que están soportados por un acelerador fuertemente acoplado en el cauce de ejecución de las instrucciones.

En el proyecto OASIS[1] se plantea la adopción de la arquitectura RISC-V para aplicaciones que requieren altas demandas de cómputo, particularmente en sistemas empotrados en las que RISC-V sirve de acceso a motores de aceleración *hardware* para IA y procesamiento de señales (DSP). La filosofía abierta de RISC-V permite a los usuarios diseñar nuevas instrucciones, facilitando su difusión en distintos sectores de la industrial. Esta adaptabilidad, combinada con un alto nivel de personalización aporta soluciones eficientes en términos de energía habilitando la integración con aceleradores especializados.

En el caso de soluciones de altas prestaciones empotradas, la flexibilidad de RISC-V permite ser utilizada en distintos dispositivos, desde sensores hasta servidores.

Una de las aplicaciones de OASIS es el desarrollo de nuevas técnicas de diagnóstico y de terapia. En este sentido, se concentra en crear una solución de neurodiganóstico en tiempo real para la ayuda en la cirugía de tumores cerebrales. La idea que subyace en la definición es la incorporación de arquitecturas basadas en RISC-V adaptables a los problemas a abordar.

RISC-V [7], es la quinta iteración dentro de una serie de proyectos nacidos en la Universidad de Berkeley (RISC) en el año 2010, como un ISA de estándar abierto, ofrece flexibilidad en el diseño, permitiendo la creación de extensiones aparte de las instrucciones oficiales, personalizadas y adaptadas a las necesidades específicas de cada aplicación en función del rendimiento y la eficiencia energética necesarias [8]. La ausencia de pagos de licencia fomenta la exploración y la creación rápida de prototipos, tanto en hábitos académicos como empresariales, impulsando la innovación y permitiendo a los diseñadores un control granular sobre la funcionalidad de su procesador. Sin embargo, debido a la inmensa dificultad que representa migrar de una arquitectura a otra [9], solo las empresas con importantes recursos que tienden a controlar toda la cadena de sus productos, desde el silicio hasta el firmware y el software, se han dado cuenta del potencial de RISC-V cuya flexibilidad es atractiva para desarrollar desde ordenadores personales e *Internet of Things* (IoT) hasta *High Performance Computing* (HPC) y aceleradores de Inteligencia Artificial (IA) [10]. En cualquier caso, hay una previsión de crecimiento de las soluciones basadas en RISC-V para distintos tipos de aplicaciones, tal como se muestra en la figura 1.1.

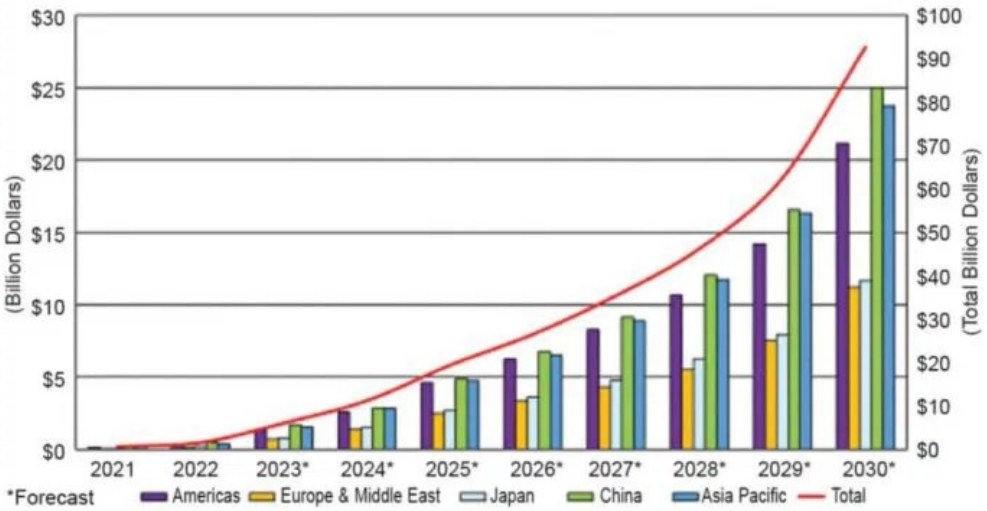
El desarrollo de este trabajo permite explorar y validar la integración de aceleradores *hardware* en la arquitectura RISC-V, mejorando el rendimiento y la eficiencia de aplicaciones dedicadas. Tiene como resultado un flujo de trabajo que da lugar a soluciones escalables enfocadas a la aceleración de aplicaciones complejas [11], que culmina en una implementación final, ya sea en FPGA o en ASIC [12]. El enfoque del TFG incluye aspectos metodológicos, de desarrollo arquitectural y de implementación.

[1] Proyecto del Plan Nacional de I+D, convocatoria de 2024, desarrollado conjuntamente por la Universidad de Las Palmas de Gran Canaria (IUMA), la Universidad de Castilla-La Mancha, Universidad Politécnica de Madrid y Universidad de Cantabria.

## RISC-V SoC regional forecasts

Here are the regional market share forecasts for RISC-V SoCs:

- **Americas:** Revenue of $1.3 billion in 2023, with market share increasing slightly from 2.6% to 2.7% by 2030
- **Europe & EMEA:** Market share growing from 7.4% in 2023 to 9.9% by 2030
- **Japan:** Market share expanding from 7.1% in 2023 to 18% by 2030
- **China:** Market share starting at 33.8% in 2023 and only slightly increasing to 34% by 2030
- **Asia Pacific:** Market share decreasing from 45.5% in 2023 to 32.6% by 2030




*Total RISC-V SoC regional revenues 2021-2030 (Source: The SHD Group, January 2024)*

The RISC-V market's application segment shares are projected to change as follows by 2030:

- **Industrial:** Remains relatively stable, from 2.6% in 2023 to 2.7%
- **Automotive:** Increases from 7.4% in 2023 to 9.9%
- **Networking:** Grows significantly from 7.1% in 2023 to 18%
- **Computer:** Slightly declines from 33.8% in 2023 to 34%
- **Consumer:** Decreases from 45.5% in 2023 to 32.6%
- **Other:** Drops from 3.6% in 2023 to 2.6%

Figura 1.1: Evolución de la penetración de RISC-V en distintos mercados y regiones.

## 1.2. RISC-V

RISC-V es la evolución del concepto RISC introducido en la década de los 80 por la Universidad de Berkeley. Se trata de la quinta versión de la arquitectura, de ahí su numeración en números romanos. El término RISC-V hace referencia a distintos conceptos: la propia ISA, la comunidad que se encarga de desarrollar la arquitectura y el organismo que lo integra (RISC-V International) y las distintas implementaciones realizadas de la ISA base y sus extensiones, ya sean de código abierto o propietarias. Cada significado se pude inferir por el contexto de su uso, aunque en este trabajo, nos centraremos principalmente en las implementaciones realizadas de la arquitectura.

Es importante destacar el papel motor de la especificación RISC-V puesta en la comunidad para su adopción. En este caso, la ISA de RISC-V ha permitido abrir a la innovación para implementar nuevas arquitecturas que permitan cubrir las nuevas necesidades de computación, sin tener a la vista el cuello de botella que supone una especificación propietaria y cerrada.

La comunidad facilita la rápida adopción, contribuyendo con modelos colaborativos, muchos de ellos heredados del desarrollo *software*. Sin embargo, para la utilización de la ISA de RISC-V es preciso tener en cuenta distintos niveles de desarrollo, heterogéneos, sobre tecnologías ya existentes, que han contribuido a la implementación final de la arquitectura. En muchos casos, se han adoptado esquemas de procesamiento en desuso (por ejemplo la inclusión de *arrays* sistólicos inicialmente utilizados en 1949 en el ordenador Colossus Mark II en 1944 y formalizado por H. T. Kung and Charles E. Leiserson en 1979 [13]. En este trabajo se hará uso de este tipo de arquitecturas para la implementación de aceleradores *hardware*.

En otros casos se han creado nuevas arquitecturas optimizadas para los retos de computación introducidos por la inteligencia artificial, como el caso de la computación de redes neuronales profundas (*Deep Neural Networks* (DNN)). Existen en la literatura numerosas aplicaciones en este sentido. Durante el desarrollo del trabajo se explican distintos ejemplos de interés.

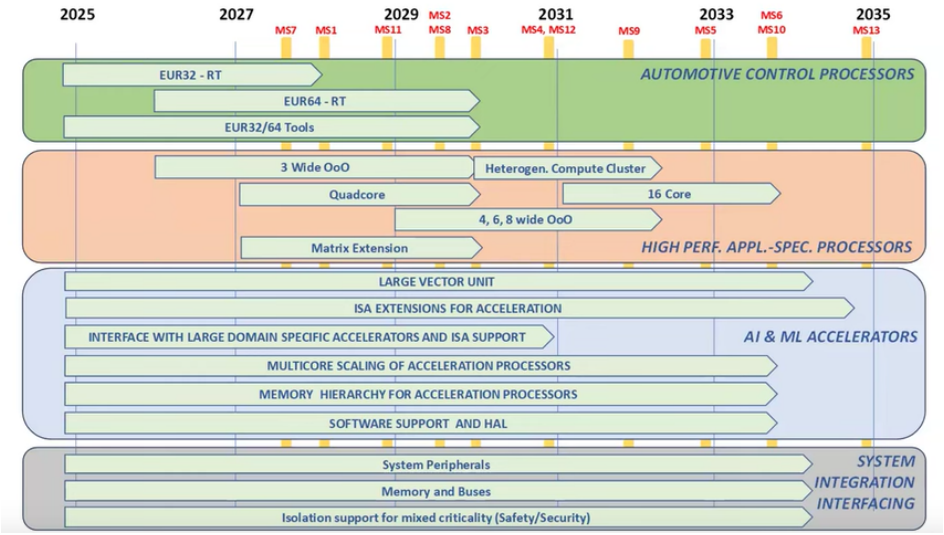
RISC-V presenta en la actualidad un interés clave en el desarrollo del microprocesador europeo en diferentes ámbitos de actuación. En la figura 1.2 se presenta la hoja de ruta prevista para la presencia masiva de RISC-V en distintas soluciones.

En términos de desarrollo de *cores* con alto interés, el consorcio OpenHW [14] está liderando la producción de distintos núcleos procesadores que distribuye como código RTL en SystemVerilog. Cada vez este tipo de núcleos procesadores son utilizados en la industria. Un ejemplo de su utilización en el sector de la automoción se muestra en la figura 1.3.

Ello se debe a que se aporta un entorno completo de utilización, incluyendo el entorno de diseño *hardware* (IPs, entorno de verificación, *scripts* de síntesis, restricciones, etc), y entorno de desarrollo *software*: compiladores, herramientas de depurado, entorno de desarrollo *baremetal* y RTOS y sistemas operativos tipo Linux 1.4.

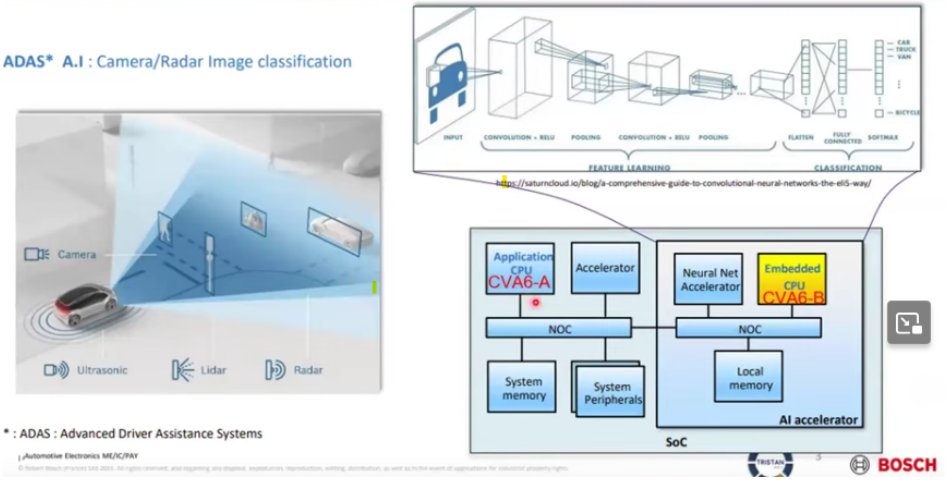
En [15] se indica que, en general las CPUs, orientadas principalmente a dar un buen rendimiento en aplicaciones de control, presentan limitaciones para aplicaciones de IA, especialmente para realizar los procesos de inferencia. Ello se debe a que las CPUs están diseñadas para ejecutar código aleatorio, o en el mejor de los casos realizar cálculo vectorial. Sin embardo presentan limitaciones en el ancho de banda para la carga y almacenamiento de datos.

Los enfoques que pretenden utilizar RISC-V para la IA deben incluir un motor matricial independiente, lo que introduce problemas de partición de la aplicación. La innovación adecuada para abordar la IA es la creación de



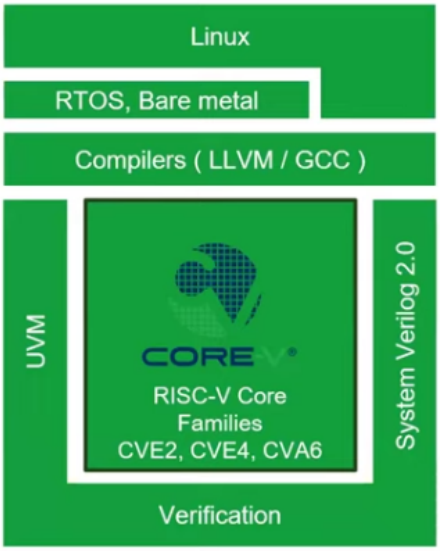

Fuente: https://www.youtube.com/watch?v=azIk6dGWrOU

Figura 1.2: Hoja de ruta de RISC-V en Europa




Fuente: https://www.youtube.com/watch?v=azIk6dGWrOU

Figura 1.3: Utilización de RISC-V para aplicaciones en automoción.




Fuente: https://www.youtube.com/watch?v=azIk6dGWrOU

Figura 1.4: OpenHW *stack*

arquitecturas que estén intrínsecamente optimizadas para problemas de $matriz\ x\ matriz$ o $tensor\ x\ tensor$, y que rompan la dependencia centrada únicamente en la CPU, la caché de memoria y los *pipelines* especulativos fuera de orden.

Una de las ventajas claras de RISC-V permite bloques aceleradores, encapsulados en un procesador de control convencional, instrucciones tensoriales específicas. Esa flexibilidad de RISC-V permite desarrollar instrucciones personalizadas para cargas de trabajo específicas. También es interesante estandarizar y ampliar las especificaciones para compartir los costes de mantenimiento de los compiladores, las cadenas de herramientas y las bibliotecas.

## 1.3. Objetivos

El objetivo de este proyecto es el estudio de los mecanismos necesarios para la inclusión de extensiones dedicadas en el juego de instrucciones RISC-V y su implementación en núcleos RISC-V disponibles. Se estudiarán soluciones fuertemente acopladas en el flujo de ejecución de las instrucciones de RISC-V (*tightly-coupled accelerators*) y soluciones desacopladas, conectadas a través de las interfaces estándar del procesador (interfaces AXI4MM por ejemplo). Finalmente, se integra el conocimiento adquirido en la implementación en la ruta de procesamiento de la ruta de datos del procesador.

Este objetivo general queda desglosado en los siguientes objetivos operativos:

O1 Realizar un estudio de la arquitectura RISC-V y presentar las distintas estrategias para incluir aceleración *hardware*.

O2 Realizar el prototipado de algunas implementaciones RISC-V.

O3 Estudiar los mecanismos de integración de extensiones de instrucciones con RISC-V.

O4 Estudiar los mecanismos de aceleración de aplicaciones dedicadas utilizando RISC-V y aceleradores específicos.

O5 Realizar la implementación y prototipado.

O6 Documentar del trabajo realizado.

## 1.4. Organización de la memoria

La organización de esta memoria es la siguiente. En este capítulo se introduce la necesidad del proyecto, definiendo los antecedentes del proyecto, su contexto de desarrollo y los objetivos definidos. En el capítulo 2 se da una visión general de la arquitectura ISA de RISC-V, explicando la arquitectura ISA (Registros, formatos de instrucciones, ...). Tal como se ha indicado, en RISC-V es posible extender el juego de instrucciones de tal forma que se genera una arquitectura eficiente en función de la aplicación. Los mecanismos necesarios se explican en el capítulo 3. En el capítulo 4 se presentan distintas arquitecturas relacionadas con los aceleradores *hardware*, poniendo énfasis en los mecanismos de comunicación de datos y su integración en el flujo de ejecución de las instrucciones. El capítulo 5 muestra el flujo de integración de distintos tipos de aceleración *hardware* y presenta los resultados obtenidos. Finalmente en el capítulo 6 se aportan las conclusiones del trabajo y los trabajo futuros a abordar.

# Capítulo 2

# Arquitectura RISC-V

> RISC-V es una ISA estándar que sirve como base para el diseño de microprocesadores
> *RISC-V International*

La idea principal de RISC-V es definir un ISA universal que permita:

- Estar presente en todo el rango de los sistemas de procesamiento, desde los microcontroladores más pequeños para sistemas empotrados hasta las supercomputadoras más potentes.
- Tener un amplio soporte tanto de paquetes y programas como de lenguajes de programación.
- Poder implementarse en todo tipo de tecnologías: desde FPGA, ASIC, chips personalizados e incluso en tecnologías aún no inventadas.
- Eficiencia para todo tipo de microarquitecturas con distintos tipos de *pipelines*, en-orden, desacopladas, fuera-de-orden; instrucciones secuenciales o superescalar.
- Especializarse para su utilización como base en aceleradores personalizados, cada vez más importantes en distintos ámbitos de procesamiento, como es el caso del *Edge-processing* (procesamiento en el borde).

## 2.1. Conjunto de instrucciones relacionadas

Aunque es más sencillo hablar de RISC-V como un conjunto de instrucciones único, RISC-V en realidad, es una familia de ISA relacionadas, en la actualidad existen cuatro ISA base. Cada uno de estos conjuntos de instrucciones relacionados se caracteriza por tres facetas:

1. Ancho de sus registros para enteros.
2. Tamaño correspondiente del espacio de direcciones.

3. Número de dichos registros.

Existen dos variantes de enteros principales, RV32I y RV64I que proporcionan espacios de direcciones de 32 o 64 bits respectivamente. Además, en paralelo podemos encontrar las variantes 'E', RV32E y RV64E derivadas de las anteriores, que dan soporte a microcontroladores y que cuentan con la mitad de registros de enteros. Finalmente, se esboza una futura variante, RV128I con un espacio de direcciones de 128 bits. Esta familia de conjuntos de instrucciones utiliza una representación en complemento a dos para los valores enteros con signo [16].

## 2.2. Excepciones, *traps* e interrupciones

El término excepción es empleado para referirse a una condición inusual que ocurre en tiempo de ejecución y está asociada con una instrucción en el *hart* actual de RISC-V. En cambio, el término interrupción se utiliza para referirnos a un evento asíncrono externo que puede hacer que un *hart* experimente un traslado de control inesperado. Finalmente, el término *trap* o trampa hace referencia a la delegación de control a un manejador de *traps* provocado tanto por una excepción como por una interrupción.

- ***Trap* contenida**: Es visible y manejada por el *software* que se ejecuta dentro del entorno de ejecución.
- ***Trap* solicitada**: Es una excepción síncrona que llama al entorno de ejecución, solicitando una acción en nombre del *software* dentro del entorno de ejecución.
- ***Trap* invisible**: Es manejada de forma transparente por el entorno de ejecución, tras esto la ejecución regresa a su flujo normal.
- ***Trap* fatal**: Representa un fallo fatal y causa que el entorno de ejecución finalice la ejecución.

## 2.3. Extensiones del conjunto de instrucciones

Como antes fue mencionado RISC-V es un conjunto de instrucciones diseñado con la personalización y la especialización en mente. Para cumplir con este requisito, se utilizan extensiones al juego de instrucciones, que se clasifican en tres categorías:

- ***standard***: Definidas por "RISC-V Internacional". Todas aquellas extensiones no definidas por "RISC-V Internacional" son *no* estándar.
- ***reserved***: No están definidas, pero se reservan para su futura estandarización.
- ***custom***: Las codificaciones *custom* no se utilizarán por parte de "RISC-V Internacional" para extensiones estándar y quedan disponibles para extensiones propietarias.

El conjunto de instrucciones base se referencia con la letra "I" y el prefijo RV32 o RV64 dependiendo del ancho de los registros para enteros, como se expuso con anterioridad. Dentro de esta extensión hay instrucciones para operaciones computacionales, *load/store* de enteros y diferentes controles de flujo.

Para permitir el desarrollo de *software* general existe un conjunto de extensiones estándar que dan soporte a diferentes operaciones comunes:

- M: Multiplicación y División.
- F: Punto Flotante de Precisión Simple.
- D: Punto Flotante de Precisión Doble.
- A: Atómico.
- C: Instrucciones Comprimidas.
- V: Vector.
- S: Modo supervisor.

Por ejemplo, para referirse a una ISA en concreto, que tenga soporte para las extensiones M, A, F, D, se concatena cada una de las letras: RV32IMAFD, este conjunto en particular también se puede representar por la letra "G", RV32G.

Mientras que otros conjuntos de extensiones son diseñados con la idea de ser una entidad monolítica que va cambiando con el tiempo, RISC-V, en cambio, está ideado con la idea de mantener la base y las extensiones estándar sin cambios e ir añadiendo instrucciones mediante extensiones opcionales. Esto es principalmente debido a que es poco probable que una instrucción para una aplicación específica tenga un impacto positivo significativo en la computación general, sin embargo, esa misma instrucción para esa aplicación concreta puede tener un impacto dramático en su ejecución.

## 2.4. Formato de instrucciones RISC-V

Entendiendo la codificación del conjunto de instrucciones RV32I permite comprender qué estructura general sigue el resto de la familia. En RV32I las instrucciones se codifican con palabras 32 bits alineadas cada 4 bytes y en formato *little endian*. Dentro de RV32I se pueden diferenciar seis principales formatos básicos de instrucciones dentro de las cuales hay realmente cuatro formatos básicos y dos derivados (B y J), como podemos ver en la figura 2.1:

Cada uno de estos seis formatos agrupan instrucciones con diferente funcionalidad:

- Tipo R: operaciones aritméticas o entre registros.
- Tipo I: operaciones con inmediatos, *loads* y genéricas.
- Tipo S: operaciones de registros a memoria (*stores*).
- Tipo B: control de flujo, *branches*.
- Tipo U: instrucciones con los 20 bits superiores (`lui` y `auipc`).
- Tipo J: saltos incondicionales.

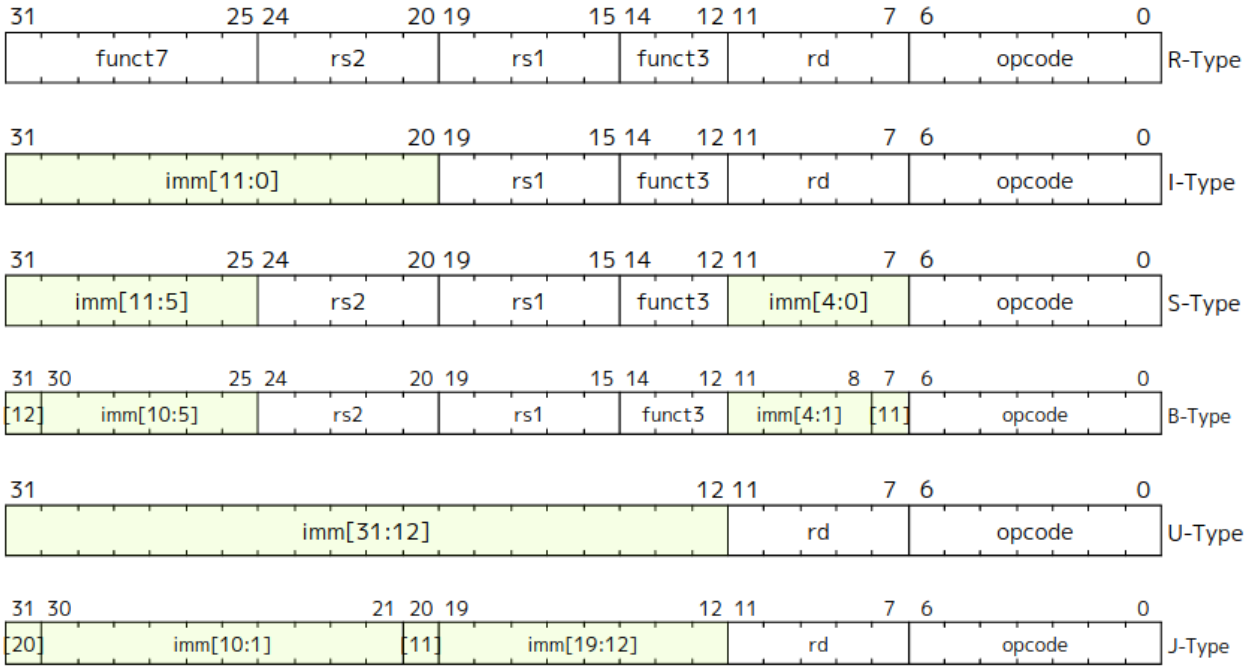



Figura 2.1: Formatos básicos y las variantes de los tipos B y J [16, p. 25]

En la figura 2.1 se puede ver como dentro de los 32 bits en los que se codifican las instrucciones se mantienen los campos en la misma posición a través de los diferentes formatos. El `opcode` se mantiene al principio en todos los formatos en `[0:6]`, a continuación los diferentes registros: `rd` (registro de destino), `rs1` (registro fuente 1) y `rs2` (registro fuente 2) si se utilizan están en `[7:11]`, `[15:19]` y `[20:24]`, respectivamente. Por último queda `funct3` y `funct7` que junto al `opcode` codifican en jerarquía (`opcode` → `funct3` → `funct7`) qué instrucción se ha de llevar a cabo, estos se ubican en `[12:14]` y `[25:31]`, respectivamente. Por ejemplo:

| opcode | | funct3 | | funct7 |
|---|---|---|---|---|
| 0110011 | → | 000 | → | 0100000 |
| tipo R | | add/sub | | sub |

Figura 2.2: Ejemplo de codificación de la instrucción `add`

RV32I está formado por 37 instrucciones, que han sido reducidas respecto a las versiones anteriores de RISC-V, pasando a extensiones opcionales, los siguientes grupos de instrucciones:

- Llamadas al entorno: `ecall`, `ebreak` (extensión **Zicsr**)
- Operaciones con Registros de Control y Estado (CSR): `csrrw`, `csrrs`, `csrrc`, `csrrwi`, `csrrsi`, `csrrci` (extensión **Zicsr**)

- Sincronización: `fence`, `fence.i` (extensión **Zifencei**)

### 2.4.1. Tipo R

Las instrucciones de tipo R realizan operaciones entre registros: `rs1`, `rs2` y escriben el resultado en `rd`, según la figura 2.3 y la tabla 2.1.

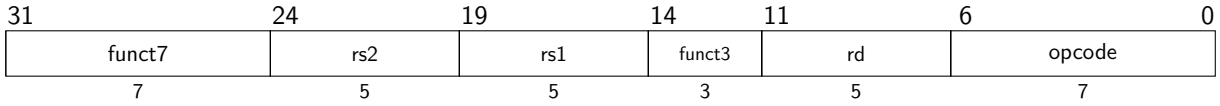



Figura 2.3: Formato de las instrucciones de tipo R

| **Instrucción** | **Descripción** | **Semántica** |
|---|---|---|
| `add` | Suma entera | $x[rd] = x[rs1] + x[rs2]$ |
| `sub` | Resta entera | $x[rd] = x[rs1] - x[rs2]$ |
| `sll` | Desplazamiento lógico a la izquierda | $x[rd] = x[rs1] \ll x[rs2]$ |
| `slt` | Comparación firmado menor | $x[rd] = x[rs1] <_s x[rs2]$ |
| `sltu` | Comparación sin signo menor | $x[rd] = x[rs1] <_u x[rs2]$ |
| `xor` | OR exclusivo bit a bit | $x[rd] = x[rs1] \wedge x[rs2]$ |
| `srl` | Desplazamiento lógico a la derecha | $x[rd] = x[rs1] \gg_u x[rs2]$ |
| `sra` | Desplazamiento aritmético a la derecha | $x[rd] = x[rs1] \gg_s x[rs2]$ |
| `or` | OR bit a bit | $x[rd] = x[rs1] \mid x[rs2]$ |
| `and` | AND bit a bit | $x[rd] = x[rs1] \,\&\, x[rs2]$ |

Tabla 2.1: Instrucciones de tipo R en RV32I

### 2.4.2. Tipo I

Las instrucciones de tipo I, involucran un inmediato de 12 bits en una gran variedad de instrucciones, entre las cuales se destacan 3 grupos, según la figura 2.4 y las tablas 2.2 y 2.3.

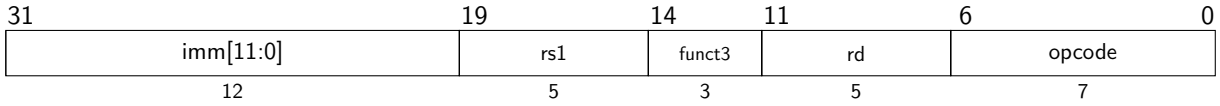



Figura 2.4: Formato de las instrucciones de tipo I

**Instrucción de salto relativo (`jalr`):**

- `jalr` realiza un salto indirecto: el nuevo Contador de Programa (PC) es $rs1 + imm_{11:0}$ (con el bit 0 forzado a cero), y guarda PC+4 en `rd`.

**Instrucciones de carga:**

| Instrucción | Descripción | Semántica |
|---|---|---|
| `lb rd, imm(rs1)` | Carga un byte con signo | $x[rd] = sext(M[x[rs1] + sext(offset)][7:0])$ |
| `lh rd, imm(rs1)` | Carga un *halfword* con signo | $x[rd] = sext(M[x[rs1] + sext(offset)][15:0])$ |
| `lw rd, imm(rs1)` | Carga una *word* | $x[rd] = sext(M[x[rs1] + sext(offset)][31:0])$ |
| `lbu rd, imm(rs1)` | Carga un *byte* sin signo | $x[rd] = M[x[rs1] + sext(offset)][7:0]$ |
| `lhu rd, imm(rs1)` | Carga un *halfword* sin signo | $x[rd] = M[x[rs1] + sext(offset)][15:0]$ |

Tabla 2.2: Instrucciones de tipo I de carga (load) en RV32I

**Instrucciones de tipo I con inmediatos:**

| Instrucción | Descripción | Semántica |
|---|---|---|
| `addi rd, rs1, imm` | Suma inmediata | $x[rd] = x[rs1] + sext(immediate)$ |
| `slti rd, rs1, imm` | Compara con signo y establece | $x[rd] = x[rs1] <_s sext(immediate)$ |
| `sltiu rd, rs1, imm` | Compara sin signo y establece | $x[rd] = x[rs1] <_u sext(immediate)$ |
| `xori rd, rs1, imm` | XOR inmediato | $x[rd] = x[rs1] \wedge sext(immediate)$ |
| `ori rd, rs1, imm` | OR inmediato | $x[rd] = x[rs1] \mid sext(immediate)$ |
| `andi rd, rs1, imm` | AND inmediato | $x[rd] = x[rs1] \,\&\, sext(immediate)$ |
| `slli rd, rs1, shamt` | Desplazamiento lógico a la izquierda con inmediatos | $x[rd] = x[rs1] \ll shamt$ |
| `srli rd, rs1, shamt` | Desplazamiento lógico a la derecha con inmediatos | $x[rd] = x[rs1] \gg_u shamt$ |
| `srai rd, rs1, shamt` | Desplazamiento aritmético a la derecha con inmediatos | $x[rd] = x[rs1] \gg_s shamt$ |

Tabla 2.3: Instrucciones de tipo I de operaciones inmediatas en RV32I

### 2.4.3. Tipo S

El formato S para instrucciones de almacenamiento (*stores*), tabla 2.4. Divide el inmediato en dos trozos según la figura 2.5.

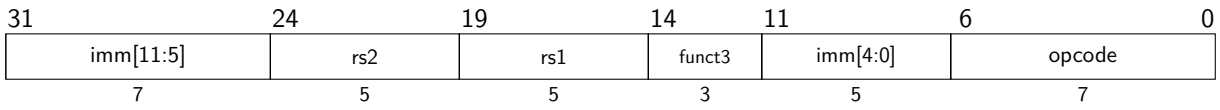



Figura 2.5: Formato de las instrucciones de tipo S

| Instrucción | Descripción | Semántica |
|---|---|---|
| `sb rs2, imm(rs1)` | Store Byte | $\texttt{M}[x[rs1] + sext(offset)] = x[rs2][7:0]$ |
| `sh rs2, imm(rs1)` | Store Halfword | $\texttt{M}[x[rs1] + sext(offset)] = x[rs2][15:0]$ |
| `sw rs2, imm(rs1)` | Store Word | $\texttt{M}[x[rs1] + sext(offset)] = x[rs2][31:0]$ |

Tabla 2.4: Instrucciones de tipo S (stores) en RV32I

### 2.4.4. Tipo B

Para saltos condicionales (*branches*), tabla 2.5. El inmediato de 13 bits se divide en varios campos, figura 2.6:

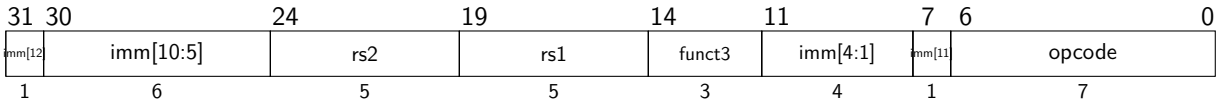



Figura 2.6: Formato de las instrucciones de tipo B

| Instrucción | Descripción | Semántica |
|---|---|---|
| `beq rs1, rs2, offset` | Salto si iguales | $\texttt{if}(rs1 == rs2)\ pc+ = sext(offset)$ |
| `bne rs1, rs2, offset` | Salto si distintos | $\texttt{if}(rs1 \neq rs2)\ pc+ = sext(offset)$ |
| `blt rs1, rs2, offset` | Salto si menor (con signo) | $\texttt{if}(rs1 <_s rs2)\ pc+ = sext(offset)$ |
| `bge rs1, rs2, offset` | Salto si mayor o igual (con signo) | $\texttt{if}(rs1 \geqslant_s rs2)\ pc+ = sext(offset)$ |
| `bltu rs1, rs2, offset` | Salto si menor (sin signo) | $\texttt{if}(rs1 <_u rs2)\ pc+ = sext(offset)$ |
| `bgeu rs1, rs2, offset` | Salto si mayor o igual (sin signo) | $\texttt{if}(rs1 \geqslant_u rs2)\ pc+ = sext(offset)$ |

Tabla 2.5: Instrucciones de tipo B (branches) en RV32I

### 2.4.5. Tipo U

Las instrucciones U de operaciones especiales, tabla 2.6, manejan inmediatos de 20 bits alineados a la parte más significativa, figura 2.7.

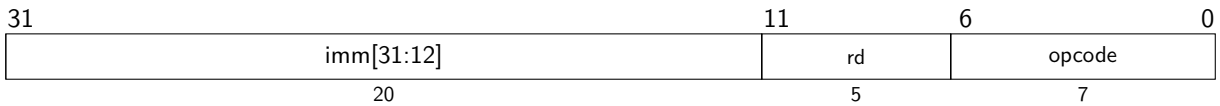



Figura 2.7: Formato de las instrucciones de tipo U

| Instrucción | Descripción | |
|---|---|---|
| `lui rd, imm` | Carga el inmediato de 20 bits en los bits altos de `rd`, dejando los 12 bits bajos a cero (`rd` $\leftarrow$ $\texttt{imm}_{31:12} \ll 12$) | $\texttt{x}[rd] = sext(immediate[31:12] \ll 12)$ |
| `auipc rd, imm` | Calcula PC + ($\texttt{imm}_{31:12} \ll 12$) y almacena el resultado en `rd` (`rd` $\leftarrow$ PC + ($\texttt{imm}_{31:12} \ll 12$)), útil para PC relativo | $\texttt{x}[rd] = pc + sext(immediate[31:12] \ll 12)$ |

Tabla 2.6: Instrucciones de tipo U en RV32I

### 2.4.6. Tipo J

El formato J codifica saltos con enlace (jal) con un inmediato de 21 bits, figura 2.8.

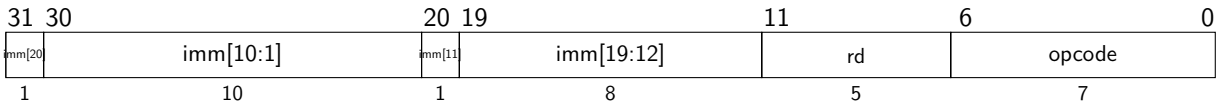



Figura 2.8: Formato de las instrucciones de tipo J

- `jal` (tipo J) efectúa un salto PC relativo: el destino se calcula como PC + $\text{offset}_{20:1} \times 2$ y el valor de retorno (PC+4) se almacena en `rd`.

## 2.5. Registros en RISC-V

En la tabla 2.7 se especifican los registros para RV32I no privilegiado. En RV32I los 32 registros tienen cada uno un ancho de 32 bits (`XLEN=32`). El registro 0x tiene en todo momento sus bits a 0. Los registros de propósito general `x1-x32` guardan valores que las diferentes instrucciones interpretan como una colección de valores booleanos o enteros en complemento a dos con signo o enteros sin signo. Hay un registro adicional que guarda la dirección de la instrucción que está en ejecución, `pc`.

| Registro | Alias | Registro | Alias | Registro | Alias |
|---|---|---|---|---|---|
| x0 | zero | x11 | a1 | x22 | s6 |
| x1 | ra | x12 | a2 | x23 | s7 |
| x2 | sp | x13 | a3 | x24 | s8 |
| x3 | gp | x14 | a4 | x25 | s9 |
| x4 | tp | x15 | a5 | x26 | s10 |
| x5 | t0 | x16 | a6 | x27 | s11 |
| x6 | t1 | x17 | a7 | x28 | t3 |
| x7 | t2 | x18 | s2 | x29 | t4 |
| x8 | s0 / fp | x19 | s3 | x30 | t5 |
| x9 | s1 | x20 | s4 | x31 | t6 |
| x10 | a0 | x21 | s5 | pc | contador de programa |

Tabla 2.7: Registros en modo no privilegiado en RV32I

El conjunto de Instrucciones base no detalla usos específicos para los registros, pudiéndose utilizar cualquiera de los registros para cualquier uso aunque normalmente, tal como está definida en la *Application Binary Interface* (ABI), se utiliza el registro `x1` para guardar la dirección de retorno, `x5` como alternativa a `x1` (*alternate link register*) y `x2` como puntero a la pila.

### 2.5.1. Registros de CSR

Los CSR se emplean principalmente en modos privilegiados, también resultan fundamentales en el código de usuario para la gestión de contadores, temporizadores y del estado de la unidad de punto flotante; la extensión Zicsr de la arquitectura RISC-V reserva un espacio de direcciones con capacidad para 4096 CSR por cada hart; todas las instrucciones privilegiadas se codifican bajo el opcode SYSTEM y se organizan en dos familias principales: aquellas que posibilitan la lectura, modificación y escritura de los CSR definidos en Zicsr y aquellas que comprenden el resto de operaciones de control del hart, ajenas directamente a los registros de control y estado.

## 2.6. Ejecución de instrucciones

La especificación oficial de RISC-V define la semántica de cada instrucción oficial, incluyendo así, la codificación de los bits, efectos sobre registros y memoria, comportamiento según el nivel de privilegios, entre otros. Sin embargo, no se específica ninguna estrategia de microarquitectura.

El manual de referencia explica en detalle qué debe ocurrir cuando se ejecuta una instrucción, por ejemplo, `ADD x5, x1, x2`, pero deja completamente abierto cómo el *hardware* implementa esa ejecución: un solo ciclo o multiciclo, número de etapas del *pipeline*, predicción de saltos, tamaño de la memoria caché, etc.

De este modo se garantiza la compatibilidad del *software* dejando total libertad al diseñador para tomar decisiones. De esta libertad para tomar decisiones surge un ecosistema con implementaciones RISC-V muy diferentes, diseñadas con enfoques en algunos casos, completamente diferentes. A continuación se muestran algunos ejemplos.

## 2.7. Modos de ejecución

Del mismo modo que en otras arquitecturas, RISC-V define una jerarquía de modos de ejecución o niveles de privilegios para aislar los diferentes programas entre sí y prevenir que código con un menor nivel de fiabilidad interfiera con procesos críticos. Este nivel de privilegio determina qué acciones puede llevar a cabo un hilo *hardware* o *hart*, siempre que se esté en ejecución lo hará bajo un privilegio concreto codificado en uno o varios CSR [17]. Los *harts* normalmente ejecutan código en modo-U hasta que un *trap* (una llamada del modo supervisor o la interrupción de un *timer*) fuerza el cambio al sistema de manejo de ese *trap* que llegado el momento regresará al contexto anterior. Los *traps* que incrementan el nivel de privilegios se determinan *traps* verticales, mientras que los que mantienen el nivel de privilegios se determinan *traps* horizontales [18, secc. 1.2].

| **Modo de Privilegio** | **Codificación** | **Uso** | **Permisos** |
|---|---|---|---|
| Usuario/aplicación | 00 | Código de aplicación | No puede ejecutar instrucciones privilegiadas, acceder a CSR de supervisor o máquina y tiene mapeos de memoria virtual limitados. |
| Supervisor | 01 | *Kernel* | Puede configurar la memoria virtual, gestionar excepciones/*traps* de Modo Usuario, acceso a CSR de supervisor. |
| *Reservado* | 10 | - | |
| Máquina | 11 | *Firmware* y *bootloader* | Acceso total e irrestricto a todas las instrucciones, memoria y CSR. |

Tabla 2.8: Niveles de Privilegio en RISC-V y permisos [18, cap. 1]

### 2.7.1. Modo máquina

Es el único modo obligatorio, propio de sistemas que no disponen de sistema operativo como sistemas empotrados sencillos. Los *harts* tienen acceso completo a los registros y memoria, son capaces de interactuar y manejar todos los CSR y excepciones a continuación:

#### Registros de modo máquina

- **ISA de máquina (`misa`):** Presenta el ancho de la dirección del procesador (32, 64 o 128 bits) e identifica las extensiones soportadas por la arquitectura.
- **ID de proveedor (`mvendorid`):** Provee el ID del fabricante JEDEC[1] del proveedor del procesador.
- **ID de arquitectura (`marchid`):** Muestra la microarquitectura base, junto a `mvendorid` identifica de manera única la microarquitectura implementada.
- **ID de implementación de máquina (`mimpid`):** Versión de la implementación de la microarquitectura base.
- **ID de hart (`mhartid`):** Presenta el ID del hart que está en ejecución.
- **Estado de máquina (`mstatus` y `mstatush`):** Guarda el estado actual: habilitador global de interrupciones `mie` y `mpie` (valor anterior de `mie`), entre otros:

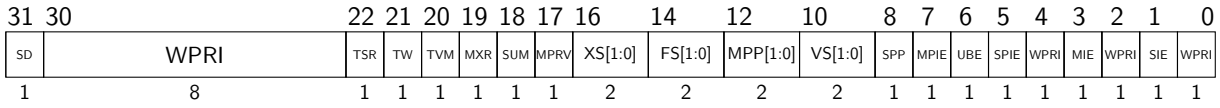



Figura 2.9: Campos del registro `mstatus`.

- **Vector de Interrupciones de Máquina (`mtvec`):** Dirección base de vectores de excepciones, a qué dirección salta el procesador cuando ocurre una excepción.
- **CSR para la delegación de *traps* (`medeleg` y `mideleg`):** El CSR `mideleg` regula qué interrupciones son delegadas al manejador de excepciones de menor privilegio (modo S), donde el índice de la posición del bit habilita la excepción o interrupción correspondiente en el registro `sip`.
- **Interrupciones de Máquina Pendientes (`mip`) y Habilitador de Interrupciones de Máquina (`mie`):** Lista qué interrupciones están pendientes (`mip`) y cuáles puede tomar el procesador o cuáles debe ignorar (`mie`).
- **CSR para la medición:**
  - `mtime`: contador de tiempo real de 64 bits.
  - `mtimecmp`: Causa una excepción cuando `mtime` iguala o excede su valor.
  - `mcounteren` y `scounteren`: controlan la visibilidad de los CSR para la monitorización del rendimiento para el siguiente nivel menos privilegiado.

[1] jedec.org/standards-documents/docs/jep-106ab

- `mcycle`, `minstret`, `mhpmcounter`: 32 CSR para la monitorización del rendimiento.

- **Registro de causa (`mcause`)**:

| Interrupción/Excepción **mcause[XLEN-1]** | Código de Excepción **mcause[XLEN-2:0]** | **Descripción** |
|---|---|---|
| 1 | 0 | *Reservado* |
| 1 | 1 | Interrupción de *software* de Supervisor |
| 1 | 2 | *Reservado* |
| 1 | 3 | Interrupción de *software* de Máquina |
| 1 | 4 | *Reservado* |
| 1 | 5 | Interrupción de temporizador de Supervisor |
| 1 | 6 | *Reservado* |
| 1 | 7 | Interrupción de temporizador de Máquina |
| 1 | 8 | *Reservado* |
| 1 | 9 | Interrupción externa de Supervisor |
| 1 | 10 | *Reservado* |
| 1 | 11 | Interrupción externa de Máquina |
| 1 | 12 | *Reservado* |
| 1 | 13 | *Counter-overflow interrupt* |
| 1 | 14-15 | *Reservado* |
| 1 | ≥16 | *Designated for platform use* |
| 0 | 0 | Dirección de instrucción desalineada |
| 0 | 1 | Fallo de acceso en instrucción |
| 0 | 2 | Instrucción ilegal |
| 0 | 3 | Breakpoint |
| 0 | 4 | Dirección de `Load` desalineada |
| 0 | 5 | Fallo de acceso en `Load` |
| 0 | 6 | Dirección de `Store` desalineada |
| 0 | 7 | Fallo de acceso en `Store` |
| 0 | 8 | Llamada al entorno desde modo U |
| 0 | 9 | Llamada al entorno desde modo S |
| 0 | 11 | Llamada al entorno desde modo M |
| 0 | 12 | Fallo de página en instrucción |
| 0 | 13 | Fallo de página en `Load` |
| 0 | 15 | Fallo de página en `Store` |
| 0 | 16-17 | *Reservado* |
| 0 | 18 | *Software check* |
| 0 | 19 | *Hardware error* |
| 0 | 20-23 | *Reserved* |
| 0 | 24-31 | *Designated for custom use* |
| - | 32-47 | *Reserved* |
| - | 48-63 | *Designated for custom use* |
| - | ≥64 | *Reserved* |

Tabla 2.9: Códigos del registro `mcause` tras un *trap*

- **Valor de excepción de máquina (`mtval`)**: Contiene información adicional sobre las excepciones: dirección, instrucción o cero, en función del tipo de excepción que ocurra.

- **Puntero hacia la configuración de la máquina (`mconfigptr`)**: Guarda la dirección física de una estructura de datos con diferentes datos y configuraciones. Las aplicaciones pueden recorrer esta estructura para averiguar información sobre los *harts*, la plataforma y su configuración.

- **Configuración del entorno de la máquina (`menvcfg`)**: controla diferentes características del entorno de ejecución para modos menos privilegiados que el modo máquina.

- **Configuración de seguridad de la máquina (`mseccfg`)**: registro opcional que controla diferentes opciones relacionadas con la seguridad de extensiones que están por desarrollarse.

### 2.7.2. Modo supervisor

El esquema de protección tratado con anterioridad es útil para dar soporte a sistemas de bajos recursos, sin embargo, presenta problemas para la computación general que se soluciona introduciendo el soporte de memoria virtual basada en páginas.

#### Registros de supervisor

La mayoría de los registros en modo supervisor realizan una función análoga a la que desarrollan los registros que comparten sufijo en modo máquina.

- **Estado del supervisor (`sstatus`):** estado actual del supervisor, figura 2.10.

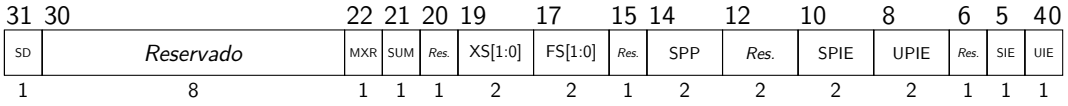



Figura 2.10: Campos del registro `sstatus`.

- **Dirección base del vector de *traps* de supervisor (`stvec`):** guardan la configuración de los vectores de excepciones, dirección base del vector `BASE` y modo del vector `MODE`.
- **Registros de interrupción de supervisor (`sip` y `sie`):** habilitador de interrupciones de supervisor (sip) e interrupciones de supervisor pendientes (sie), respectivamente.
- **Temporizadores y contadores de rendimiento de supervisor:**
  - **Registro de habilitación de contadores (`scounteren`):** controla la disponibilidad de los contadores de monitoreo de rendimiento para el siguiente modo menos privilegiado.
- **Registro de scratch de supervisor (`sscratch`):** registro de libre uso para el manejador de excepciones.
- **Contador de programa de excepciones de supervisor (`sepc`):** apunta a la instrucción que da lugar a la excepción.
- **Registro de causa de supervisor (`scause`):** recibe la causa de la excepción codificada como se muestra en la tabla 2.9.
- **Registro de valor de *trap* de supervisor (`stval`):** contiene información relativa a la excepción para ayudar al programa con el manejo de dicha excepción.
- **Registro de configuración de entorno de supervisor (`senvcfg`):** controla ciertas características del entorno de ejecución en modo usuario.
- **Registro de traducción y protección de direcciones de supervisor (`satp`):** controla el sistema de paginación, tiene el formato mostrados en la figura 2.11 y la tabla 2.11.

**Instrucciones de modo supervisor**

- Instrucción para la gestión de memoria en modo supervisor (`SFENCE.VMA`): Limpia el caché de modo que la unidad de manejo de memoria pueda observar los cambios en la memoria. Presenta multitud de configuraciones para ajustarse a cada contexto de ejecución.

### 2.7.3. Modo usuario

Debido a que no es recomendable ejecutar todo el código de una aplicación en modo de privilegios elevado, se trata de diseñar sistemas en los que la ejecución del código se realice mayormente en modos poco privilegiados, siendo el modo usuario el menos privilegiado. Dependiendo del sistema se implementarán una cierta combinación de modos de los modos de ejecución, comúnmente los indicados de la tabla 2.10.

| Combinación | Descripción |
|---|---|
| M | Sistemas empotrados simples |
| M, U | Sistemas empotrados con protección de memoria |
| M, S, U | Sistemas operativos estilo Unix con memoria virtual |

Tabla 2.10: Tipos de sistemas y modos de ejecución asociados

El código ejecutado bajo modo usuario genera una excepción de instrucción ilegal cuando se utiliza una instrucción o CSR propio de un modo de ejecución con mayores privilegios.

Para regresar al modo anterior de ejecución existen las instrucciones `SRET` y `MRET`. El *software* en modo máquina puede entrar al modo usuario mediante la instrucción `MRET`, tras haber establecido `mstatus.MPP` a la codificación correspondiente según la tabla 2.8.

### 2.7.4. Modo hipervisor

La extensión estándar "H" de RISC-V virtualiza la arquitectura a nivel de supervisor para permitir el alojamiento de sistemas operativos invitados en hipervisores de tipo 1 y tipo 2. Esta extensión introduce el concepto general de virtualización a RISC-V, el modo supervisor tradicional se convierte en modo supervisor extendido por el hipervisor (modo HS), en el cual se ejecuta un hipervisor o un sistema operativo con capacidad de alojamiento de otros sistemas. Una característica importante del modo HS es la segunda etapa de traducción de direcciones que, convierte direcciones físicas del sistema operativo invitado en direcciones físicas de supervisor, virtualizando así tanto la memoria como los dispositivos de E/S mapeados en memoria. El funcionamiento del modo HS es análogo al modo supervisor estándar, sin embargo, incluye tanto instrucciones como registros de control y estado, adicionales que gestionan la nueva capa de traducción y permiten un modo supervisor virtual (modo VS) para los invitados. En el capítulo 3 se elabora el concepto de extensiones en RISC-V, cómo se organizan y su terminología.

## 2.8. Memoria

Cada *hart* en RISC-V dispone de un único espacio de direcciones accesible al *byte*. Cada palabra de memoria (*word*) se define como 32 bits (4 bytes), media palabra (*halfword*) 16 bits (2 bytes), una doble palabra (*doubleword*) son 64 bits (8 bytes) y cuádruple palabra (*quadword*) 128 bits (16 bytes). El espacio de direcciones de memoria es circular, de modo que el byte en la dirección máxima está adyacente al byte en la dirección cero. En concordancia, los cálculos de direcciones que hace el *hardware* ignorar los desbordamientos y continúan hasta el módulo de la posición calculada.

El entorno de ejecución determina cómo se asignan los recursos de *hardware* al espacio de un *hart*. Cada rango de memoria puede: estar libre, contener *memoria principal* o contener uno o más dispositivos de entrada/salida (I/O). Cuando un dispositivo tiene múltiples *harts*, el espacio de memoria de los *harts* pueden ser completamente iguales, completamente diferentes o parcialmente distintos o compartiendo un conjunto de recursos en los mismos o diferentes rangos de direcciones.

### 2.8.1. Memoria virtual

La memoria virtual en RISC-V funciona bajo los mismos principios básicos que en otros conjuntos de instrucciones. Cuando el sistema operativo asigna memoria a un programa este tiene disponible una región continua, aislada del resto de procesos y con la posibilidad de tener mayor capacidad que la memoria física, de memoria virtual. Gracias a la memoria virtual y más concretamente a la unidad de manejo de memoria se abstrae la complejidad inherente de la memoria física, traduciendo direcciones virtuales a direcciones físicas.

De igual modo que en ARM y x86, RISC-V divide la memoria en páginas de tamaño fijo (4 KiB) y utiliza tablas de páginas multinivel. Se asocian números de página virtual (VPN) a números de página físicos (PPN) mediante diferentes esquemas.

El proceso de traducción de direcciones virtuales tiene como base el registro `satp`:

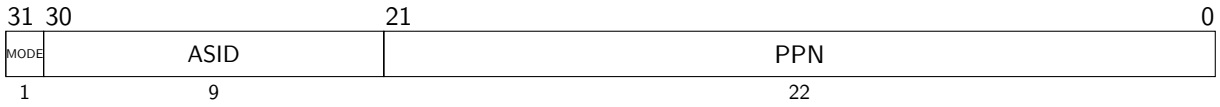



Figura 2.11: Campos del registro `satp`.

En primer lugar, se lee el campo `PPN` para obtener la dirección base física de la tabla de paginación.

1. Se obtiene el número de página físico de la tabla raíz, guardado en el campo `PPN` del registro `satp`.
2. Si `V=0`, `R=0` y `W=1` o si alguno de los bits reservados están establecidos, raise page-fault exception.
3. El PTE es válido. Si `PTE.R=1` o `PTE.X=1`, avanzamos. En caso contrario este PTE es un puntero al siguiente nivel y vamos al paso anterior.
4. Se ha encontrado un PTE leaf, se determina si el acceso a memoria está permitido (PTE.R, PTE.W, PTE.X y PTE.U) dado el nivel de privilegio actual y el valor de los registros SUM y MXR del registro `mstatus`. Si no page-fault.

5. La traducción se ha completado. La dirección física está determinada por:

   - `PA.PO = VA.PO`
   - `PA.PPN[L-1:i] = PTE.PPN[L-1:i]`
   - Donde (para Sv32):
     - `L`: niveles de páginas virtuales.
     - `i`: nivel actual 1, 0.

| ISA | MODE | ASID | Número de página físico (PPN) |
|---|---|---|---|
| **RV32** | **1 bit**<br>Bare/Sv32 | **9 bits** | **22 bits** |
| **RV64** | **4 bits**<br>Bare/Sv39/Sv48<br>/Sv57/Sv64 | **16 bits** | **44 bits** |

Tabla 2.11: Campos del registro `satp` para RV32 y RV64.

### 2.8.2. Protección física de memoria

Para evitar que el modo de ejecución realice modificaciones a diferentes regiones de memoria es posible utilizar la funcionalidad de PMP que permite especificar qué direcciones de memoria puede acceder cada modo de ejecución.

Su funcionamiento se basa en una lista de reglas, donde cada regla defina un área de memoria y qué operaciones están permitidas en ella (leer datos, escribir datos o ejecutar código). Estas reglas incluyen los permisos y direcciones relevantes y se guardan en los siguientes registros:

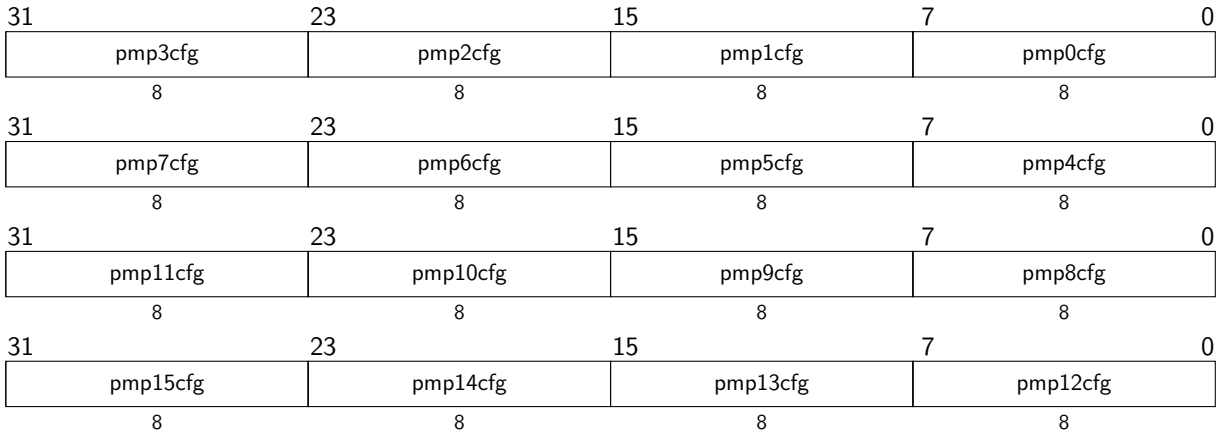



Figura 2.12: Disposición de los registros `pmp{0...15}cfg`.

Los registros PMP son CSR con nombre `pmpaddr0-pmpaddr15`. Cada registro PMP codifica los bits `[33:2]` de una dirección de 34 bits física, figura 2.13.

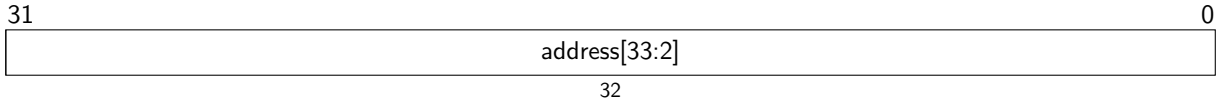



Figura 2.13: Campos del registro PMP.

La figura 2.14 muestra, entre otros, los bits R, W y X, los cuales habilitan, la lectura, escritura y ejecución, respectivamente. Cuando se trata de obtener una instrucción de una región PMP que no tiene permiso de ejecución, se eleva una excepción de acceso.

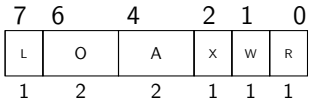



Figura 2.14: Formato de configuración de los registros PMP.

## 2.9. Ecosistema *software*

La organización RISC-V mantiene una lista [19] a modo de resumen sobre los proyectos importantes que permiten entender el estado del ecosistema *software* para RISC-V. A continuación se muestra dicha lista con algunas adiciones relevantes:

### 2.9.1. *Toolchains*

- Binutils: colección de herramientas de parte de GNU (ensamblador, enlazador, objdump, gprof, etc). Tiene soporte completo *upstream*[2] desde la versión 2.28, lanzada en 2017, implementa las ISAs base RV32GC y RV64GC con extensiones estándar y extensiones no ratificadas en ramas separadas.
- GCC: compilador que tiene *front-ends* para C, C++, Objective-C, Fortran, Ada, Go, D, etc y sus bibliotecas estándar. Tiene soporte de *upstream* completo para las ISAs estándar, variables para sistemas empotrados. Integrado soporte para RISC-V a partir de la versión 7.1 (2017). El desarrollo es continuo, con optimizaciones y extensiones experimentales en ramas de desarrollo.
- *GNU Debugger* (GDB): Depurador a nivel de código fuente o máquina del proyecto GNU. Pasa a incorporar soporte básico para RISC-V a partir de la versión 8.3 (2019).
- Glibc: Biblioteca estándar de C utilizada por sistemas GNU/Linux y otros sistemas Unix para mediar en las llamadas al sistema y funciones comunes de C. Soporte *upstream* completo, la arquitectura RISC-V fue añadida en Glibc 2.27 (2018). Entre los ABIs soportados están: ilp32, ilp32d, lp64 y lp64d permitiendo ejecutar programas en los sistemas antes mencionados.
- LLVM: Infraestructura que provee un conjunto de herramientas y bibliotecas para contruir compiladores, depuradores y otras herramientas de desarrollo de *software*. En la actualidad integra soporte completo para

[2] repositorio o proyecto fuente principal donde se integran oficialmente las mejoras, correcciones y nuevas características, antes de su distribución o incorporación en versiones derivadas.

RISC-V y de forma experimental desde LLVM 9.0 en 2019. LLVM soporta las ISAs de 32 y 64 bits y las extensiones estándar "GC".

- Newlib: Implementación de la biblioteca estándar de C orientada a sistemas empotrados (sistemas sin Sistema Operativo (SO), *bare-metal*). Soporta las versiones de 32 y 64 bits de RISC-V.

### 2.9.2. Sistemas operativos

- FreeBSD: Sistema operativo basado en Unix, derivado dé *Berkeley Software Distribution* (BSD). A partir de FreeBSD 13.0 (2021) el RISC-V en FreeBSD pasa a *Tier-2*, teniendo así soporte oficial con menor nivel de garantía que las arquitecturas *Tier-1*. Previamente, pertenecía al *Tier-3*, experimental.
- Linux: Es un *kernel* de sistema operativo tipo Unix ampliamente utilizado por las distribuciones *GNU*/Linux. El soporte comienza con la versión 4.15 (2017). Actualmente Linux soporta ambas versiones principales de RISC-V. Además varias distribuciones Linux (Debian, Fedora, Ubuntu, OpenSUSE, etc.) ya disponen de versiones oficiales para RISC-V.
- OpenBSD: Sistema operativo basado en Unix de la familia de BSD, enfocado en la seguridad y simplicidad. Dispone de una versión oficial para RISC-V de 64 bits desde la versión 7.3.

### 2.9.3. Emuladores y simuladores

- QEMU: Emulador y virtualizador de máquina genérico. Permite ejecutar sistemas operativos y programas compilados para una arquitectura en una máquina con distinta arquitectura. QEMU dispone de soporte completo para RISC-V añadido inicialmente con la versión 2.12 en 2018. Cuenta con multitud de placas RISC-V virtuales que dan soporte a las extensiones estándar, por ejemplo, para riscv64: `amd-microblaze-v-generic`, `microchip-icicle-kit`, `none`, `shakti_c`, `sifive_e`, `sifive_u`, `spike` y `virt`. En la figura 2.15 se puede ver la ejecución de OpenBSD para RISC-V sobre QEMU.
- Spike: Simulador oficial de RISC-V, existe como referencia o *golden model*. Emula el comportamiento de uno o más núcleos RISC-V a nivel de instrucciones. Puede operar en modo *full-system*, para simular un sistema completo, con memoria y dispositivos simples o en modo *proxy*, delegando las llamadas al sistema anfitrión.

### 2.9.4. Entornos de ejecución de lenguajes

La multitud de entornos de ejecución tienen soporte, entre ellos: Dart, GHC, Go, OpenJDK, PHP, Rust, Rust Embedded, V8 y .Net.

### 2.9.5. Herramientas

- *Open On-Chip Debugger* (OpenOCD): herramienta de código abierto para la depuración y programación de microcontroladores y SoCs a bajo nivel mediante interfaces *Joint Test Action Group* (JTAG), *Serial Wire Debug* (SWD), etc. Es común utilización junto a con GDB y adaptadores de interfaz para la de Unidad Central de Procesamientos (CPUs) empotradas. Soporta de forma oficial la arquitectura RISC-V,

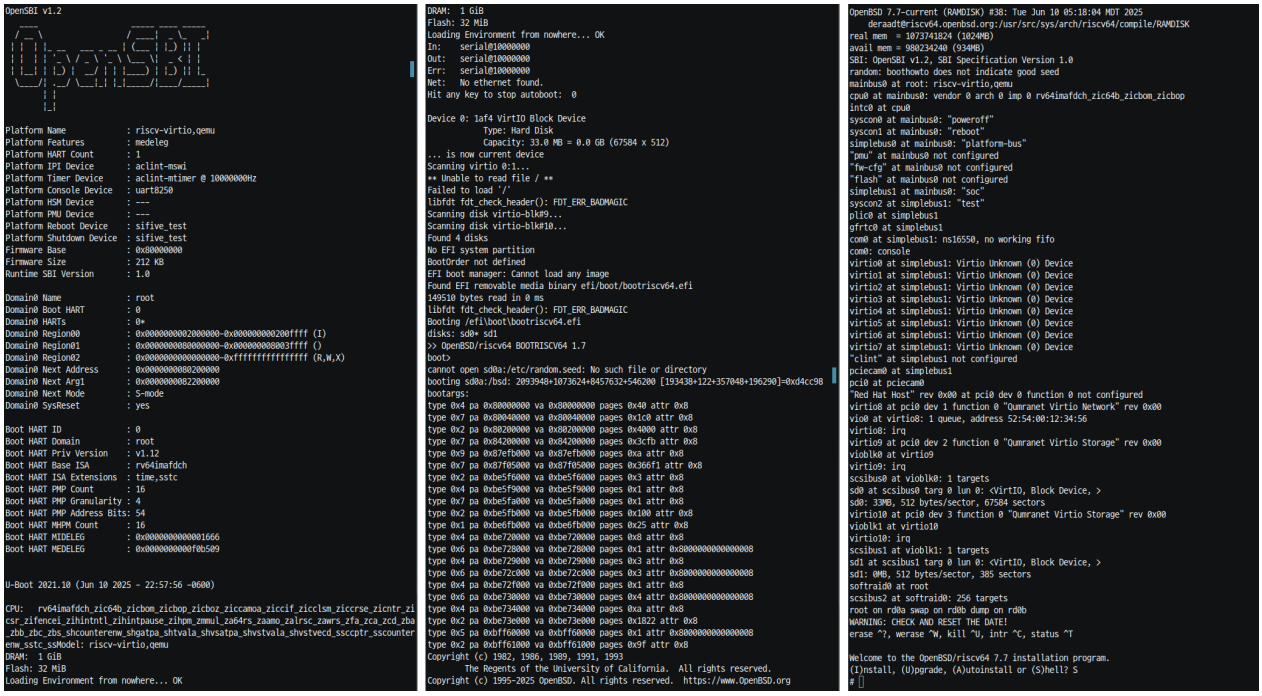



Figura 2.15: Proceso de arranque de OpenBSD para RISC-V 64 bits sobre un *host* Linux.

implementa el estádar de depuración propio de RISC-V (*Debug Spec 0.13*). Además existe un *fork* de parte de `riscv-collab`, `riscv-openocd` que, como su nombre indica, ha tratado de tener la mejor compatibilidad con RISC-V.

- Valgrind: conjunto de herramientas de análisis dinámico de programas, que incluyde mecanismos para la detección de errores en memoria (*Memcheck*) y multitud de otros errores que pueden surgir durante el desarrollo de programas, en tiempo de ejecución. A principios de 2025 integró la compatibilidad con RISC-V en el *upstream*.

### 2.9.6. RISE

Con el objetivo de acelerar la integración de la arquitectura RISC-V en un amplio abanico de programas como sistemas operativos, compiladores, bibliotecas y otros; la Fundación Linux, con el apoyo de compañías como Google, Intel, NVIDIA y Qualcom, lanza en el año 2023 el proyecto "RISE". Como se puede ver en la figura 2.16, "RISE" busca mejorar el soporte para RISC-V en todo el ecosistema de programas informáticos, con un énfasis en preparar dicho ecosistema para el desarrollo comercial.

## 2.10. Comparación con otros conjuntos de instrucciones

En la actualidad el panorama de la computación está dominado por un grupo muy reducido arquitecturas, x86 de la mano de Intel y AMD y ARM por parte de Qualcom, MediaTek y Apple, debido a la complejidad intrínseca de las arquitecturas de computación; RISC-V se establece como una solución para para un mercado con una

| | |
|---|---|
| Compilers & Toolchains | LLVM, GCC |
| System Libraries | Glibc, OpenSSL, OpenBLAS, LAPACK, OneDAL, Jemalloc |
| Kernel & Virtualization | Linux, Android |
| Language Runtimes | Python, OpenJDK/Java, V8 |
| Linux Distro Integration | Ubuntu, Debian, RHEL, Fedora, Alpine |
| Debug & Profiling Tools | Performance profiles, DynamoRIO, Valgrind |
| Simulator/Emulators | QEMU, SPIKE |
| System Software | UEFI, ACPI |

Figura 2.16: Áreas de desarrollo prioritarias para el proyecto "RISE".

diversidad tan reducida. Como se ha mencionado con anterioridad, la principal ventaja y razón del crecimiento reciente de RISC-V, entre otros factores, es su carácter abierto y totalmente modular. RISC-V se encuentra en la posición ideal para tener en cuenta los errores de las arquitecturas pasadas e integrarlas en su desarrollo, sin tener que lidiar continuamente con sistemas legados como por ejemplo x86. A continuación se muestran los aspectos que hacen de RISC-V una arquitectura con vistas a futuro:

- **Flexibilidad y simplicidad:** x86 es una arquitectura que ha tratado de adaptarse al momento actual y ha crecido a la par que la complejidad en la computación, manteniendo siempre la compatibilidad con los sistemas legados. Aunque en la computación general el incremento en la complejidad es comparativamente pequeño, debido a la complejidad de los sistemas actuales, existen áreas como los sistemas empotrados, procesos como la verificación y áreas como la enseñanza en los que la diferencia de complejidad no se puede ignorar. RISC-V lleva el concepto de flexibilidad al extremo, tanto es así que las operaciones de multiplicación y división no forman parte del conjunto de instrucciones base. Es un ISA apto tanto para sistemas empotrados con los más exigentes requisitos de consumo como para supercomputadoras.
- **Agilidad en el desarrollo:** Debido a que uno de los pilares base es la extensibilidad, los desarrolladores tienen la opción de expandir o contraer sus sistemas para adaptarlos a los retos actuales y futuros mediante caminos bien definidos.
- **Consistencia:** RISC-V es consistente en su definición; un ejemplo de esto es el formato de las instrucciones. Los registros, origen y destino mantienen su ubicación a través de los formatos, esta misma regularidad se mantiene desde los espacios de codificación y tamaño de las instrucciones hasta los protocolos de extensión del conjunto de instrucciones.

## 2.11. Conclusiones

En este capítulo se ha presentado un análisis detallado de la arquitectura de RISC-V. Se ha destacado su carácter modular y abierto, que permitirá su adaptación a un amplio espectro de se sistemas, desde microcontroladores hasta supercomputadoras. Se han descrito sus partes principales como la terminología, formatos de instrucciones base, modos de ejecución y memoria virtual. Finalmente se ha comparado RISC-V con las otras arquitecturas dominantes. El capítulo en su totalidad expone los datos técnicos para su valoración como arquitectura universal, escalable y preparada para el futuro.

# Capítulo 3

# Extensión del juego de instrucciones

> La libertad que ha traído RISC-V habilita la capacidad de innovación en la arquitectura.
> *Adaptado de Brian Balley*

Este capítulo expande el concepto de la extensión del juego de instrucciones de RISC-V. Como ya se ha discutido, aparte de permitir el uso de RISC-V para la computación general, RISC-V también tiene como fin facilitar el desarrollo de conjunto de instrucciones especializadas para aceleradores hechos a medida. Los espacios de codificación de las instrucciones están diseñados para facilitar el desarrollo del *software* mediante el *toolchain* estándar cuando se crean procesadores más personalizados. La intención es la de soportar de forma completa las implementaciones que solo cuenten con la base "I" y con posiblemente multitud de conjuntos de instrucciones no estándar [16, cap. 15]. Aunque se hará mención de algunas implementaciones RISC-V, este capítulo se ceñirá a presentar cómo dichas implementaciones hacen uso de la extensión del juego de instrucciones, mientras que en el capítulo 5, se hará un análisis detallado de cada una.

## 3.1. Fundamentos para la extensión del juegos de instrucciones

Para entender cómo se expande el juego de instrucciones con RISC-V es necesario conocer las bases en las que se fundamenta la expansión del juego de instrucciones base.

### 3.1.1. Extensiones estándares y no estándares

Como se mencionó previamente, cualquier procesador RISC-V debe soportar el conjunto de instrucciones base para enteros RV32I, RV32E, RV64I, RV64E o RV128I. Además, de forma opcional, puede prestar soporte para una o más extensiones; estas se dividen en dos categorías:

- Las extensiones estándar son aquellas que son de utilidad para la mayoría de los casos de uso y no entran en conflictos con otras extensiones estándar.

- Las extensiones no estándar son altamente especializadas, son susceptibles de entrar en conflicto con otras extensiones, incluso con las estándares [16].

### 3.1.2. Espacios de codificación y prefijos

El término "espacio de codificación" se emplea para hacer referencia a una serie de bits en los que se codifica tanto el conjunto de instrucciones base como sus extensiones, el conjunto de instrucciones base está definido en un espacio de codificación de 30 bits (bits 31 a 2 de cada instrucción de 32 bits), mientras que la extensión de instrucciones atómicas, "A" está codificado en un espacio de codificación de 25 bits. El término "prefijo" se utiliza para referirse a los bits menos significativos del espacio de codificación, es decir, debido a que las instrucciones en RISC-V son *little-endian*, los bits de la derecha. Por ejemplo, el prefijo para el conjunto de instrucciones base es "11" como se puede observar con el campo opcode para en RV32I.

Para dar soporte para el desarrollo de instrucciones personalizadas propietarias, hay partes del espacio de codificación que se garantiza que nunca serán utilizadas por extensiones estándares.

Aunque los espacios de codificación pueden ser de una longitud arbitraria, emplear un conjunto de tamaños de codificación simplifica el empaquetado de las extensiones desarrolladas de forma independiente, la tabla 3.1 muestra los tamaños que sugieren.

| **Tamaño** | **Uso** | **# Disponibles según longitud de instrucción estándar** | | | |
|---|---|---|---|---|---|
| | | 16 bits | 32 bits | 48 bits | 64 bits |
| 14 bits | Cuadrante de la codificación comprimida de 16 bits | 3 | | | |
| 22 bits | Subcódigo de operación en la codificación base de 32 bits | | $2^8$ | $2^{20}$ | $2^{35}$ |
| 25 bits | Código de operación principal en la codificación base de 32 bits | | 32 | $2^{17}$ | $2^{32}$ |
| 30 bits | Cuadrante de la codificación base de 32 bits | | 1 | $2^{12}$ | $2^{27}$ |
| 32 bits | Subcódigo de operación en la codificación de 48 bits | | | $2^{10}$ | $2^{25}$ |
| 37 bits | Código de operación principal en la codificación de 48 bits | | | 32 | $2^{20}$ |
| 40 bits | Cuadrante de la codificación de 48 bits | | | 4 | $2^{17}$ |
| 45 bits | Sub-subcódigo de operación en la codificación de 64 bits | | | | $2^{12}$ |
| 48 bits | Subcódigo de operación en la codificación de 64 bits | | | | $2^9$ |
| 52 bits | Código de operación principal en la codificación de 64 bits | | | | 32 |

Tabla 3.1: Tamaños sugeridos para el espacio de codificación de instrucciones estándar RISC-V.

Para comprender la tabla 3.1 conviene atender a los siguientes ejemplos:

- Extensión "C": utiliza 16 bits para la codificación de las instrucciones y tiene como prefijo "10", dentro de este espacio de codificación quedan libres otros 3 códigos de operación.
- RV32I: utiliza 32 bits para la codificación de las instrucciones y tiene como prefijo "11". Dentro de este espacio de codificación no queda más espacio, teniendo en cuenta los prefijos asignados a la extensión "C".
- Extensión "A": el espacio de codificación empleado para esta extensión tiene la particularidad de que los 3 bits que se utilizan para codificar el subcódigo no son contiguos con el código de operación principal y se ubican en el campo `funct3`.

Es importante tener en cuenta que el espacio de decodificación está diseñado con las particularidades antes vistas para agilizar y simplificar el proceso de decodificación, de modo que, por ejemplo, conociendo los dos primeros bits de la instrucción, conocemos en qué espacio está codificada la instrucción.

### 3.1.3. Extensiones *greenfield* y *brownfield*

El término "extensión *greenfield*", se utiliza para referirse a una extensión que ocupa un nuevo espacio de codificación y, por tanto, solo puede causar conflictos de codificación a nivel de prefijo, por otra parte, las extensiones "*brownfield*" son aquellas que que se introducen en un espacio de codificación ya existente. Por ejemplo, el conjunto de instrucciones base son extensiones *greenfield* del espacio de codificación de 30 bits, mientras que las extensiones de punto flotante son extensiones *brownfield* que expanden dicho espacio de codificación.

Hay que tener en cuenta que la extensión A se considera que tiene una codificación *greenfield* debido a que se ubica en un espacio de codificación de 25 bits que está en la parte izquierda de los 32 bits del espacio de codificación base aunque su prefijo la ubique dentro del espacio de codificación de 30 bits antes mencionado. Al cambiar sus 7 bits de prefijo se podría mover la extensión A otro espacio de codificación de 30 bits pudiendo surgir solo conflictos a nivel de prefijo pero no dentro del espacio de codificación.

### 3.1.4. Filosofía para la extensión

Uno de los objetivos de RISC-V es permitir el desarrollo de extensiones de forma independiente. Para ello, habilita de espacios de codificación de instrucciones y herramientas que permitan empaquetarlas en una codificación global compatible con el estándar mediante la asignación de prefijos únicos.

## 3.2. Extensiones estándar

La lista de extensiones que están ratificadas está en continua evolución. La tabla 3.2 muestra las instrucciones que están actualmente ratificadas que pueden o no ser parte de la especificación, pero en algún momento formarán parte de la especificación final.

| **Nombre de la especificación** | **Fecha de ratificación** | **Nuevas extensiones o perfiles** |
|---|---|---|
| Load/Store Pair for RV32 (Zilsd & Zclsd) | Febrero de 2025 | Zilsd, Zclsd |

*Continuación en la siguiente página*

| Nombre de la especificación | Fecha de ratificación | Nuevas extensiones o perfiles |
|---|---|---|
| The RISC-V Debug Specification | Febrero de 2025 | Sdext, Sdtrig |
| RISC-V Control Transfer Records | Noviembre de 2024 | Smctr, Ssctr |
| RISC-V Pointer Masking | Octubre de 2024 | Smmpm, Smnpm, Ssnpm, Supm, Sspm |
| The RISC-V Instruction Set Manual Volume II: Privileged Architecture | Octubre de 2024 | Sm1p13, Ss1p13 |
| Double Trap | Agosto de 2024 | Ssdbltrp, Smdbltrp |
| RISC-V Quality-of-Service (QoS) Identifiers | Junio de 2024 | Ssqosid |
| Obviating Memory-Management Instructions after Marking PTEs Valid | Junio de 2024 | Svvptc |
| Resumable Non-Maskable Interrupts | Junio de 2024 | Smrnmi |
| Shadow Stacks and Landing Pads | Junio de 2024 | Zicfiss, Zicfilp |
| BF16 Extensions | Junio de 2024 | Zfbfmin, Zvfbfmin, Zvfbfwma |
| Zaamo and Zalrsc Extensions | Abril de 2024 | Zaamo, Zalrsc |
| B Standard Extension for Bit Manipulation Instructions | Abril de 2024 | B |
| Byte and Halfword Atomic Memory Operations (Zabha) | Abril de 2024 | Zabha |
| RISC-V Supervisor Counter Delegation | Marzo de 2024 | Smcdeleg, Ssccfg |
| May-Be-Operations | Marzo de 2024 | Zimop, Zcmop |
| RISC-V Indirect CSR Access (Smcsrind/Sscsrind) | Febrero de 2024 | Smcsrind, Sscsrind |
| RISC-V Integer Conditional (Zicond) operations extension | Noviembre de 2023 | Zicond |
| Hardware Updating of PTE A/D Bits (Svadu) | Noviembre de 2023 | Svadu |
| RISC-V Cycle and Instret Privilege Mode Filtering (Smcntrpmf) | Noviembre de 2023 | Smcntrpmf |
| Atomic Compare-and-Swap (CAS) Instructions (Zacas) | Noviembre de 2023 | Zacas |
| RISC-V Cryptography Extensions Volume II: Vector Instructions | Septiembre de 2023 | Zvbb, Zvbc, Zvkb, Zvkg, Zvkn, Zvknc, Zvkned, Zvkng, Zvknha, Zvknhb, Zvks, Zvksc, Zvksed, Zvksg, Zvksh, Zvkt |
| "Zfa" Standard Extension for Additional Floating-Point Instructions | Septiembre de 2023 | Zfa |
| RISC-V Advanced Interrupt Architecture | Junio de 2023 | Smaia, Ssaia |

*Continuación en la siguiente página*

| Nombre de la especificación | Fecha de ratificación | Nuevas extensiones o perfiles |
|---|---|---|
| "Zvfh/Zvfhmin" Vector Extension for Half-Precision FP Arithmetic | Junio de 2023 | Zvfh, Zvfhmin |
| "Zihintntl" Non-Temporal Locality Hints | Mayo de 2023 | Zihintntl |
| RISC-V Code Size Reduction | Abril de 2023 | Zca, Zcb, Zcd, Zce, Zcf, Zcmp, Zcmt |
| RISC-V Profiles | Marzo de 2023 | RVA20, RVI20, RVA22 |
| "Zicntr" and "Zihpm" Counters | Marzo de 2023 | Zicntr, Zihpm |
| RV32E and RV64E Base Integer Instruction Sets | Enero de 2023 | RV32E, RV64E |
| "Ztso" Standard Extension for Total Store Ordering | Enero de 2023 | Ztso |
| RISC-V Wait-on-Reservation-Set (Zawrs) extension | Noviembre de 2022 | Zawrs |
| Zmmul Extension | Junio de 2022 | Zmmul |
| PMP Enhancements for Machine-mode Memory Protection (Smepmp) | Noviembre de 2021 | Smepmp |
| RISC-V Base Cache Management Operation ISA Extensions | Noviembre de 2021 | Zicbom, Zicbop, Zicboz |
| RISC-V Bit-Manipulation ISA-extensions | Noviembre de 2021 | Zba, Zbb, Zbc, Zbs |
| RISC-V Count Overflow and Mode-Based Filtering Extension | Noviembre de 2021 | Sscofpmf |
| RISC-V Cryptography Extensions Vol I: Scalar & Entropy Source Instructions | Noviembre de 2021 | Zbkb, Zbkc, Zbkx, Zknd, Zkne, Zknh, Zksed, Zksh, Zkn, Zks, Zkt, Zk, Zkr |
| RISC-V State Enable Extension | Noviembre de 2021 | Smstateen |
| RISC-V "stimecmp / vstimecmp" Extension | Noviembre de 2021 | Sstc |
| RISC-V Vector Extension | Noviembre de 2021 | Zve32x, Zve32f, Zve64x, Zve64f, Zve64d, Zve, Zvl32b, Zvl64b, Zvl128b, Zvl256b, Zvl512b, Zvl1024b, Zvl |
| The RISC-V Instruction Set Manual Volume II: Privileged Architecture | Noviembre de 2021 | Sm1p12, Ss1p12, Sv57, Hypervisor, Svinval, Svnapot, Svpbmt |
| "Zfh" and "Zfhmin" Half-Precision FP Extensions | Noviembre de 2021 | Zfh, Zfhmin |
| "Zfinx", "Zdinx", "Zhinx", "Zhinxmin" FP-in-Integer Registers Extensions | Noviembre de 2021 | Zfinx, Zdinx, Zhinx, Zhinxmin |

*Continuación en la siguiente página*

| Nombre de la especificación | Fecha de ratificación | Nuevas extensiones o perfiles |
|---|---|---|
| "Zihintpause" Pause Hint | Febrero de 2021 | Zihintpause |
| The RISC-V Instruction Set Manual Volume I: Unprivileged ISA | Diciembre de 2019 | A, D, F, RV32I, RV64I, Zaamo, Zalrsc, Zicsr, Zifencei |
| The RISC-V Instruction Set Manual Volume II: Privileged Architecture | Diciembre de 2019 | C, M, Q, Sm1p11, Ss1p11, Sv32, Sv39, Sv48 |

Tabla 3.2: Especificaciones de RISC-V ratificadas y sus correspondientes extensiones o perfiles.

### 3.2.1. Extensiones destacadas

#### Extensión Vectorial (RVV)

La Extensión Vectorial de RISC-V (RVV) posibilita la computación paralela. A diferencia de la extensión "P" la cual utiliza registros de propósito general (GPR) para ejecución de instrucciones de tipo *packed-SIMD*, RVV introduce un archivo de registros adicional con 32 registros dedicados a las operaciones SIMD. Una característica importante de RVV es la flexibilidad para definir la longitud del vector. En lugar de estar determinado por la arquitectura, la longitud de los vectores la determina la implementación, permitiendo así, que diferentes microarquitecturas tengan diferentes longitudes de vector. Esto permite a los programas que usen esta extensión escalar de forma automática entre implementaciones sin la necesidad de reescritura o recompilación [20]. Algunos ejemplos de procesadores que implementan esta extensión son el Ara2 [21],

#### Extensión P para instrucciones *Packed-SIMD*

La Extensión P para instrucciones *Packed-SIMD* de RISC-V habilita operaciones SIMD (Single Instruction, Multiple Data) para operar sobre múltiples elementos. A diferencia de la Extensión Vectorial (RVV), que introduce un archivo de registros dedicado para grandes operaciones SIMD, la Extensión P busca aprovechar los registros de propósito general (GPRs) ya existentes. Inicialmente, se consideraron operaciones de punto flotante empaquetadas, pero el desarrollo y enfoque de la Extensión V para operaciones SIMD de punto flotante más amplias ha dirigido el objetivo principal de la Extensión P hacia las operaciones de *punto fijo* empaquetadas. Esta reorientación la posiciona como una opción para habilitar cierto nivel de paralelismo de datos en registros enteros, siendo particularmente relevante para implementaciones RISC-V más pequeñas o con recursos limitados. Actualmente, la especificación de la Extensión P es un trabajo en progreso llevado a cabo por un grupo de tarea dedicado dentro de la comunidad RISC-V.

## 3.3. Extensiones no estándares

Las principales líneas de desarrollo en materia de extensiones de no estándares son en Internet de las cosas, inteligencia artificial, comunicaciones, computación gráfica, criptografía postcuántica, computación de alto rendimiento, virtualización y seguridad [22].

### 3.3.1. Internet de las cosas

Los dispositivos IoT generalmente basan su funcionamiento en la realización de algunas tareas de computación y en la transmisión de datos de forma inalámbrica. Hay que tener en cuenta que, para realizar estas tareas, este tipo de dispositivos cuenta con recursos, cantidad de memoria y consumo de energía, entre otros. Las extensiones desarrolladas dan respuesta a estos problemas, por ejemplo, reduciendo el consumo de energía de operaciones de punto flotante [23], mejorando la eficiencia del procesado de señales digital (*Digital Signal Processing* (DSP)) [24], [25].

La extensión CHERIoT [26] surge de un proyecto de investigación por parte de Microsoft, ofrece un conjunto de instrucciones de 32 bits optimizado para entornos IoT, aporta la garantía de punteros infalsificables y compartimentación ligera, mejorando así la seguridad de microcontroladores con recursos limitados. Las soluciones de seguridad de memoria tradicionales, originadas en sistemas de escritorio o servidores, no se trasladan bien debido a las restricciones de tiempo real, tamaño de código, potencia y compatibilidad

### 3.3.2. Inteligencia artificial

Del mismo modo que se expande la adopción de la inteligencia artificial, crece la necesidad de llevar a cabo las computaciones complejas que requieren los últimos paradigmas como las redes neuronales profundas (DNN) y las redes neuronales convolucionales (Convolutional Neural Network (CNN)). La gran demanda de recursos de almacenamiento y computación que requieren limita las mejoras de rendimiento y su expansión hacia nuevos ámbitos. Los principales avances tratan de acelerar las diferentes arquitecturas para inteligencia artificial como los transformadores y las redes neuronales gráficas, además de las ya mencionadas DNN y CNN.

### 3.3.3. Comunicación

La utilidad de los dispositivos con comunicación inalámbrica y la demanda de los consumidores de comunicaciones de baja latencia y alto ancho de banda han dado lugar a la quinta generación de tecnologías de telefonía móvil (5G). La sexta generación de telefonía móvil (6G) se espera que tenga un mayor énfasis en la latencia y capacidad de procesamiento en comparación con su predecesor. Los diseñadores de la infraestructura para la comunicación inalámbrica pueden sortear las barreras en cuanto a adaptabilidad y escalabilidad que presentan estos sistemas mediante la implementación de procesadores con conjunto de instrucciones específico de la aplicación (**ASIPs!** (**ASIPs!**)). RISC-V ofrece la oportunidad de llevar sus principios de personalización e innovación a los *Application-specific Instruction Set Processor* (ASIP).

### 3.3.4. Computación gráfica

Las aplicaciones multimedia como el procesado de imagen y vídeo requieren de computación gráfica, la principal forma de aceleración de tipo de computación ha sido mediante el uso de tarjetas gráficas. El hecho de que RISC-V carezca de un estándar para la computación gráfica ha empujado el desarrollo de extensiones de tipo Single Instruction Multiple Threads (SIMT), para renderizado gráfico y para la computación eficiente de gráficos.

### 3.3.5. Criptografía postcuántica

Los sistemas que mantienen tanto la información como las comunicaciones seguras a nivel global están cimentadas en la criptografía asimétrica. Este sistema criptográfico se basa en métodos criptográficos, que son difíciles de atacar mediante fuerza bruta con la capacidad de computación que tenemos actualmente. Sin embargo, un futuro con ordenadores cuánticos de suficiente capacidad vuelve obsoletos la mayoría de algoritmos empleados en la actualidad.

Aquellos métodos criptográficos resistentes a ataques criptográficos, pertenecen a la categoría de criptografía postcuántica (PQC) y aunque se han desarrollado multitud de métodos, algunos como la criptografía basada en retículos necesitan gran capacidad de cómputo. Para suplir la demanda que imponen tecnologías como las comunicaciones celulares en cuanto a anchos de banda y latencia, multitud de extensiones RISC-V se han desarrollado para acelerar estos algoritmos.

### 3.3.6. Computación de alto rendimiento

Uno de los campos hacia el que se está expandiendo RISC-V es la industria de computación de alto rendimiento (HPC), para su uso principalmente en predicciones del tiempo, simulaciones de accidentes de tráfico y simulaciones de materiales, entre otros. Las principales extensiones de RISC-V para HPC, incluyen,

### 3.3.7. Virtualización

Pese a que dentro de las extensiones estándares está la extensión "H" que introduce soporte para la virtualización, esta tecnología se encuentra en sus estados iniciales y necesita de multitud de mejoras. Sin embargo, RISC-V provee una base que incentiva el desarrollo de la plataforma hacia un sistema de virtualización completo.

### 3.3.8. Seguridad

La cantidad de dispositivos y el grado de interconexión a nivel global entre estos están en aumento. Con tecnologías como IoT se ha comprobado el daño que pueden ocasionar masas de dispositivos comercializados a escala que no incorporan las medidas de seguridad apropiadas.

## 3.4. Extensiones no estándares

Debido a la que RISC-V facilita la integración de extensiones personalizadas a la arquitectura base de referencia, se han propuesto un amplio número de extensiones, específicas para distintos dominios de aplicación. Dichas extensiones estarán soportadas por aceleradores *hardware* específicos para el correspondiente dominio de aplicación Domain-specific accelerator (DSA). A continuación se resume algunas extensiones no estándares que es posible encontrar en distintas implementaciones realizadas, según se recogen en el trabajo de [22].

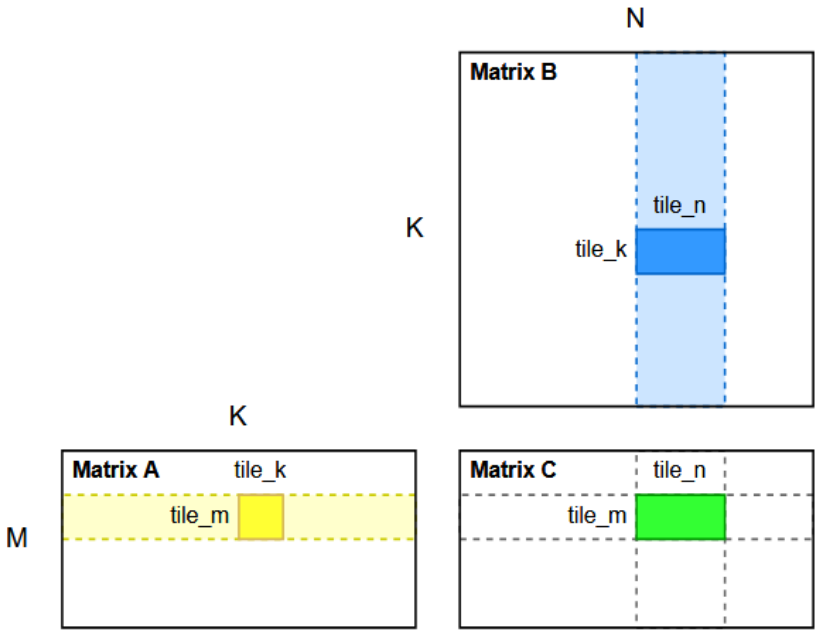



Figura 3.1: Ejemplo de multiplicación de matrices usando las extensiones "Matrix Extension"

### 3.4.1. RISC-V Matrix extension

La extensión para la multiplicación de matrices (Matrix Extension) [27] implementa multiplicaciones de matrices dividiendo la matriz de entrada y salida en mosaicos, que luego se almacenan en registros de matrices. El tamaño del mosaico se reduce con respecto al tamaño completo de la matriz, por lo que se consigue una gestión eficiente de memoria (figura 3.1). Cada instrucción de multiplicación de matrices calcula su mosaico de salida recorriendo la dimensión K en mosaicos, cargando los valores necesarios de las matrices A y B, y multiplicándolos y acumulándolos en la salida. La extensión de matrices está fuertemente inspirada en la extensión «V» vectorial de RISC-V.

### 3.4.2. Extensiones *custom*

Se ha indicado anteriormente que la ISA de RISC-V está diseñada de forma modular, partiendo de un juego de instrucciones muy simples. De esta forma, se añade un conjunto de extensiones al juego de instrucciones (ISA extensions) que permite llevar un control fino en el impacto en área y potencia del procesador implementado, no teniendo que consumir área y potencia si la aplicación final no lo requiere.

En los apartados anteriores se han indicado las extensiones estandarizadas y, por tanto, congeladas, que permite que mantener la compatibilidad de RISC-V. En general, añadir un conjunto de extensiones hechas a medida de la aplicación final utilizando aproximaciones DSA supone un aspecto diferenciador y competitivo para la empresa, favoreciendo por lo tanto la innovación.

Existen diferentes alternativas para mantener la compatibilidad de las instrucciones extendidas [28]. En la figura 3.2 (a) se muestra cómo encajar una extensión ISA personalizada en una pila de *software*. En el nivel más bajo,

hay un procesador compatible con RISC-V con una extensión ISA personalizada. La aplicación se ejecuta en un sistema operativo, ya sea RTOS o un sistema operativo completo. Se puede compilar con cualquier compilador compatible con un procesador RISC-V estándar (sin extensiones ISA especiales). Encima del sistema operativo hay tres aplicaciones. App1 es una aplicación genérica que no requiere ninguna aceleración. Se puede utilizar un compilador estándar disponible (por ejemplo, GCC) para compilarla, o incluso se puede utilizar una aplicación precompilada; el procesador RISC-V podrá ejecutarla. App2 y App3 son las aplicaciones que requieren instrucciones especiales ya que deben ejecutarse lo más rápido posible. Estas deben compilarse con un compilador que conozca específicamente la extensión ISA personalizada. El compilador puede utilizar las nuevas instrucciones que acelerarán App2 y App3.

La figura 3.2 (b) muestra otro ejemplo de un procesador compatible con RISC-V con una extensión ISA personalizada. La aplicación 1 no utiliza la extensión ISA personalizada. Las aplicaciones 2 y 3 utilizan una API genérica. La API se implementa mediante una biblioteca que reconoce la extensión ISA personalizada, lo que a su vez puede acelerar las aplicaciones 2 y 3. Tanto la aplicación 2 como la 3 se pueden reutilizar en el procesador RISC-V estándar. Es necesario una biblioteca que implementa la API requerida. En este sistema, trasladar App2 y App3 de RISC-V con una extensión ISA personalizada a RISC-V sin la extensión no requiere ninguna migración de aplicaciones.

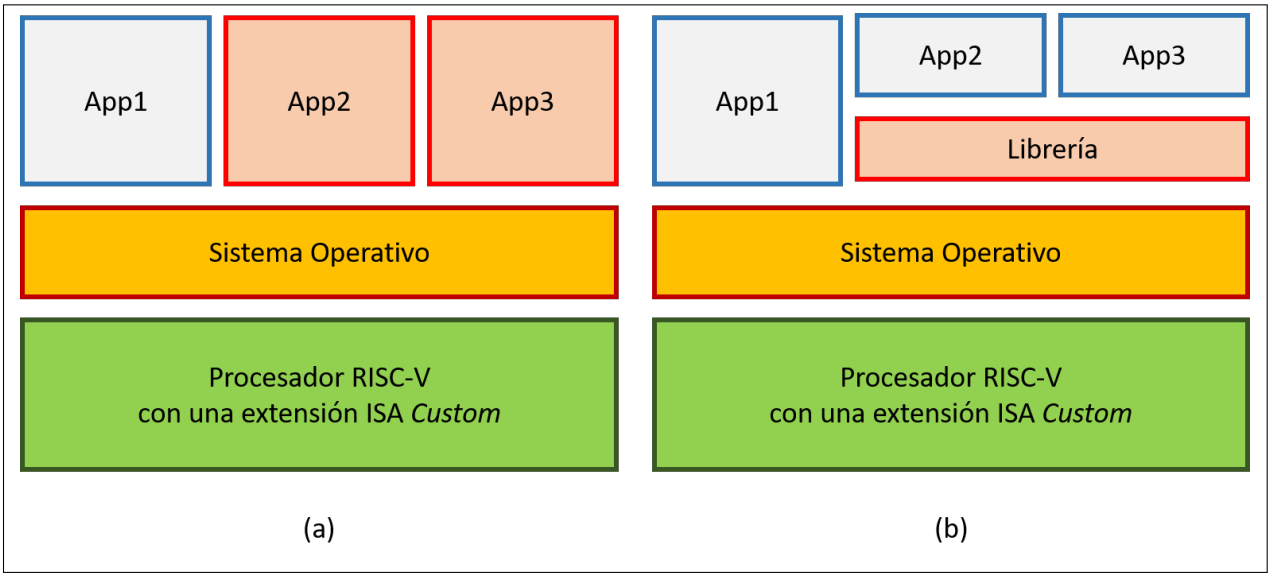



Figura 3.2: Integración de instrucciones *custom* en el *stack software*. Adaptado de [28]

El proceso de crear una nueva extensión a la ISA de RISC-V es un proceso ciertamente complejo. En primer lugar, hay que identificar las instrucciones a implementar. A continuación, hay que añadirlas al compilador C, a los simuladores, a los depuradores y a otras herramientas del entorno de desarrollo, verificando la integridad y eficiencia del código generado. Por último, hay que ampliar la descripción del *hardware* a nivel RTL y verificar cualquier cambio que se haya realizado en él.

A continuación se muestran algunos ejemplos del proceso de implementación de una instrucción *custom* para el intercambio de bytes (`swapbytes`)[28]. En la figura 1 se presenta un ejemplo de utilización de la función `byteswap` implementada como una instrucción *custom*. Como se puede apreciar, el código está preparado para ser compilado

en una arquitectura que soporta dicha extensión (usando la macro ISA_BYTE_SWAP) o para otra más general que no incluya dicha función.

En las figuras 2 y 3 se puede apreciar la diferencia en el código ensamblador generado por el compilador en ambos caso. La consecuencia directa es la disminución significativa del tiempo de cómputo para la instrucción *custom*.

```
// universal byteswap() function
inline uint32_t byteswap(const uint32_t word) {
#ifdef ISA_BYTE_SWAP
    // C compiler intrinsic
    return __byteswap(word);
#else
    return ((word >> 24) & 0x000000ff) |
           ((word <<  8) & 0x00ff0000) |
           ((word >>  8) & 0x0000ff00) |
           ((word << 24) & 0xff000000);
#endif
}
```

Código fuente 1: Ejemplo de código con soporte a funciones *custom*. Adaptado de [28].

```
byteswap:
  byteswap x10, x10
  c.jr ra // return from function
```

Código fuente 2: Ejemplo de generación de código con soporte a ISA con extensiones *custom*. Adaptado de [28].

```
byteswap:
  srl x15, x10, 8
  lui x14, %hi( 65280 )
  add x14, x14, 65280 & 0xfff
  c.and x15, x14
  srl x14, x10, 24
  c.or x15, x14
  sll x14, x10, 8
  lui x13, 16711680>>12 & 0xfffff
  c.and x14, x13
  sll x13, x10, 24
  c.or x14, x13
  or x10, x14, x15
  c.jr ra // return from function.
```

Código fuente 3: Ejemplo de generación de código con soporte a ISA estándar. Adaptado de [28].

Finalmente, en la figura 4 se aprecian las modificaciones necesarias a realizar en la arquitectura del procesador, generalmente a nivel RTL, y particularmente en este caso en la ALU, para dar soporte a la nueva instrucción.

```
always @(*) begin
  case ( alu_opcode )
      ...
      // <file>.codal:<line>:<column>
      4'h9: mux_ex_result_D = {alu_op1_Q[ 7: 0],alu_op1_Q[15: 8],
                               alu_op1_Q[23:16],alu_op1_Q[31:24]};
      ...
  endcase
end
```

Código fuente 4: Modificación de la descripción SystemVerilog de la ALU de RISC-V para la instrucción *custom*. Adaptado de [28].

## 3.5. Conclusiones

Este capítulo profundixa en el estudio de las extensiones del juego de instrucciones de RISC-V, ya ean extensiones estándares definidas por RISC-V International o extensiones *custom*. Estas extensiones permiten facilitar el desarrollo de conjunto de instrucciones especializadas para aceleradores hechos a medida DSA.

Los espacios de codificación de las instrucciones están diseñados para facilitar el desarrollo del *software* mediante el *toolchain* estándar cuando se crean procesadores más personalizados. La intención es la de soportar de forma completa las implementaciones que solo cuenten con la base "I" y con posiblemente multitud de conjuntos de instrucciones no estándar [16, cap. 15].

# Capítulo 4

# Aceleradores *hardware*

> RISC-V está revolucionando el desarrollo de la IA al proporcionar una ISA flexible y abierta que integra *software* y *hardware*.
> *RISC-V International*

En este capítulo se presenta el concepto de acelerador *hardware*, la razón de su necesidad, su clasificación a través de cuatro dimensiones: (1) Aspectos generales, (2) Acoplamiento con el host, (3) Arquitectura y (4) Aspectos de *software* basados en la taxonomía propuesta por Peccerillo et al en [29]. Estas son las clasificaciones que formarán la base para decidir sobre qué plataforma RISC-V se desarrollará el acelerador.

## 4.1. Justificación

El avance de la tecnología establecido por la Ley de Moore ha resultado en décadas de crecimiento de la potencia de computación sin precedentes. El *Dennard Scaling* ha reforzado esta tendencia permitiendo que la densidad de potencia se haya mantenido estable sin incrementos en el consumo. Juntos, la Ley de Moore y el *Dennard Scaling* han llevado a un bucle positivo en el que el área se ha reducido, el número de transistores incrementado, junto a las velocidades de reloj y a la potencia por vatio dando lugar a procesadores de propósito general cada vez más capaces.

Sin embargo, alrededor de mediados de los 2000 la industria empezó a toparse con límites físicos. Tras esto las densidades de potencias empezaron a aumentar, resultando necesariamente en regiones de *dark silicon* o silicio oscuro, para limitar la potencia disipada en cada momento.

La solución provista por los arquitectos de computación para este reto ha alejado el desarrollo de los núcleos de propósito general de cada vez mayor frecuencia en favor de la especialización a nivel de arquitectura y heterogeneidad en los sistemas.

## 4.2. Tipos de aceleradores *hardware* y su clasificación

Debido a las razones antes expuestas los aceleradores *hardware* juegan en la actualidad un papel fundamental en la optimización del rendimiento y del consumo energético. Sin embargo, debido a la propia naturaleza heterogénea de esta tecnología existe una gran diversidad de arquitecturas y criterios para su categorización lo que dificulta la comparativa y toma de decisiones. Como ya se mencionó, Peccerillo et al. en [29] presentan una taxonomía que clasifica los aceleradores *hardware* atendiendo a cuatro macro-categorías: Aspectos generales, Acoplamiento con el host, Arquitectura y Aspectos de *software*:

## 4.3. Aspectos generales

Son aquellos aspectos que describen el acelerador en unos niveles altos de abstracción y facilita su contextualización.

### 4.3.1. Tipo de arquitectura

#### ASIC

Los ASIC se diseñan para ejecutar una única aplicación o un conjunto muy acotado de funciones. Esta especialización les confiere una gran eficiencia energética y de área inigualable, sin embargo, no son reconfigurables y conllevan elevados costes *Non-Recurring Engineering* (NRE). Su uso de justifica mediante un alto volumen de producción y la función a implementar está completamente fijada.

#### FPGA

Las FPGA son aceleradores reconfigurables compuestos por millones de bloques lógicos, celdas de memoria y bloques especializados (DSP, *Input/Output* (I/O), etc.) interconectados mediante una red programable. Su reconfiguración a nivel de grano fino permite abarcar una gran variedad de aplicaciones, lo que los convierte en candidatos idóneos para la computación de propósito general. El tiempo de reconfiguración suele ser del orden de milisegundos a segundos.

#### Spatial

Los aceleradores *spatial* se basan en una matriz de unidades aritmético-lógicas de *Processing Engine* (PE) relativamente simples que se comunican directamente mediante redes en chip *Network on a Chip* (NoC), buses o conexiones ad-hoc. Las PE pueden encadenarse y transferir datos en modo *stream*, cada una con su propia lógica de control y memoria local, lo que favorece la paralelización de datos y tareas.

#### CGRA

Las *Coarse-Grained Reconfigurable Array*s (CGRAs) constituyen un caso particular de aceleradores *spatial*: disponen de PEs reconfigurables y de una red de interconexión también reconfigurable. Ofrecen una reconfigurabilidad de baja resolución que las hace flexibles para dominios concretos, con tiempos de reconfiguración del

orden de nanosegundos a microsegundos. Ello permite combinar cómputo espacial y temporal, así como modos de ejecución impulsados tanto por la configuración como por el flujo de datos.

### Sistólico

En los aceleradores sistólicos la computación principal se realiza en una matriz 1D o 2D de Unidades Aritmético-lógicas (ALUs) síncronas entre sí (*lock-step*). Los datos de entrada penetran por el primer nivel; cada unidad aplica una función fija y propaga el resultado al nivel siguiente, generando un flujo en forma de onda hasta producir el resultado final. Se trata, por tanto, de un subtipo de acelerador *spatial* con dos restricciones: (i) las ALU no son programables y (ii) el flujo de datos sigue una dirección fija.

### Vector

Los aceleradores vectoriales ejecutan la computación mediante procesadores vectoriales que integran uno o varios núcleos vectoriales. Cada núcleo dispone de registros capaces de almacenar decenas o cientos de elementos y aplicar instrucciones vectoriales a todos ellos simultáneamente. Su diseño aprovecha de forma intensiva el *pipelining* y permite cargar o almacenar datos contiguos o con saltos regulares en memoria, resultando muy eficientes para bucles densos y operaciones matemáticas repetitivas.

### Manycore

Los aceleradores *manycore* agrupan cientos o miles de núcleos físicos simples—normalmente en grupos de multiprocesadores—para maximizar el paralelismo a nivel de datos. Frente a los núcleos complejos de los procesadores convencionales, se opta por diseños más sencillos (in-order, sin predicción de saltos) a fin de aumentar la densidad de cómputo por área. Este paradigma se considera la evolución natural del multicore tradicional.

### GPU

Las Unidades de Procesamiento Gráfico (GPUs) representan el exponente más destacado de los aceleradores *manycore* y, al mismo tiempo, comparten similitudes con los vectoriales. Una GPU moderna agrupa decenas de multiprocesadores formados por núcleos simples que explotan un paralelismo masivo a nivel de hilos. Cada multiprocesador se comporta como un núcleo vectorial multihilo, donde la latencia se oculta mediante la conmutación rápida entre cientos de hilos activos, en lugar de emplear tuberías profundas.

### PIM

Los aceleradores *Processing in Memory* (PIM) acercan el cómputo a la memoria con el fin de reducir el consumo energético asociado al movimiento de datos y aumentar el ancho de banda utilizable. Existen dos enfoques: *near-memory*, que integra motores de cómputo junto a bancos Memoria Dinámica de Acceso Aleatorio (DRAM)/Memoria Estática de Acceso Aleatorio (SRAM) convencionales, e *in-memory*, donde las propias celdas de memoria se reutilizan para operaciones lógicas, con ayuda de tecnologías emergentes como la Memoria de Acceso Aleatorio Resistiva (ReRAM) o la Memoria Magnética de Torque de Transferencia de Espín (STTMRAM). Es un tipo de acelerador especialmente útil para cargas de trabajo intensivas en movimiento de datos.

### 4.3.2. Dominio de aplicación

Hace referencia al tipo de aplicaciones que se pueden ejecutar en cada acelerador, aunque no estén limitadas al mismo.

#### Paralelismo de datos

El paradigma de paralelismo de datos permite expresar problemas en términos de colecciones de elementos que pueden procesarse de manera concurrente, mediante la aplicación del mismo flujo de instrucciones a cada elemento.

#### Flujo de datos

El paradigma de flujo de datos representa los problemas mediante nodos de procesamiento y el flujo de información entre ellos, generalmente a través de un *DataFlow Graph* (DFG). Las operaciones se ejecutan en cuanto los operandos están disponibles en el nodo correspondiente.

#### Propósito general

Se refiere a la ejecución de programas sin estructura, restricciones ni dominio de aplicación específicos, lo que proporciona máxima flexibilidad en el diseño e implementación de algoritmos.

**Dominios específicos:**

- **Entrenamiento de *Machine Learning* (ML)**: El entrenamiento de ML consiste en la ejecución de un modelo de aprendizaje automático con el objetivo de determinar los valores de sus parámetros y minimizar una función de coste durante el proceso de optimización.
- **Inferencia de ML**: La inferencia de ML se refiere a la ejecución de un modelo cuyas parámetros ya han sido establecidos en una fase previa de entrenamiento, con el fin de generar predicciones o clasificaciones sobre nuevos datos.
- **Realidad Aumentada (RA)**: La RA proporciona una vista directa o indirecta del entorno físico real, enriquecida con información virtual generada por ordenador en tiempo real.
- **Visión por computadora**: La visión por computadora es el campo de la inteligencia artificial que busca dotar a los sistemas de la capacidad de analizar e interpretar imágenes o vídeo, emulando funciones del sistema visual humano.
- **Criptografía**: La criptografía estudia métodos para transformar mensajes de manera que resulten ininteligibles para todo sujeto distinto del receptor previsto, garantizando así la confidencialidad de la información.
- **Compresión y descompresión de datos**: La compresión de datos engloba algoritmos destinados a reducir el número de bytes necesarios para representar información, ya sea preservando la calidad original (sin pérdidas) o aceptando cierta degradación (con pérdidas).

- **Manipulación de bases de datos**: La manipulación de bases de datos comprende las operaciones sobre colecciones de información estructurada, típicamente almacenada electrónicamente en sistemas de gestión de bases de datos, con el fin de insertar, consultar, actualizar o eliminar registros.

- **Alineamiento de secuencias genómicas**: El alineamiento de secuencias genómicas consiste en comparar y ordenar secuencias de bases nucleotídicas obtenidas de un individuo para identificar similitudes y diferencias, facilitando estudios genéticos y filogenéticos.

- **Procesamiento de grafos**: El procesamiento de grafos se ocupa de la manipulación de datos organizados en estructuras de grafo, abarcando algoritmos de recorrido, búsqueda de rutas, detección de comunidades y otros análisis topológicos.

- **Gráficos por ordenador**: Los gráficos por ordenador se centran en la generación, manipulación y visualización de imágenes digitales mediante el uso de técnicas de representación y renderizado en sistemas informáticos.

- **Procesamiento multimedia**: El procesamiento multimedia implica el manejo de flujos de datos de diversos formatos (audio, vídeo, voz, texto, ...) para realizar tareas como codificación, transmisión, edición y análisis.

- **Autómata finito no determinista (NFA)**: El cálculo de NFA estudia máquinas abstractas en las que pueden existir múltiples transiciones posibles desde un mismo estado, lo que facilita el reconocimiento de patrones complejos.

- **Encriptación de flujo de red**: La encriptación de flujo de red comprende operaciones criptográficas aplicadas a los datos que ingresan o salen a través de la interfaz de red, con el fin de proteger la información en tránsito.

- **Detección de patrones**: La detección de patrones agrupa un conjunto de operaciones destinadas a verificar la presencia o ausencia de una secuencia o estructura concreta dentro de un conjunto de datos.

- **Aceleración de búsqueda**: La aceleración de búsqueda reúne técnicas y operaciones diseñadas para proporcionar resultados relevantes en motores de búsqueda web con tiempos de respuesta consistentemente bajos.

### 4.3.3. Programabilidad del acelerador

Los aceleradores permiten proveen a su usuario formas de ejecutar las computaciones sobre datos aportados por el mismo, de modo que se obtenga una salida. La programabilidad hace referencia al grado en el que las operaciones del acelerador se pueden configurar.

### 4.3.4. Reconfigurabilidad del acelerador

Un acelerador es reconfigurable cuando existe un procedimiento para alterar las funciones lógicas llevadas a cabo por el *hardware*, ejemplo de esto son las FPGAs y las CGRAs, donde se puede utilizar operaciones mediante *software* para alterar las operaciones ejecutadas en los elementos lógicos y las interconexiones entre estos que las componen.

## 4.4. Comunicaciones con el host

A continuación se detallan las diferentes opciones de conexión del acelerador con el resto del sistema, el *host*.

### 4.4.1. Estrategia de conexión

- **Bump-in-the-wire**: Conexión especial situada entre dos componentes existentes, que consiste en redirigir los cables de entrada y salida de dichos componentes hacia los del acelerador.
- ***Cache Coherent Interconnect for Accelerators* (CCIX)**: Interconexión coherente de caché definida por el consorcio CCIX. Permite la comunicación chip a chip en sistemas heterogéneos con elevado ancho de banda y baja latencia, y extiende la memoria paginada del sistema operativo para incluir la del acelerador en un espacio virtual unificado.
- **Ranura DRAM**: Conector estándar de placa base donde se insertan los módulos de memoria DRAM. Los aceleradores que usan esta interfaz se presentan como Módulos de Memoria de dos Lineas (DIMMs) con unidades de computación en memoria integradas.
- ***Hybrid Memory Cube* (HMC)**: Paquete de alto rendimiento que apila cuatro u ocho dispositivos DRAM junto con una capa lógica de computación. Puede conectarse al sistema host mediante ranuras DIMM o interconexiones como Intel QuickPath.
- **Integrado en chip**: El acelerador forma parte de un SoC más amplio que incluye, al menos, una CPU de propósito general.
- **M.2**: Especificación flexible para dispositivos externos que soporta buses *Peripheral Component Interconnect Express* (PCIe), *Serial AT Attachment* (SATA) y *Universal Serial Bus* (USB).
- ***OCP Accelerator Module* (OAM)**: Interconexión de alta velocidad basada en un factor de forma mezzanine, diseñada para garantizar la interoperabilidad entre componentes.
- **PCIe**: Bus de expansión de alta velocidad, la interfaz más común en placas base para dispositivos externos.
- **Interconexión de red**: Conexión física estándar para dispositivos de red. Los aceleradores que emplean esta interfaz suelen ser dispositivos complejos, a menudo diseñados como *Point of Delivery* (PoD).
- **USB**: Estándar industrial más común para conectar periféricos externos.
- ***Core-V eXtension interface* (CV-X-IF)**: La interfaz Core-V eXtension (CV-X-IF) [30] es una extensión de RISC-V que proporciona un marco generalizado para implementar coprocesadores personalizados y extensiones de ISA en procesadores RISC-V existentes. Cuenta con canales independientes para la descarga de instrucciones de forma agnóstica al acelerador y la escritura de los resultados.

### 4.4.2. Interfaz *software*

Describe las operaciones que se pueden llevar a cabo mediante programación para interactuar con el acelerador. El desarrollador de la aplicación que será acelerada necesita ejecutar una serie de operaciones para iniciar la ejecución, se pueden clasificar en:

- **Gestión de memoria**: asignación y liberación de memoria del acelerador, así como transferencia de datos entre la memoria del host y la del acelerador.
- **Configuración de *software***: ajuste de parámetros que refinan una operación predefinida o carga del binario que representa la tarea acelerada.
- **Configuración de lógica programable**: procedimiento *software* para configurar la lógica programable de un dispositivo reconfigurable, generalmente expresado en RTL/*Hardware Description Language* (HDL).
- **Lanzamiento de kernel**: invocación y ejecución de una función de kernel definida por el usuario en el acelerador.
- **Ejecución de operación**: ejecución de una única operación, habitualmente mediante el envío de un paquete de comandos.

### 4.4.3. Memoria común

Específica si el acelerador comparte alguna región dentro de la jerarquía de la memoria con el *host* y puede acceder dicha región sin intervención de la CPU.

- **ninguna**: el acelerador y el host no comparten ningún nivel de la jerarquía de memoria.
- **LLC**: el acelerador tiene acceso directo al *Last-Level Cache* (LLC) del host.
- **DRAM**: el acelerador tiene acceso directo a la DRAM del host.
- **Otros modelos**: diversos modelos de memoria.

### 4.4.4. Comunicación con el sistema operativo

Los aceleradores requieren diferentes grados de esfuerzo por parte del sistema operativo, mientras que algunos al conectarse funcionan de forma transparente, otros necesitan ser reconocidos por el SO.

- **Ninguno**: el acelerador interactúa con el sistema de forma transparente, sin necesidad de que el SO lo reconozca.
- **Controladores**: es necesario instalar controladores en espacio de usuario y/o espacio del *kernel*.
- **SO personalizado**: el acelerador requiere un sistema operativo personalizado instalado en el host.
- **Sin detalles**: no se disponen de más datos sobre el acoplamiento con el SO.

## 4.5. Arquitectura

Los aceleradores presentan una gran diversidad, para ver las similitudes y diferencias con los sistemas de propósito general, conviene analizar estas dos categorías: recursos de propósito general y recursos de propósito específico.

### 4.5.1. Recursos de propósito general

Componentes que se encuentran normalmente dentro de sistemas de propósito general, agrupados en memorias y motores de cómputo.

- **Memoria**:
  - L1, L2 y L3 (niveles de caché).
  - DDR3, DDR4 y DDR5 (memorias DRAM).
  - LPDDR4, LPDDR4X, LPDDR5 y LPDD5X (memorias DRAM de bajo consumo).
- **Motor de cómputo**:
  - ARM CPU: procesador de propósito general basado en ARM.
  - x86 CPU: procesador de propósito general basado en x86.
  - Unidad escalar: procesador simple de propósito general con un conjunto de instrucciones que no se especifica.

En cuanto a los motores de cómputo, se distinguen ARM, x86 y otras ISAs no especificadas (por ejemplo, RISC-V, MIPS) dentro de la categoría de unidad escalar.

### 4.5.2. Recursos de propósito específico

Componentes de *hardware* específicos de los aceleradores. Incluye aquellos recursos que han sido diseñados explícitamente para satisfacer las necesidades de las aplicaciones aceleradas, siguiendo principios diferentes a los adoptados para componentes de propósito general.

**Memoria:**

- **GDDR6, GDDR6X**: tecnologías de DRAM orientadas a gráficos.
- **HBM, HBM2, MCDRAM, HMC-RAM**: tecnologías de memoria DRAM apiladas en 3D de alta velocidad, con vias de silicio y microbump.
- **eDRAM**: DRAM integrada en el mismo chip del acelerador.
- **Memoria local**: memoria SRAM en chip, tanto direccionable (gestionada explícitamente por *software*, como *scratchpad*) como no direccionable (utilizada de forma transparente para almacenar resultados parciales, parámetros de configuración, instrucciones u otros).

**Motores de cómputo:**

- **Matriz espacial de procesamiento** (Spatial array): matriz 1D o 2D de unidades aritméticas/interconectadas que procesan y comparten datos de forma independiente.

- **Procesador vectorial** (Vector processor): procesador que ejecuta una sola instrucción sobre matrices de datos, incluyendo procesadores vectoriales avanzados y procesadores SIMD.
- **Matriz sistólico** (Systolic array): matriz 1D o 2D de unidades aritméticas que procesan datos en sincronía y los transmiten hacia adelante.
- **Núcleo tensorial** (Tensor core): núcleo único con una matriz 2D o 3D de unidades aritméticas, generalmente especializado en operaciones MAC.

**Componentes de función fija:**

- **Motor de renderizado**: pipeline con todas las operaciones necesarias para renderizar modelos 3D en pantallas 2D.
- **Motor físico**: unidad especializada en la emulación de comportamiento físico.
- **Motor de funciones auxiliares de ML**: unidad especializada en operaciones auxiliares de aprendizaje automático (por ejemplo, funciones de activación).
- **Motor criptográfico**: unidad especializada en operaciones criptográficas (por ejemplo, *hashing*, cifrado/descifrado).
- **Convertidor AD/DA**: convertidor analógico-digital / digital-analógico.
- **Motor de compresión/descompresión**: unidad especializada en la compresión y descompresión de datos.
- **Motor de alineamiento**: unidad especializada en operaciones de alineamiento genómico.
- **Motor de operaciones de BD**: unidad especializada en operaciones comunes de manipulación de bases de datos.

**Lógica programable:**

- **Baja resolución**: permite flexibilidad de propósito general, con *hardware* altamente personalizable e interconexiones entre componentes. Es típico de los FPGAs.
- **Alta resolución**: la estructura general es fija y dictada por la flexibilidad específica del dominio. Es típico de los CGRAs, que presentan un matrices 2D de unidades reconfigurables con una interconexión también reconfigurable, usualmente menos compleja que la de los FPGAs.

**Memoria y Cómputo:**

- **ReRAM**: memoria no volátil que utiliza el cambio de resistencia para almacenar datos.
- **STT-MRAM**: memoria no volátil basada en uniones túnel magnéticas con capas magnéticas que cambian de valor por corrientes de espín polarizado.
- **Lógica en DRAM**: unidades de cómputo integradas dentro de la memoria DRAM.
- **Lógica en SRAM**: unidades de cómputo integradas dentro de la memoria SRAM.
- **Lógica cercana a memoria**: unidades de cómputo ubicadas cerca, pero no dentro, de la memoria.

## 4.6. Aspectos *software*

### 4.6.1. Capa de programación

Las herramientas que los programadores pueden utilizar para programar, configurar o comunicarse con el acelerador. Representa el nivel de abstracción que los desarrolladores pueden aprovechar y, en consecuencia, influye en la productividad con la que pueden escribir código acelerado.

- **HDL**: lenguajes de descripción de *hardware* usados para describir circuitos digitales. Pueden emplearse para reconfigurar la lógica programable de aceleradores que lo permitan.
- **Assembly**: equivalente al lenguaje ensamblador de los procesadores de propósito general, pero específico para aceleradores.
- **Bibliotecas**: colecciones de funciones y clases que invocan código acelerado de forma interna; pueden utilizarse en código fuente de alto nivel para acceder al acelerador.
- **Lenguajes de alto nivel**: equivalente para aceleradores de los lenguajes de programación de propósito general; permiten generar código para el acelerador mediante compilación o interpretación del código fuente de alto nivel.

### 4.6.2. Ecosistema

Hace referencia al entorno de desarrollo, los *frameworks*, compiladores, bibliotecas, entre otros, de los que disponen los programadores y los usuarios para hacer uso del acelerador. Pueden ser de código abierto o exclusivos de cada empresa.

### 4.6.3. Granularidad

Describe el tipo de tareas que se esperan del acelerador y está muy relacionado con la flexibilidad de este:

- **Tarea preconfigurada**: operación única cableada directamente en el *hardware* del acelerador. Generalmente puede ser "activada" mediante un paquete de comandos.
- **Nivel de función**: función definida por el usuario, tal como se entiende en lenguajes de programación de alto nivel como C o C++. No necesariamente se invoca de manera directa, sino que puede formar parte de un flujo de ejecución más complejo. Por ejemplo, puede representar la función de activación de las neuronas en una red neuronal, o la operación realizada por cada nodo de un grafo sobre sus datos de entrada.
- **Nivel de aplicación**: una aplicación completa puede ejecutarse directamente en el acelerador.

## 4.7. Aceleradores

A continuación se hará una exposición de los aspectos clave y de la clasificación de los aceleradores que se han considerado para, mediante la expansión del conjunto de instrucciones base, acelerar diversas aplicaciones. En el capítulo 5, están las plataformas RISC-V genéricas (SoCs y CPUs) que, de igual modo, se han tenido en cuenta.

## 4.8. Marcellus

El Marcellus [31], figura 4.1 es un SoC orientado hacia el *edge computing*, específicamente a nodos de IA-IoT, planteado para la tecnología de 22nm FDX de GlobalFoundries. Combina un clúster de 16 núcleos RISC-V para DSP diseñados para la ejecución de diversos flujos de trabajo, un *Reconfigurable Binary Engine* (RBE) para acelerar convoluciones para DNNs y un conjunto de bloques *On-Chip Monitoring* (OCM) conectados a un generador *Adaptive Body Biasing* (ABB) y que permite la modificación de los voltajes de *threshold* de los transistores.

Las principales contribuciones del Marcellus son:

1. Un clúster de alto rendimiento y propósito general de 16 núcleos RISC-V programables con *software*.
2. Extensión de RISC-V, `XpulpNN` que añade instrucciones para operaciones Single Instruction Multiple Data (SIMD) para operaciones de producto escalar en DNNs con operaciones MAC&LOAD.
3. El RBE, un acelerador dedicado para capas de convolución 3×3 y 1×1 punto a punto con un *datapath* con precisión reconfigurable de 2 a 8 bits.
4. Un mecanismo ABB basado en bloques OCM y un lazo de control que permite la modificación al momento de los voltajes de *threshold*.

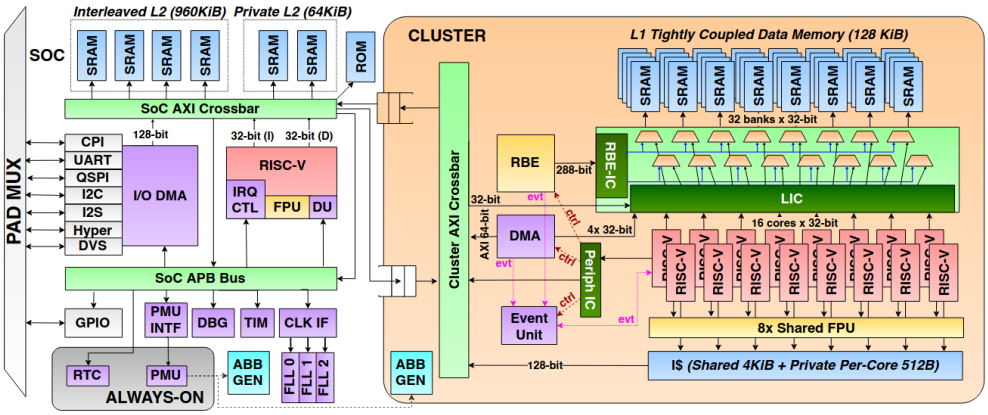



Figura 4.1: SoC Marcellus [31].

Atendiendo a la clasificación basada en [29], antes presentada y comenzando por los aspectos generales, es un acelerador *manycore* de dominio específico para la inferencia de, DNNs aunque se pueda utilizar para todo tipo de computación que aproveche el paralelismo de datos. Su programabilidad es limitada porque solo es programable a nivel de parámetros y no es reconfigurable debido a que no es posible alterar su microarquitectura durante el flujo de trabajo para el que se ha diseñado.

El acoplamiento en cuanto a conexión física es "integrado en chip", la conexión desde el SoC al clúster se ha realizado mediante AXI4. La interfaz de *software* permite el lanzamiento del *kernel* y la gestión de la memoria. La memoria común al clúster y al *host* es la L2 SRAM de 960 KiB, el motor *Direct Memory Access* (DMA) permite organizar y transferir 64 bits por ciclo entre la memoria L2 SRAM del SoC y el clúster. El acoplamiento con el SO

requiere de un controlador específico para el soporte del *software* del SoC (drivers del DMA, llamada de eventos, etc.) para orquestar los trabajos y sincronizar los eventos.

## 4.9. Ara

El acelerador Ara [32] permite la ejecución de la extensión vectorial de RISC-V, es escalable y eficiente energéticamente, tiene soporte para punto flotante. Ha sido planteado para la tecnología 22FDX FD-SOI de GlobalFoundries. Está compuesto por una serie de *lanes* idénticos, como se puede ver en la figura 4.2b, cada uno con una parte de los *register files* y las unidades funcionales.

Ara [32] fue diseñado para operar en conjunto con Ariane (posteriormente CVA6) [33] un procesador, *in-order*, *single-issue* y preparado para aplicaciones, comúnmente denominado, *application class*. Tiene soporte para operaciones de multiplicación/división, operaciones atómicas e incluye una Floating-Point Unit (FPU). Cuenta con un *pipeline* de seis etapas: generación de contador de programa (PC), búsqueda, decodificación, despacho, ejecución y *commit* de instrucciones. Las instrucciones vectoriales se decodifican parcialmente en el decodificador de instrucciones del Ariane para averiguar si la instrucción actual es vectorial y se terminan de decodificar dentro del Ara.

Las partes principales del Ara son:

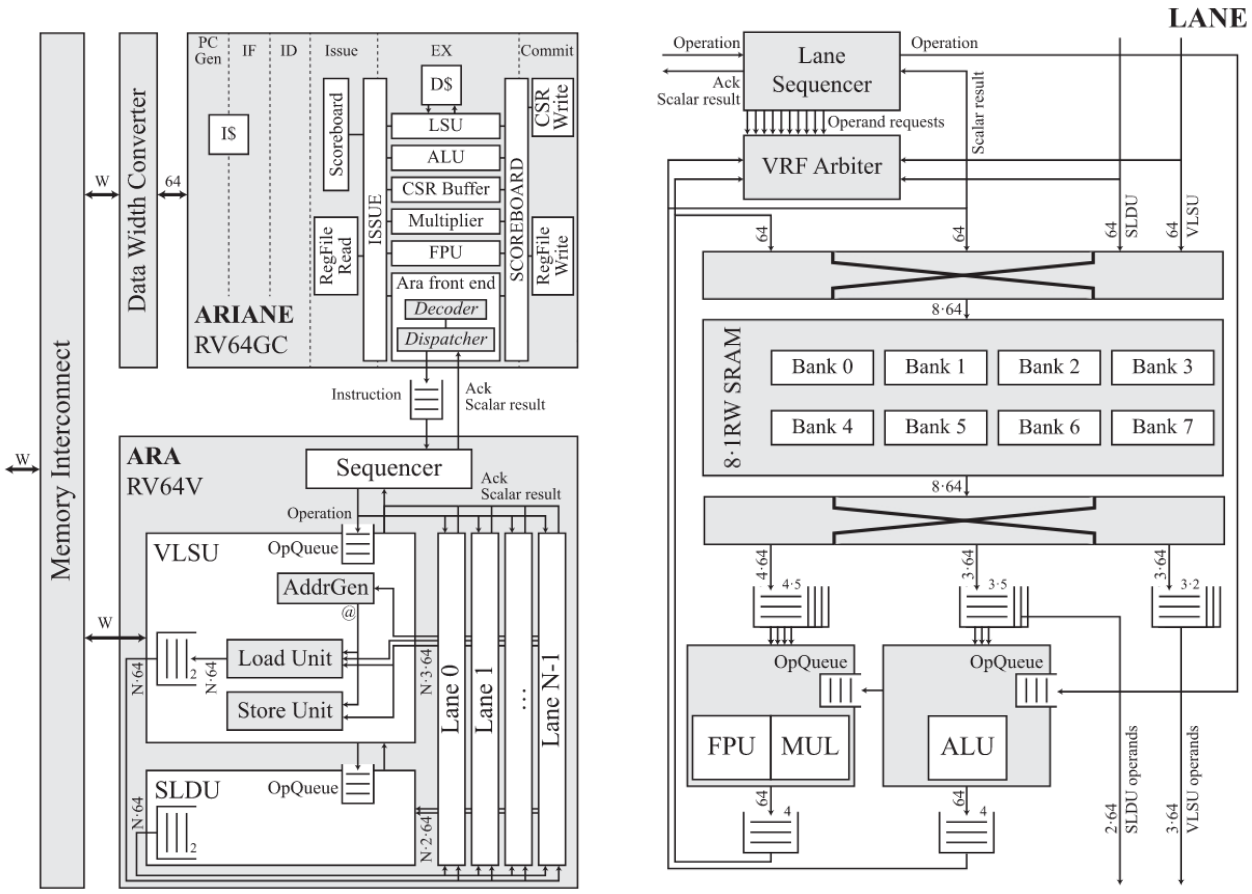
1. *Sequencer*: tiene en cuenta las instrucciones vectoriales que se están ejecutando en el Ara, las destina a las diferentes unidades de ejecución y notifica al Ariane.

2. *Slide unit (SLDU)*: maneja las operaciones vectoriales como *shuffles*, *slides* y potencialmente reducciones, aunque esto no está soportado. Está representado por el bloque "SLDU" en la figura 4.2a.

3. *Vector Load/Store Unit (VLSU)*: maneja los accesos a la memoria para obtener los datos vectoriales o guardar el resultado del cómputo. Genera las direcciones dependiendo del tipo de acceso de memoria y se comunica con el sistema de memoria a través de AXI. Está representado por el bloque "VLSU" en la figura 4.2a.

4. Organización de los *lanes*: es posible configurar en número de *lanes*, cada uno con la microarquitectura mostrada en la figura 4.2b, cada una con su propio *lane sequencer*, encargado de manejar la ejecución de hasta ocho instrucciones vectoriales.

Atendiendo a la clasificación de los aceleradores, este es un acelerador vectorial que ha adoptado la especificación RVV 1.0 para RISC-V, por tanto, su dominio es principalmente el de paralelismo de datos aunque es posible su utilización en todos aquellos ámbitos que la computación vectorial sea útil: ML, DSP, álgebra lineal, entre otros.




(a) Diagrama global del Ara.

(b) Diagrama detallado de un *lane*.

Figura 4.2: Diagramas del Ara

Su programabilidad es alta debido a que el acelerador está diseñado para ejecutar instrucciones vectoriales según la especificación, sin embargo, del mismo modo que el Marcellus [31], no es reconfigurable, puesto que la microarquitectura permanece fija. La estrategia de conexión con el *host*, es de tipo "integrada en chip", el *dispatcher* del Ara controla la interfaz entre el Ara y el puerto dedicado para el *scoreboard* del Ariane. La interfaz de *software* es de tipo "ejecución de operación", a diferencia de otros aceleradores, el hecho de implementar todo un conjunto estandarizado de instrucciones permite un mayor grado de flexibilidad en comparación a aceleradores en los que solo se pueden ejecutar un número reducido de comandos.

Tanto el *host* como el acelerador acceden a la memoria mediante la interconexión AXI. Una gran ventaja que presenta el Ara es su acoplamiento con el sistema operativo, debido a que el ara acelera exclusivamente operaciones que forman parte de la una extensión del ISA estandarizada, el sistema no necesita de controladores específicos, aquellas operaciones que hagan uso de las instrucciones vectoriales serán ejecutadas de forma transparente por el Ara. Finalmente, en cuanto a la arquitectura, tenemos los recursos de propósito general como las cachés L1, L2 y la FPU, los recursos específicos forman el procesador vectorial como el VRF de 64k bits y el Slide Unit (SLDU).

## 4.10. Ztachip

El Ztachip [34] es un acelerador especializado y diseñado para mejorar el rendimiento de tareas de inferencia de inteligencia artificial, está pensado para una implementación tanto en FPGA como en ASIC. El valor principal del Ztachip reside en la capacidad que presenta para la aceleración de 20 a 50 veces de tareas de IA cuando se compara con implementaciones RISC-V sin aceleración, incluso contando con extensiones vectoriales.

La innovación clave del Ztachip es su procesador tensorial a medida, figura 4.3, a diferencia de muchos otros aceleradores contemporáneos que están optimizados para un conjunto reducido de operaciones, normalmente, redes neuronales convolucionales, el procesador tensorial del Ztachip está diseñado para ofrecer versatilidad.

Se adapta a multitud de flujos de trabajo, desde algoritmos básicos de visión artificial como detección de bordes, flujo óptico, detección de movimiento y conversión de color hasta la ejecución de modelos complejos de IA desarrollados con TensorFlow [35]. Esta filosofía basada en la versatilidad se presenta como una solución hacia los problemas con el cómputo de tareas de procesamiento y postprocesamiento que pueden llegar a ser tan demandantes como la propia inferencia, especialmente en entornos restringidos como la IA *edge*.

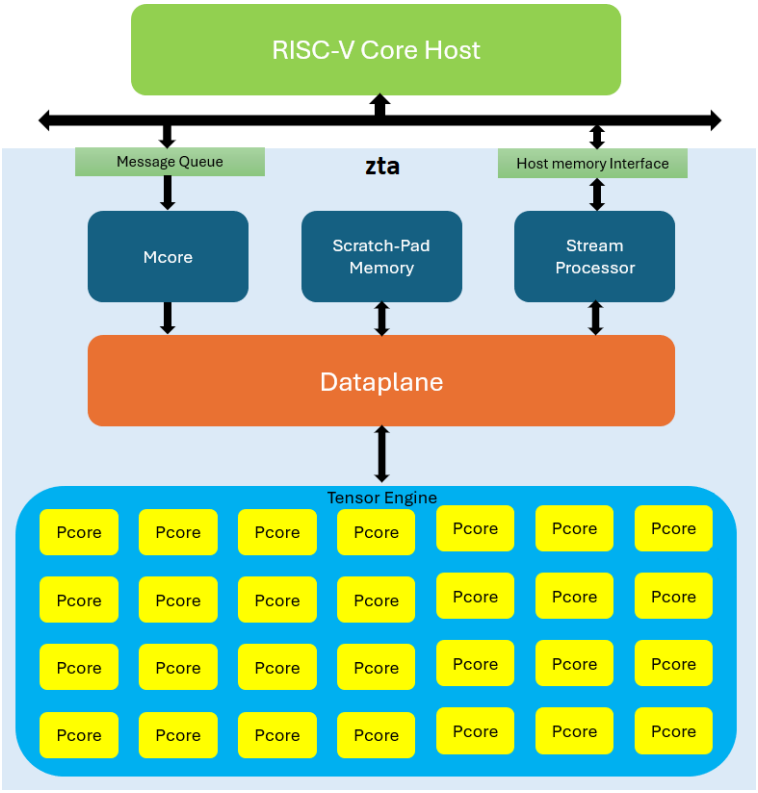



Figura 4.3: Arquitectura del Ztachip [34].

Para aprovechar el paralelismo disponible, el Ztachip introduce un nuevo paradigma de programación tensorial mediante un Domain-specific language (DSL) de sintaxis similar a C y su correspondiente compilador, juntos permiten a los programadores describir tanto el paralelismo de los datos como el de la computación.

Los componentes principales del Ztachip son la CPU VexRiscv [36], una DRAM y los siguientes periféricos que forman el propio acelerador:

- Un Mcore, el procesador de planificación.
- Un plano de comunicación para transmitir los datos e instrucciones al motor tensorial.
- Una memoria *scratch-pad* para almacenar datos temporalmente.
- Un procesador de flujo para gestionar la E/S de datos.
- Un motor tensorial con 28 Pcores que pueden configurarse para funcionar como matriz sistólica y realizar cómputo en memoria, como se puede ver en la figura 4.3. Cada uno de estos Pcores cuenta con: una ALU escalar y una ALU vectorial y soporta 16 hilos *hardware* de ejecución con memoria privada organizados según la figura 4.4.

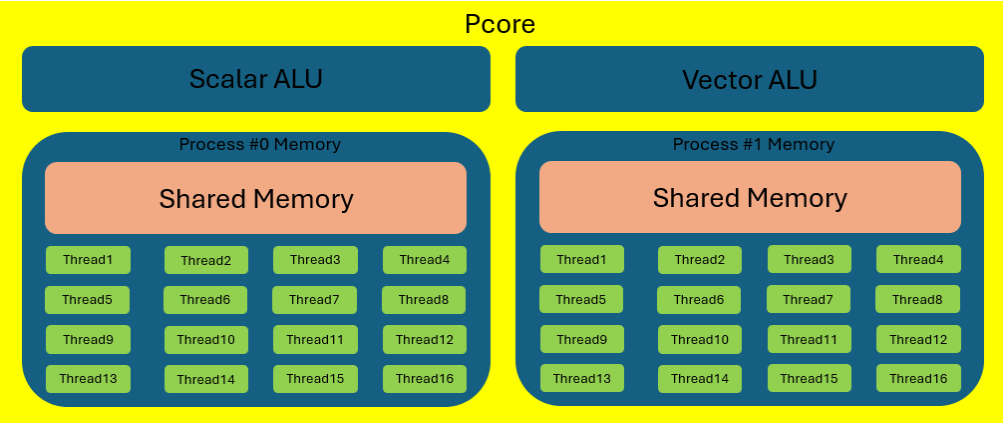



Figura 4.4: Organización interna de los Pcores.

Una vez más, atendiendo a la clasificación propuesta en el capítulo 4: los 28 Pcores configurables operan como una matriz sistólica, cada Pcore es un procesador vectorial multi-hilo de arquitectura Very Large Instruction Words (VLIW). Su dominio principal es el de visión por computadora e inferencia de ML (detección de bordes, flujos ópticos, etc.). La programabilidad del acelerador es alta, a nivel de operación, sin embargo, se introduce la problemática de la adaptación del desarrollador al DSL que se compila a las instrucciones propias del acelerador.

Su reconfigurabilidad es nula, puesto que el entramado lógico en ningún momento se puede alterar durante un flujo de ejecución común. En relación con el acoplamiento con el *host*, la estrategia de conexión es "integrado en chip", una vez más se utiliza AXI para su conexión con el *host*, en cuanto a la interfaz de *software*, las acciones que puede llevar a cabo el *host* son las de lanzamiento del *kernel* y gestión de memoria mediante las operaciones tensoriales.

La memoria común al *host* y el propio acelerador, es externa y se acopla mediante el bus AXI. En lo que respecta al acoplamiento con el SO, se propone la ejecución de aplicaciones *bare-metal*, por lo que no existe tal, en el caso de que se necesitara utilizar Linux, serían necesarios los controladores para la comunicación mediante el bus AXI. Finalmente, los recursos específicos son el Mcore y los Pcores.

### 4.10.1. Arquitectura *hardware*

**`ztachip` (módulo *top*)**

Este es el módulo principal del Ztachip, emplea los buses señalizados como `axilite_*` y `axi_*`, el bus AXILite permite la comunicación con el *host* RISC-V a través del cual se envían las instrucciones tensoriales al Ztachip. El bus Advanced eXtensible Interface (AXI) es empleado por el Ztachip para las transferencias de tipo DMA con la memoria externa. La ejecución de las instrucciones tensoriales se basa en: (1) coordinar las operaciones con datos tensoriales que mueven dichos datos entre el `sram_core`, la memoria interna del *core* y la memoria externa y (2) el despacho de las solicitudes de ejecución al *core* las cuales, a su vez, envían las peticiones de ejecución a una matriz de procesadores VLIW.

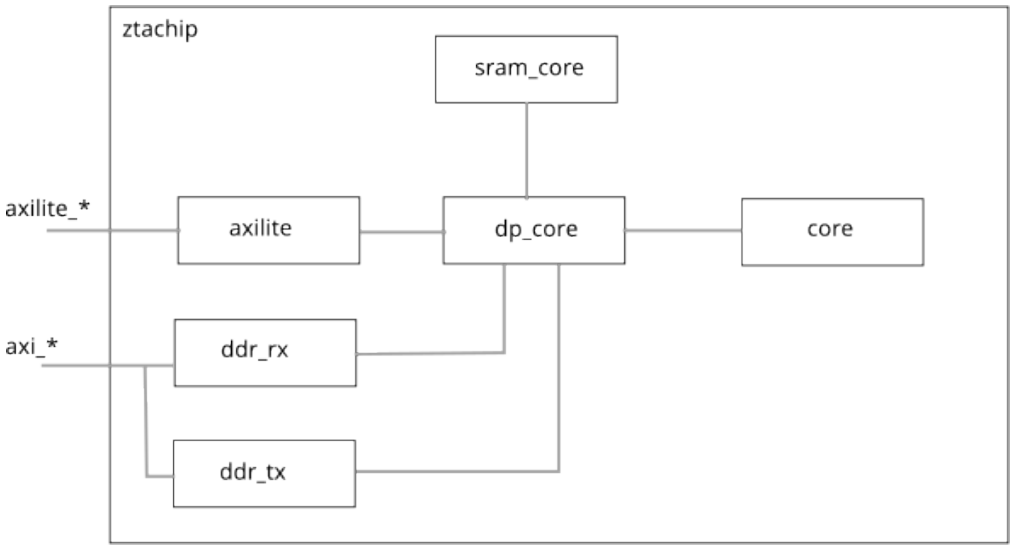



Figura 4.5: Esquema del Ztachip.

Los submódulos que se muestran en las figuras 4.5 y 4.6 son:

- **`axilite:`** bus AXILite que permite la conexión del Ztachip con el *host* RISC-V.
- **`sram_core:`** memoria temporal de tipo *scratch* que guarda memoria de forma temporal durante la transferencia de los datos tensoriales.
- **`ddr_rx:`** transmisión de los datos mediante DMA desde la memoria interna del *core* hacia la memoria externa.
- **`ddr_tx:`** recepción de los datos mediante DMA desde la memoria externa del *core* hacia la memoria interna.
- **`core:`** unidad de ejecución de aritmética tensorial, compuesta por una matriz de procesadores VLIW.
- **`dp_core:`** procesador central que coordina todas las operaciones dentro del Ztachip como las transferencias de memoria y la ejecución en la matriz de procesadores VLIW.

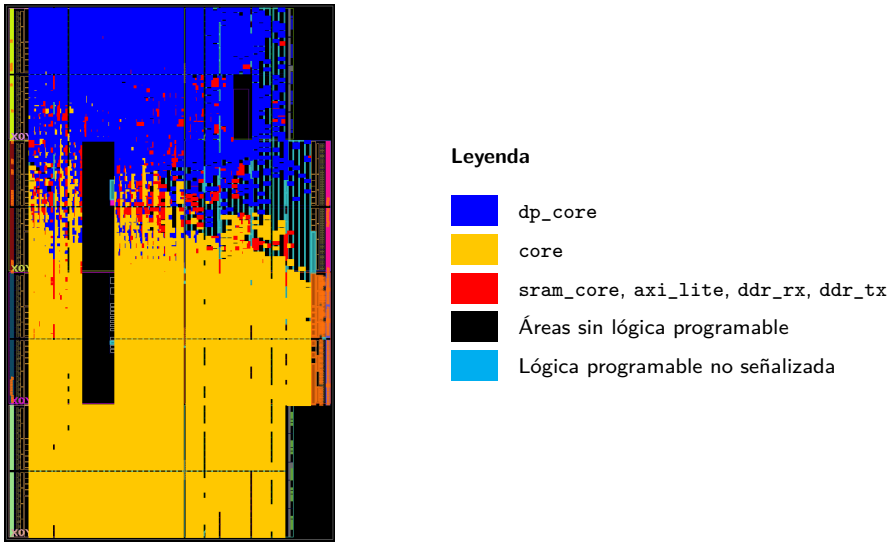



Figura 4.6: Resultado de la implementación con los submódulos señalizados.

**`ztachip.dp_core`**

Es el módulo encargado de ordenar las instrucciones tensoriales (operaciones con datos o de ejección). A través de `dp_fetch` recibe las instrucciones tensoriales del *host*, cada instrucción tiene asociado uno de los hilos *hardware* disponibles, de modo que uno puede encargarse de la ejecución un operador tensorial, mientras que el otro realiza la transferencia de datos tensoriales. Este módulo, además, es responsable de decodificar las instrucciones tensoriales. Las operaciones de transferencia de datos son procesadas por `dp_gen_core`, `dp_source` y `dp_sink`; se pueden ejecutar hasta dos operaciones con datos de forma simultánea, por ejemplo, entre la memoria interna y la memoria externa y entre la memoria de tipo *scratch* y la memoria interna.

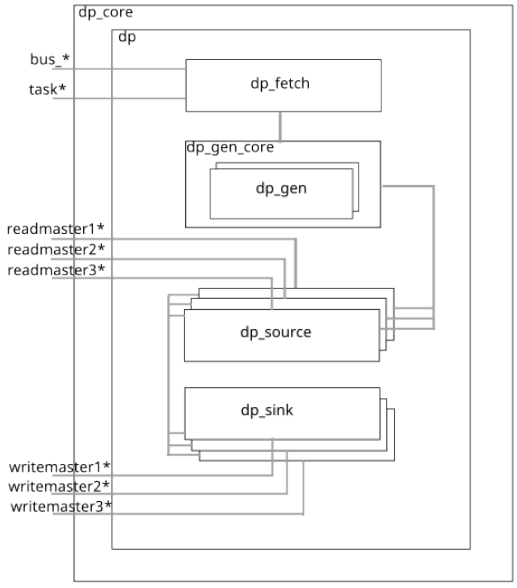



Figura 4.7: Esquema del procesador tensorial.

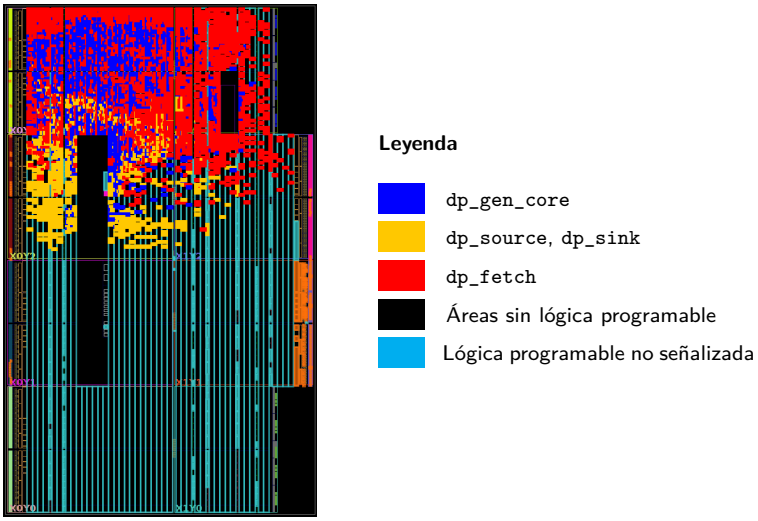



Figura 4.8: Resultado de la implementación con los submódulos del módulo `dp_core`.

Los submódulos que se muestran en las figuras 4.7 y 4.8 son los siguientes:

- **`dp_fetch`:** recibe las instrucciones tensoriales del *host*, aquellas que son de datos tensoriales son enviadas al `dp_gen_core` en el momento apropiado. En cambio, las instrucciones de operadores tensoriales son enviadas al `core` a través de la interfaz `task*`.
- **`dp_gen_core`:** recibe las instrucciones de operaciones de datos tensoriales provenientes del `dp_fetch`, tras esto se generan las direcciones de memoria para la transferencia de datos.

- **`dp_source:`** responsable de generar las transferencias DMA de lectura para la fuente de las operaciones con datos tensoriales; los datos recibidos son reenviados al `dp_sink` apropiado, responsable de la transferencia DMA de los datos recibidos al destino.

- **`dp_sink:`** genera la transferencia de escritura DMA para el destino de la operación de datos tensoriales, recibe la información relativa a los datos y las direcciones para la transferencia del módulo `dp_source`, ya mencionado.

#### `ztachip.core`

Módulo responsable de la ejecución de las operaciones tensoriales, está compuesto por una matriz de pcores. Las peticiones de ejecución provienen del módulo `dp_core` a través de las señales `task*`. El módulo `instr` controla la ejecución en todos los pcores. Estos son procesadores de tipo VLIW de baja complejidad que están formados principalmente por una ALU con memoria, en configuración *lock-step* con el resto. Los módulos que se muestran en las figuras 4.9 y 4.10 son:

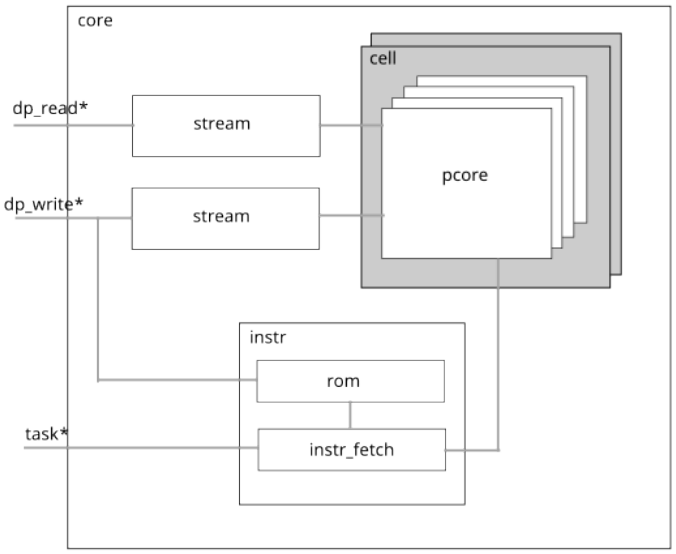



Figura 4.9: Esquema del módulo `core`, responsable de la ejecución de operaciones tensoriales.

- **`stream:`** mapea los flujos de datos entre la entrada y la salida, uno de los dos mapea los datos que serán escritos en la memoria interna, el `core`, el segundo mapea los datos según se leen de dicha memoria interna.

- **`cell:`** agrupaciones de 4 pcores con el propósito de mejorar el *fan-out* de las señales del bus.

- **`cell.pcore:`** array de procesadores VLIW, las operaciones tensoriales se ejecutan en multitud de estos pcores.

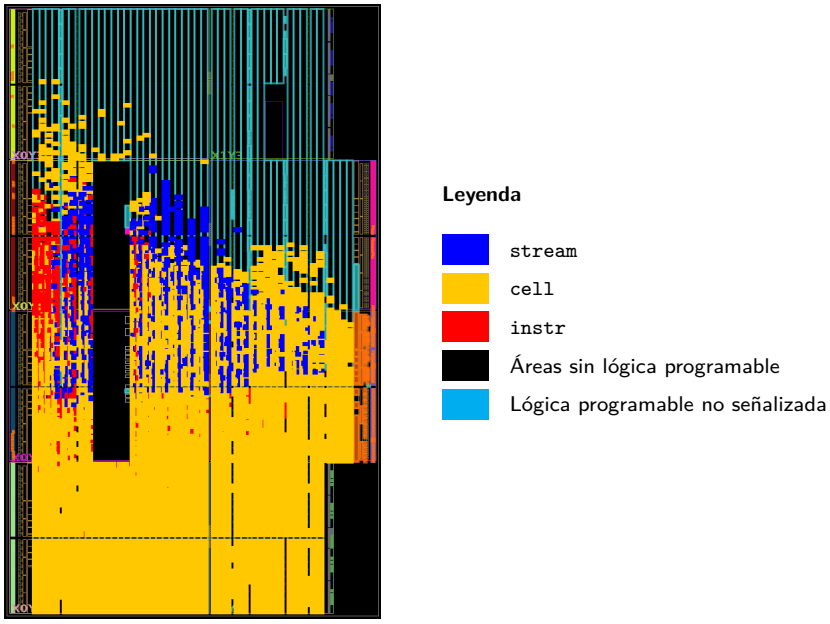



Figura 4.10: Resultado de la implementación con los submódulos del módulo `core`.

- **`instr:`** procesador maestro de todos los pcores, son ALUs que se ejecutan en modo *lock-step* con el resto.

- **`instr.instr_fetch:`** realiza la programación de las ejecuciones, sobre cada hilo. Los pcores tienen 16 hilos de *hardware* cada uno, que realizan su ejecución en modo *round-robin*. Debido a que las instrucciones VLIW son complejas y necesitan de un *pipeline* muy profundo, se utiliza un esquema multi-hilo para reducir el impacto de la latencia de las instrucciones.

- **`instr.rom:`** guarda la instrucción VLIW, todos los pcores comparten la misma instrucción, debido a que la ejecución es en *lock-step*, solo una instrucción se ejecuta en cada ciclo de reloj.

#### `ztachip.core.stream`

Permite mapear datos de forma arbitraria entre la entrada y la salida, según la figura 4.11. En el Ztachip realiza mapeos de memoria justo antes de la escritura y justo después de la lectura, siguiendo la ecuación 4.1.

$$y = \text{lookup\_quotient}(x[11:5]) \; + \; x[4:0] \cdot \text{lookup\_remainder}(x[4:0]) \tag{4.1}$$

El módulo `stream` permite realizar hasta 4 operaciones de mapeo de forma simultánea, habilitando la interpolación lineal multipunto entre la entrada y la salida, comúnmente utilizada en las aplicaciones de IA para las funciones de activación según se escriben los datos a memoria. Tiene una contribución limitada en recursos, según se puede ver en la figura 4.12.

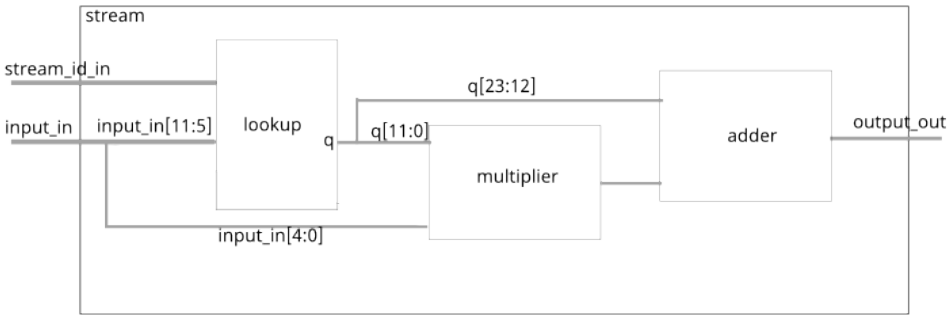



Figura 4.11: Esquema del módulo `stream`.

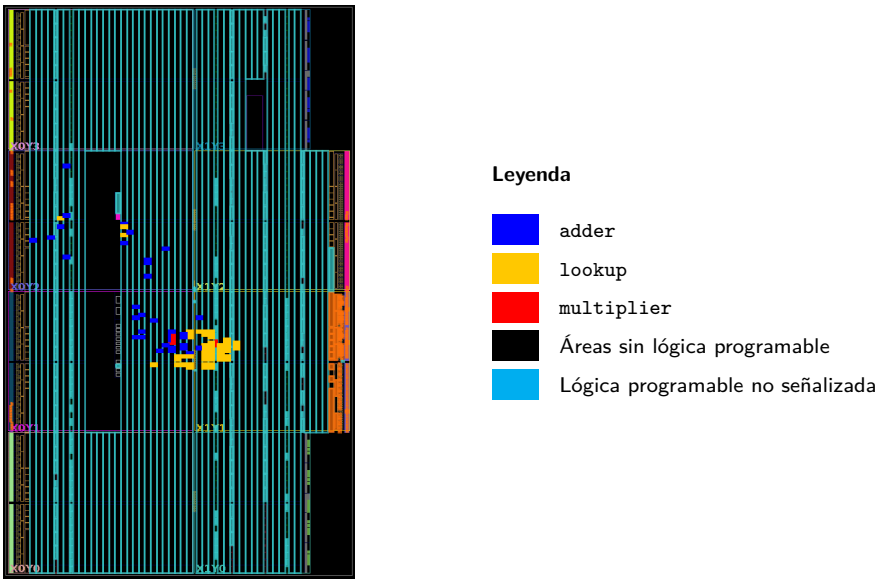



Figura 4.12: Resultado de la implementación con los submódulos del módulo `stream`.

**`ztachip.core.pcore`**

Como se ha mencionado con anterioridad, ejecuta las operaciones tensoriales en paralelo. Las instrucciones VLIW son, como su nombre indica, de mayor longitud que las instrucciones comunes, en las que se codifican multitud de sub-operaciones:

- Cálculo de las direcciones del parámetro de entrada y los 2 de salida.
- Búsqueda en el `register_back` de los vectores de entrada.
- Búsqueda del operando *accumulator* del `xregister_file`, dichos operandos se emplean en el cálculo de las operaciones de tipo Fused Multiply Add (FMA).

- Cálculos enteros en la `ialu` y cálculos vectoriales en banco de la `alu`.

- Guardado de los resultados, *accumulator* de 32 bits o vector de 12 bits de la `alu` en el `xregister_file` o en el `register_bank`.

Como todas las sub-operaciones antes mencionadas se ejecutan de forma simultánea, el *pipeline* de ejecución puede durar hasta 14 ciclos. Sin embargo, cuentan con 16 hilos *hardware* que mitigan las penalizaciones de latencia que introduce la complejidad intrínseca de instrucciones VLIW, de modo que cada hilo ejecuta una de las sub-operaciones, resultando así, una instrucción por ciclo por pcore. Presenta una gran contribución al uso de recursos, como se puede ver en la figura 4.14. En la ecuación 4.2 se muestra un ejemplo de operación que se puede realizar en una instrucción VLIW en el Ztachip, en un ciclo de reloj.

$$z[i++] = x[i+2] + y[i+3]; \tag{4.2}$$

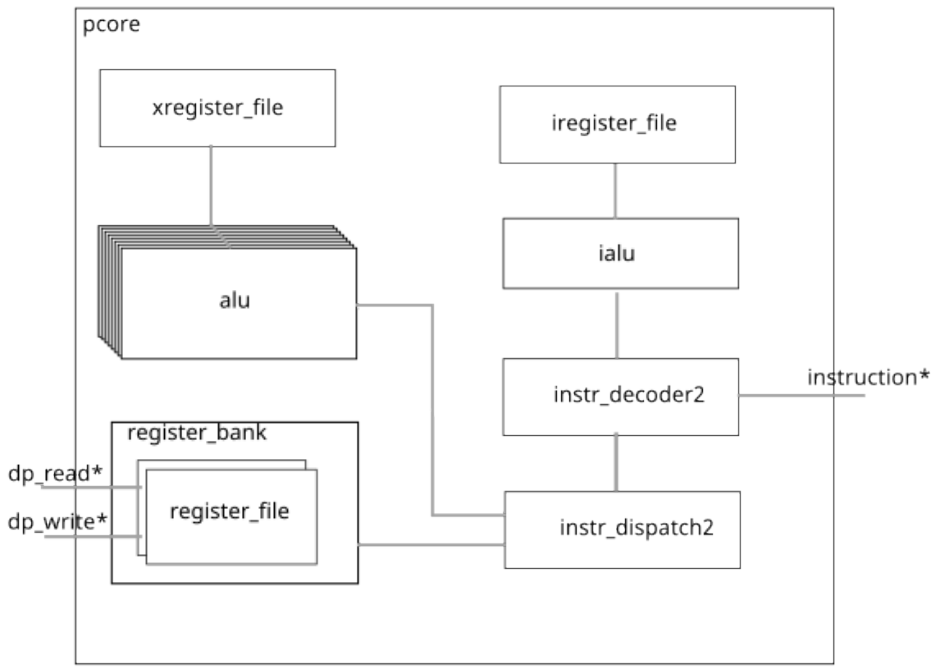



Figura 4.13: Esquema del módulo `pcore`.

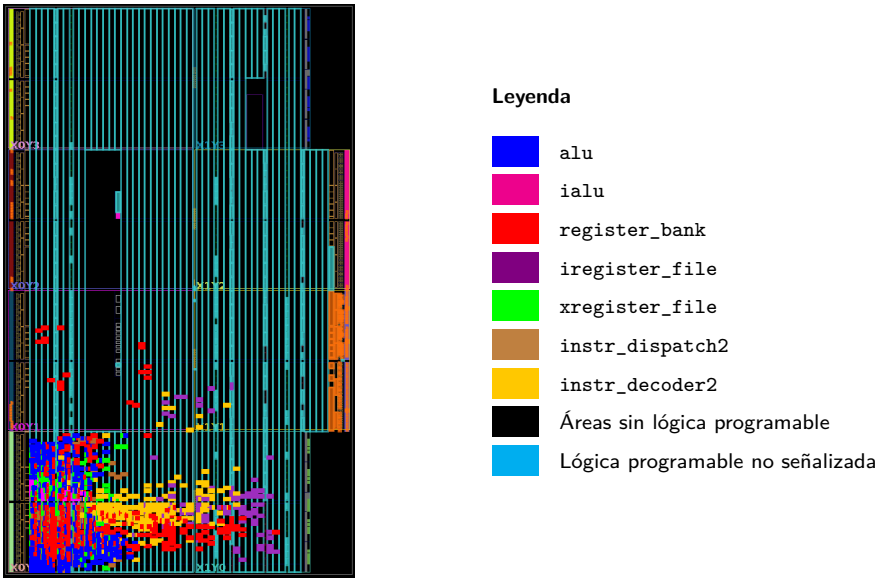



Figura 4.14: Resultado de la implementación con los submódulos del módulo `pcore`.

#### `ztachip.core.pcore.register_bank`

Implementa la memoria interna de los pcores, de ahí su uso de recursos, figura 4.16. Está particionada en dos páginas, cada una asociada a uno de los 2 hilos del procesador tensorial, es la razón por la que Ztachip es capaz de acceder la primera página de memoria interna con el primer hilo y ejecutar una operación tensorial con el segundo hilo sobre la segunda página de memoria.

Es importante incidir en que las operaciones tensoriales dependen únicamente de la memoria interna sin referencia alguna a la externa. Las operaciones con datos (instrucciones tensoriales con datos) están completamente desacopladas de las operaciones de ejecución (instrucciones tensoriales de ejecución).

La memoria interna está estructurada en dos tipos:

- **Privada**. La memoria es privada entre los 16 hilos, las palabras de memoria privada están entrelazadas según el hilo, como se puede ver en la figura 4.15.
- **Compartida**. La memoria es compartida entre los 16 hilos dentro de cada pcore.

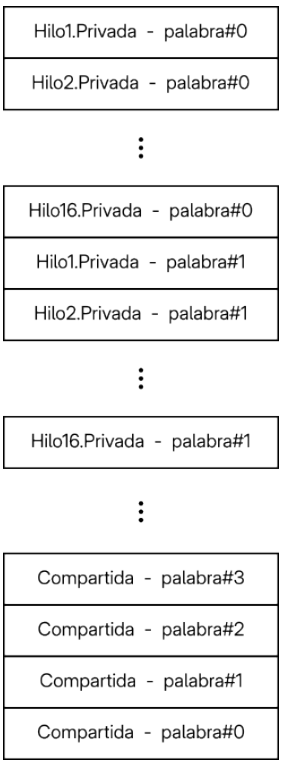



Figura 4.15: Esquema de organización de la memoria interna de los **pcores**.

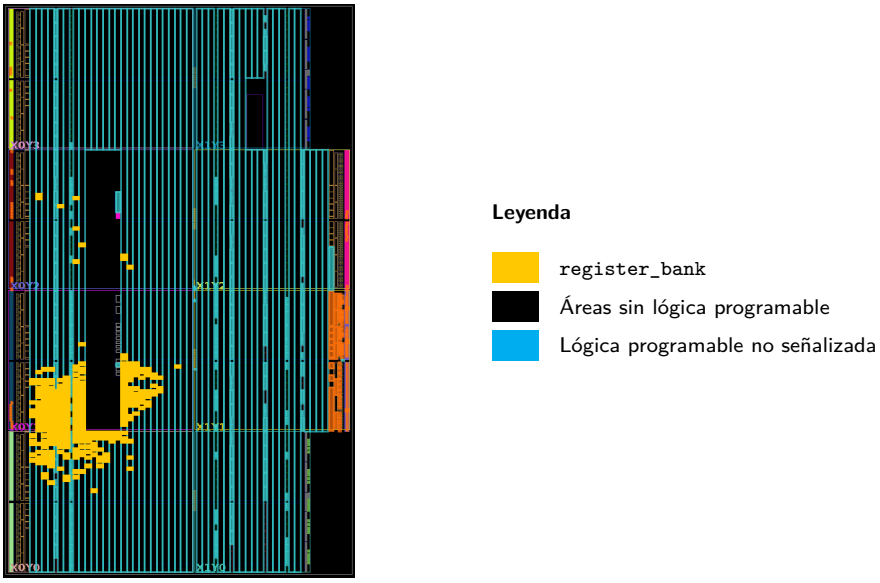



Figura 4.16: Resultado de la implementación con los submódulos del módulo `register_bank`.

`ztachip.core.pcore.instr_decoder2`

Componente que decodifica las instrucciones VLIW, estas realizan 3 operaciones principales:

- Operaciones en la `alu`: Operaciones matemáticas sobre datos vectoriales.
- Operaciones en la `ialu`: Operaciones matemáticas sobre enteros.
- Operaciones con saltos: Las lleva a cabo el procesador maestro, `instr` debido a que todos los procesadores VLIW se ejecutan en modo *lockstep* y comparten saltos.

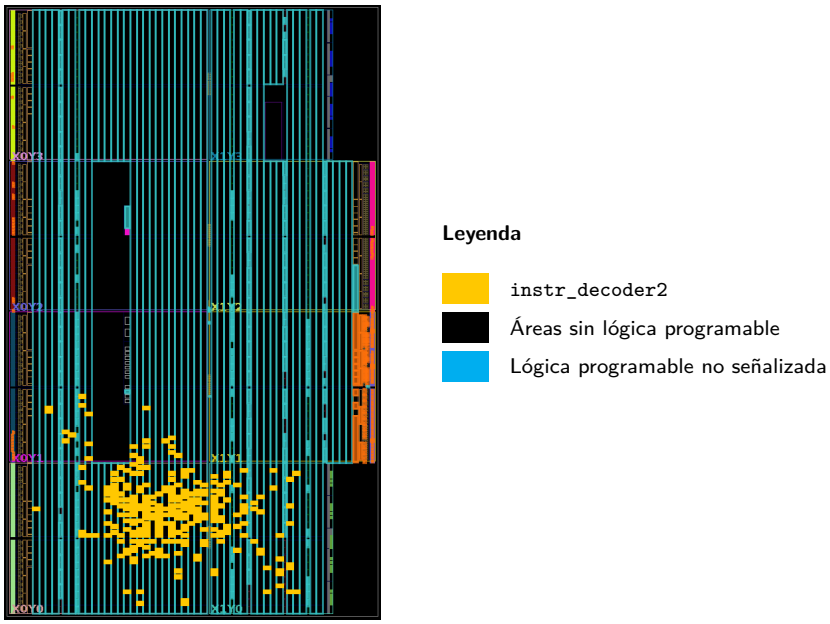



Figura 4.17: Resultado de la implementación del `instr_decoder2`.

| **MU – instrucción para `pcore.alu`** | | |
|---|---|---|
| Campo | Bits | Descripción |
| `oc` | 5 | Código de operación |
| `save` | 1 | Guardar `alu.y_out` → `xregister_file` |
| **Parámetro `x3`** | | |
| `vector` | 1 | 1 si es vector, 0 si es escalar |
| `addr` | 12 | Dirección de memoria interna de `x3` |
| `attr` | 4 | Atributo |
| **Parámetro `x1`** | | |
| `vector` | 1 | 1 si es vector, 0 si es escalar |
| `addr` | 12 | Dirección de memoria interna de `x1` |
| `attr` | 4 | Atributo |
| **Parámetro `x2`** | | |
| `vector` | 1 | 1 si es vector, 0 si es escalar |
| `addr` | 12 | Dirección de memoria interna de `x2` |
| `attr` | 4 | Atributo |
| **Parámetro `y`** | | |
| `vector` | 1 | 1 si es vector, 0 si es escalar |
| `addr` | 12 | Dirección de memoria interna de `y` |
| `attr` | 4 | Atributo |
| **IMU – instrucción para `pcore.ialu`** | | |
| `oc` | 5 | Código de operación |
| `x1` | 4 | Parámetro `x1`, se mapea a `ialu.x1_in` |
| `x2` | 4 | Parámetro `x2`, se mapea a `ialu.x2_in` |
| `y` | 4 | Parámetro `y`, se mapea a `ialu.y_out` |
| `const` | 13 | Campo de constante |
| **CTRL – instrucción de salto (en función del resultado de IMU)** | | |
| `oc` | 5 | Código de operación de salto basado en el valor de `IMU.y` |
| `addr` | 11 | Dirección de código a la que saltar |

Tabla 4.1: Formato de las instrucciones tensoriales.

La suma total de los bits en la 4.1 es 120, los 8 bits restantes pertenecen a las señales:

- **`instruction_vm_in`** (1 bit): *flag* de modo vector.
- **`instruction_data_model_in`** (2 bits): selector de modelo de datos para la generación de direcciones.
- **`instruction_tid_in`** (5 bits): identificador de hilo.

**`ztachip.core.pcore.ialu`**

Componente encargado de las operaciones con enteros descritas como parte del formato de instrucción visto en la tabla, 4.1. Emplea los siguientes registros `R0`–`R7` y `VMASK`. El cómputo realizado por la `ialu` es comúnmente utilizado para el cálculo de las condiciones en los bucles, índices de matrices, cálculo de punteros a memoria, cálculo de direcciones, etc.

`ztachip.core.pcore.alu`

Es la ALU encargada de realizar la aritmética lineal necesaria para las tareas de IA y procesado de vídeo. Las operaciones no lineales son realizadas por el módulo `stream`, como se puede ver la figura 4.11. Este módulo `alu`, figuras 4.18 y 4.19, está compuesto por los siguientes módulos aritméticos:

- Multiplicación de dos valores de 12 bits.
- Suma del resultado de la multiplicación con un valor de 32 bits, *accumulator*.
- Operación de desplazamiento.
- Comparación booleana del resultado.
- Conversión del resultado de 32 a 12 bits.

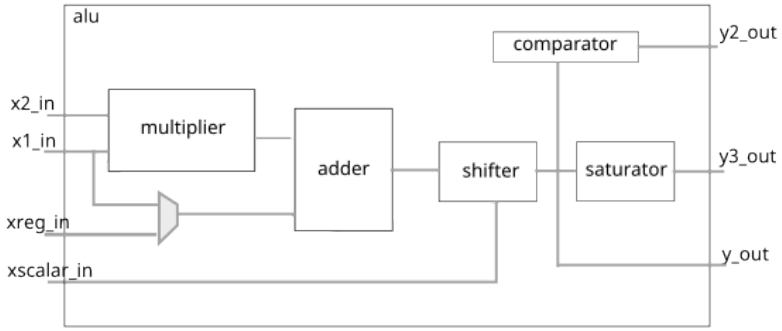



Figura 4.18: Esquema de organización de la `alu`.

### 4.10.2. Diseño *software*

Como se ha analizado con anterioridad, el dominio del Ztachip pertenece a aquellas aplicaciones que se pueden expresar como una serie de operaciones tensoriales. Se describen dos tipos principales de operaciones:

- Operaciones de datos tensoriales: incluyen operaciones como la copia de datos, transposición de tensores y redimensionamientos.
- Operaciones de operadores tensoriales: realizan tareas de cálculo sobre uno o varios tensores.

Para mejorar la eficiencia a la hora de interactuar con el acelerador se requiere de DSL para abstraer las complejidades de la arquitectura, en el caso del Ztachip está compuesto por dos tipos de programas: tensor-core (*.m) y p-core (*.p).

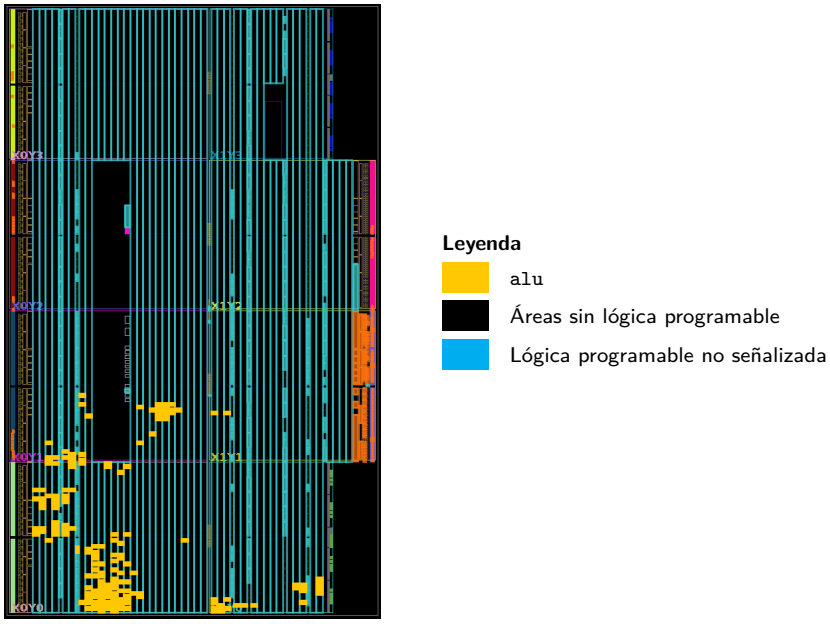



Figura 4.19: Resultado de la implementación de una `alu`.

### Programas tensor-core

Los programas tensor-core se ejecutan en el *host* RISC-V, no en el Ztachip. Tienen la extensión *.m, son visualmente parecidos a un programa escrito en C, solo que con las instrucciones tensoriales, antes de que se pueda compilar con un compilador estándar para C con soporte para RISC-V es necesario utilizar el compilador específico para este tipo de programas. En el bloque de código 5 se puede ver un ejemplo de un programa tensor-core.

```
// vector_add.m
// Suma de of 2 vectores de 4096 palabras de longitud

>DTYPE(INT16)PCORE[0:7].THREAD[0:15].example::x[0:3][0:7] <=
 ↪  DTYPE(INT16)MEM(x_p)[0:4095];
>DTYPE(INT16)PCORE[0:7].THREAD[0:15].example::y[0:3][0:7] <=
 ↪  DTYPE(INT16)MEM(y_p)[0:4095];
>EXE_LOCKSTEP(example:vector_add);
>DTYPE(INT16)MEM(z_p)[0:4095] <=
 ↪  DTYPE(INT16)PCORE[0:7].THREAD[0:15].example::z[0:3][0:7];
```

Código fuente 5: Ejemplo de programa tensor core.

En el ejemplo 5, `x`, `y` y `z` son matrices de 4x8 alojadas en la memoria privada de los pcores, como hay 8 pcores con 16 hilos *hardware* por pcore, `x`, `y` y `z` son tensores de 8x16x4x8. En el ejemplo antes mencionado, se copian 4096 palabras desde la memoria externa al tensor `x` y a `y`. Finalmente, se ejecuta una operación tensorial,

`example::vector_add` que suma `x` e `y` para dar lugar a `z`.

#### Programas p-core

Los programas p-core (*.p) implementan las operaciones tensoriales a las que se hace referencia en los programas tensor-core. Estas implementaciones se materializan como operaciones dentro de un objeto C++. Debido a las limitaciones de memoria, todos estos objetos se instancian solo una vez y ocupan la misma región de memoria, de modo que el usuario ha de diseñar estas operaciones para que nunca se requiera más de una en el mismo instante.

```
// vector_add.p

vint16 example::x[4];
vint16 example::y[4];
vint16 example::z[4];

_kernel void example::vector_add()
{
   int i;
   for(i=0;i < 3;i++)
      z[i]=x[i]+y[i];
}
```

Código fuente 6: Ejemplo de programa p-core.

### 4.10.3. Comparativa con la programación tradicional

En la programación convencional, los datos y la ejecución están entrelazados entre sí, de modo que si se toma el ejemplo anterior de referencia, en primer lugar será necesario cargar los vectores a memoria uno a uno y luego realizar la suma. Esto da lugar a grandes penalizaciones en cuanto a *delay* si los datos no están disponibles en el caché. En cambio, con el Ztachip los datos se cargan desde la memoria externa de forma continua con precargas y sin introducir *delays*, esto se traduce en un gran incremento en la eficiencia con la que se emplea el ancho de banda de la memoria. Además, hay que tener en cuenta que en el ejemplo 6 cada suma vectorial se lleva a cabo en diferentes hilos y en diferentes pcores, esto confiere al Ztachip grandes capacidades de procesamiento en paralelo.

## 4.11. Conclusiones

En este capítulo se ha presentado el concepto de acelerador *hardware*, la razón de su necesidad, su clasificación a través de cuatro dimensiones: (1) Aspectos generales, (2) Acoplamiento con el host, (3) Arquitectura y (4) Aspectos de *software*.

Además se han descrito diferentes casos de implementación siguiendo diferentes arquitecturas existentes, como es el caso de Marcellus, un sistema basado en un cluster de RISC-V diseñado para DNN. Se explica Ara, planteado para acelerar extensiones vectoriales y finalmente se entra en detalle en Ztachip, una arquitectura sistólica que

implementa un motor tensorial de 28 cores, fuertemente paralelizada, construida como acelerador *hardware* de un sistema basado en RISC-V.

# Capítulo 5

# Implementación de procesadores con arquitectura RISC-V

> Las implementaciones RISC-V en FPGA son excelentes aceleradores debido a la capacidad de seleccionar extensiones para una implementación específica.
> *Adaptado de Efinix, Inc.*

En este capítulo se presentan las diversas implementaciones RISC-V estudiadas y se concluye con la solución elegida tras haber estudiado tanto implementaciones independientes de RISC-V como aceleradores completos.

En primer lugar, se sintetizan las principales características funcionales de cada implementación independiente (NEORV32, VexRiscv y CVA6) y se realiza el prototipado durante la fase de exploración, para comprobar el funcionamiento básico de cada caso, atendiendo en cada caso también a cuestiones subjetivas como calidad de la documentación y facilidad de integración. A continuación se justifica la utilización del acelerador final seleccionado para la realización del trabajo y se concluye con el flujo adoptado para la implementación del acelerador sobre la placa de prototipado VCU128. El capítulo concluye con un estudio de distintas implementaciones ASIC realizadas a partir de aceleradores vectoriales, estableciendo una ruta hacia el ASIC.

## 5.1. Metodología

Para llegar a la solución final, se ha llevado a cabo un estudio preliminar para determinar los objetivos a cumplir y qué conocimientos han sido necesarios adquirir para abarcar el problema que soluciona el presente TFG. Tras haber adquirido unos conocimientos iniciales, presentados en los capítulos 2, 3 y 4, se ha realizado un prototipado de los aceleradores considerados al final del capítulo 4 y de las implementaciones que se presentan en este capítulo. Es importante tener en cuenta que, aunque las implementaciones RISC-V a continuación se presentan de forma breve, el ejercicio de prototipado representa un esfuerzo en multitud de líneas y a diversos niveles (*software*, *hardware*,

documentación, etc) y que, de ser incluido, opacaría el principal fin del TFG. Tras encontrar una plataforma que, de forma preliminar, se ha alienado con los objetivos mediante prototipados adicionales y la validación de las capacidades, se determina si es realmente apta para, una vez más, cumplir con los objetivos fijados. En aquellos casos en que la plataforma no se ha adaptado, se ha regresado a la fase de prototipado hasta alcanzar una solución preliminar adicional, siguiendo el diagrama de flujo en la figura 5.1.

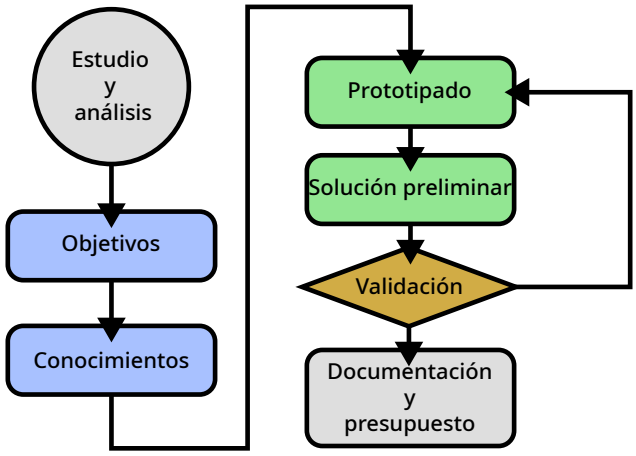



Figura 5.1: Metodología seguida para llegar a la solución final.

## 5.2. NEORV32

El NEORV32 [37] es un procesador *open-source*, basado en la arquitectura RISC-V de 32 bits. Está diseñado para trabajar como procesador auxiliar en conjunto a diseños SoC de mayor complejidad o como un microcontrolador *standalone* personalizable. Entre las posibilidades que habilita el SoC existe tanto una configuración de núcleo único como Symmetric Multiprocessing (SMP) de doble núcleo. Además de todas las extensiones estándar (RV32 I/A/B/C/E/M/U/X) y multitud de extensiones opcionales (Zicsr, Zifencei, Zfinx, Zicond, etc.). Dispone de una unidad de funciones personalizadas *Custom Functions Unit* (CFU) para incluir instrucciones específicas definidas por el usuario y permitir así, acelerar las aplicaciones. El NEORV32 está separado en módulos opcionales según se puede ver en la figura 5.2 que pueden ser configurados dependiendo de las necesidades del sistema.

### 5.2.1. Arquitectura de la CPU

Muchos procesadores priman el rendimiento haciendo uso de ejecución fuera de orden y pipelines profundos, sin embargo, estas decisiones son a cambio de un incremento de la complejidad y en última instancia, área en el silicio. En el otro extremo optimizando la simplicidad, existen los diseños monociclo en los que cada instrucción se completa transcurrido un periodo, esto da lugar a la menor cantidad posible de lógica de control, pero también a una ruta crítica muy larga que deriva en frecuencias de funcionamiento bajas. El NEOV32, implementa un diseño multiciclo híbrido, que ofrece un compromiso entre los niveles de consumo, área, frecuencia máxima y rendimiento entre los dos extremos expuestos con anterioridad. En este modelo de diseño, cada instrucción se descompone en una serie de microoperaciones que se ejecutan en ciclos sucesivos. Para incrementar el rendimiento el *frontend* (fetch) y el *back-end* (execution) están desacoplados mediante un *First-In First-Out* (FIFO). De modo que se

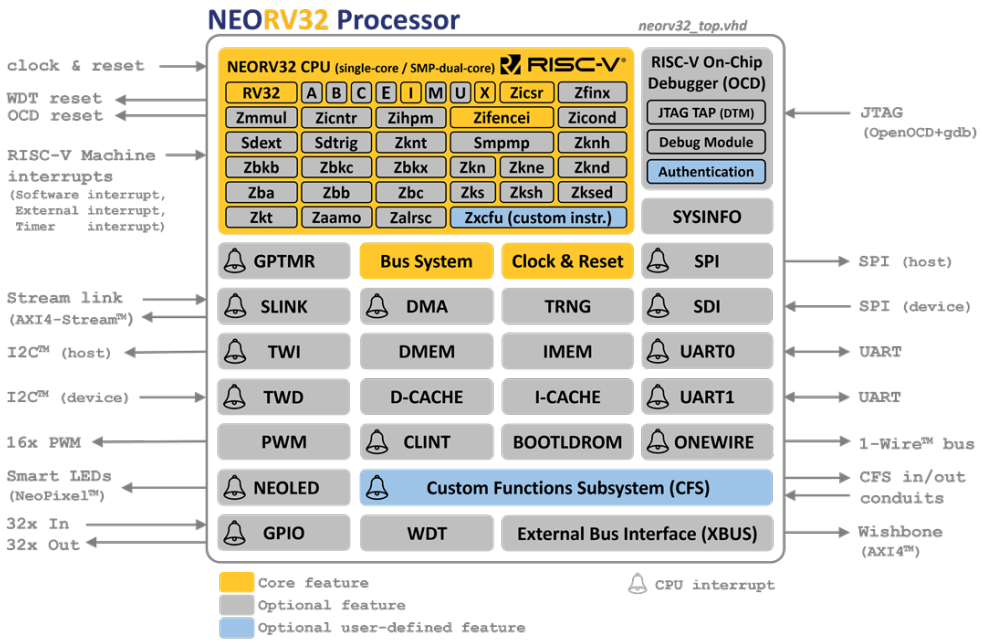



Figura 5.2: Procesador NEORV32 (diagrama de bloques).

puede hacer la búsqueda de la instrucción mientras que se terminan de ejecutar las instrucciones previas.

### 5.2.2. Estructura de archivos del proyecto

El repositorio del NEORV32 está compuesto por cuatro directorios que representan a su vez las cuatro categorías distintivas de ficheros:

- **docs:** Documentación, ficheros a parir de los cuales se generan las páginas de información (*userguide* y *datasheet*).
  - *Userguide*: Guía de usuario con los aspectos generales orientado al desarrollo de programas para su futura ejecución en el NEORV32, detalles sobre la simulación y cómo utilizar el proyecto sobre diferentes plataformas.
  - *Datasheet*: Información detallada de la microarquitectura del procesador y de las capacidades del SoC.
- **rtl:** Ficheros VHDL que describen al procesador en su conjunto, como se puede ver en la figura 5.3, junto con plantillas VHDL para instanciar el procesador y ficheros para la verificación de la ISA.

```
rtl/core
├─neorv32_application_image.vhd - Imagen de inicialización de aplicación IMEM (package)
├─neorv32_boot_rom.vhd          - ROM del bootloader
├─neorv32_bootloader_image.vhd  - Imagen de memoria ROM del bootloader (package)
├─neorv32_bus.vhd               - Módulos de infraestructura del bus SoC
├─neorv32_cache.vhd             - Módulo de caché genérico
├─neorv32_clint.vhd             - Interruptor local del núcleo
├─neorv32_clockgate.vhd         - Interruptor de control de reloj genérico
├─neorv32_cfs.vhd               - Subsistema de funciones personalizadas
├─neorv32_cpu.vhd               - ENTIDAD PRINCIPAL DE LA CPU NEORV32
├─neorv32_cpu_alu.vhd           - Unidad aritmética/lógica
├─neorv32_cpu_control.vhd       - Control de la CPU, sistema de excepciones y CSRs
├─neorv32_cpu_counters.vhd      - Contadores de hardware (extensiones Zicntr y Zihpm)
├─neorv32_cpu_cp_bitmanip.vhd   - Coprocesador de manipulación de bits (extensión B)
├─neorv32_cpu_cp_cfu.vhd        - Coprocesador de instrucciones personalizadas (extensión Zxcfu)
├─neorv32_cpu_cp_cond.vhd       - Coprocesador condicional entero (extensión Zicond)
├─neorv32_cpu_cp_crypto.vhd     - Coprocesador de criptografía escalar (extensiones Zk*/Zbk*)
├─neorv32_cpu_cp_fpu.vhd        - Coprocesador de coma flotante (extensión Zfinx)
├─neorv32_cpu_cp_muldiv.vhd     - Coprocesador de multiplicación/división (extensión M)
├─neorv32_cpu_cp_shifter.vhd    - Coprocesador de desplazamiento de bits (ISA base)
├─neorv32_cpu_decompressor.vhd  - Decodificador de instrucciones comprimidas (extensión C)
├─neorv32_cpu_frontend.vhd      - Instrucciones fetch e issue
├─neorv32_cpu_icc.vhd           - Unidad de comunicación entre núcleos
├─neorv32_cpu_lsu.vhd           - Unidad de load/store
├─neorv32_cpu_pmp.vhd           - Unidad de protección de memoria física (extensión Smpmp)
├─neorv32_cpu_regfile.vhd       - Banco de registros de datos
├─neorv32_crc.vhd               - Unidad de verificación de redundancia cíclica (CRC)
├─neorv32_debug_auth.vhd        - Depurador en chip: módulo de autenticación
├─neorv32_debug_dm.vhd          - Depurador en chip: módulo de depuración
├─neorv32_debug_dtm.vhd         - Depurador en chip: módulo de transferencia de depuración
├─neorv32_dma.vhd               - Controlador de acceso directo a memoria (DMA)
├─neorv32_dmem.vhd              - Memoria de datos interna genérica del procesador
├─neorv32_fifo.vhd              - Componente FIFO genérico
├─neorv32_gpio.vhd              - Unidad de puerto de entrada/salida de propósito general (GPIO)
├─neorv32_gptmr.vhd             - Temporizador de propósito general de 32 bits
├─neorv32_imem.vhd              - Memoria de instrucciones interna genérica del procesador
├─neorv32_neoled.vhd            - Interfaz de LED inteligente compatible con NeoPixel (TM)
├─neorv32_onewire.vhd           - Controlador de interfaz serial One-Wire
├─neorv32_package.vhd           - Archivo principal de paquete VHDL
├─neorv32_pwm.vhd               - Controlador de modulación por ancho de pulso
├─neorv32_sdi.vhd               - Controlador de interfaz de datos serial (dispositivo SPI)
├─neorv32_slink.vhd             - Interfaz de enlace de flujo
├─neorv32_spi.vhd               - Controlador de SPI (host SPI)
├─neorv32_sys.vhd               - Módulos de infraestructura del sistema
├─neorv32_sysinfo.vhd           - Memoria de información de configuración del sistema
├─neorv32_top.vhd               - ENTIDAD PRINCIPAL DEL PROCESADOR/SOC NEORV32
├─neorv32_trng.vhd              - Generador de números verdaderamente aleatorios
├─neorv32_twd.vhd               - Controlador de TWD (dispositivo)
├─neorv32_twi.vhd               - Controlador de TWI (interfaz)
├─neorv32_uart.vhd              - Receptor/transmisor asíncrono universal (UART)
├─neorv32_wdt.vhd               - Temporizador watchdog
└─neorv32_xbus.vhd              - Puertas de enlace de interfaz de bus externo (Wishbone)
```

Figura 5.3: Esquema de los archivos RTL del NEORV32, archivos más relevantes en azul.

- **sim:** Archivos de VHDL que se emplean durante la simulación para optimizarla, junto al *script* que llama a GHDL.

- **sw:** Todo lo relacionado con la ejecución de código, el *bootloader*, *scripts* de enlazado, códigos de ejemplo, la generación del código binario que se ha de cargar en el NEORV32, bibliotecas, configuración para OpenOCD y el archivo *System View Description* (SVD).

### 5.2.3. Aceleradores en NEORV32

El SoC incorpora interfaces independientes para la búsqueda de instrucciones y el acceso de datos, es capaz de diseccionar la totalidad del espacio de 32 bits desde cada interfaz y ambas se pueden configurar de modo que empleen un caché, *dCache* e *iCache*, respectivamente.

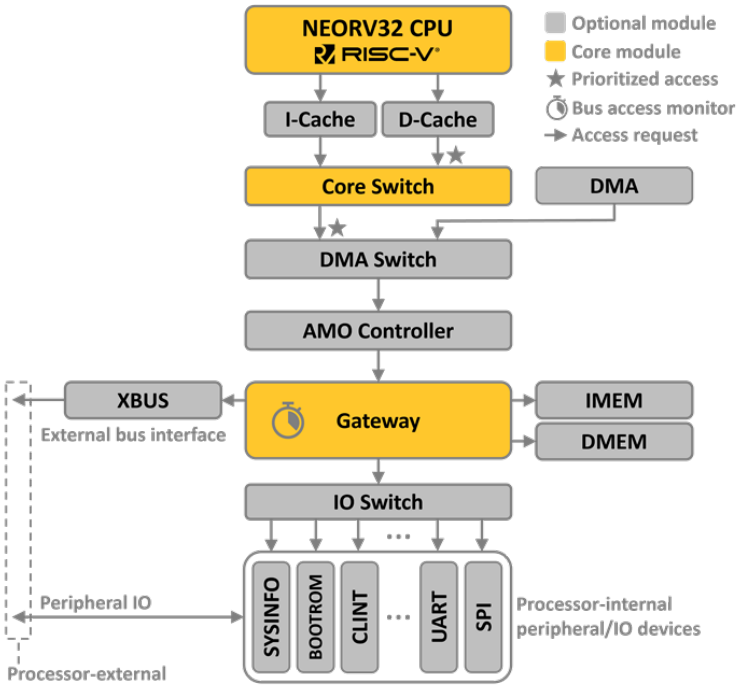



Figura 5.4: Sistema de buses en NEORV32 [37, Datasheet - 2.6.1 Bus System].

Las dos interfaces (`I` y `D`), están multiplexadas, de modo que el flujo de información con la CPU es mediante un solo bus, este bus a su vez, está multiplexado con el controlador de DMA para permitir que este pueda acceder al conjunto del espacio de memoria, dando lugar al bus de SoC. Los accesos que originen en el bus de SoC son redirigidos por el *Bus Gateway* que redirige los accesos de acuerdo con las regiones de memoria presentes en la figura 5.5. Siguiendo la jerarquía descrita, se organiza la velocidad de acceso, siendo los accesos a los periféricos lentos y los buses de instrucciones y datos de la CPU rápidos.

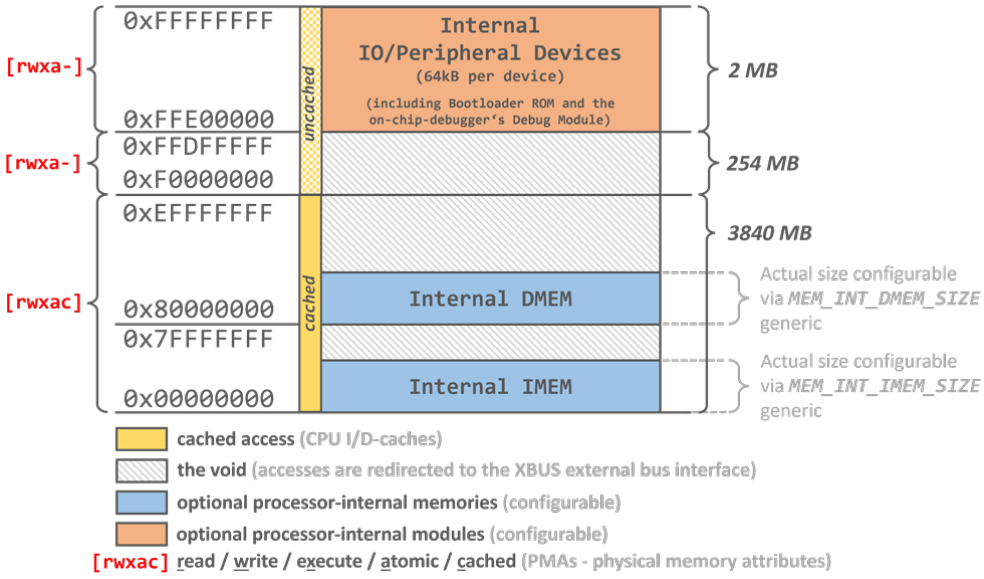



Figura 5.5: Espacio de memoria del NEORV32.

Es posible acelerar diferentes flujos de trabajo utilizando el NEORV32. Las dos principales vías son: la implementación de instrucciones mediante el CFU del NEORV32 y la conexión de un acelerador externo mediante XBUS.

#### XBUS

La interfaz de bus externo XBUS expone una interfaz de bus en-chip de compatible con Wishbone. Es posible utilizarlo para conectar desde aceleradores externos hasta memorias, pasando por todo tipo de periféricos. Aunque no está asociado a ninguna dirección de memoria particular, todos aquellos accesos a regiones que no tuvieran asociado otra región de interna son redirigidas a esta. De forma general, tenemos las regiones marcadas con líneas diagonales grises, como se puede ver en 5.5, disponibles para este tipo de accesos.

#### NEORV32 CFU

El NEORV32 [37] presenta una unidad *hardware* llamada CFU (*Custom Functions Unit*) responsable de implementar la extensión del conjunto de instrucciones, `Zxcfu`, que permite, a su vez, implementar instrucciones personalizadas desde *software*.

### 5.2.4. Funcionamiento básico del NEORV32

A continuación se demuestra el funcionamiento básico del NEORV32 mediante la ejecución de un programa tipo *Hello World!*, la configuración previa a la implementación sobre la FPGA Nexys4 DDR [38] se ha realizado, con `Vivado 2024.1` [39].

El NEORV32 se ha tratado de desarrollar de forma agnóstica, sin uso de primitivas propias de ningún flujo de

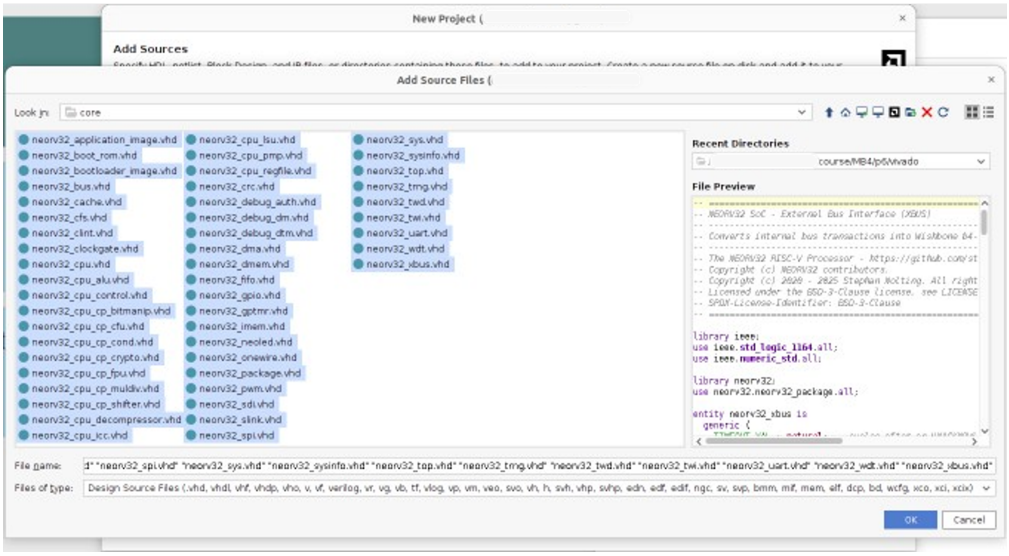



Figura 5.6: Importación del los archivos RTL al proyecto de Vivado.

trabajo propietario. En este caso, se ha utilizado Vivado 2024.1 debido al grado de familiarización previo con el mismo. Se ha creado un nuevo proyecto con Vivado al que se han añadido todos los archivos RTL que describen el NEORV32 como se muestra en la figura 5.6.

Una vez se han añadido los archivos que componen el NEORV32 y completado el flujo de creación del proyecto, se muestra el resumen de la configuración elegida, en esta se muestra la placa de prototipado FPGA elegida, en este caso la Nexys4 DDR. La Nexys4 DDR es una placa de prototipado básica con recursos más que suficientes, como se puede ver en 5.8, para implementar el NEORV32.

Para implementar el NEORV32 con Vivado en la FPGA es necesario llegar al paso VII del flujo de trabajo. En el primer paso se añaden los archivos RTL que sean necesarios, como ya se ha hecho. Los pasos II y III permiten utilizar el editor de bloques y ejecutar simulaciones, respectivamente, son opcionales y en este ejemplo no se utilizarán. El paso IV permite validar el código RTL y presenta un esquema generado a partir de los archivos RTL que se han provisto con anterioridad.

Los pasos que mayor relevancia tienen son los tres últimos, empezando por el paso V, síntesis, con el que se convierte la descripción RTL a una lista de conexiones que utiliza las primitivas propias de Xilinx que se encuentren disponibles en FPGA la que se ha configurado en la selección durante la creación del proyecto.

La implementación asocia la lista de conexiones creada durante la síntesis a los recursos físicos dentro de la FPGA, es posible optimizar la implementación dependiendo de las preferencias en cada caso, en potencia, área o rendimientos de diferentes tipos.

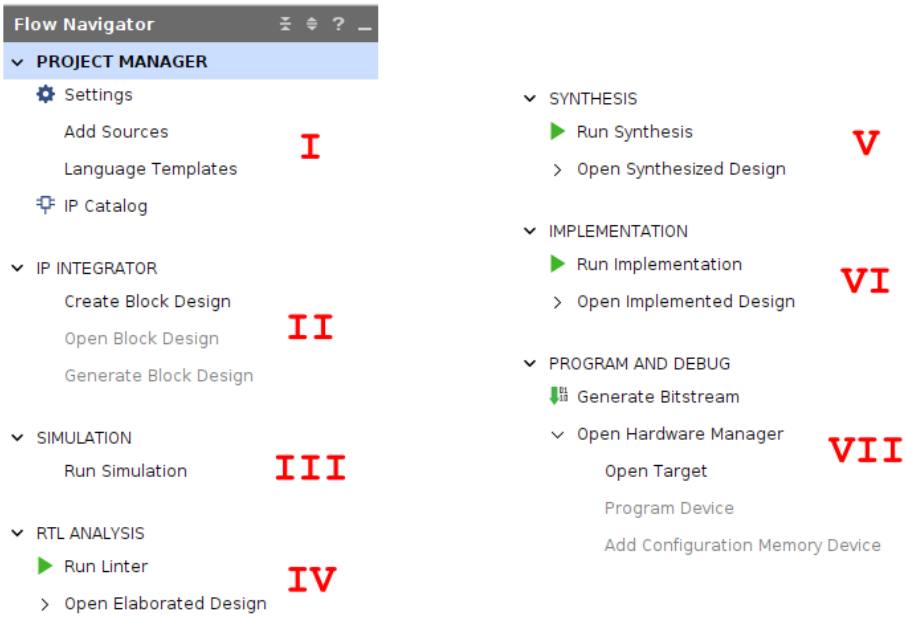



Figura 5.7: Siete pasos del flujo de trabajo con Vivado.

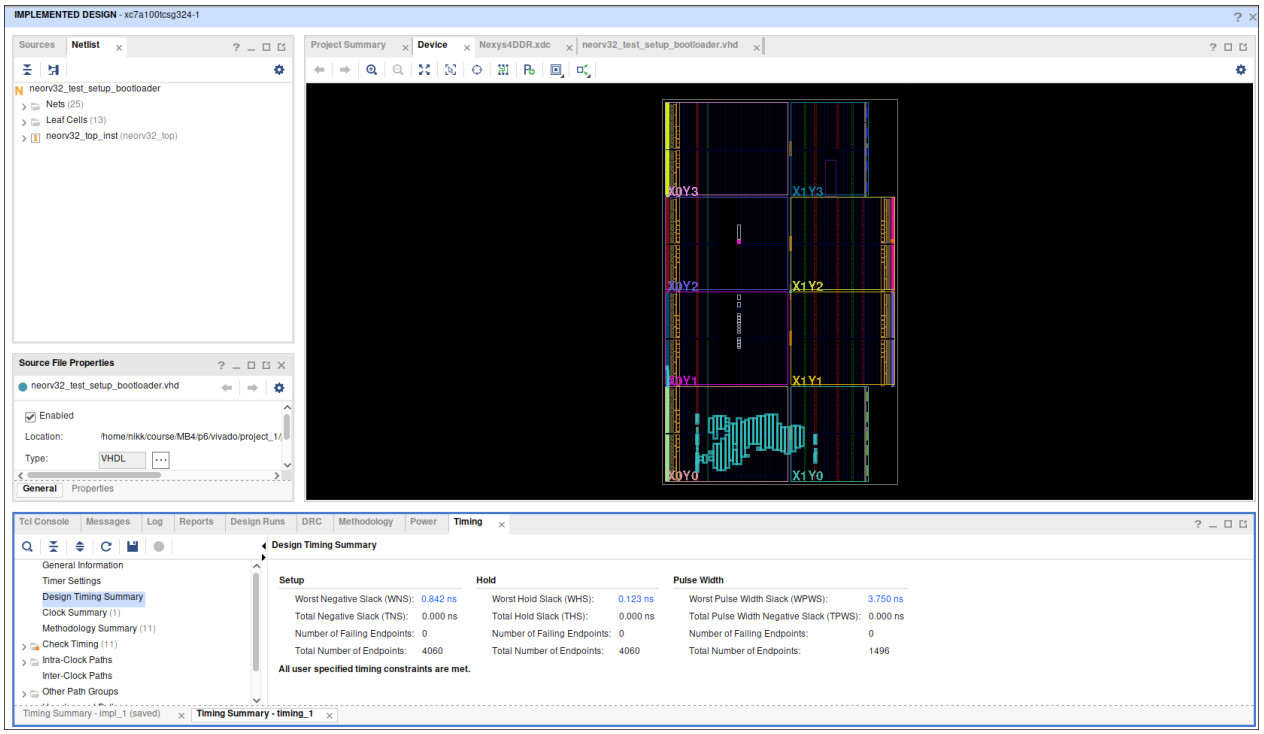
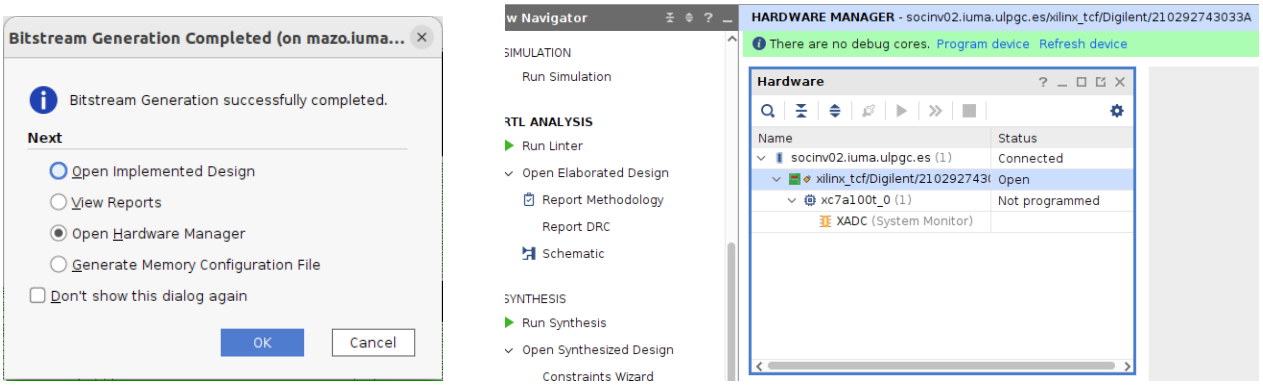



Figura 5.8: Resultado de la implementación del NEORV32 sobre la placa de prototipado Nexys4 DDR, en cian, la pequeña región de recursos utilizados.

El último paso es la generación del *bitstream* donde se genera la imagen que, una vez volcada en la FPGA configurará los recursos físicos de la misma con las estructuras descritas en los archivos HDL que componen el NEORV32.




(a) Bitstream completado.

(b) *Hardware manager* en Vivado.

Figura 5.9: Generación del *bitstream* completada e interfaz de carga (*hardware manager*).

Una vez se ha generado y volcado el *bitstream* sobre la FPGA, según el esquema de conexión en la figura 5.10, por medio del puerto USB principal, es necesario utilizar un programa de comunicación mediante puerto serie que permita leer y escribir al dispositivo serial que Linux crea a partir de la conexión física con el chip FT2232 integrado en la placa de prototipado. En este caso se utilizará CuteCom, pero es posible utilizar cualquier otro programa similar (minicom, picocom, screen, socat, etc.).


Figura 5.10: Esquema de conexión del NEORV32 con la computadora.

En la figura 5.11 se muestra la interfaz gráfica de CuteCom con la que es posible ver la salida de la UART principal del NEORV32, *UART0* y enviar los diferentes comandos necesarios para ejecutar las aplicaciones sobre el NEORV32.

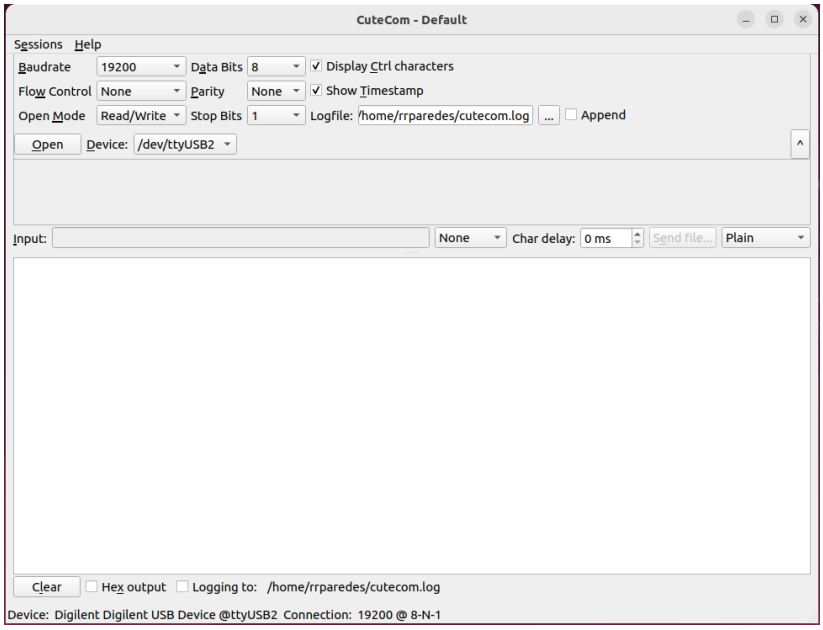



Figura 5.11: Interfaz gráfica del CuteCom.

Sin embargo, antes de cargar ningún programa es necesario escribirlo, compilarlo y generar la imagen que programará la memoria del NEORV32, para ello se han utilizado el *makefile* accesible desde el directorio de ejemplo. Una vez se ejecuta el comando `make all` como se puede ver en la figura 5.12 se generan todos los archivos que pueden ser utilidad a la hora de trabajar con el NEORV32.

```
user@ubuntuvbox:~/course/MB4/p6/neorv32/sw/example/hello_world$ make all
Memory utilization:
text    data     bss     dec     hex filename
5692       0     116    5808    16b0 main.elf
Generating neorv32_exe.bin
Executable size in bytes:
5704
Generating neorv32_raw_exe.hex
Generating neorv32_raw_exe.bin
Generating neorv32_raw_exe.coe
Generating neorv32_raw_exe.mem
Generating neorv32_raw_exe.mif
Generating neorv32_application_image.vhd
Installing application image to ../../../rtl/core/neorv32_application_image.vhd
```

Figura 5.12: Ejecución del comando `make all` dentro del directorio correspondiente al ejemplo `hello_world`.

Los archivos generados mediante el `make all` son:

- `*.hex`: fichero con valores hexadecimales codificados con caracteres ASCII.

- `*.bin`: fichero binario de propósito general.
- `*.coe`: fichero COE para la inicialización de la memoria de la FPGA.
- `*.mem`: fichero MEM para la inicialización de la memoria de la FPGA.
- `*.mif`: fichero MIF para la inicialización de la memoria de la FPGA.
- `*_application_image.vhd`: fichero VHDL que permite inicializar la memoria IMEM interna del procesador.

Después de haber establecido la conexión física entre el computador y el NEORV32, se dispone de un canal de comunicación serial que comienza en las conexiones RX y TX de la UART primaria del NEORV32 y termina en la interfaz gráfica de CuteCom, pasando el cable USB y el chip FT2232, mencionado con anterioridad.

Inmediatamente después de haber volcado correctamente el *bitstream* en la FPGA, se transmite por la UART el mensaje del *bootloader*, figura 5.13.

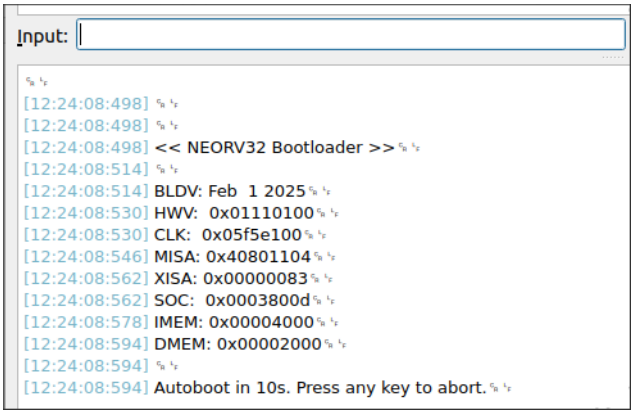



Figura 5.13: Mensaje de *bootloader*.

El *bootloader* presenta las direcciones de memoria donde se ubican los diferentes módulos clave del NEORV32 y espera 10 segundos para leer el programa que pudiera haber en la memoria *flash*, si se interrumpe este proceso, se presenta un menú con las opciones que se pueden ver en la figura 5.14.

- `h: Help`: Muestra la ayuda de la figura 5.14.
- `r: Restart`: Reinicia el procesador.
- `u: Upload`: Inicia el proceso de carga de un programa y queda a la espera de que mande un archivo, por ejemplo con *Send file* en CuteCom. (1º)
- `s: Store to flash`: Vuelca el programa en la memoria *flash* que previamente se ha cargado con la operación `u`.
- `l: Load from flash`: Carga el programa que previamente se ha volcado en la memoria *flash*.
- `e: Execute`: Ejecuta el programa cargado. (2º)

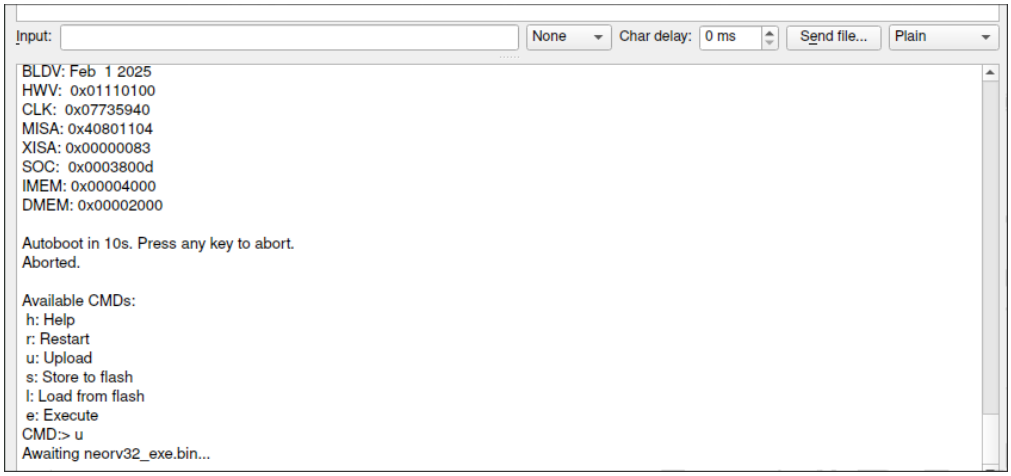



Figura 5.14: Menú de comandos que presenta el *bootloader* después de los 10 segundos.

Durante la fase de prototipado se han ejecutado los programas exclusivamente mediante dos pasos: (1º) `Upload` y (2º) `Execute`, esto aplicado (5.15) al `hello_world` da como resultado el texto que se puede apreciar en la figura 5.16.


Figura 5.15: Carga del programa representado como imagen binaria, `neorv32_exe.bin`.

```
CMD:> e
Booting from 0x00000000


                                                                                ##       ##   ##   ##
 ##     ##   #########   ########    ########   ##      ##   ########    ########    ##      ################
####    ##   ##          ##      ##  ##      ##  ##      ##  ##      ##  ##      ##  ##      ##  ####          ####
## ##   ##   ##          ##      ##  ##      ##  ##      ##          ##          ##  ##      ##    ##   ######   ##
##  ##  ##   #########   ##      ##  #########   ##      ##      #####       ##      ##      ##  ####   ######   ####
##   ## ##   ##          ##      ##  ##      ##    ##    ##          ##    ##        ##      ##    ##   ######   ##
##    ####   ##          ##      ##  ##      ##     ##  ##   ##     ##    ##        ##      ##  ####           ####
##     ##    #########   ########   ##      ##      ##      ########   ##########  ##      ################
                                                                                ##       ##   ##   ##

Hello world! :)
```

Figura 5.16: Salida de la UART al ejecutar el programa de ejemplo `hello_world`.

### 5.2.5. Aceleración de aplicaciones criptográficas mediante la CFU de NEORV32

NEORV32 permite la extensión de su funcionalidad a través de la Custom Function Unit (CFU). En este ejemplo se ha utilizado para la implementación del algoritmo de cifrado ligero XTEA [40] [41]. Tiny Encryption

Algorithm (TEA) utiliza operaciones muy simples para contribuir a la confusión, como XOR, desplazamiento y operaciones de adición de módulo 32, trabajando en entradas de datos de 32 bits. Dado que tiene un tamaño de código reducido y un bajo requerimiento de memoria, es ideal para realizar el cifrado en sistemas IoT ligeros. XTEA es una extensión realizada para aumentar su seguridad mediante el mezclado de los bytes de la clave. Sin embargo, se afirma que serían necesarios 32 rondas para operaciones de alta seguridad. La clave criptográfica utilizada es de 128 bits dividida en 4 partes de 32 bits cada uno.

La Figura 5.17 muestra el pseudocódigo para el cifrado y el descifrado XTEA. En NEORV32 se ha implementado XTEA como una funcionalidad Custom a través de la CFU indicada. Por tanto, es necesario activar en el *core* NEORV32 las extensiones Zxcfu.

La implementación realizada se encuentra en el fichero `rtl/core/neorv32_cpu_cp_cfu.vhd`. En la carpeta `sw/example/demo_cfu` se dispone de un ejemplo de utilización. Para ejecutar dicho ejemplo, es necesario activar también las extensiones Zicntr que permite acceder a los contadores internos definidos en la arquitectura. Dichos contadores permiten medir el número de ciclos utilizados en la ejecución de un programa y, por tanto, su tiempo de ejecución.

En este caso se ha recurrido a configurar el NEORV32 utilizado mediante la personalización de propiedades del bloque *Intellectual property* (IP), aunque la misma configuración se puede realizar mediante modificaciones la archivo *top* que actúa como *wrapper* del resto del proyecto. Tras realizar las modificaciones el diagrama de bloques resultante queda como se puede ver en la figura 5.18.

***Algorithm 1 Pseudo Code XTEA Encryption***

```
sum = 0
for i = 0 to N do
        v0+ = ((v1 << 4) xor (v1 >> 5) + v1) xor (sum +
        key[sum &3])
        sum+ = delta
        v1+ = ((v0 << 4) xor (v0 >> 5) + v0) (sum +
        key[sum>>11 &3])
end for
```

***Algorithm 2 Pseudo Code XTEA Decryption***

```
sum = 0
for i = 0 to N do
        v1- = ((v0 << 4) xor (v1 >> 5) + v0) xor (sum +
        key[sum>>11 &3])
        sum+ = delta
        v0- = ((v1 << 4) xor (v1 >> 5) + v1) (sum +
        Key[sum &3])
end for
```

Figura 5.17: Cifrado y descifrado en XTEA.

En la figura 5.19 se puede apreciar la diferencia entre la ejecución del programa de ejemplo con y sin las extensiones para acelerar la ejecución del algoritmo de encriptación ligero XTEA. Al final de la figura 5.19 se ven los resultados presentados mediante la UART: la encriptación realizada únicamente por *software* resulta en 68517 ciclos, esta misma operación realizada mediante las instrucciones específicas para el cálculo del algoritmo, es realizada en 12643 ciclos. La diferencia es aún más crítica en el caso de la desencriptación, dando lugar a una

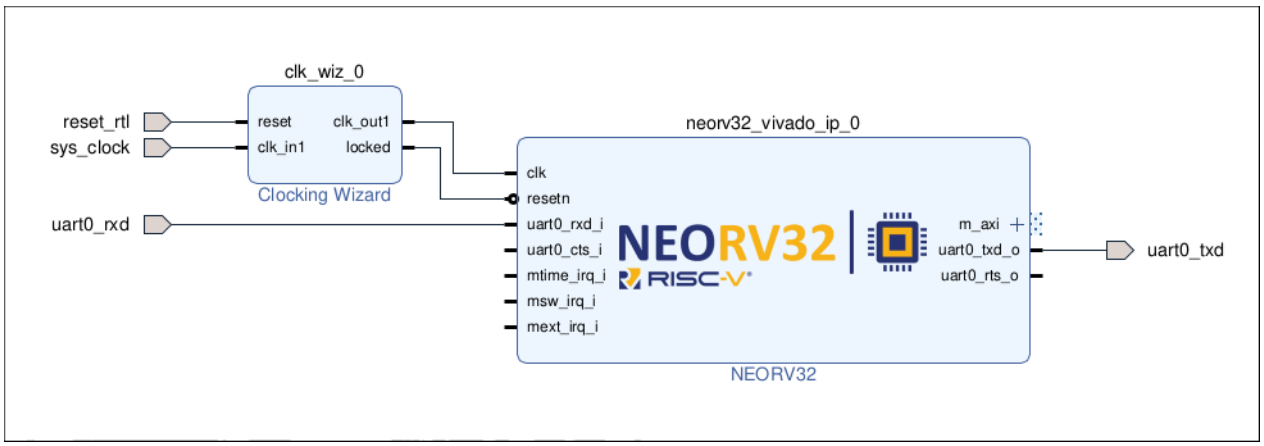



Figura 5.18: Diagrama de bloques que expone las señales de reloj, reset y transmisión/recepción de la UART del NEORV32.

diferencia de 79157 ciclos. Al concluir la ejecución de determina que la implementación del algoritmo que hace uso de las instrucciones dedicadas requiere de 6 veces menos ciclos de reloj.

```
x: Boot from flash (XIP)
e: Execute
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> e
Booting from 0x00000000...

<<< NEORV32 Custom Functions Unit (CFU) - Custom Instructions Example >>>

[NOTE] This program assumes the default CFU hardware in
'rtl/core/neorv32_cpu_cp_cfu.vhd' that implements
the Extended Tiny Encryption Algorithm (XTEA).

XTEA key: 0x207230ba1ffba710c45271efdd01768a

XTEA SW encryption (40 rounds, 64 words)...
XTEA HW encryption (40 rounds, 64 words)...
Comparing results... OK

XTEA SW decryption (40 rounds, 64 words)...
XTEA HW decryption (40 rounds, 64 words)...
Comparing results... OK

Execution timing:
ENC SW = 68517 cycles
ENC HW = 12643 cycles
DEC SW = 91811 cycles
DEC HW = 12654 cycles
Average speedup: ~6x

CFU demo program completed.
```

Figura 5.19: Resultado de la ejecución de las dos implementaciones del algoritmo de cifrado ligero XTEA.

## 5.3. VexRiscv

VexRiscv [36] es una implementación de RISC-V de 32 bits escrita en SpinalHDL [42]. Soporta las extensiones "M", "A", "F", "D", "C" y cuenta con un pipeline configurable de 2 a más de 5 etapas, con capacidad de alcanzar de 1.44 a 1.57 DMIPS/MHz. Entre sus opciones de configuración tiene buses como AXI4, Avalon y Wishbone, un FPU de 32 o 64 bits, cachés de instrucción y datos, *Memory Management Unit* (MMU), extensión que permite la depuración mediante GDB, OpenOCD y JTAG. Está optimizado para FPGAs y es posible ejecutar aplicaciones *barebones* o con sobre un sistema operativo como Linux, Zephyr o FreeRTOS.

### 5.3.1. VexRiscv con Ztachip

En la exposición realizada sobre el prototipado con Ztachip del capítulo 4, se mencionó cómo dicho acelerador emplea el VexRiscv como *host* el desarrollado de dicho proyecto ha incluido una copia de dicha implementación RISC-V suficientemente reducida para cumplir con la gran restricción de área con la que se cuenta en la FPGA de destino. Este hecho, junto con la numerosa cantidad de configuraciones estándar presentadas en el repositorio del proyecto, figura 5.21, pone de manifiesto la flexibilidad con la que se ha planteado y que remarca una vez más esta tendencia dentro del panorama de desarrollo de RISC-V. Como resultado de estas decisiones de diseño y como refleja la figura 5.20 resulta una implementación en la que el *host* RISC-V ocupa unos recursos muy limitados, en comparación con el acelerador en su conjunto.

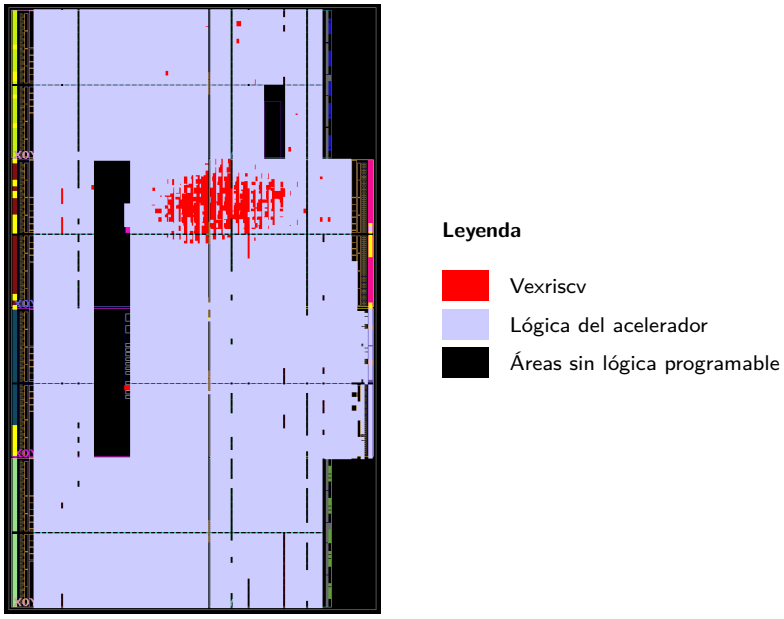



Figura 5.20: Implementación del Ztachip con el área que ocupa el *host* del acelerador resaltado.

```
VexRiscv small (RV32I, 0.52 DMIPS/MHz, no datapath bypass, no interrupt)

VexRiscv small (RV32I, 0.52 DMIPS/MHz, no datapath bypass)

VexRiscv small and productive (RV32I, 0.82 DMIPS/MHz)

VexRiscv small and productive with I$ (RV32I, 0.70 DMIPS/MHz, 4KB-I$)

VexRiscv full no cache (RV32IM, 1.21 DMIPS/MHz 2.30 Coremark/MHz, single cycle barrel shifter, debug
module, catch exceptions, static branch)

VexRiscv full (RV32IM, 1.21 DMIPS/MHz 2.30 Coremark/MHz with cache trashing, 4KB-I$,4KB-D$, single
cycle barrel shifter, debug module, catch exceptions, static branch)

VexRiscv full max perf (HZ*IPC) -> (RV32IM, 1.38 DMIPS/MHz 2.57 Coremark/MHz, 8KB-I$,8KB-D$, single
cycle barrel shifter, debug module, catch exceptions, dynamic branch prediction in the fetch stage,
branch and shift operations done in the Execute stage)

VexRiscv full with MMU (RV32IM, 1.24 DMIPS/MHz 2.35 Coremark/MHz, with cache trashing, 4KB-I$, 4KB-D$,
single cycle barrel shifter, debug module, catch exceptions, dynamic branch, MMU)

VexRiscv linux balanced (RV32IMA, 1.21 DMIPS/MHz 2.27 Coremark/MHz, with cache trashing, 4KB-I$, 4KB-
D$, single cycle barrel shifter, catch exceptions, static branch, MMU, Supervisor, Compatible with
mainstream linux)
```

Figura 5.21: Posibles configuraciones estándares de la implementación RISC-V, Vexriscv.

## 5.4. CVA6

Es un procesador RISC-V altamente configurable, forma parte de la familia de procesadores *CORE-V* del OpenHW Group. Tiene un *pipeline* compuesto por 6 etapas, que despacha una instrucción por ciclo, la ejecución de las instrucciones es en orden e implementa las extensiones I, M, A y C. Es capaz de ejecutar Linux y está diseñado para ofrecer un alto rendimiento con un *pipelining* optimizado. Implementa los tres niveles de privilegio (M, S, U) de RISC-V principales y es compatible con la especificación de depuración externa, RISC-V *Debug Spec 0.13*.

El CVA6, inicialmente llamado, "Ariane", nace dentro del proyecto PULP [43] mediante una colaboración entre el ETH Zürich y la Universidad de Bolonia. Es concebido como una plataforma RISC-V abierta apta tanto para entornos académicos como industriales, con soporte completo a nivel de sistema y que se ha llevado a producción como ASIC experimental. Tras un periodo inicial fue migrado al OpenHW Group para convertirse en un proyecto comunitario de código abierto, consolidándose como uno de los núcleos RISC-V más activos en la actualidad.

### 5.4.1. Arquitectura

Como ya se ha mencionado, el CVA6 implementa un pipeline en seis etapas: (1) Generación del PC y búsqueda de instrucciones, (2) decodificación de las instrucciones, (3) etapa de emisión, (4) etapa de ejecución y (5) fase de *commit*, según se puede ver en la figura 5.22.

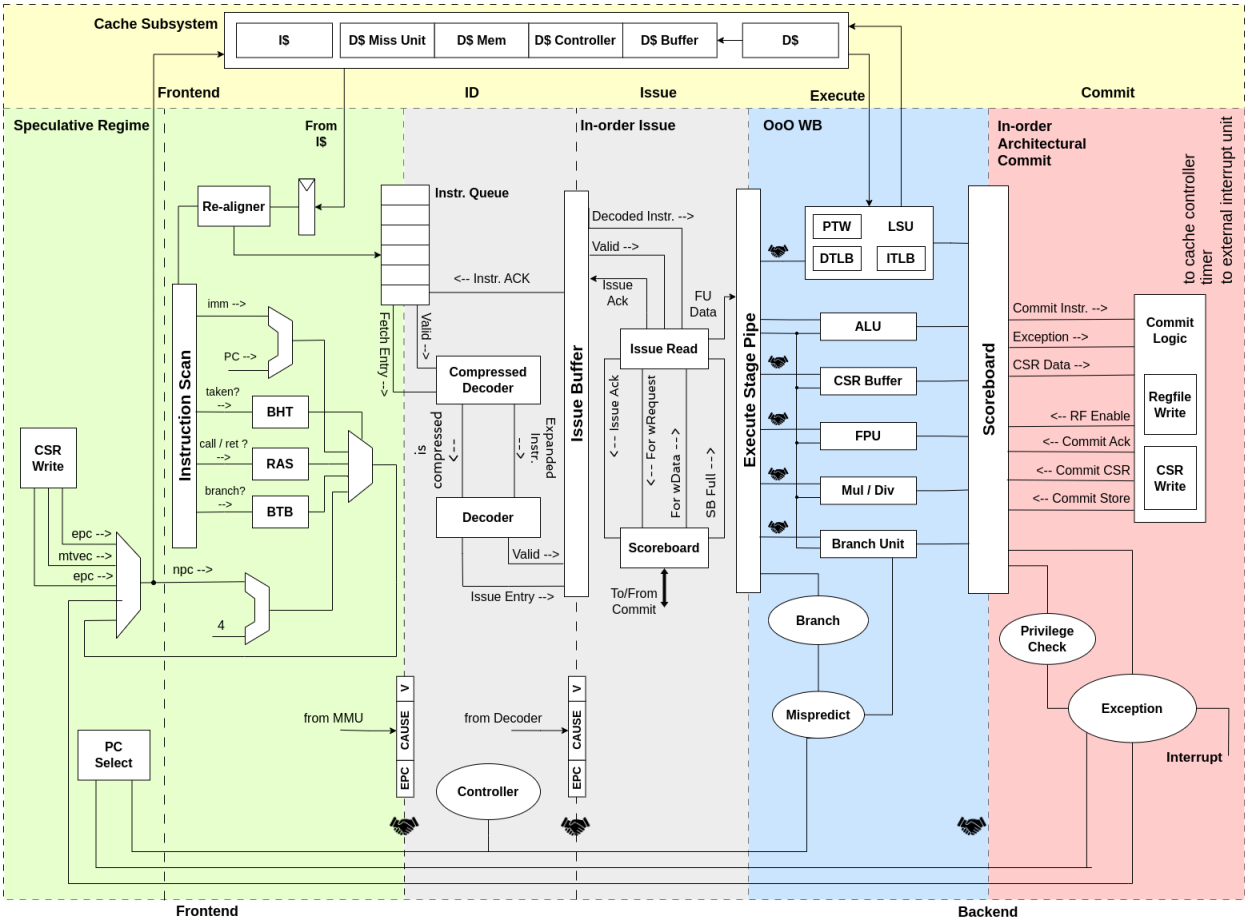



Figura 5.22: Arquitectura del CVA6.

Uno de los principios base del CVA6 es la parametrización, es posible instanciarlo como procesador tanto de 32 como 64 bits, habilitar las extensiones que se consideren oportunas, deshabilitar el MMU, establecer tamaños de los registros y comportamiento de los saltos, entre muchas otras configuraciones[1].

Las principales características del CVA6 son:

- Predictor de saltos dinámicos avanzados.
- Soporte de memoria virtual mediante *Translation Lookaside Buffer*s (TLBs) y un mecanismo de recorrido de tabla de páginas (*Page Table Walker* (PTW)) integrado en su MMU.
- Compatibilidad con la especificación para depuración externa de RISC-V 0.13.
- Pipeline de seis etapas con *scoreboard*.

### 5.4.2. Implementación del CVA6 en Genesys 2

La arquitectura CVA6 está integrada en un SoC que se puede implementar tanto en dispositivos FPGA y en tecnología ASIC. Se ha realizado su implementación FPGA para la placa de prototipado FPGA Genesys 2 [44]. Los resultados de las prestaciones obtenidos de su implementación se muestran en la figura 5.23. El *layout* de la implementación se muestra en la figura 5.24, en la que se ha diferenciado la implementación del CVA6 y del resto del SoC (MMU, buses del sistema, etc.).

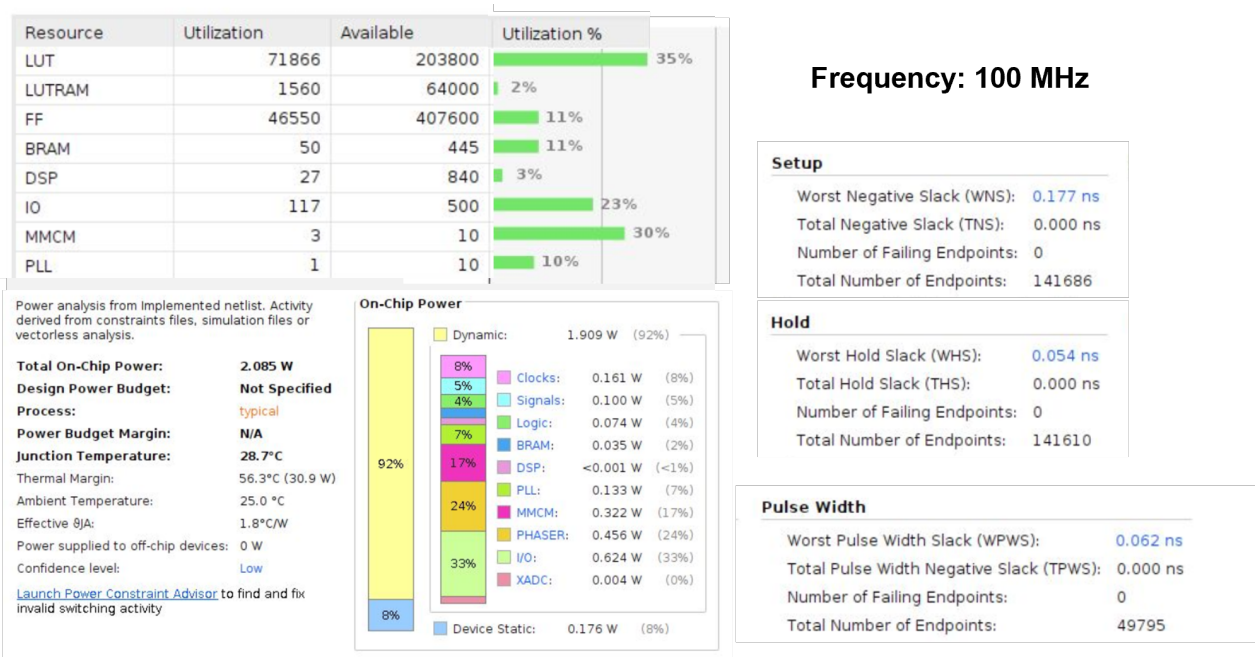



Figura 5.23: Prestaciones de la implementación de CVA6 en Genesys 2.

Para utilizar Linux es necesario contar con la imagen Linux, Berkeley boot loader (BBL), rootfs y OpenSBI. RISC-V Open Source Supervisor Binary Interface (OpenSBI) [45] es la implementación de RISC-V SBI. SBI

[1] Parameters_Configuration.html

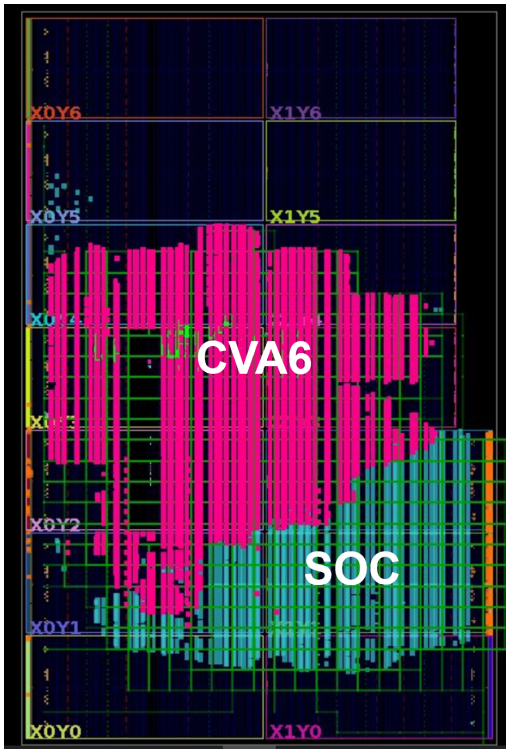



Figura 5.24: Layout de la implementación de CVA6 en Genesys 2.

permite que el *software* que funcione en modo supervisor (S-mode o VS-mode) sea portable en las distintas implementaciones de RISC-V definiendo una abstracción para la funcionalidad específica de la plataforma. Su diseño sigue la filosofía modular de RISC-V: funcionalidad reducida, acompañada de extensiones modulares opcionales. En la figura 5.25 se muestra el esquema de abstracción: U-Mode (*user mode*), S-mode (*supervisor mode*), M-mode (*machine mode*).

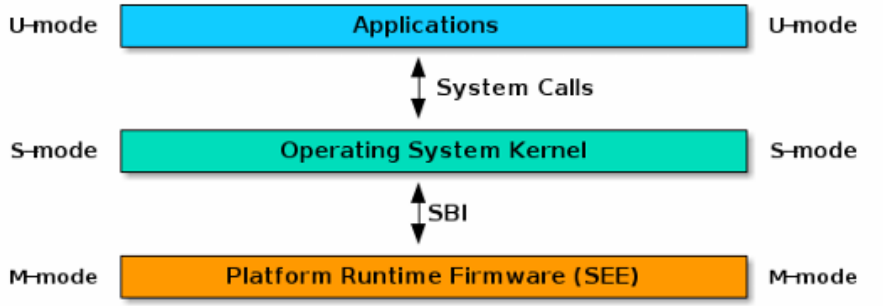



Figura 5.25: Estructura de RISC-V SBI.

Una vez realizado el proceso de *boot*, se puede acceder a ejecutar los comandos y aplicaciones incluidos en el rootfs configurado. El *kernel* de Linux configura los *drivers* necesarios para acceder a los recursos del

sistema, incluyendo el almacenamiento en la tarjeta SD y en otros dispositivos presentes. También dispone de los componentes del *stack* TCP/IP para comunicaciones de alta velocidad (Gigabit Ethernet). En la figura 5.28 se muestra la configuración de DHCP para facilitar el acceso a la red.

En las figuras 5.26 y 5.27 se aprecia el proceso de arranque del procesador CVA6 en la placa mencionada.

```
Presione CTRL-A Z para obtener ayuda sobre teclas especiales

Hello World!
init SPI
status: 0x0000000000000025
status: 0x0000000000000025
SPI initialized!
Initializing SD...
SD command cmd0               response : 01
SD command cmd55              response : 01
SD command cmd41              response : 01
SD command cmd55              response : 01
SD command cmd41              response : 01
SD command cmd55              response : 01
SD command cmd41              response : 01
SD command cmd55              response : 01
SD command cmd41              response : 01
SD command cmd55              response : 01
SD command cmd41              response : 01
SD command cmd55              response : 01
SD command cmd41              response : 01
SD command cmd55              response : 01
SD command cmd41              response : 01
SD command cmd55              response : 01
SD command cmd41              response : 01
SD command cmd55              response : 01
SD command cmd41              response : 01
SD command cmd55              response : 01
SD command cmd41              response : 00
sd initialized!
gpt partition table header:
    signature:                5452415020494645
    revision:                 00010000
    size:                     0000005C
    crc_header:               BA533437
    reserved:                 00000000
    current lba:              0000000000000001
    backup lba:               0000000076E4FFFF
    partition entries lba:                        0000000000000002
    number partition entries:                     00000080
    size partition entries:                       00000080
gpt partition entry 00
    partition type guid:      D5F7127456A1134B81DC867174929325
    partition guid:           C2EC084A17531E47B09C59744F1A10F7
    first lba:                0000000000000800
    last lba:                 0000000000001AFD
    attributes:               0000000000000000
    name:                     0000000000000000000000000000000000000000000000000000
gpt partition entry 01
    partition type guid:      AF3DC60F838472478E793D69D8477DE4
```

Figura 5.26: Arranque de CVA6 en la placa Genesys 2 (1).

```
last lba:          000000000001AFD
attributes:        0000000000000000
name:              0000000000000000000000000000000000000000000000000000

gpt partition entry 01
    partition type guid: AF3DC60F838472478E793D69D8477DE4
    partition guid:      CA17F0C0E40E3B4FA8E64F9592469F0C
    first lba:           0000000000020000
    last lba:            00000000076E4FDE
    attributes:          0000000000000000
    name:                0000000000000000000000000000000000000000000000000000

gpt partition entry 02
    partition type guid: 00000000000000000000000000000000
    partition guid:      00000000000000000000000000000000
    first lba:           0000000000000000
    last lba:            0000000000000000
    attributes:          0000000000000000
    name:                0000000000000000000000000000000000000000000000000000

gpt partition entry 03
    partition type guid: 00000000000000000000000000000000
    partition guid:      00000000000000000000000000000000
    first lba:           0000000000000000
    last lba:            0000000000000000
    attributes:          0000000000000000
    name:                0000000000000000000000000000000000000000000000000000

copying boot image .... done!

OpenSBI v0.9
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name             : ARIANE RISC-V
Platform Features         : medeleg
Platform HART Count       : 1
Platform IPI Device       : aclint-mswi
Platform Timer Device     : aclint-mtimer @ 1000000Hz
Platform Console Device   : uart8250
Platform HSM Device       : ---
Platform Reboot Device    : ---
Platform Shutdown Device  : ---
Firmware Base             : 0x80000000
Firmware Size             : 220 KB
Runtime SBI Version       : 0.3

Domain0 Name              : root
Domain0 Boot HART         : 0
Domain0 HARTs             : 0*
Domain0 Region00          : 0x0000000020080000-0x00000000200bffff (I)
```

Figura 5.27: Arranque de CVA6 en la placa Genesys 2 (2).

```
# ifconfig -a
eth0      Link encap:Ethernet HWaddr 00:18:3E:02:E3:7F
          BROADCAST MTU:1500 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          LOOPBACK  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

sit0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-A8-42-00-00-00-00-00-00-00-00
          NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

# udhcpc
udhcpc: started, v1.33.1
[ 185.795852] Open device, request interrupt 11
[ 185.795852] Open device, request interrupt 11
udhcpc: sending discover
udhcpc: sending discover
[ 190.359881] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ 190.359881] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
udhcpc: sending select for 10.13.24.43
udhcpc: lease of 10.13.24.43 obtained, lease time 10800
deleting routers
adding dns 193.145.138.100
adding dns 193.145.138.200
# ifconfig -a
eth0      Link encap:Ethernet HWaddr 00:18:3E:02:E3:7F
          inet addr:10.13.24.43  Bcast:10.13.24.255  Mask:255.255.255.0
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:227 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:88539 (86.4 KiB)  TX bytes:1434 (1.4 KiB)
```

Figura 5.28: Configuración de DHCP para facilitar el acceso a la red Ethernet de CVA6.

## 5.5. Elección de la implementación

Una vez se ha estudiado las posibles implementaciones de RISC-V y aceleradores y prototipado con las diferentes implementaciones, se procede a definir la plataforma final a utilizar.

Después de realizar el estudio y prototipado de las plataformas expuestas en los capítulos 4 y 5, se ha abordado la experimentación con el sistema Ara2, capaz de implementar extensiones vectoriales siguiendo la especificación RVV 1.0.

Está integrado en la plataforma del procesador CVA6 que, por sí solo, presenta altas capacidades en cuanto a rendimiento, pudiendo alcanzar 4.35 CoreMark/MHz con las configuraciones *superescalar* y *speculative scoreboard* activas [46], además de tener un mayor rendimiento base que el resto de procesadores, incluyendo 0.9523

CoreMark/MHz con la configuración de alto rendimiento del NEORV32[2] y 2.27 Coremark/MHz para VexRiscv[3]. Además de las métricas anteriores, los proyectos que rodean a Ara de parte de PULP Platform han presentado hasta la actualidad un continuo desarrollo, como se puede ver mediante el volumen de contribuciones y frecuencia de las mismas en los repositorios del Ara, Cheshire y CVA6 cuentan el compromiso total de la organización, se espera que continúen expandiendo en la misma línea. Finalmente, es necesario resaltar el aspecto clave que hace el desarrollo del presente TFG posible: la disponibilidad del código fuente, la amplia documentación y la facilidad para aportar *feedback* mediante *issues* en la plataforma Github, estableciendo así un canal de comunicación directo con aquellos contribuyentes clave.

## 5.6. Aceleración de la Transformada Discreta *Wavelet*

En las secciones a continuación se presenta cómo se puede acelerar una aplicación mediante el uso de un acelerador *hardware*, en concreto, una ejecución acelerada por medio de instrucciones vectoriales RISC-V y el acelerador *hardware* Ara.

### 5.6.1. Transformada *wavelet*

La transformada *wavelet* extiende la idea básica de la transformada de Fourier de reconstruir una señal compleja a partir de señales sencillas. La diferencia crucial radica en la naturaleza de las señales empleadas para la descomposición, mientras que en la transformada de Fourier se emplea la contribución de multitud de senos y cosenos; en la transformada *wavelet* se emplean *wavelets*. Esta familia de funciones de corta duración que ocupan un instante de tiempo concreto confiere a la transformada *wavelet* la capacidad de preservar la información relativa al tiempo, además de la relativa a la frecuencia, como únicamente es capaz la transformada de Fourier.

En definitiva, la transformada *wavelet* permite caracterizar a las señales que son susceptibles de sufrir transitorios abruptos, preservando el aspecto temporal de dichos transitorios.

A continuación se muestra un caso práctico que ilustra la utilidad de la transformada *wavelet*. Partiendo de una señal formada por la suma de dos *chirps*, figuras 5.29 y 5.30.

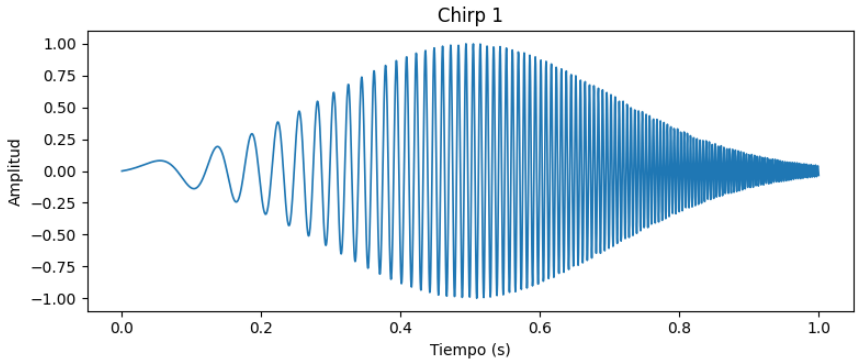



Figura 5.29: Señal `chirp1`.

[2] neorv32/#_cpu_performance
[3] area-usage-and-maximal-frequency

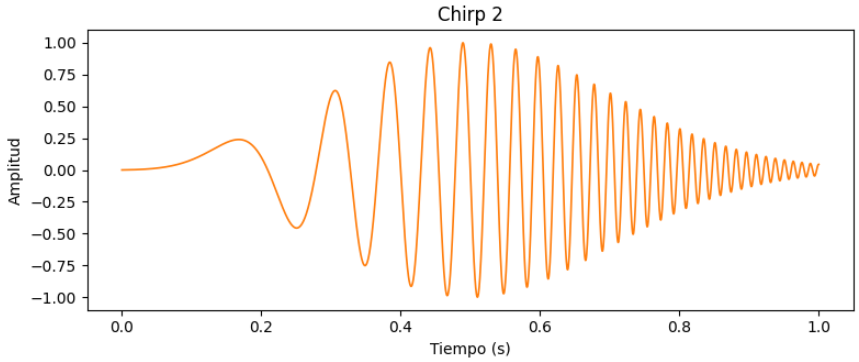



Figura 5.30: Señal `chirp2`.

Las señales *chirp* de las figuras 5.29 y 5.30, se caracterizan por una variación de la frecuencia con el tiempo, peculiaridad que concede a la señal de interés que está formada por la suma de ambos *chirps* una naturaleza poco intuitiva.

Mientras que la transformada de Fourier puede arrojar algo de claridad, no permite relacionar de forma clara el par de señales originales con la señal 5.31, como queda de manifiesto en la figura 5.32.

En cambio, debido a que la transformada *wavelet* conserva la información relativa al tiempo, es posible crear un escalograma como el de la figura 5.33 que posibilita entender de una forma más clara las contribuciones de las dos señales que conforman la señal, 5.31.

Las figuras de la sección actual han sido creadas mediante Python, en concreto, el código en 21 y 22, incluido en los anexos 5.

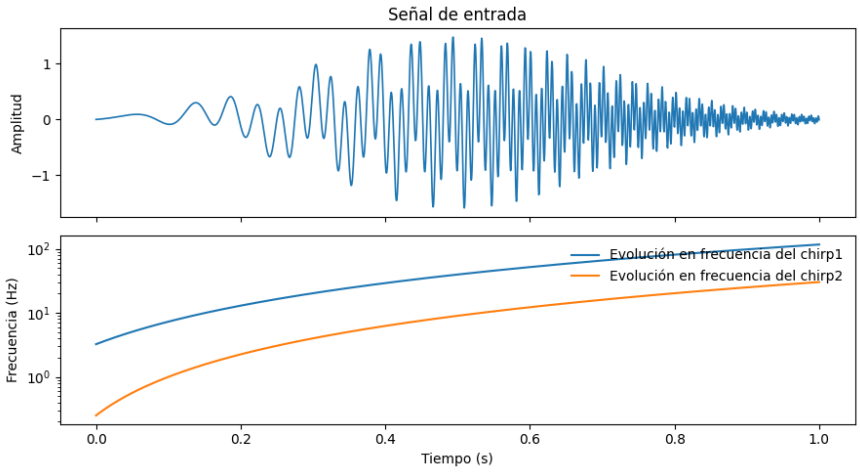



Figura 5.31: Señal de entrada resultado de la suma de `chirp1` y `chirp2`, justo con la evolución de sus frecuencias en función del tiempo.

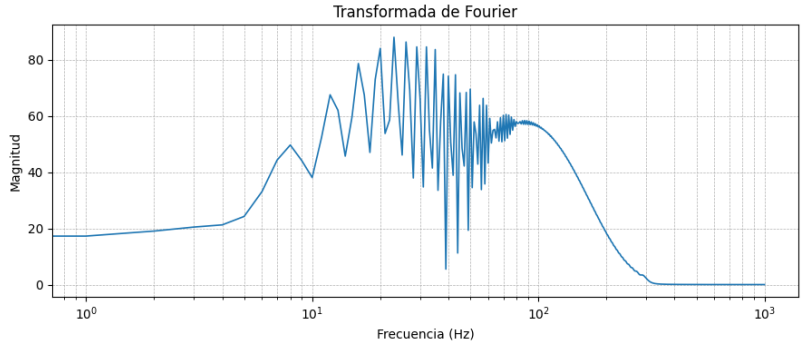



Figura 5.32: Espectro de frecuencia resultante de la transformada de Fourier.

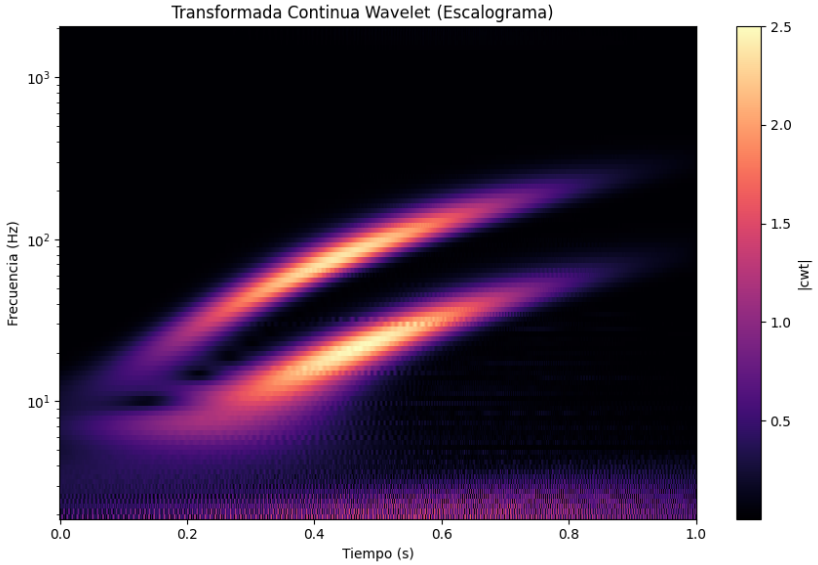



Figura 5.33: Escalograma obtenido como resultado de la transformada continua wavelet.

### 5.6.2. Fast Wavelet Transform

El repositorio del acelerador *hardware* Ara incluye un directorio con ejemplos que demuestran de forma práctica cómo se pueden acelerar diversos algoritmos de uso común como: convoluciones 2D/3D, Fast Fourier Transform (FFT), operación *dropout* para redes neuronales, entre otros.

A continuación se explicará en detalle cómo se acelera la ejecución del algoritmo FWT mediante el correspondiente ejemplo `apps/dwt`, que computa el algoritmo FWT sobre una matriz unidimensional de números de tipo `float` de 32 bits (muestras de entrada).

El ejemplo realiza la comparación del tiempo de ejecución (en concepto de ciclos) de la FWT entre para una implementación que hace uso de la extensión vectorial y, por tanto, del Ara, y una implementación tradicional que no hace uso del acelerador.

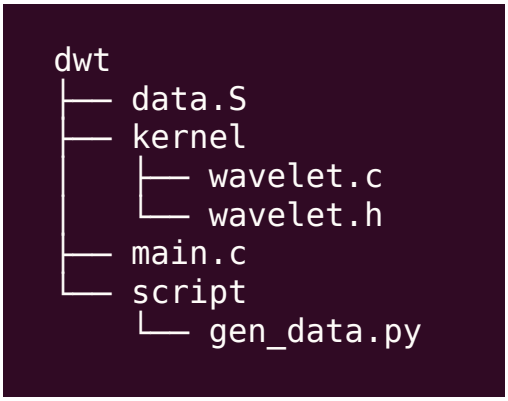



Figura 5.34: Estructura del directorio `apps/dwt` dentro del repositorio.

La FWT pasa a la matriz de entrada `a`, a través de dos filtros: `g`, que da lugar a los llamados coeficientes de aproximación y `h` que genera los coeficientes de detalle, figura 5.35 y tras esto se reduce el número de muestras en un factor de 2. Esto se repite tantas veces como se desee, es decir, tantos niveles como se desee, en este caso se emplean dos niveles.

#### Implementación tradicional

La función principal, 7, representa un nivel, con sus correspondientes filtros. Como el ejemplo únicamente computa dos niveles, se hacen dos llamadas a esta función. La función tiene 4 argumentos:

- **`gsl_wavelet`:** estructura que guarda la información relativa al *wavelet* utilizado:
  - **`*h1`:** puntero a la matriz de tipo `float` con los coeficientes del filtro `h`.
  - **`*g1`:** puntero a la matriz de tipo `float` con los coeficientes del filtro `g`.
  - **`nc`:** número de coeficientes.
  - **`offset`:** desplazamiento de los coeficientes.
- **`a`:** matriz con las muestras de entrada.
- **`n`:** número de muestras.
- **`buf`:** búfer que guarda de forma temporal los coeficientes de detalle.

La información relativa al *wavelet* utilizado se guarda en la estructura `w`, en este caso el *wavelet* de Haar, cuyos filtros asociados están definidos como se puede ver en la ecuación 5.1.

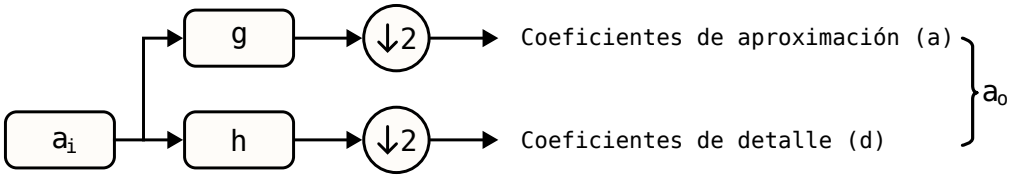



Figura 5.35: Esquema de los filtros calculados con la función `dwt_step`, donde `g` y `h` son los filtros asociados al Wavelet de Haar.

```
static inline void dwt_step(const gsl_wavelet *w, float *a, size_t n,
                            float *buf) {
  size_t i, ii;
  size_t jf;
  size_t k;
  size_t n1, ni, nh, nmod;
  float h, g;

  // Setup
  nmod = w->nc * n;
  nmod -= w->offset; // center support
  n1 = n - 1;
  nh = n >> 1;

  // Compute and downsample (factor 2)
  for (i = 0, ii = 0; i < n; i += 2, ++ii) {
    h = 0;
    g = 0;
    ni = i + nmod;
    for (k = 0; k < w->nc; k++) {
      // If offset == 0, then jf == (i + k)
      jf = n1 & (ni + k);
      h += w->h1[k] * a[jf];
      g += w->g1[k] * a[jf];
    }
    a[ii] = g;
    buf[ii] = h;
  }

  for (k = 0; k < n / 2; ++k) {
    a[k + nh] = buf[k];
  }
}
```

Código fuente 7: Implementación tradicional de cada nivel de la FWT.

$$g = \left[\frac{1}{\sqrt{2}},\ \frac{1}{\sqrt{2}}\right], \quad h = \left[\frac{1}{\sqrt{2}},\ -\frac{1}{\sqrt{2}}\right] \tag{5.1}$$

$$a_o \ = \ \left((a_i * g) \downarrow 2 \ \middle|\ (a_i * h) \downarrow 2\right) \tag{5.2}$$

El primer bucle `for` realiza la convolución entre el filtro `h` y al señal de entrada, guardando el resultado en `a`. Es importante tener en cuenta que aunque se lee y escribe en la matriz `a`, en ningún momento hay interferencia entre el resultado de la convolución (`a[ii]`) y la señal original, en uso (`a[jf]`).

Por otra parte, la convolución entre el filtro `g` y la señal de entrada se guarda en el búfer, `buf`, al terminar todas las convoluciones, se guarda la contribución de dicho filtro en la parte final de `a`, siguiendo la operación que se puede ver en la ecuación 5.2.

En la implementación se considera que la señal de entrada es finita, por tanto, el cálculo de los índices requiere de la operación módulo para generar los índices de la indexación circular.

La implementación presenta una optimización, que conlleva la restricción de que el tamaño de la matriz de entrada tiene que ser una potencia de 2. A cambio, se elimina la necesidad de tener que realizar la operación módulo.

#### Implementación con funciones intrínsecas vectoriales

La función `dwt_step_vector` es el equivalente a la función `dwt_step` anterior, solo que esta función hace uso de las funciones intrínsecas correspondientes a la extensión vectorial de RISC-V. Los principales factores que diferencian ambas implementaciones son:

- **Directivas del preprocesador `#ifndef`**: controlan qué código final se compila.
  - `SMALL_PROBLEM`: en el caso de que el número sea inferior a 2048 se pueden cargar en su totalidad en los registros vectoriales y no es necesario hacer múltiples cargas eliminando la necesidad de bucles.
  - `SEGMENT`: controla la estrategia que se utiliza para cargar las muestras.
- **Funciones intrínsecas vectoriales:** Cada función intrínseca se corresponde con una instrucción de la extensión vectorial de RISC-V con una configuración específica.

La ejecución de las operaciones vectoriales se basa en una serie de valores de configuración que, una vez escritos a sus correspondientes CSRs definen, junto a la implementación física de la unidad de cómputo vectorial, en este caso el acelerador Ara, cómo se organizan los datos sobre los que las instrucciones vectoriales operan. Estos valores son:

- **ELEN:** El tamaño de cada elemento vectorial ha de ser mayor o igual a 8 y ser una potencia de 2. En este caso, el algoritmo trabaja con elementos de 32 bits.
- **VLEN:** El tamaño de los registros vectoriales. Ha de ser igual o mayor que ELEN y una potencia de 2. En el Ara $vlen = 4096$.
- **SEW:** Divide el espacio configurado en una serie de elementos: $8 \leq sew \leq elen$.

- **LMUL:** Multiplicador que permite utilizar múltiples registros o parte de uno para albergar los elementos con los que se trabaja: $lmul = 2^k, -3 \leq k \leq 3$.
- **VLMAX:** Determinado por SEW y LMUL, establece el número máximo de elementos con los que se opera. $vlmax = lmul \cdot vlen/sew$.
- **VL:** Número de elementos con los que se opera con una instrucción concreta. $VL \leq vlmax$.
- **VSTART:** Elemento inicial a partir del cual se opera.

Según los parámetros anteriores, es posible trabajar con elementos con más o menos bits y una cantidad de los mismos variable. Se puede utilizar multitud de elementos pequeños en un solo registro o en múltiples registros o cualquier combinación que sea necesaria. En conclusión, flexibilidad total sobre la organización de los datos. En las figuras 5.37 y 5.36 se pueden ver diferentes combinaciones de los parámetros.

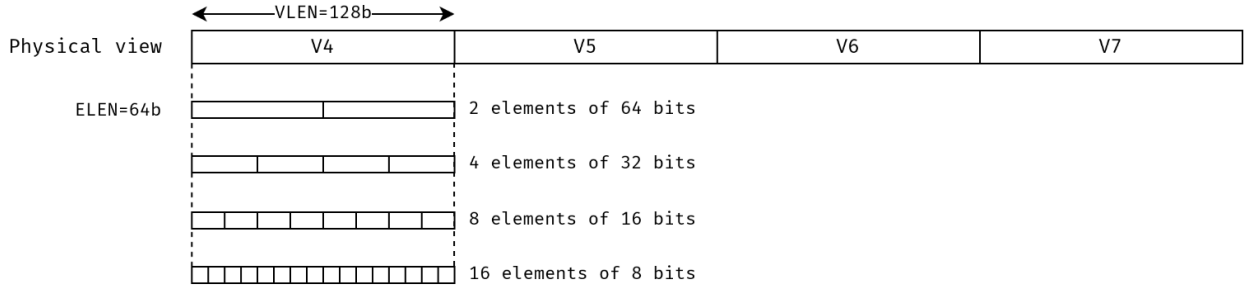



Figura 5.36: Ejemplo de posibles configuraciones de vectoriales, énfasis en ELEN [47].

En esta segunda implementación se emplean instrucciones vectoriales por medio de funciones intrínsecas que son las que permiten establecer los parámetros antes vistos y operar con los registros vectoriales.

Las funciones intrínsecas de RISC-V están definidas en [48], donde se detalla la nomenclatura, 8 de referencia:

```
__riscv_{V_INSTRUCTION_MNEMONIC}_{OPERAND_MNEMONIC}_{RETURN_TYPE}_{ROUND_MODE}_{POLICY{(...)}
```

Código fuente 8: Formato de funciones intrínsecas según la especificación *RISC-V Vector C Intrinsic* [48].

Las funciones intrínsecas que se utilizan en la implementación, mostrada en el código fuente 10, están compuestas por:

- `V_INSTRUCTION_MNEMONIC:`
  - `vsetvl`: Establece el tamaño de los vectores (`VL`). por tanto, el número de vectores sobre los que cada instrucción realizará las operaciones.
  - `vlseg2e32`: Carga las muestras de entrada (`samples_r`) a los registros vectoriales siguiendo la configuración indicada por el nombre de la función intrínseca. Hay que tener en cuenta que las variables que

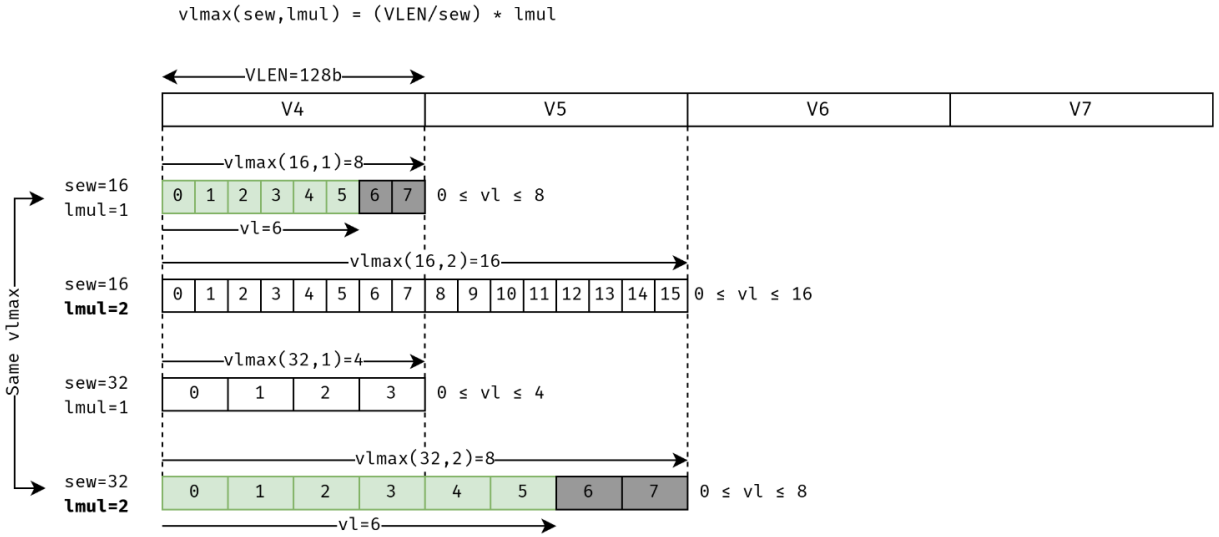



Figura 5.37: Ejemplo de posibles configuraciones vectoriales [47].

se han declarado al inicio y que utilizan en los argumentos no son variables convencionales; en su lugar, es una forma de indicar al compilador, cómo se han de cargar los datos en los registros, en concreto:

  - 2: La carga toma los valores pares e impares y los guarda los 2 vectores resultantes en los registros, los segmentos donde se cargan los vectores tienen una longitud de 1024 `floats`, como hace referencia la ecuación 5.3.
  - e32: Cada unidad es de 32 bits, $sew = 32\ bits$.

- `vlse32`: Forma alternativa de cargar los vectores, resulta de igual modo en la carga de los dos vectores, uno con las muestras pares y otro con las impares. Es necesario 2 instrucciones en lugar de una.
- `vfmul`: Multiplica todos los elementos del vector por un número. Como se puede ver en 10, se multiplica el primer coeficiente de un filtro por los valores impares, y se guarda en el vector `g_vec`, obteniéndose así, el resultado parcial, de forma similar para `h_vec`.
- `vfmacc`: Multiplica todos los elementos del vector por un número y lo suma con un vector. En ese caso, se multiplica el segundo coeficiente de cada filtro por el vector de valores pares y se suma con el valor de `*_vec`.
- `vse32`: Guarda un vector a memoria. Se guarda `g_vec` en `samples_w` y `h_vec` en `buf_w`. Al final de la función se utiliza para guardar el resultado del cómputo como en la ecuación 5.2.
- `vle32`: Carga de memoria a un vector. Se emplea para leer `buf_r` y guardarlo en `h_vec`.

- `RETURN_TYPE:`
  - `e32m4`: Establece SEW=32b, indicando que cada elemento es de 32 bits y LMUL=4 para que la operación utilice un grupo de 4 registros vectoriales, siendo cada registro de 4096 bits.

- `f32m4`: Establece los mismos valores antes indicados, especificando que se realiza una conversión a tipo `float`.

Dada la información que ha expuesto hasta el momento, es posible determinar la cantidad máxima de elementos de tipos `float` (muestras de la señal de entrada) con los que es posible trabajar prescindiendo de bucles (controlado con la directiva `#ifndef SMALL_PROBLEM`), según la ecuación 5.3.

$$N_{\text{máx}} = L_{\text{hw}} \times S_{\text{seg}} \times \text{LMUL} \times \frac{\text{VLEN}}{\text{SEW}} \tag{5.3}$$

donde:

$$\begin{aligned} L_{\text{hw}} &= \text{número de } \textit{lanes} \text{ físicas} && = 2, \\ S_{\text{seg}} &= \text{segmentos por instrucción (}\texttt{vlseg2}\text{)} && = 2, \\ \text{LMUL} &= \text{agrupación de los registros} && = 4, \\ \text{VLEN} &= \text{tamaño de los registros vectoriales del Ara (bits)} && = 4096, \\ \text{SEW} &= \text{tamaño de las muestras (bits)} && = 32. \end{aligned}$$

Sustituyendo:

$$N_{\text{máx}} = 2 \cdot 2 \cdot 4 \cdot \frac{4096}{32} = 2048$$

Hasta el momento se han detallado las peculiaridades asociadas a las funciones intrínsecas para instrucciones vectoriales en RISC-V, a continuación, se analizará cómo se implementa el algoritmo FWT, con dichas instrucciones.

Debido a la extensión de la función, esta se ha dividido en tres bloques de código:

- en la sessión (9) se incluyen de declaraciones,
- en la sección (10) se incluyen las convoluciones de los filtros, y finalmente
- en se incluyen (11) consolidación de los resultados en una única variable, `samples_w`.

Estos tres pasos utilizan las ideas subyacentes que forma la implementación tradicional antes vista.

En el primer segmento de código, sección 9, se declaran las variables de las que se harán uso a través del programa, en la implementación tradicional, se hace uso de `a` para guardar los resultados. Ahora, en cambio, los resultados se guardan en `samples_w`. Es importante tener en cuenta que el tipo `vfloat32m4_t` no es una estructura clásica de `C`; en cambio, es la forma mediante la que se hace referencia al hecho de cargar esos datos a los registros vectoriales.

En la sección de código 10, se concentra el núcleo del algoritmo. En primer lugar, se establece `vl`, al número de elementos con los que se operará con cada instrucción. En caso de que el número de muestras de entrada exceda lo calculado en 5.3, se recurrirá al uso de bucles (en función de `#ifndef SMALL_PROBLEM`). Inmediatamente después, se cargan los segmentos desde la variable `samples_r` (memoria) siguiendo una de las dos estrategias de carga, según `#ifdef SEGMENT`:

- Carga segmentada: Toma los segmentos iniciales y los carga en dos vectores `sample_vec_0` y `sample_vec_1` utilizando de forma alternada los segmentos para completar los vectores.

```
static inline void dwt_step_vector(const gsl_wavelet *w, float *samples,
                                   size_t n, float *buf) {

  size_t avl = n;
  vfloat32m4_t sample_vec_0;
  vfloat32m4_t sample_vec_1;
  vfloat32m4_t g_vec;
  vfloat32m4_t h_vec;

  float *samples_r = samples;
  float *samples_w = samples;
  float *buf_r = buf;
  float *buf_w = buf;
```

Código fuente 9: Principio de la función `dwt_step_vector`.

- Cargas de paso fijo: Se fija un paso o tamaño de salto, 2 * 32 bits que tiene cada muestra y el inicio que será para ambos vectores resultantes: la primera y segunda muestras de la matriz de muestras. `samples_r` y `samples_r + 1`.

Una vez se han guardado las muestras en los registros vectoriales como dos segmentos continuos de elementos referidos como "muestras", se lleva cabo la computación, en concreto, la ecuación 5.38 implementada en dos pasos: una multiplicación por una constante (instrucciones `fmul`) seguida de una multiplicación y suma (operaciones `fmacc`).

Las instrucciones `vse32` guardan en memoria el contenido de los vectores `g_vec` y `h_vec`, ambos contienen `vl / 2` elementos, como se ha hecho referencia con anterioridad.

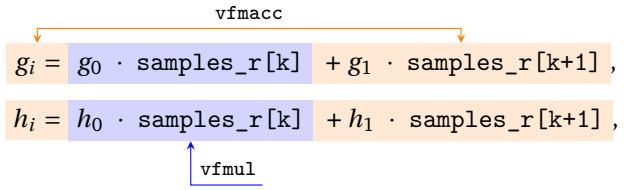
Tras esto se avanzan los punteros de: (1) la memoria donde se guardan las muestras inicialmente (`samples_r`), (2) la memoria donde se guardan los resultados (`samples_w`), y (3) el búfer temporal para el resultado con los coeficientes de detalle (`buf_w`).

Finalmente, se guardan los coeficientes de detalle que residen en el búfer posición de memoria indicada por el puntero `samples_w`. En el caso de que se comience con un número de muestras mayor a 2048, es necesario realizar la operación recurriendo al bucle delimitado por `#ifndef SMALL_PROBLEM`.

A partir de estas instrucciones, se termina la ejecución de la función, en el caso de que el número de muestras fuera inferior a 2048, cifra resultante de las decisiones de implementación plasmadas en la ecuación 5.3 ó se realizan los preparativos para la segunda ejecución del bucle delimitado por `#ifndef SMALL_PROBLEM`.

```
  // Strip-Mining loop
#ifndef SMALL_PROBLEM
  for (size_t vl = __riscv_vsetvl_e32m4(avl); avl > 0; avl -= vl) {
#endif
    size_t vl = __riscv_vsetvl_e32m4(avl);
    // If we have enough samples, fill the vector registers!
    if (avl >= 2 * vl)
      vl *= 2;
#ifdef SEGMENT
    // Segment load the vectors. ToDo: check if vl/2 is correct
    __riscv_vlseg2e32_v_f32m4(sample_vec_0, sample_vec_1, samples_r, vl / 2);
#else
  // Strided load (inefficient!)
  sample_vec_0 =
      __riscv_vlse32_v_f32m4(samples_r, 2 * sizeof(*samples_r), vl / 2);
  sample_vec_1 =
      __riscv_vlse32_v_f32m4(samples_r + 1, 2 * sizeof(*samples_r), vl / 2);
#endif

    // First implementation
    // nc == 2!

    // Generate the g vector and store it back
    // Generate the h vector and store it back
    g_vec = __riscv_vfmul_vf_f32m4(sample_vec_0, w->g1[0], vl / 2);
    h_vec = __riscv_vfmul_vf_f32m4(sample_vec_0, w->h1[0], vl / 2);

    g_vec = __riscv_vfmacc_vf_f32m4(g_vec, w->g1[1], sample_vec_1, vl / 2);
    h_vec = __riscv_vfmacc_vf_f32m4(h_vec, w->h1[1], sample_vec_1, vl / 2);

    __riscv_vse32_v_f32m4(samples_w, g_vec, vl / 2);
    __riscv_vse32_v_f32m4(buf_w, h_vec, vl / 2);

    // Bump pointers
    samples_r += vl;    samples_w += vl / 2;
    buf_w += vl / 2;
#ifndef SMALL_PROBLEM
  }
#endif
```

Código fuente 10: Cálculo de la convolución de ambos filtros.




Ecuación 5.38: Operaciones matemáticas realizadas con las instrucciones vectoriales `vfmul` y `vfmacc`.

```
  // Memcpy h_vec to the samples vector
  avl = n / 2;
#ifndef SMALL_PROBLEM
  for (size_t vl = __riscv_vsetvl_e32m4(avl); avl > 0; avl -= vl) {
#else
  vl = __riscv_vsetvl_e32m4(avl);
#endif
    h_vec = __riscv_vle32_v_f32m4(buf_r, vl);
    __riscv_vse32_v_f32m4(samples_w, h_vec, vl);
    buf_r += vl;
    samples_w += vl;
#ifndef SMALL_PROBLEM
  }
#endif
}
```

Código fuente 11: Consolidación de los resultados en una variable, según lo visto en la ecuación 5.2.

### 5.6.3. Simulación con QuestaSim

Tras haber efectuado un análisis teórico de la implementación del Discrete Wavelet Transform (DWT), en esta sección se analizan los resultados de la compilación y la simulación del programa mediante QuestaSim, estudiando las particularidades de este programa para *baremetal*. El análisis ha sido planteado tomando como base 3 referencias: el *dump* del binario compilado (`dwt.dump`), el código escrito en C y la simulación con QuestaSim. Para iniciar la simulación y abrir QuestaSim, se puede ejecutar el comando shell mostrado en el código 12, de este modo se han obtenido las figuras 5.39 y 5.40.

```
cd hardware
app=hello_world make simc
```

Código fuente 12: Comando para realizar la simulación y abrir QuestaSim.

Una de las particularidades que presenta el `main.c` del programa `apps/dwt` es la declaración de las variables globales que guardan en número de muestras con las que se opera, la matriz de datos para la ejecución de la

implementación estándar, la matriz de datos para la ejecución de la implementación vectorial y la variable `buf`, utilizada para guardar parte de los coeficientes de forma momentánea, declaradas como se lista en el código 13.

```
extern uint64_t DWT_LEN;
extern float data_s[] __attribute__((aligned(4 * NR_LANES)));
extern float data_v[] __attribute__((aligned(4 * NR_LANES)));
extern float buf[] __attribute__((aligned(4 * NR_LANES)));
```

Código fuente 13: Declaración de las variables globales utilizadas a través del programa.

Dichas variables se guardan en memoria siguiendo las instrucciones en el extracto del código ensamblador presentado en el código 14, y son las que posteriormente se cargarán en los 4096 bits (`VLEN`) de registros vectoriales distribuidos a través de los 4 *lanes* utilizados en la simulación.

```
.section .data,"aw",@progbits
.global DWT_LEN
.balign 8
DWT_LEN:
    .word 0x00000200
    .word 0x00000000
.global data_s
.balign NR_LANES*4
data_s:
    .word 0x3f4768ee
    .word 0x3f115a76
...

.global data_v
.balign NR_LANES*4
data_v:
    .word 0x3f4768ee
    .word 0x3f115a76
...

.global buf
.balign NR_LANES*4
buf:
    .word 0x00000000
    .word 0x00000000
...
```

Código fuente 14: Código ensamblador generado con una *script* de Python en el que se declaran las variables declaradas en el código 13.

Una vez que las muestras están en memoria, se acceden a través del bus AXI por el módulo VLSU del acelerador Ara (Vector Load Store Unit). El módulo VLSU, presente en la figura 4.2a es el encargado de realizar las operaciones

de acceso a la memoria. Las muestras se cargan en bloques de 128 bits y son dirigidos a los bancos de memoria que forman los 1024 bits de registros vectoriales por *lane*, 4.2b del acelerador *hardware* Ara.

En la figura 5.39, el instante de tiempo indicado por el marcador vertical amarillo hace referencia al inicio de la computación con el Ara. Es en este instante de tiempo en el que el módulo VLSU realiza la primera carga de memoria a través del bus AXI en la dirección `80001bd0`, el bus AXI responde con el contenido de esa posición de memoria, que se corresponde con las 4 primeras muestras de 32 bits en orden inverso `3F7F5AD2`, `3F7BCC15` y en especial `3F115A76` y `3F4768EE` que son las dos primeras muestras de del vector `data_v`, según se puede ver en el ensamblador del listado mostrado en el código 14.

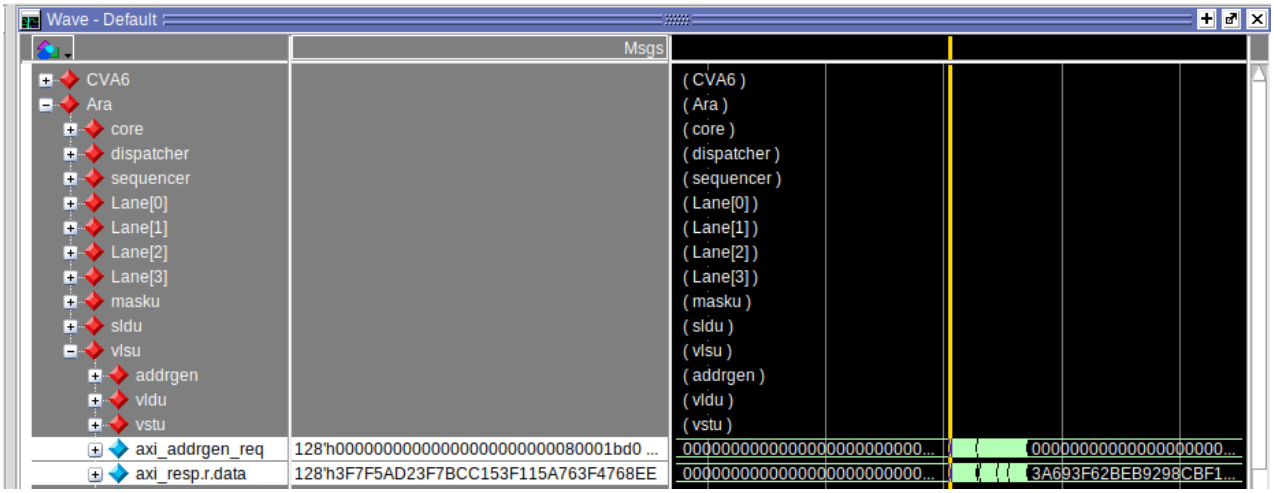



Figura 5.39: Instante de tiempo en el que el VLSU recupera de la memoria las 4 primeras, muestras contenidas en 128 bits.

### Ejemplo de ejecución de instrucción vectorial - instrucción `vfmacc`

En la figura 5.40 se aprecia la ejecución de la operación `vfmacc`. Esto se corresponde con una de las instrucciones del código de la implementación vectorial del código 10 y se corresponden con de la ecuación 5.38. En las señales que representan los operandos de la instrucción `vfmacc` (`operands_i`) se pueden encontrar las mismas muestras que previamente han sido cargadas por el VLSU y que fueron declaradas inicialmente en el código ensamblador 15. El operando restante contiene uno de los coeficientes por los que se han de multiplicar las muestras, 0.70710677, hay que tener en cuenta que, debido a que el programa opera con valores de tipo flotante, la codificación de dichos valores, en consecuencia, es en punto flotante y de acuerdo a la figura 5.41.

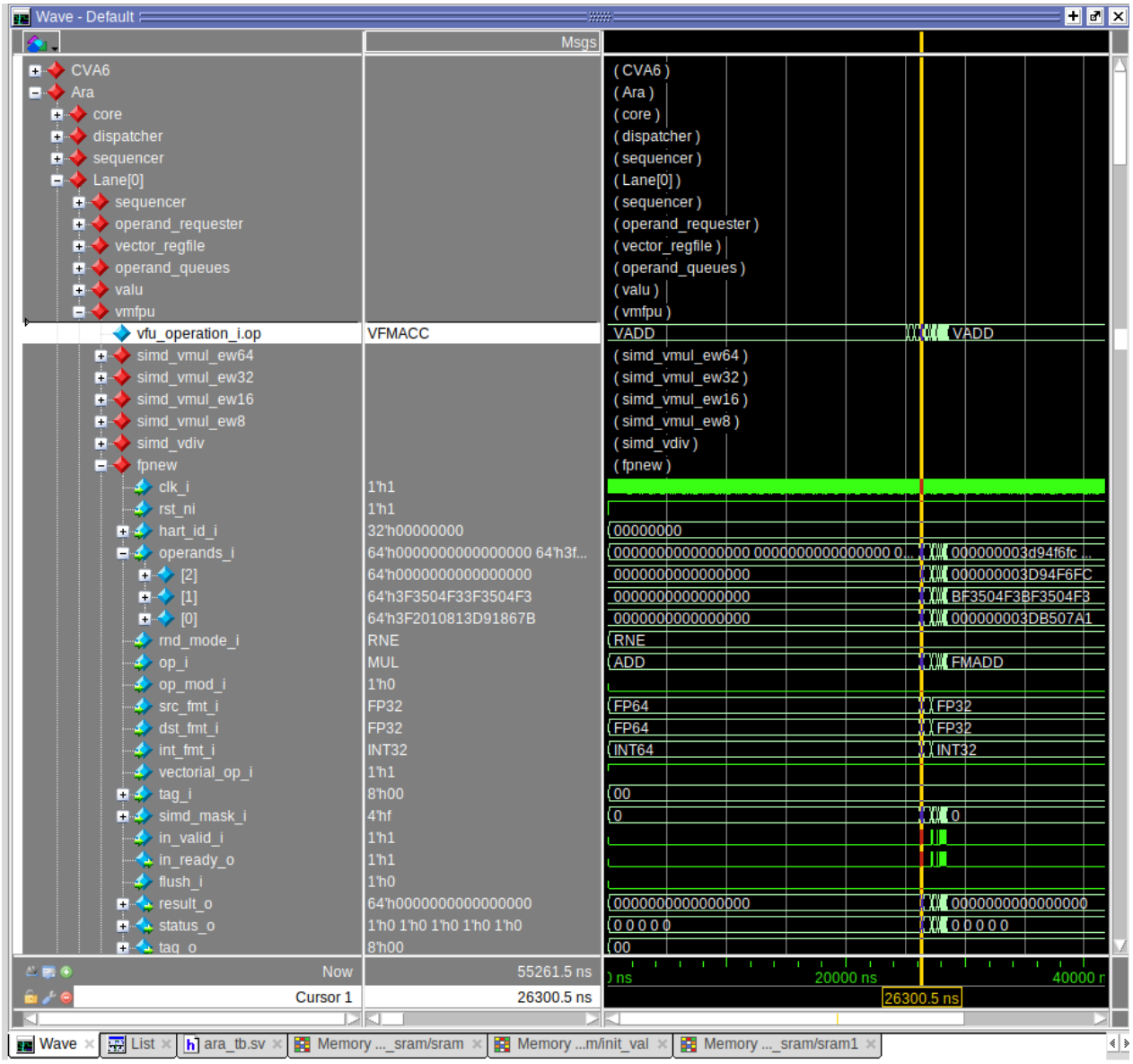



Figura 5.40: Operación de tipo `vfmacc` como se puede ver en la ecuación 5.38.

```
.global data_v
.balign NR_LANES*4
data_s:
    .word 0x3f4768ee
    .word 0x3f115a76
...
    .word 0x3f6f973a
    .word 0x3f201081
```

Código fuente 15: Ejemplo de muestra con la que se opera durante el ejemplo de operación `vfmacc` que se puede ver en la figura 5.40.

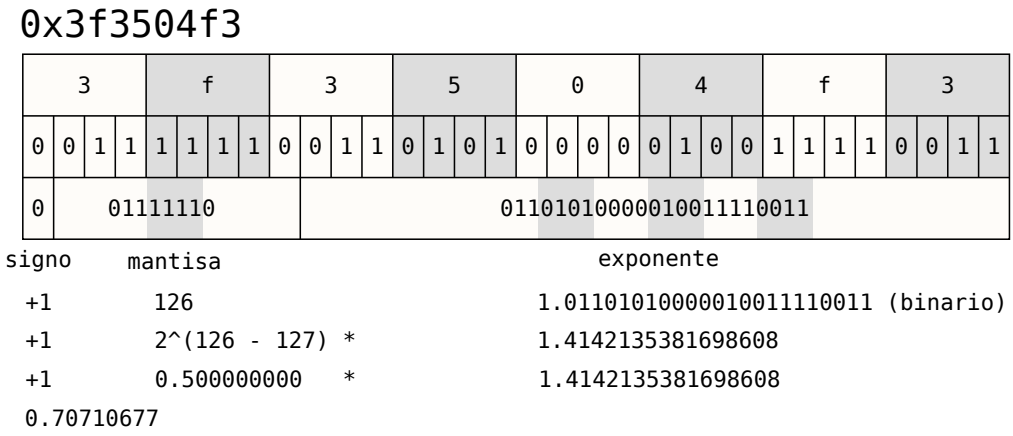



Figura 5.41: Conversión de punto flotante hexadecimal a punto flotante decimal, `0x3f3504f3 = 0.70710677`.

### 5.6.4. Resultados de la simulación completa con QuestaSim

Además de ejecutar la simulación gráfica para estudiar en detalle los valores que toman cada una de las señales del sistema, es posible simular el programa en su totalidad viendo el resultado del mismo por la consola, junto a la salida de la UART, que es redirigida a la salida estándar del mismo terminal en el que se ejecuta la simulación.

Según los resultados de la simulación de la figura 5.42, la **implementación estándar** requiere de **17576 ciclos**, mientras que la **implementación vectorial** se completa en **3344 ciclos**. El rendimiento obtenido es un ≈30 % del rendimiento máximo teórico. Los valores cuyos nombres están escritos entre corchetes, empezando con `[hw-cycles]`, corresponden a mediciones realizadas mediante contadores *hardware*, entre estos se observa que no se han producido *stalls* relacionados con el caché de datos y, en cambio, se han producido 4 *stalls* relacionados con el caché de instrucciones.

```
# run -a
# Dump results on ../gold_results.txt
# Loading ELF file /home/users/rvproject/ara/ara/apps/bin/dwt
# Loading section 0000000080000000 of length 00000000000013be
# Loading section 00000000800013c0 of length 0000000000001410
# Loading section 00000000800027d0 of length 0000000000000550
# Loading section 0000000080002d20 of length 0000000000000008
# [TRACER] Output filename is: trace_hart_0.log
#
# =========
# =  DWT  =
# =========
#
#
# Computing DWT with 512 samples
# Scalar DWT...
# The scalar DWT execution took 17576 cycles.
# Vector DWT...
# The vector DWT execution took 3344 cycles.
# Max ideal performance is 4.000000, max real performance is 3.000000, actual performance is 0.916866.
# Performance over max real: 30.562199%.
# Test result: PASS. No errors.
# [hw-cycles]:        3235
# [cva6-d$-stalls]:           0
# [cva6-i$-stalls]:           4
# [cva6-sb-full]:           0
# ** Info: Core Test *** SUCCESS *** (tohost = 0)
#    Time: 55261500 ps  Scope: ara_tb File: /home/users/rvproject/ara/ara/hardware/tb/ara_tb.sv Line: 213
# End time: 12:16:24 on Jul 11,2025, Elapsed time: 0:00:15
# Errors: 0, Warnings: 0
./scripts/return_status.sh build/transcript

Simulation returned 0
```

Figura 5.42: Resultado de la simulación del acelerador *hardware* Ara con QuestaSim.

## 5.7. Cheshire

Cheshire [49] es una plataforma con capacidad para ejecutar Linux, que ha sido diseñada inicialmente como plataforma basadas en CVA6. Permite el desarrollo de SoCss de distinta complejidad, desde sistemas empotrados hasta aceleradores con multitud de núcleos, con la configurabilidad y el minimalismo como valores clave. Del mismo modo que CVA6, el Cheshire, ha sido desarrollado en conjunto entre la Escuela Politécnica Federal de Zúrich y la Universidad de Bolonia.

### 5.7.1. Arquitectura del Cheshire

Es posible utilizar configuraciones con hasta 31 núcleos CVA6 conectados entre sí. Está formado por 5 grupos de módulos:

1. Núcleos CVA6,

2. Periféricos multiplexados (UART, $I^2C$, SPI, GPIO, Boot ROM, etc.),

3. Módulo de DMA,

4. Periféricos no multiplexados (JTAG Debug, Serial Link, VGA y USB 1.1) y

5. El *crossbar* que une todos los elementos que conforman el sistema, según se indica en la figura 5.43

Cheshire ha sido llevado a ASIC por medio del chip *standalone*, *Neo*[4] y el módulo de 65nm de TSMC, alcanzando una velocidad de reloj de 325 MHz y un consumo inferior a 300 mW durante el desarrollo de tareas de computación con amplio uso de datos.

[4] http://asic.ethz.ch/2022/Neo.html

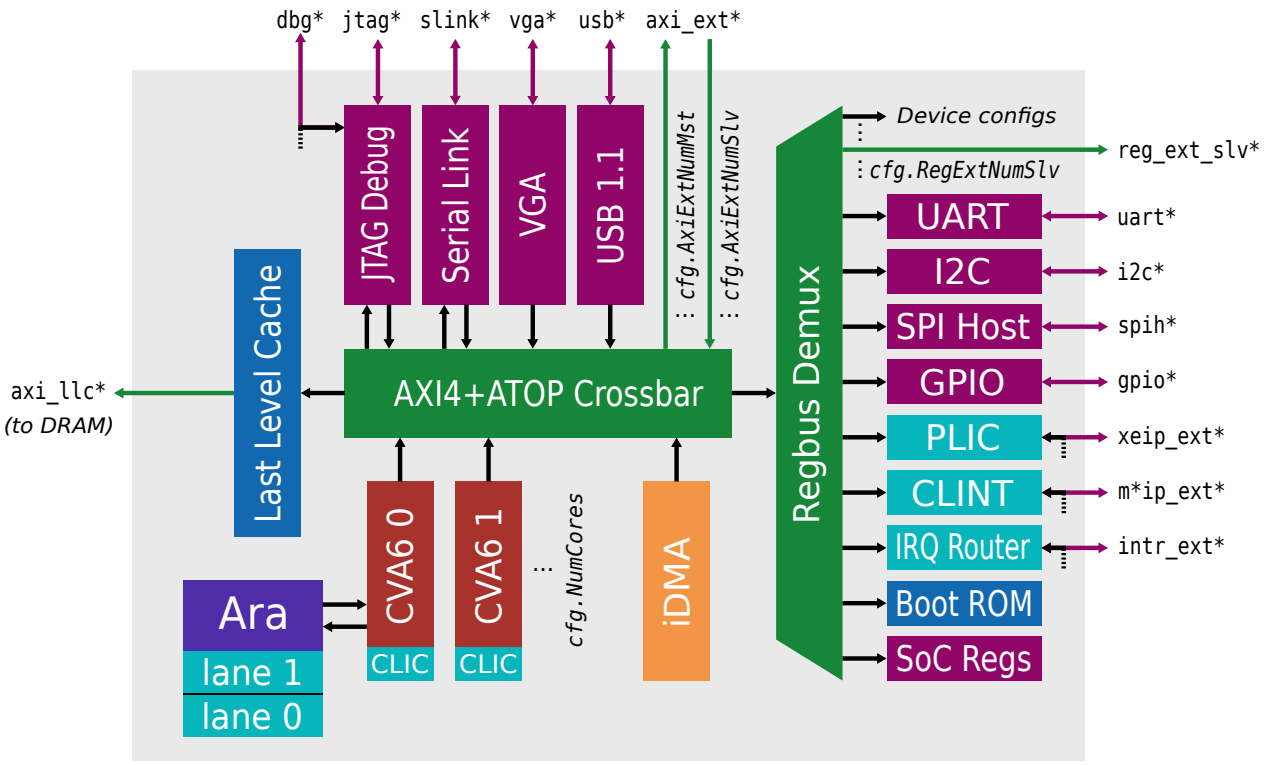



Figura 5.43: Diagrama de bloques del SoC Cheshire con los periféricos, el número de núcleos CVA6 configurados, 1 en este caso, y con el acelerador *hardware* Ara.

## 5.8. Implementación de Ara en Cheshire para VCU128

La implementación del acelerador *hardware* Ara sobre el VCU128 se ha realizado mediante la plataforma Cheshire y, por tanto, con el CVA6 como *host*. El proceso que se detalla a continuación ha sido completado parcialmente mediante diferentes sistemas operativos Linux, debido al gran número de dependencias que tiene cada proceso, aunque en teoría es posible llevar a cabo las instrucciones[5] en un sistema Linux genérico. Para compilar las dependencias de cada proyecto es necesario contar con la *toolchain* de RISC-V [50], además de las dependencias mostradas en el comando del código 16.

```
sudo apt install autoconf automake autotools-dev curl python3 python3-pip
 ↪  python3-tomli libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex
 ↪  texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build git
 ↪  cmake libglib2.0-dev libslirp-dev
```

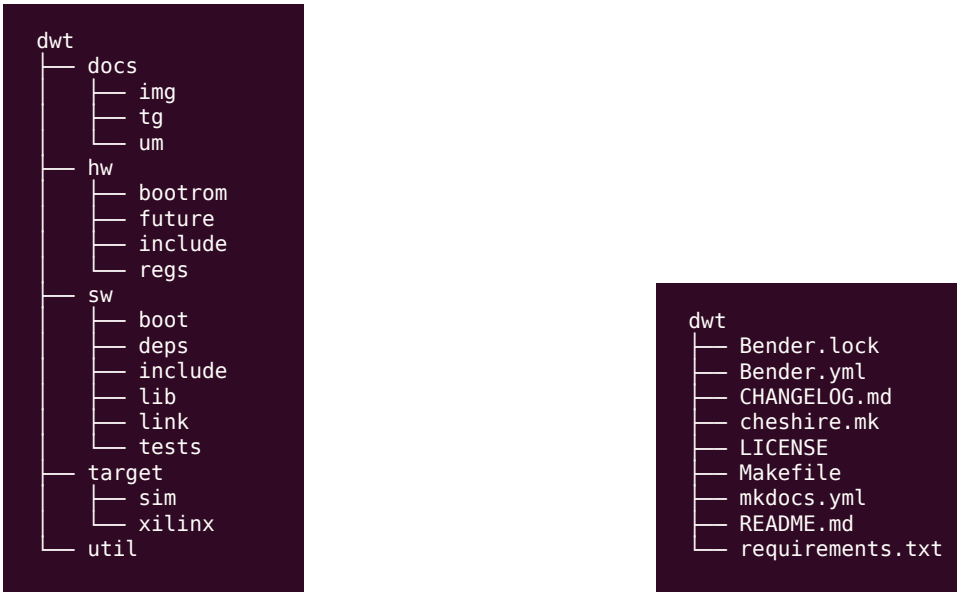
Código fuente 16: Instalación de dependencias principales.

[5] https://github.com/pulp-platform/ara/tree/main/cheshire

Además de las dependencias indicadas (código 16), es necesario disponer de un entorno de Python en el que descargar las dependencias que dicta el archivo `requirements.txt` y tener en el `$PATH` el binario de `bender` que es la herramienta de manejo de dependencias *hardware* que utiliza la plataforma PULP y mediante la que se puede obtener el directorio bajo el que se encuentra el Ara, como dependencia.

```
ARA_ROOT=$(bender path ara)
echo $ARA_ROOT
/home/users/user/ara/cheshire/.bender/git/checkouts/ara-2c7b103275a16c87
```

Código fuente 17: Localización del acelerador *hardware* Ara como dependencia de la rama `mp/ara-pulp-v2` de Cheshire.




(a) Directorios principales del repositorio.

(b) Ficheros principales del repositorio.

Figura 5.44: Estructura del repositorio `cheshire` (mp/ara-pulp-v2).

Una vez se ha compilado el *toolchain* de RISC-V es necesario tener especial cuidado para construir una variable `$PATH` que contenga únicamente los ejecutables que esperamos y no aquellos incluidos por otros programas ya instalados en el sistema como puede darse con el *toolchain* para RISC-V que incluye Vivado para el Microblaze-V.

Tras haber instalado las dependencias y tener disponible el *toolchain* se puede clonar el repositorio del Cheshire, en concreto la rama `mp/ara-pulp-v2` que incluye los tres componentes anteriores: el CVA6 conectado con el acelerador *hardware* Ara y todo esto sobre el SoC Cheshire, como se puede ver en la figura 5.43.

La estructura de directorios en la que se organiza el repositorio del Cheshire está compuesta por los directorios de la figura 5.44a y los archivos de la figura 5.44b. Siguiendo las instrucciones, se han compilado los *kernels* que se ejecutarán, es decir, las aplicaciones finales, en este caso el `${ARA_ROOT}/apps/dwt`. La compilación de los *kernels* se lleva a cabo mediante el comando mostrado en el código 18.

```
kernel=dwt
make ${kernel}-linux
```

Código fuente 18: Comando para compilar el *kernel* `dwt`.

El siguiente paso es generar la imagen de Linux y la imagen de Linux para Cheshire, en concreto, para el VCU128, que contendrá el sistema de archivos (`rootfs`) y el *kernel* antes compilado, mediante los comandos bash mostrados en el código 19.

```
cd ${ARA_ROOT}/cheshire/sw
make linux-img

BOARD=vcu128
make -C ${ARA_ROOT}/cheshire ara-chs-image BOARD=${BOARD}
```

Código fuente 19: Comando para compilar la imagen de Linux y la imagen de Linux para Cheshire.

Finalmente, para generar el *bitstream* utilizando Vivado para el sistema de prototipado VCU128 es necesario ejecutar los comandos detallados en el código 20.

```
make -C ${ARA_ROOT}/cheshire ara-chs-xilinx BOARD=${BOARD}
```

Código fuente 20: Generación del proyecto con Vivado y posterior *bitstream*.

De la finalización del proceso 20, resultan en directorio `target/xilinx/build`, los archivos del proyecto de Vidado, en este caso el proyecto `vcu128.cheshire/cheshire.xpr`. Utilizando dicho proyecto se ha efectuado el análisis final de la sección a continuación.

### 5.8.1. Prestaciones de la implementación de Ara

Según se puede ver en la figura 5.45, los recursos disponibles en el VCU128, permiten implementar el acelerador *hardware* Ara con 2 *lanes* y 4096 bits de VLEN a 100 MHz de frecuencia nominal del sistema[6]. Se ha alcanzado un uso del **25 %** Look-Up Tables (LUTs), **7 %** de ***flip flops***, **6 %** de Block RAMs (BRAMs) y tan solo **1 %** de DSPs.

En la figura 5.46 observa que la mayor parte del área está dedicada al propio acelerador aunque la DRAM y el CVA6 del mismo modo ocupan una parte significativa de los recursos utilizados. Poniendo el foco en el acelerador *hardware* Ara se puede ver cómo la mayoría del área está destinada a los *lanes* y por tanto han de ser considerados en el caso de que se estime aumentar el rendimiento incluyendo más canales de procesamiento en el sistema.

[6]El sistema contiene distintos relojes generados internamente para las comunicaciones con memoria DDR y otros dispositivos seriales.

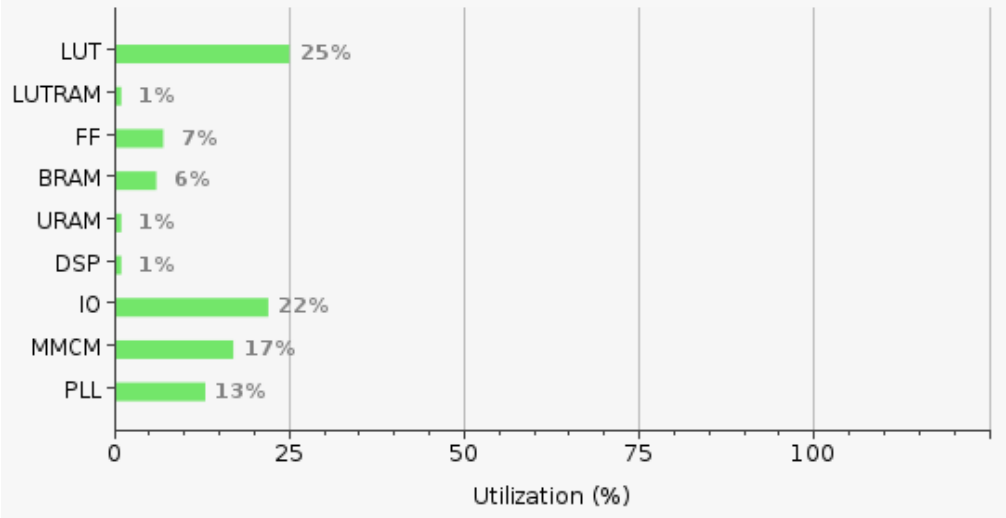



Figura 5.45: Porcentaje de utilización de recursos para VCU128.

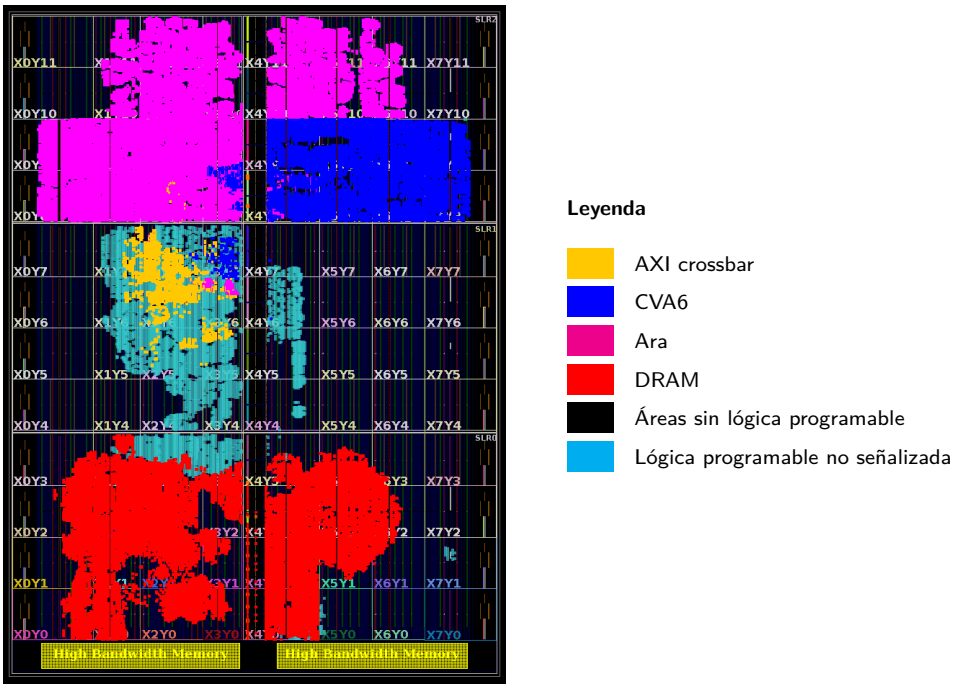



Figura 5.46: Distribución del uso de los recursos para VCU128 con los módulos más demandantes señalizados según la leyenda.

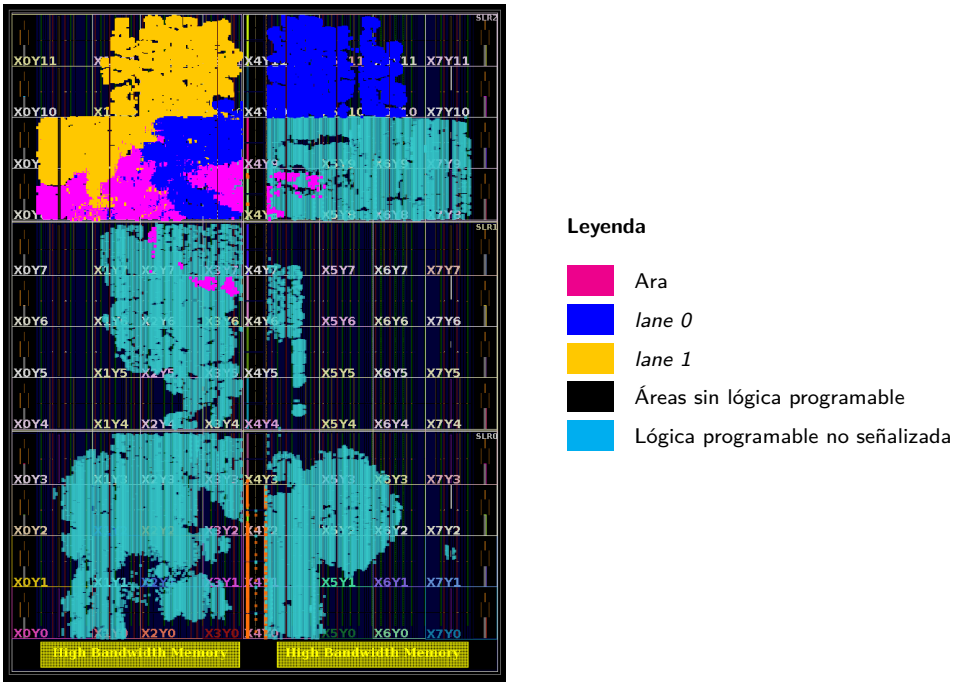



Figura 5.47: Distribución de los recursos utilizados por los *lanes* del acelerador *hardware* Ara.

**Setup**

| | |
|---|---|
| Worst Negative Slack (WNS): | 0.015 ns |
| Total Negative Slack (TNS): | 0.000 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 460055 |

**Hold**

| | |
|---|---|
| Worst Hold Slack (WHS): | 0.010 ns |
| Total Hold Slack (THS): | 0.000 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 456591 |

**Pulse Width**

| | |
|---|---|
| Worst Pulse Width Slack (WPWS): | 0.000 ns |
| Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 175263 |

Figura 5.48: Resumen de la temporización con los tiempos de *setup*, *hold* y *ancho de pulso* de la implementación.

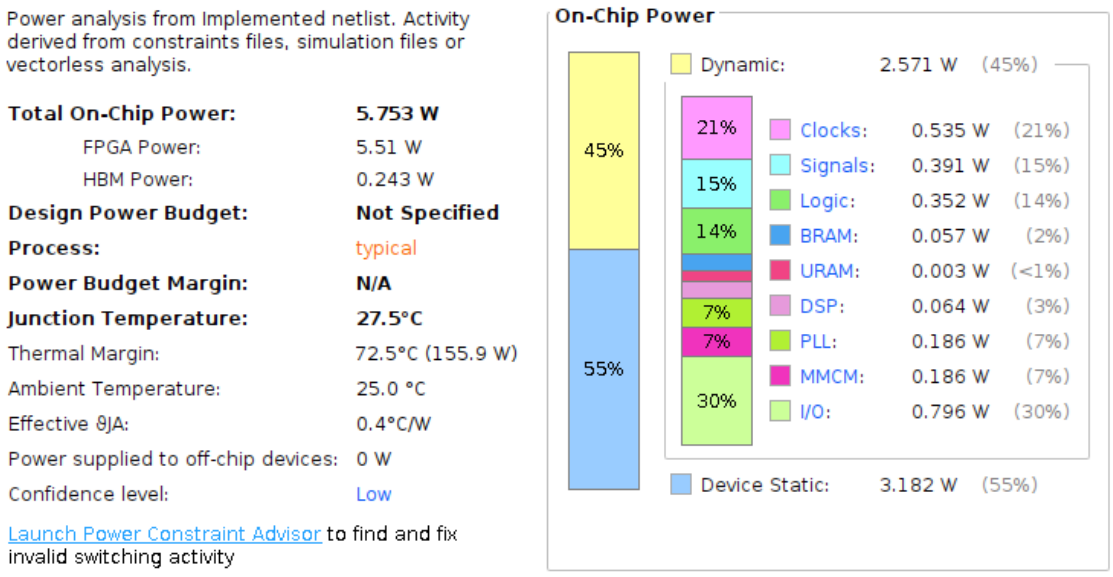



Figura 5.49: Distribución de potencia estática y dinámica, potencial total y reparto de la misma a través de los diferentes recursos del sistema.

## 5.9. Estudio de diseños ASIC basados en RISC-V

### 5.9.1. Estudio de trabajo previos

Como ruta hacia la implementación, ASIC se ha hecho un estudio de diferentes implementaciones existentes en línea con el trabajo realizado en este TFG. El estudio analiza distintas implementaciones de la especificación RISC-V RV64GC usando tecnologías ASIC.

En [49] se utiliza la tecnología CMOS 65nm de TSMC para la implementación de la arquitectura CVA6 en un demostrador que han denominado Neo, capaz de ejecutar Linux. El demostrador utiliza una tensión de alimentación de 1.2 V, alcanzando una frecuencia de funcionamiento de 325 MHz en condiciones típicas y consumiendo menos de 300 mW para una carga computacional intensiva. Para su implementación se ha hecho uso del entorno **Cheshire** descrito en este trabajo. Algunos aspectos a destacar de la implementación se resumen a continuación:

1. Se trata de una plataforma *host* que puede integrarse con aceleradores específicos *on-chip* o a nivel de *chiplets* en DSA, disponiendo por tanto de un bus AXI-4 y de una interfaz Die-to-Die (D2D).

2. La interconexión con memoria utiliza una interfaz con número de pines reducido (RPC-DRAM) de alta eficiencia energética, lo que reduce el número de E/S del dado.

3. La tecnología seleccionada utiliza 9 capas de metales. El proceso de síntesis se ha realizado en Synopsys Design Compiler [51] y la implementación física (*placement & routing*) se completó usando Cadence Innovus

[52]. La tecnología se caracterizó para una frecuencia de 200 MHz en las esquinas del proceso *Slow-Slow (SS)* (125ºC) y celdas con baja tensión umbral ($V_t$).

4. La arquitectura (figura 5.50) incluye, además del núcleo procesador CVA6, los bloques necesarios para crear una plataforma completa.

5. El chip obtenido (figura 5.51) tiene unas dimensiones de 3200 $\mu m^2$ X 2000 $\mu m^2$. Incluye las distintas unidades funcionales descritas en la arquitectura.

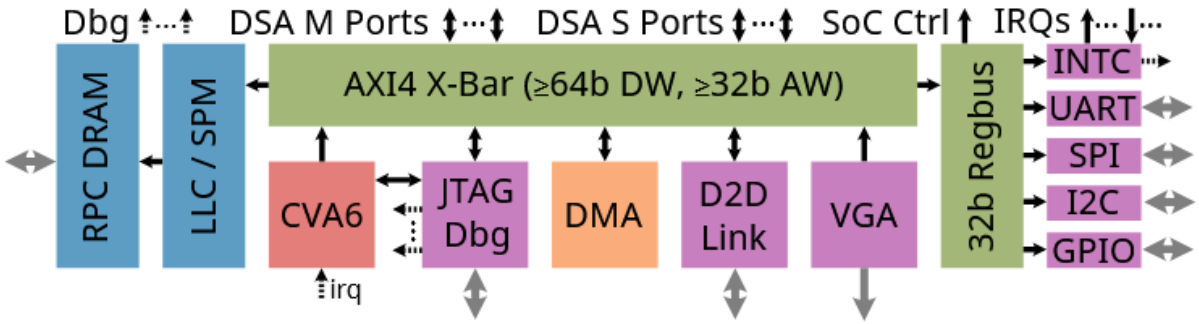



Figura 5.50: Arquitectura de Neo usando la plataforma Cheshire [49]

.

Por otra parte, en [32] se presenta un acelerador *hardware* específico DSA para el dominio de procesamiento vectorial, que cumple con la especificación *draft* 0.5 de las extensiones vectoriales de RISC-V. Está implementado en tecnologías de Global Foundries (GF) Complementary Metal-Oxide-Semiconductor (CMOS) de 22 nm (22FDX) Fully Depleted Silicon-on-Insulator (FD-SOI). La síntesis lógica se ha realizado en Synopsys Design Compiler y para la fase de implementación física se emplea Cadence Innovus. La tecnología está caracterizada con una área de 0,199 $\mu m^2$ por puerta.

Las restricciones impuestas al diseño incluye la posibilidad de trabajar con 2, 4, 8 o 16 bloques de procesamiento vectoriales o *lanes*, el ancho de memoria de 32 bits y se ha caracterizado para unas condiciones típicas de la tecnología (esquina TT, tensión de 0,80 Voltios y 25ºC). La frecuencia objetivo es de 1 GHz. La estrategia de diseño utilizada ha sido forzar un periodo de reloj inferior durante la fase de síntesis debido a los retardos introducidos durante el diseño físico. En concreto se ha utilizado un periodo de reloj de 250 ps, la cuarta parte del objetivo planteado de 1 ns.

Debido a la presencia de células estándar con distinto tipo de tensión umbral, es posible mezclar células de muy bajo consumo con otras de muy baja tensión umbral, consiguiendo aumentar la frecuencia de funcionamiento a costa de incrementar el consumo de potencia. En este diseño se han usado un 72,9 % de celdas de Low Voltage Threshold (LVT) y un 27,1 % de Super Low Voltage Threshold (SLVT).

El diseño con 4 *lanes* ocupa un área de 1,125$mm$ x 1,000$mm$ para la tecnología GF CMOS 22 FDX, lo que equivale aproximadamente al tamaño del procesador *host* (Ariadne), unas 524.000 puertas equivalentes. En cuanto a las prestaciones organizadas en términos de Performance, Power and Area (PPA) el diseño funciona a una frecuencia nominal de 1.25 GHz para condiciones típicas (TT/0.80 V/25 ºC) y 930 MHz para el caso más

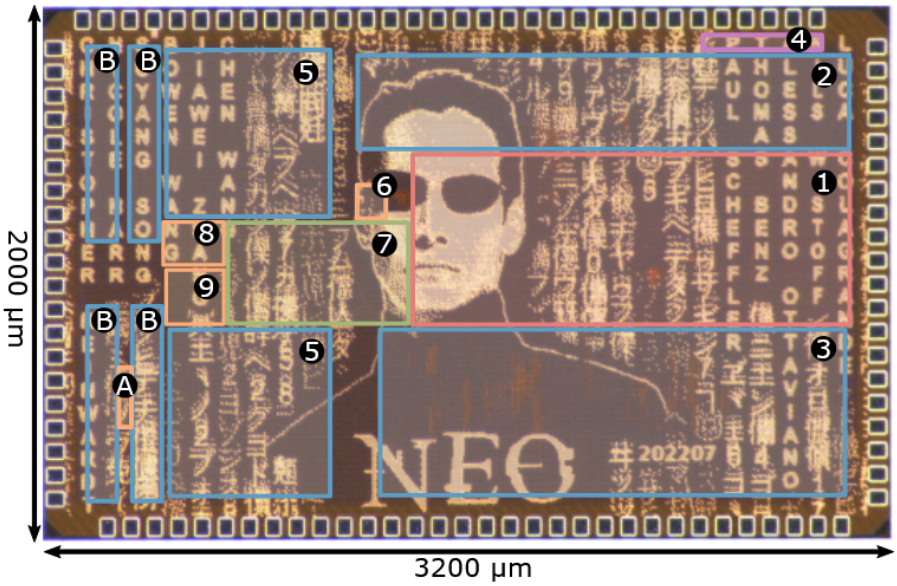



Figura 5.51: *Layout* final de Neo [49].

desfavorable. La potencia disipada por el *core* es de 259 mW (27 para el host RISC-V y 232 para las extensiones vectoriales), lo que implica una potencia por *lane* de 65 mW. El diseño es 2.5 veces más eficiente en términos de energía que si se ejecuta solamente el procesador RISC-V sin extensiones vectoriales (Ariadne).

Perotti et al. presentan en [53] la implementación de la versión 1.0 (*Frozen*) de las extensiones vectoriales de RISC-V (RISC-V Vector Extension (RVV)). Al igual que en [32] utilizan la tecnología GF CMOS 22FDX FD-SOI para implementar la arquitectura que se muestra en la figura 5.52. El *layout* del circuito se muestra en la figura 5.53.

El diseño se ha implementado usando las herramientas del entorno de Synopsys, tanto para la síntesis lógica (Synopsys Design Compiler) como la la síntesis física (Synopsys IC Compiler II). La simulación post-síntesis se ha realizado con Siemens QuestaSim y los procesos de análisis temporal estático y análisis de potencia con Synopsys PrimeTime.

Los resultados obtenidos de la implementación física, en términos de PPA, son los siguientes:

- Frecuencia: 920 MHz para el peor caso, y 1.34 GHz para condiciones típicas.
- Área, incluyendo las memorias cache de primer nivel: 0,81 $mm^2$
- Potencia disipada ejecutando el programa de pruebas `fmatmul`: 280 mW

El desarrollo de la arquitectura NewARA se evoluciona con las mejoras introducidas en ARA2 [21], en la que se evalúa una nueva propuesta de implementación de RVV 1.0 incluyendo hasta 16 *lanes*. La arquitectura ARA2

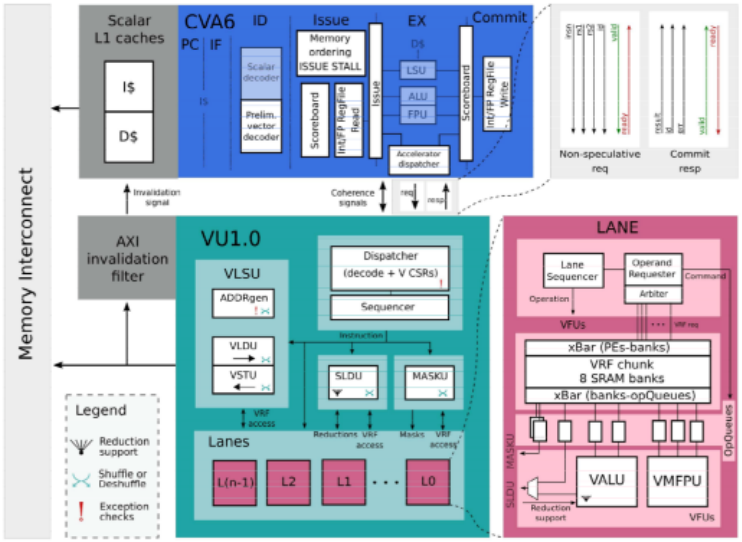



Figura 5.52: Diagrama de bloques de la arquitectura New Ara


Figura 5.53: *Layout* de New Ara

ha sido implementada en un ASIC utilizando la tecnología GF CMOS 22FDX FD-SOI, alcanzando una frecuencia de 1.08 GHz para unas condiciones típicas de la tecnología (TT/0.80 V/25 ºC). El área utilizada es de 4,47$mm^2$, lo que implica 14773K puertas equivalentes.

Recientemente se ha publicado una mejora significativa de la arquitectura denominada ARA-XL [54] que permite la inclusión de hasta 64 *lanes*. La arquitectura se estructura como un *cluster* de unidades de procesamiento vectorial ARA2 modificadas, que se interconectan, entre ellos y con memoria L2, utilizando las siguientes interfaces:

- Request Interface (REQUI)
- Global Load Store Unit (GLSU)
- Ring Interface (RINGI)

La nueva arquitectura permite una reducción del 14 % del área ocupada mientra mantiene los requerimientos de frecuencia máxima de 1.15 GHz (64 *Lanes*). En la figura 5.54 se muestra el *floorplan* de la arquitectura de ARAXL.

### 5.9.2. Trabajos en desarrollo para la implementación ASIC

Teniendo en cuenta los trabajos descritos, se ha abordado la implementación ASIC utilizando una tecnología de referencia de TSMC CMOS de 65nm. En una primera exploración de la arquitectura se ha realizado la síntesis lógica del diseño, incluyendo los bloques del SoC, en el entorno Design Vision de Synopsys. Para ello se ha seguido el flujo de síntesis mostrado en la figura 5.55 que se incluye en distintos scripts programados en lenguaje Tcl.

En la figura 5.56 se muestra el entono de diseño Design Vision, incluyendo la jerarquía del diseño y el análisis temporal realizado. En dicho análisis se muestra el *Path histogram* con la ruta crítica del diseño, mostrando que cumple las restricciones temporales para una frecuencia de 200 MHz. En la figura 5.57 (a) se indica el área ocupada y en la figura 5.57 (b) se muestra la potencia disipada en el diseño.

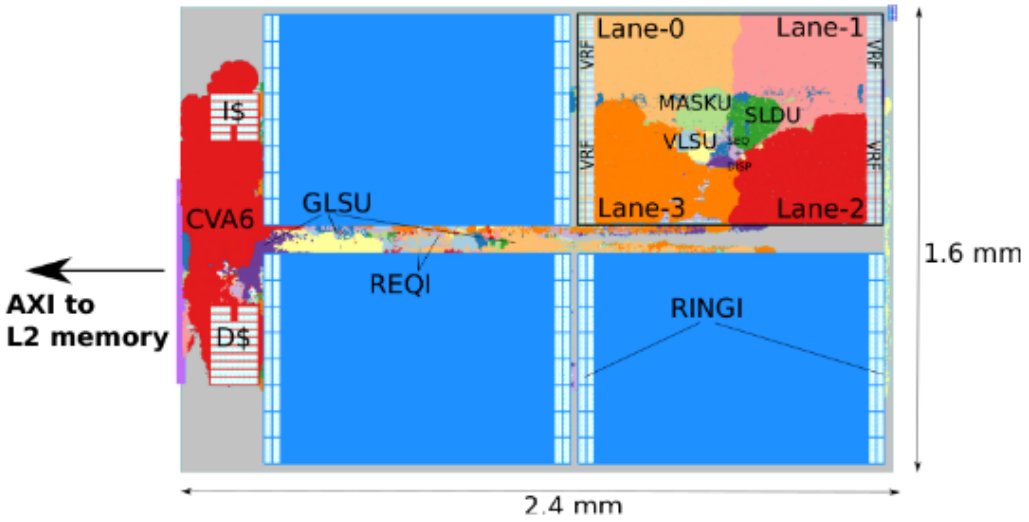



Figura 5.54: *Floorplan* para una implementación de ARAXL con 16 *lanes*. Cada *cluster* incluye 4 *lanes*, con conexiones A2A.

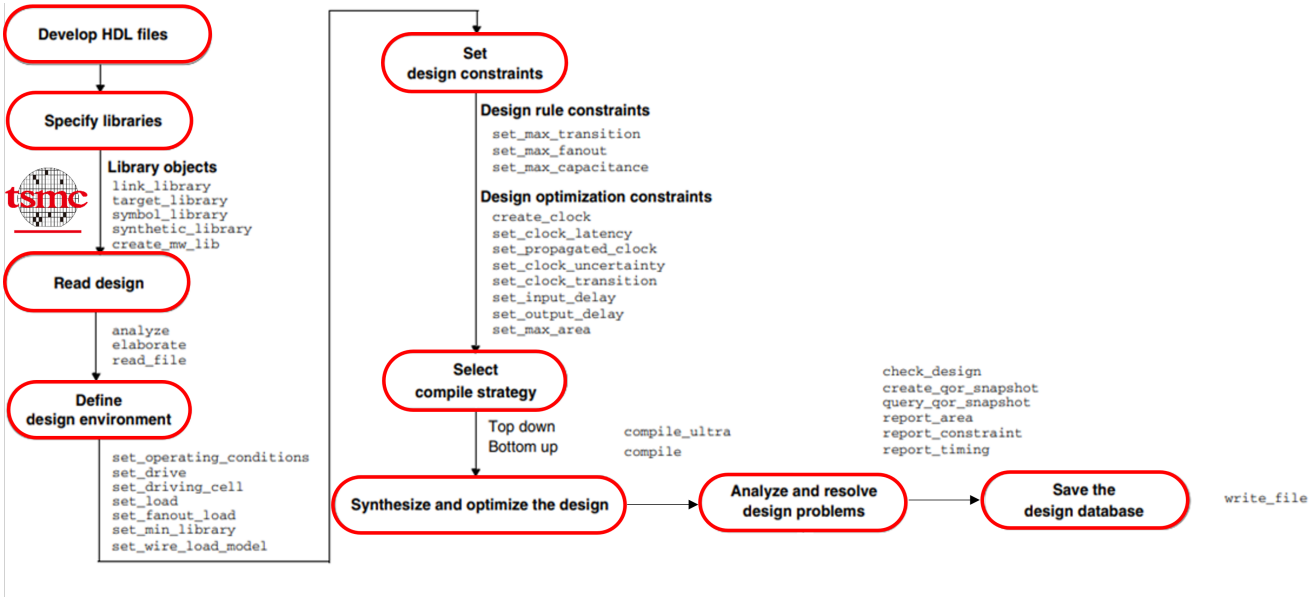



Figura 5.55: Flujo de síntesis con Synopsys Design Vision para el procesador CVA6

La síntesis realizada utiliza 428000 puertas equivalentes. Se ha considerado como puerta equivalente una NAND2 básica con un área de 1,440$\mu m^2$ (0,8$\mu m$ de ancho x 1,8$\mu m$ de alto). La frecuencia utilizada en esta implementación es de 200 MHz.

Para obtener unas prestaciones dentro del estado del arte, se está trabajando hacia una implementación en tecnología TSMC FinFET de 16nm. En diseños preliminares se ha conseguido realizar la implementación de un

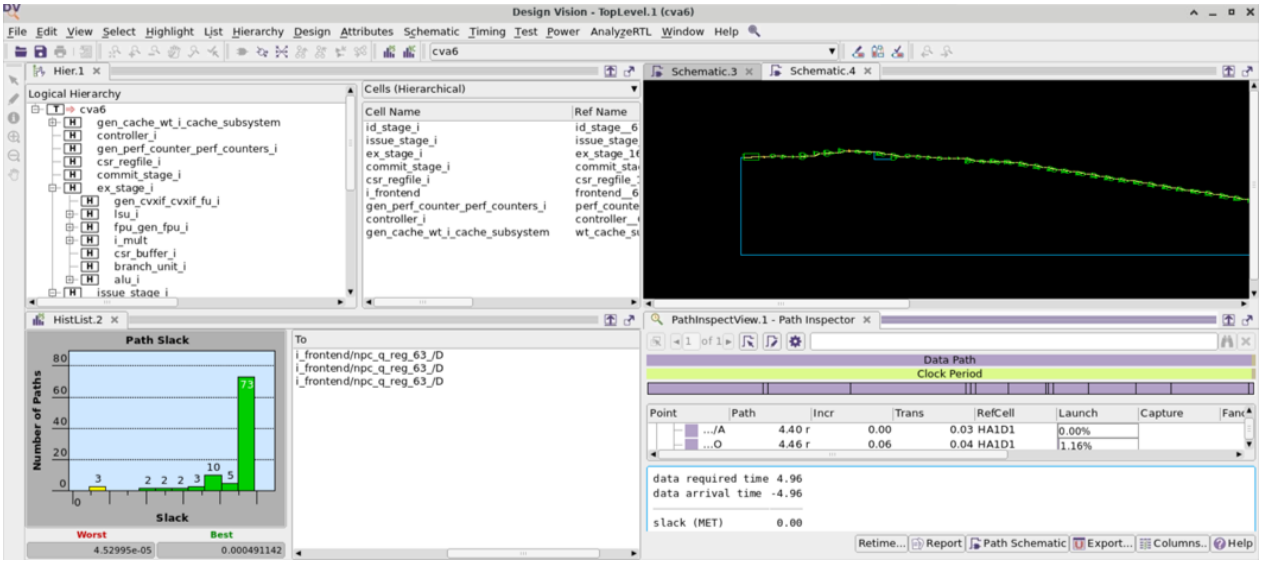



Figura 5.56: Entorno de síntesis Synopsys Design Vision

```
Number of ports:                       105871
Number of nets:                        266975
Number of cells:                       193677
Number of combinational cells:         167084
Number of sequential cells:             26246
Number of buf/inv:                      28117
Number of references:                      35

Combinational area:            402469.564077
Noncombinational area:         214207.202278
Total cell area:               616676.766355
```

(a)

```
Global Operating Voltage = 1.2
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units
    Leakage Power Units = 1nW


  Cell Internal Power  =  50.8611 mW   (99%)
  Net Switching Power  = 762.8559 uW    (1%)
                         ---------
Total Dynamic Power    =  51.6239 mW  (100%)

Cell Leakage Power     =  18.7646 uW
```

(b)

Figura 5.57: Informe de prestaciones de la síntesis realizada para la versión ASIC de CVA6: (a) área, (b) potencia

sistema multiprocesador RISC-V (3 cores y un PLL), con una frecuencia nominal de 1 GHz. El diseño sigue bajo optimización.

Como conclusión se puede indicar que la mayoría de los aceleradores *hardware* estudiados utilizan tecnologías de CMOS avanzadas (22 nm) para conseguir prestaciones adaptadas a la carga de trabajo. Los entornos de diseño utilizados se centran en Synopsys Design Compiler, Synopsys Design Fusion Compiler (versión avanzada de Synopsys IC Compiler II), Candence Innovus y Siemens QuestaSim. Por ello, la aproximación de usar tecnología FinFET de 16 nm supone una apuesta competitiva con respecto al estado del arte.

## 5.10. Conclusiones

En este capítulo se han presentado diversas implementaciones RISC-V y se concluye con la solución elegida tras haber estudiado tanto implementaciones independientes de RISC-V como aceleradores completos.

Se han sintetizado distintos procesadores RISC-V independientes, tales como NEORV32, VexRiscv y CVA6) y se realiza el prototipado durante la fase de exploración, para comprobar el funcionamiento básico de cada caso, incluyendo en la valoración aspectos tales como la calidad de la documentación y facilidad de integración.

A continuación se justifica la utilización del acelerador final seleccionado para la realización del trabajo y se concluye con el flujo adoptado para la implementación del acelerador sobre la placa de prototipado VCU128. El capítulo concluye con un estudio de distintas implementaciones ASIC realizadas a partir de aceleradores vectoriales, obteniendo recomendaciones claves para seguir con la ruta hacia el ASIC.

# Capítulo 6

# Conclusiones y trabajos futuros

> Cada solución da pie a una nueva pregunta.
> *David Hume.*

## 6.1. Conclusiones

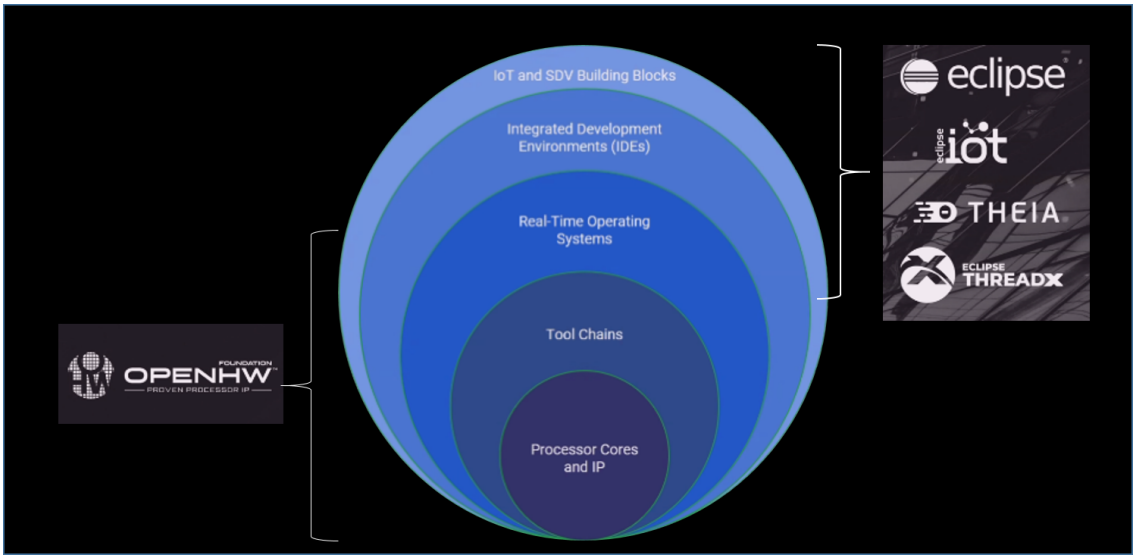
El presente trabajo fin de grado se ha centrado en el estudio de la arquitectura RISC-V y de las distintas posibilidades para la inclusión de extensiones que dan soporte a aceleradores *hardware* para dominios específicos DSA. En este trabajo se han considerado el *stack* requerido para poder abordar soluciones basadas en RISC-V con éxito (figura 6.1).

El estudio inicial realizado de la ISA RISC-V ha permitido conocer a fondo el formato de instrucciones y su regularidad, los registros definidos en la ISA (tanto desde el punto de vista *hardware* como de la especificación ABI) y su funcionalidad. Este análisis detallado del formato de instrucciones permite entender a posteriori el proceso de decodificación de la instrucciones en la descripción RTL, ya sea en VHDL o en SystemVerilog.

Como RISC-V se ha diseñado como una ISA flexible y extendible, se han analizado las capacidades de extensión de la ISA para dar soporte a nuevas instrucciones, ya sea del conjunto estandarizado por RISC-V International, como de otras extensiones *custom*.

Como resultado del estudio de la arquitectura, también se aborda el estudio del ecosistema *software*, conocido como *RISC-V toolchain*, que permite compilar los programas desde C/C++ hacia código máquina, realizar su ensamblado y su emulación en herramientas dedicadas como `spike`, o QEMU, depurando el código en el depurador GDB. Se ha realizado un esfuerzo importante en poner en marcha el protocolo OpenOCD para facilitar el depurado de aplicaciones en el dispositivo *target*.

El análisis del conjunto de extensiones existentes, aceptadas por RISC-V International (estado *frozen*) o en fase de desarrollo, da una idea clara del tipo de interés de la comunidad internacional en las funcionalidades necesarias reales que se pueden incluir en las arquitecturas avanzadas. Igualmente permite al diseñador discernir si es necesario su inclusión o no en el procesador dedicado. La consecuencia inmediata es la reducción de la




Fuente: Adaptado de https://www.youtube.com/watch?v=azIk6dGWrOU

Figura 6.1: *Stack* completo para el desarrollo *hardware/software* basado en RISC-V

utilización de recursos (FPGA) o del área ocupada (ASIC) y la reducción en el consumo de potencia, manteniendo las prestaciones necesarias.

Con objeto de comprender, desde el punto de vista arquitectural, las distintas posibilidades de integrar un acelerador *hardware* en el sistema, se han estudiado distintas alternativas, ya sea utilizando interfaces de buses estandarizadas, como por ejemplo AXI4, o por el contrario, ampliando la funcionalidad del sistema en las propias unidades funcionales preexistentes. Durante el desarrollo del trabajo se ha experimentado con distintos esquemas de interconexión en función de la necesidad de datos del acelerador, e incluyendo o no el acceso a los datos a través de las interfaces con memoria cache.

Para experimentar con los conceptos explicados se ha realizado el prototipado de diferentes procesadores que incluyen algún tipo de acelerador *hardware* de los citados con anterioridad. Previamente se han estudiado y clasificado en distintos aceleradores completos, analizado aspectos esenciales del diseño *hardware*, de su integración con el procesador RISC-V que actúa como *host* y de los aspectos del *stack software* y facilita la compilación y ejecución de las aplicaciones.

Tras haber ganado los conocimientos necesarios para conocer en detalle las capacidades de las plataformas probadas, se ha elegido la plataforma *hardware* Ara como referencia para la implementación de un procesador con extensiones vectoriales. Para ello se ha hecho uso del entorno Cheshire que facilita la integración del procesador CVA6 y de los distintos *lanes* de procesamiento vectorial. Finalmente todo el sistema se ha implementado en la plataforma AMD/Xilinx VCU128, obtenido un a ocupación del 25 % de LUTs, con una disipación de potencia de 5.75 W, funcionando a una frecuencia de 100 MHz para el reloj del sistema (la plataforma contiene múltiples relojes generados en la propia FPGA).

La plataforma FPGA representa un paso intermedio en el camino hacia la implementación ASIC. Por ese motivo se ha incluido un apartado específico del estudio de distintas implementaciones existentes en la literatura de la plataforma Ara. Se han incluido los datos experimentales obtenidos de la síntesis lógica del procesador CVA6 para

una tecnología madura TSMC CMOS de 65 nm y la previsión del camino hacia una implementación en tecnología TSMC CMOS FinFET de 16nm.

Se puede concluir que este trabajo ha contribuido de forma significativa a la compresión del ecosistema RISC-V, a establecer la metodología necesaria para abordar un proyecto de cierta complejidad usando RISC-V y plantea nuevos retos en el camino a difundir esta tecnología de referencia en el contexto europeo.

## 6.2. Trabajos futuros

Una vez completado el trabajo realizado, se han identificado distintos trabajos futuros a considerar:

1. Evaluar un algoritmo complejo sobre la arquitectura descrita. Se está valorando su utilización en el proyecto OASIS del IUMA, que tiene por objeto medir la deformación del cerebro con tiempos de respuesta aceptables (minutos).

2. Si bien se ha hecho la implementación de Ara en FPGA, será necesario realizar su prototipado sobre una placa FPGA avanzada que disponga de los recursos suficientes para implementar un sistema básico (2 *lanes*), como la utilizada de referencia en la implementación. Se estudiará la viabilidad para utilizar sistemas de prototipado basados *on-premise*, como por ejemplo los sistema Alveo U200 o la tarjeta VCK5000 Versal, ambas disponibles en el IUMA.

3. Avanzar en la implementación usando tecnologías ASIC avanzada, tal como la de TSMC FinFET de 16 nm citada en el documento. Ello requerirá un estudio detallado de la arquitectura de memoria y de la metodología de diseño sobre Cadence Genus e Innovus [55] para tecnologías nanométricas. También es de interés evaluar los resultados en términos de PPA de forma comparativa con el entorno Fusion Compiler de Synopsys.

4. Una vez que se conoce la arquitectura de referencia utilizada en este trabajo, estudiar otras posibles arquitecturas de código abierto de aceleradores, tanto vectoriales, como procesamiento de matrices y de tensores, que permitan realizar una comparación detallada de las prestaciones y su viabilidad de utilización para aplicaciones complejas.

# BIBLIOGRAFÍA


[1] «AMD Virtex™ UltraScale+™ HBM VCU128 FPGA Evaluation Kit,» AMD. (s.f.), dirección: https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/vcu128.html (visitado 17-07-2025).

[2] A. Raveendran, V. B. Patil, D. Selvakumar y V. Desalphine, «A RISC-V Instruction Set Processor-Micro-Architecture Design and Analysis,» en *2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*, 2016, págs. 1-7. DOI: 10.1109/VLSI-SATA.2016.7593047.

[3] A. Kelleher, *Ley de Moore – Ahora y En El Futuro*, feb. de 2022. dirección: https://www.intel.la/content/www/xl/es/newsroom/opinion/moore-law-now-and-in-the-future.html (visitado 05-02-2025).

[4] H. Wong, «On the CMOS Device Downsizing, More Moore, More than Moore, and More-than-Moore for More Moore,» en *2021 IEEE 32nd International Conference on Microelectronics (MIEL)*, 2021, págs. 9-15. DOI: 10.1109/MIEL52794.2021.9569101.

[5] RISC-V International. «RISC-V Technical Specifications. ISA Specifications.» (s.f.), dirección: https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications.

[6] T. Fritzmann, G. Sigl y J. Sepúlveda, *RISQ-V: Tightly Coupled RISC-v Accelerators for Post-Quantum Cryptography*, Cryptology ePrint Archive, Paper 2020/446, 2020. DOI: 10.13154/tches.v2020.i4.239-280. dirección: https://eprint.iacr.org/2020/446.

[7] RISC-V Foundation. «About RISC-V International.» (5 de feb. de 2025), dirección: https://riscv.org/about/.

[8] D. A. Patterson y A. Waterman, *The RISC-V Reader. An Open Architecture Atlas*. Strawberry Canyon LLC, 1 de ene. de 2017. dirección: http://riscvbook.com/.

[9] P. Sane, «Navigating ARM-based Application Adoption: Software Engineer's Insights on Challenges & Solutions,» en *2024 International Conference on Wireless Communications Signal Processing and Networking (WiSPNET)*, 2024, págs. 1-7. DOI: 10.1109/WiSPNET61464.2024.10533089.

[10] A. Shilov, «RISC-V in AI and HPC Part 1: Per Aspera Ad Astra?,» 6 de ene. de 2025. dirección: https://www.eetimes.com/risc-v-in-ai-and-hpc-part-1-per-aspera-ad-astra/.

[11] N. T. Binh, B. Kieu-Do, T.-T. Hoang, P. Cong-Kha y C. Pham-Quoc, «FPGA-based Secured and Efficient Lightweight IoT Edge Devices with Customized RISC-V,» en *2023 RIVF International Conference on Computing and Communication Technologies (RIVF)*, 2023, págs. 31-36. DOI: 10.1109/RIVF60135.2023.10471777.

[12] *Next Gen MTIA - Meta's Recommendation Inference Accelerator*, 27 de ago. de 2024. dirección: https://hc2024.hotchips.org/assets/program/conference/day2/82_HC2024.Meta.MadduryKansal.final.pdf.

[13] C. Mead y L. Conway, «Algorithms for VLSI Processor Arrays,» en *Introduction to VLSI Systems*, USA: Addison-Wesley Longman Publishing Co., Inc., 1979, ISBN: 0-201-04358-0.

[14] «OpenHW Group | OpenHW Foundation.» (s.f.), dirección: https://openhwfoundation.org/ (visitado 12-06-2025).

[15] B. Bailey. «RISC-V's Increasing Influence,» Semiconductor Engineering. (12 de jun. de 2025), dirección: https://semiengineering.com/risc-vs-increasing-influence/ (visitado 17-07-2025).

[16] RISC-V International, «The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture,» 8 de mayo de 2025. dirección: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view.

[17] S. L. Harris y D. M. Harris, *Digital Design and Computer Architecture*, RISC-V edition. Cambridge: Morgan Kaufmann, 2022, 733 págs., ISBN: 978-0-12-820064-3.

[18] RISC-V International, «The RISC-V Instruction Set Manual: Volume II: Privileged Architecture,» 8 de mayo de 2025. dirección: https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view.

[19] «RISC-V Software Ecosystem - Home - RISC-V Tech Hub.» (23 de abr. de 2024), dirección: https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154661/RISC-V+Software+Ecosystem (visitado 11-06-2025).

[20] J.-H. Li, J.-K. Lin, Y.-C. Su, C.-W. Chu, L.-T. Kuok, H.-M. Lai, C.-L. Lee y J.-K. Lee. «SIMD Everywhere Optimization from ARM NEON to RISC-V Vector Extensions.» arXiv: 2309.16509 [cs]. (28 de sep. de 2023), dirección: http://arxiv.org/abs/2309.16509 (visitado 23-05-2025), prepublicado.

[21] M. Perotti, M. Cavalcante, R. Andri, L. Cavigelli y L. Benini, «Ara2: Exploring Single- and Multi-Core Vector Processing With an Efficient RVV 1.0 Compliant Open-Source Processor,» *IEEE Transactions on Computers*, vol. 73, n.º 7, págs. 1822-1836, jul. de 2024, ISSN: 0018-9340, 1557-9956, 2326-3814. DOI: 10.1109/TC.2024.3388896. dirección: https://ieeexplore.ieee.org/document/10500752/ (visitado 07-05-2025).

[22] E. Cui, T. Li y Q. Wei, «RISC-V Instruction Set Architecture Extensions: A Survey,» *IEEE Access*, vol. 11, págs. 24 696-24 711, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3246491. dirección: https://ieeexplore.ieee.org/document/10049118 (visitado 26-05-2025).

[23] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu y L. Benini, «Design and Evaluation of SmallFloat SIMD Extensions to the RISC-V ISA,» en *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, mar. de 2019, págs. 654-657. DOI: 10.23919/DATE.2019.8714897. dirección: https://ieeexplore.ieee.org/document/8714897 (visitado 31-05-2025).

[24] H. B. Amor, C. Bernier y Z. Přikryl, «A RISC-V ISA Extension for Ultra-Low Power IoT Wireless Signal Processing,» *IEEE Transactions on Computers*, vol. 71, n.º 4, págs. 766-778, abr. de 2022, ISSN: 1557-9956. DOI: 10.1109/TC.2021.3063027. dirección: https://ieeexplore.ieee.org/document/9366907 (visitado 31-05-2025).

[25] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak y L. Benini, «Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices,» *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, n.º 10, págs. 2700-2713, oct. de 2017, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2017.2654506. dirección: https://ieeexplore.ieee.org/document/7864441 (visitado 31-05-2025).

[26] S. Amar, D. Chisnall, T. Chen, N. W. Filardo, B. Laurie, K. Liu, R. Norton, S. W. Moore, Y. Tao, R. N. M. Watson y H. Xia, «CHERIoT: Complete Memory Safety for Embedded Devices,» en *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ép. MICRO '23, New York, NY, USA: Association for Computing Machinery, 8 de dic. de 2023, págs. 641-653, ISBN: 979-8-4007-0329-4. DOI: 10.1145/3613424.3614266. dirección: https://dl.acm.org/doi/10.1145/3613424.3614266 (visitado 31-05-2025).

[27] «Stream Computing RISC-V Matrix Extension Open Source Project Upgrades to Version 0.5, Supporting Vector+Matrix Implementation – RISC-V International.» (7 de oct. de 2024), dirección: https://riscv.org/blog/2024/11/stream-computing-risc-v-matrix-extension-open-source-project-upgrades-to-version-0-5-supporting-vectormatrix-implementation/ (visitado 16-07-2025).

[28] «Extending RISC-V ISA With a Custom Instruction Set Extension.» (3 de mayo de 2019), dirección: https://www.design-reuse.com/article/61135-extending-risc-v-isa-with-a-custom-instruction-set-extension/ (visitado 16-07-2025).

[29] B. Peccerillo, M. Mannino, A. Mondelli y S. Bartolini, «A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives,» *Journal of Systems Architecture*, vol. 129, pág. 102 561, ago. de 2022, ISSN: 13837621. DOI: 10.1016/j.sysarc.2022.102561. dirección: https://linkinghub.elsevier.com/retrieve/pii/S1383762122001138 (visitado 29-04-2025).

[30] core-v-xif contributors, *Openhwgroup/Core-v-Xif*, OpenHW Group, 31 de mayo de 2025. dirección: https://github.com/openhwgroup/core-v-xif (visitado 07-06-2025).

[31] F. Conti, G. Paulin, A. Garofalo, D. Rossi, A. D. Mauro, G. Rutishauser, G. Ottavi, M. Eggimann, H. Okuhara y L. Benini, «Marsellus: A Heterogeneous RISC-V AI-IoT End-Node SoC with 2-to-8b DNN Acceleration and 30 %-Boost Adaptive Body Biasing,» *IEEE Journal of Solid-State Circuits*, vol. 59, n.º 1, págs. 128-142, ene. de 2024, ISSN: 0018-9200, 1558-173X. DOI: 10.1109/JSSC.2023.3318301. arXiv: 2305.08415 [cs]. dirección: http://arxiv.org/abs/2305.08415 (visitado 18-06-2025).

[32] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner y L. Benini, «Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-Nm FD-SOI,» *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, n.º 2, págs. 530-543, feb. de 2020, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2950087. dirección: https://ieeexplore.ieee.org/document/8918510/ (visitado 23-04-2025).

[33] F. Zaruba y L. Benini, «The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-Nm FDSOI Technology,» *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, n.º 11, págs. 2629-2640, nov. de 2019, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114. dirección: https://ieeexplore.ieee.org/document/8777130 (visitado 19-06-2025).

[34] ztachip, *Ztachip/Ztachip*, 3 de jun. de 2025. dirección: https://github.com/ztachip/ztachip (visitado 03-06-2025).

[35] T. Developers, *TensorFlow*, ver. v2.19.0, Zenodo, 12 de mar. de 2025. DOI: 10.5281/ZENODO.4724125. dirección: https://zenodo.org/doi/10.5281/zenodo.4724125 (visitado 04-06-2025).

[36] «SpinalHDL/VexRiscv: A FPGA Friendly 32 Bit RISC-V CPU Implementation.» (s.f.), dirección: https://github.com/SpinalHDL/VexRiscv (visitado 20-06-2025).

[37] S. Nolting y A. T. A. Contributors, *The NEORV32 RISC-V Processor*, ver. v1.11.3, Zenodo, 21 de abr. de 2025. DOI: 10.5281/ZENODO.5018888. dirección: https://zenodo.org/doi/10.5281/zenodo.5018888 (visitado 28-04-2025).

[38] «Nexys 4 DDR (Legacy) - Digilent Reference.» (s.f.), dirección: https://digilent.com/reference/programmable-logic/nexys-4-ddr/start (visitado 17-07-2025).

[39] «AMD Vivado™ Design Suite,» AMD. (s.f.), dirección: https://www.amd.com/es/products/software/adaptive-socs-and-fpgas/vivado.html (visitado 17-07-2025).

[40] M. H. AlMeer, «FPGA Implementation of a Hardware XTEA Light Encryption Engine in Co-Design Computing Systems,» en *2017 Seventh International Conference on Innovative Computing Technology (INTECH)*, ago. de 2017, págs. 26-30. DOI: 10.1109/INTECH.2017.8102419. dirección: https://ieeexplore.ieee.org/document/8102419 (visitado 20-06-2025).

[41] P. Israsena, «On XTEA-based Encryption/Authentication Core for Wireless Pervasive Communication,» en *2006 International Symposium on Communications and Information Technologies*, oct. de 2006, págs. 59-62. DOI: 10.1109/ISCIT.2006.339887. dirección: https://ieeexplore.ieee.org/document/4141513 (visitado 20-06-2025).

[42] C. Papon e Y. Xiao, *SpinalHDL*, 18 de jun. de 2025. dirección: https://github.com/SpinalHDL/SpinalHDL (visitado 20-06-2025).

[43] PULP Platform. «PULP Project Information.» (s.f.), dirección: https://pulp-platform.org/projectinfo.html (visitado 12-06-2025).

[44] «Genesys 2 Reference Manual - Digilent Reference.» (s.f.), dirección: https://digilent.com/reference/programmable-logic/genesys-2/reference-manual (visitado 17-07-2025).

[45] OpenSBI contributors, *Riscv-Software-Src/Opensbi*, RISC-V Software, 7 de jul. de 2025. dirección: https://github.com/riscv-software-src/opensbi (visitado 07-07-2025).

[46] C. Allart, J.-R. Coulon, A. Sintzoff, O. Potin y J.-B. Rigaud. «Using a Performance Model to Implement a Superscalar CVA6.» arXiv: 2410.01442 [cs]. (3 de oct. de 2024), dirección: http://arxiv.org/abs/2410.01442 (visitado 20-06-2025), prepublicado.

[47] R. F. Ibáñez, «Introduction to the RISC-V Vector Extension,» dirección: https://eupilot.eu/wp-content/uploads/2022/11/RISC-V-VectorExtension-1-1.pdf.

[48] «V-Intrinsic-Spec.Pdf,» Google Docs. (25 de abr. de 2025), dirección: https://drive.google.com/file/d/1RTZi2iOLKzqaX95JCCnzwOm7iCIN3JEq/view?usp=drive_link&usp=embed_facebook (visitado 02-07-2025).

[49] A. Ottaviano, T. Benz, P. Scheffler y L. Benini. «Cheshire: A Lightweight, Linux-Capable RISC-V Host Platform for Domain-Specific Accelerator Plug-In.» arXiv: 2305.04760 [cs]. (6 de jul. de 2023), dirección: http://arxiv.org/abs/2305.04760 (visitado 23-04-2025), prepublicado.

[50] *Riscv-Collab/Riscv-Gnu-Toolchain*, RISC-V Collaboration, 10 de jul. de 2025. dirección: https://github.com/riscv-collab/riscv-gnu-toolchain (visitado 10-07-2025).

[51] «Design Compiler: Timing, Area, Power, & Test Optimization | Synopsys.» (s.f.), dirección: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html (visitado 17-07-2025).

[52] «Innovus Implementation System.» (s.f.), dirección: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html (visitado 17-07-2025).

[53] M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli y L. Benini, «A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design,» en *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, jul. de 2022, págs. 43-51. DOI: 10.1109/ASAP54787.2022.00017. dirección: https://ieeexplore.ieee.org/document/9912071/ (visitado 23-04-2025).

[54] N. K. Purayil, M. Perotti, T. Fischer y L. Benini. «AraXL: A Physically Scalable, Ultra-Wide RISC-V Vector Processor Design for Fast and Efficient Computation on Long Vectors.» arXiv: 2501.10301 [cs]. (17 de ene. de 2025), dirección: http://arxiv.org/abs/2501.10301 (visitado 23-04-2025), prepublicado.

[55] «Genus Synthesis Solution.» (s.f.), dirección: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html (visitado 17-07-2025).

[56] «Continuous Wavelet Transform (CWT) — PyWavelets Documentation.» (s.f.), dirección: https://pywavelets.readthedocs.io/en/latest/ref/cwt.html (visitado 29-06-2025).

# Pliego de condiciones

## 1. Condiciones Administrativas

De acuerdo con la norma UNE 157001-2014: Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico: "El pliego de condiciones tiene como misión establecer las condiciones técnicas, económicas, administrativas, facultativas y legales para que el objeto del Proyecto pueda materializarse en las condiciones especificadas, evitando posibles interpretaciones diferentes de las deseadas. Su contenido y extensión queda a criterio de su autor y en función del tipo de Proyecto."

### 1.1. Normativa de aplicación

Al tratarse éste de un proyecto de diseño, en el marco de un Trabajo de Fin de Grado universitario, la normativa de aplicación es la relativa a la redacción del proyecto:

- Orden CIN/352/2009 de 9 de febrero de 2009, por la que se establecen los requisitos para la verificación de los títulos universitarios oficiales que habiliten para el ejercicio de la profesión de Ingeniero Técnico de Telecomunicación.
- Proyecto Docente de Trabajo de Fin de Grado de la EITE para el curso 2024/2025.
- UNE 157001:2014 - Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico, junio de 2014.

## 2. Condiciones técnicas generales

Este TFG realiza el estudio detallado de las arquitecturas RISC-V, tratando de adquirir el conocimiento de la tecnología necesaria para el desarrollo de un coprocesador *hardware* para el procesamiento vectorial, para ser aplicado a problemas complejos, como puede ser el procesamiento de algoritmos para Inteligencia Artificial. Por ese motivo es necesario abordar una serie de tecnologías, metodologías de diseño de amplio espectro, apoyándose en distintas herramientas de diseño electrónico (EDA) y en herramientas de compilación. Muchas de estas herramientas son de dominio público, por lo que se atenderá a las licencias de utilización de cada uno de los paquetes, mientras que otras son de uso comercial, que se utilizan al estar enmarcado el trabajo dentro de las actividades del IUMA. El resultado del trabajo realizado estará disponible para su utilización bajo las mismas licencias que el paquete de origen.

# 3. Condiciones técnicas particulares

Para la realización de este Trabajo Fin de Grado, se han utilizado los siguientes recursos *hardware* y *software*. Se asegura el correcto funcionamiento del sistema utilizando las versiones de las herramientas y aplicaciones citadas en el documento.

## 3.1. Requisitos a cumplir de los equipos y sistemas *hardware*

A continuación se especifican las condiciones bajo las que se han llevado a cabo los trabajos y estudios realizados en este TFG.

**Estaciones de trabajo y servidores de cómputo utilizados en el trabajo**

1. Estación de trabajo Dell Precision 3660:
    - CPU Intel Core i7-13700
    - GPU NVIDIA
    - 32 GB DDR4 RAM
    - x2 1 TB NVME SSD
    - 3 puertos USB 3.0 y 2 puertos USB
    - Sistema operativo Linux Ubuntu 22.04 LTS
2. Ordenador personal:
    - CPU Intel Core i5-14600k
    - GPU NVIDIA 4070
    - 32 GB RAM
    - 1 TB HDD
    - 3 puertos USB 3.0 y 3 puertos USB 2.0
3. Servidores de cómputo Dell EMC PowerEdge R640:
    - Procesador Intel Xeon Gold 6240 (18 cores)
    - 128 GB de RAM
    - Discos SSD 480 GB
    - Red Ethernet 10GbE

**Plataformas de prototipado *Hardware* utilizadas en el trabajo**

1. Placa de prototipado FPGA Digilent inc. Nexys A7:
    - FPGA Part#: C7A100T-1CSG324C

- 15,850 Slices de lógica programable, cada uno con 4 LUTs de seis entradas y 8 flip/flops
- 4,860 Kbits BRAM
- 240 DSPs

2. Placa de prototipado FPGA Digilent inc. Genesys 2:
   - FPGA Part#: C7K325T-2FFG900C
   - 50,950 Slices de lógica programable, cada uno con 4 LUTs de seis entradas y 8 flip/flops
   - 16 Mbits BRAM rápida
   - 840 DSPs

## 3.2. Versiones de programas *software* utilizados

A continuación se enumeran los sistemas operativos utilizados, y las herramientas de compilación, verificación y diseño utilizados.

1. Sistema Operativo Ubuntu Linux 24.04 de Canonical para estación de trabajo.
2. Sistema Operativo Ubuntu Linux 22.04.1 de Canonical para ordenador personal.
3. Sistema operativo Red Hat Linux 8.5.0-26 para servidor de cómputo.
4. Sistema de Edición en línea para LaTeX Overleaf.
5. Programa para gestión de referencias bibliográficas Zotero Desktop 7.0.15 (64-bit).
6. Visualizador de PDF Adobe Reader 24.001.30225 de Adobe.
7. RISC-V *toolchain* version 15.1.0
8. Python 3.10.12
9. Siemens/Mentor Graphics QuestaSim-64 2022.4.
10. AMD/Xilinx Vivado 2024.1.

# 4. Condiciones Económicas

Para la elaboración de presupuesto del TFG se han incluido los costes directos relacionados con los recursos humanos utilizados para el desarrollo del trabajo y los costes materiales, tanto *hardware* como *software* utilizados. Cuando se trata de un recurso compartido con otros proyectos, se ha tenido en cuenta el tiempo de utilización para determinar la parte proporcional de su amortización.

En relación con los recursos humanos empleados, se ha tenido en cuenta el número de horas invertidas y el salario correspondiente a un Técnico TCP4 con titulación de Grado Universitario o equivalente (MECES 2) que trabaje para el Instituto Universitario de Microelectrónica Aplicada de la Universidad de Las Palmas de Gran Canaria, teniendo en cuenta los complementos necesarios condicionados por los salarios del mercado en la situación

actual. Además, se han contabilizado las horas de supervisión del trabajo. Asimismo, se ha aplicado un porcentaje de gastos generales del 17 %. Esto incluye los costes indirectos relacionados con los anteriores, como el coste de mantenimiento de los equipos, coste proporcional del personal que los administra, soporte técnico, etc. Igualmente, se ha estimado un beneficio industrial del 10 %, que junto con todos los costes anteriores suman el Presupuesto de Ejecución, que se usa de base imponible para calcular el Impuesto General Indirecto Canario (IGIC) del 7 %.

# 5. Condiciones Facultativas

A la hora de ejecutar los trabajos definidos en el proyecto, la memoria detalla los procedimientos necesarios para su ejecución, la metodología a utilizar y los resultados esperados. Todo ello se ha documentado con el nivel de detalle requerido para su seguimiento utilizando buenas prácticas de diseño. En las Condiciones Técnicas Particulares, se aportan detalles adicionales sobre requerimientos técnicos del entorno de diseño necesarios.

# 6. Condiciones Legales

El presente proyecto se desarrolla en el marco del Trabajo de Fin de Grado en Ingeniería en Tecnologías de la Telecomunicación, impartido en el Escuela de Ingenieros de Telecomunicación y Electrónica de la Universidad de Las Palmas de Gran Canaria.

El trabajo se ha concebido como un trabajo académico y de iniciación a investigación y al desarrollo de sistemas en chip basados en RISC-V. Se ha desarrollado en el Instituto Universitario de Microelectrónica Aplicada de la Universidad de Las Palmas de Gran Canaria, bajo la dirección de investigadores de la División SICAD de dicho Instituto.

El proyecto se entrega como tal, y se descarta el pago de algún tipo de indemnización a quienes intentaran implementar la solución propuesta en el mismo en el hipotético caso de que se produjera algún fallo y/o daño (material y/o inmaterial) como consecuencia de algún error y/o imprecisión en el diseño y/o la descripción del mismo.

El autor se compromete a realizar el trabajo con el máximo rigor académico y actuar en todo momento de buena fe, con el objeto de que el resultado del mismo tenga la calidad necesaria para que sea útil para su posterior desarrollo en base a los resultados presentados.

# Presupuesto

## 1. Introducción

Para la elaboración de presupuesto del TFG se han tenido en cuenta los costes directos relacionados con los recursos humanos utilizados para el desarrollo del trabajo y los costes materiales, tanto *hardware* como *software* utilizados. Cuando se trata de un recurso compartido con otros proyectos, se ha tenido en cuenta el tiempo de utilización para determinar la parte proporcional de su amortización.

En relación con los recursos humanos empleados, se ha calculado en base al número de horas invertidas y el salario correspondiente a un Técnico TCP4 con titulación de Grado Universitario o equivalente (MECES 2), que desarrolle su actividad para el Instituto Universitario de Microelectrónica Aplicada de la Universidad de Las Palmas de Gran Canaria, teniendo en cuenta los complementos necesarios condicionados por los salarios del mercado en la situación actual. Además, se han contabilizado las horas de supervisión del trabajo, en este caso realizado por un profesor Doctor (titulación MECES 4).

Asimismo, se ha aplicado un porcentaje de gastos generales del 17 %. Esto incluye los costes indirectos relacionados, como el coste de mantenimiento de los equipos, parte proporcional de costes del personal que los administra, soporte técnico, etc. Igualmente, se ha estimado un beneficio industrial del 10 %, que junto con todos los costes anteriores suman el Presupuesto de Ejecución, que se usa de base imponible para calcular el Impuesto General Indirecto Canario (IGIC) del 7 %.

Dado el carácter orientado a la I+D de este TFG, el presupuesto se ha calculado en base a costes de desarrollo. No se han incluido otros conceptos relacionados con costes de visado ni otros aspectos de costes colegiales.

El presupuesto se han desglosado en los siguientes conceptos:

1. Recursos materiales (*hardware* y *software*)
2. Recursos humanos
3. Costes indirectos y otros gastos
4. Beneficio
5. Coste total con impuestos

# 2. Recursos materiales

Durante la realización de este Trabajo de Fin de Grado se han utilizado diferentes recursos, *hardware* y *software* fundamentales para el estudio, prototipado y desarrollo de la memoria final del proyecto. Para calcular el coste de estos recursos dentro del presupuesto es necesario considerar la amortización de cada uno, para ello, se ha considerado una amortización lineal según la ecuación 1.

$$\text{Cuota} = \frac{\textit{Valor de adquisición} - \textit{Valor residual}}{\textit{Número de años de vida útil}} \tag{1}$$

## 2.1. Recursos *hardware*

En la tabla 1 se presentan los resultados del coste total de los recursos *hardware* habiendo considerado la amortización.

Tabla 1: Costes del equipamiento *hardware*.

| N | Equipamiento | Valor adquisición (Euros) | Tiempo amortización (Meses) | Coste/mes (Euros) | Uso (Meses) | Importe (Euros) |
|---|---|---|---|---|---|---|
| **Estaciones de trabajo y servidores de cómputo** | | | | | | |
| 1 | Ordenador personal | 1500,00 | 24 | 62,50 | 5 | 312,50 |
| 2 | Estación de trabajo Dell Precision 3660 | 3000,00 | 36 | 69,44 | 5 | 416,67 |
| 3 | Servidores de cómputo Dell EMC PowerEdge R640 | 8000,00 | 48 | 166,67 | 5 | 541,67 |
| **Placas de prototipado FPGA** | | | | | | |
| 4 | Placa de prototipado FPGA Digilent inc. Nexys A7 | 468,75 | 24 | 19,53 | 5 | 97,66 |
| 5 | Placa de prototipado FPGA Digilent inc. Genesys 2 | 1437,50 | 24 | 59,90 | 5 | 299,48 |
| **Coste total** | | | | | | **1959,64** |

## 2.2. Recursos *software*

En la tabla 2 se presenta los resultados del cálculo realizado para los recursos *software*. En el caso de las herramientas *software* se han considerado los costes de mantenimiento anual ya que el *software* se considera en régimen de mantenimiento anual. Es importante tener en cuenta que hay ciertos recursos que han sido provistos libre de coste, aunque sean productos comerciales al estar dentro de los programas de EUROPRACTICE en los que participa el IUMA. Otras herramientas están disponible con acceso libre y se han descargado directamente desde el correspondiente repositorio. Los costes de instalación y mantenimiento están incluidos en los costes indirectos del proyecto.

Tabla 2: Costes de herramientas *software* utilizadas.

| N | Herramientas | Valor adquisición (Euros) | Tiempo amortización (Meses) | Coste/mes (Euros) | Uso (Meses) | Importe (Euros) |
|---|---|---|---|---|---|---|
| 1 | Ubuntu Linux 24.04 de Canonical para estación de trabajo. | Libre de coste | N/A | N/A | 5 | 0,00 |
| 2 | Ubuntu Linux 22.04.1 de Canonical para ordenador personal. | Libre de coste | N/A | N/A | 5 | 0,00 |
| 3 | Red Hat Linux 8.5.0-26 para servidor de cómputo (soporte) | 100,00 | 12 | 8,33 | 5 | 41,67 |
| 4 | Sistema de Edición en línea para LaTeX Overleaf (licencia Estudiante) | N/A | N/A | 9,00 | 5 | 45,00 |
| 5 | Programa para gestión de referencias bibliográficas Zotero Desktop 7.0.15 (64-bit). | Libre de coste | N/A | N/A | 5 | 0,00 |
| 6 | Visualizador de PDF Adobe Reader 24.001.30225 de Adobe | Libre de coste | N/A | N/A | 5 | 0,00 |
| 7 | RISC-V *toolchain* version 15.1.0 | Libre de coste | N/A | N/A | 5 | 0,00 |
| 8 | Python 3.10.12 | Libre de coste | N/A | N/A | 5 | 0,00 |
| 9 | Siemens/Mentor Graphics QuestaSim-64 2022.4. (Coste anual licencia universitaria) | 1630,00 | 12 | 135,83 | 5 | 679,17 |
| 10 | AMD/Xilinx Vivado 2024.1. (Coste anual licencia universitaria) | 220,00 | 12 | 18,30 | 5 | 91,67 |
| **Coste total** | | | | | | **857,50** |

N/A: No Aplicable. Libre de coste: código abierto o donación.

## 2.3. Resumen de recursos materiales

El coste total de los recursos materiales se muestra en la tabla 3.

Tabla 3: Costes totales de los recursos materiales.

| N | Concepto | Importe (Euros) |
|---|---|---|
| 1 | Costes del equipamiento *hardware*. | 1959,64 |
| 2 | Costes de herramientas *software* utilizadas | 857,50 |
| **Coste total** | | **2817,14** |

# 3. Recursos humanos

Para el cálculo de la estimación de los costes del proyecto asociados a las horas de trabajo, se toman como referencia los precios oficiales de la ULPGC para un titulado MECES-2, a los que se les aplican una serie de complementos en función de la complejidad del proyecto. Asimismo se han imputado al proyecto las horas de tutela de un titulado MECES-4 en el área de diseño microelectrónico avanzado, como es el caso de este proyecto. La tabla de costes salariales utilizados está publicada en la web de la ULPGC[1]. Dichos costes se han complementado en función del mercado laboral existente. Para el cálculo del coste por hora se ha tomado como referencia 1768 horas laborales al año.

Tabla 4: Costes salariales utilizados.

| N | Titulación | Base (Euros) | Complemento (Euros) | Coste anual) (Euros) | Coste/Hora (Euros) |
|---|---|---|---|---|---|
| 1 | Titulado MECES2 | 24.090,16 | 4.818,03 | 28.908,19 | 16,35 |
| 2 | Titulado MECES4 | 34.281,77 | 6.856,35 | 41.138,12 | 23,27 |

Con los datos obtenidos se hace el cálculo de los costes debido a los recursos humanos del proyecto (tabla 5).

Tabla 5: Costes recursos humanos.

| N | Concepto | Horas | Coste/Hora (Euros) | Coste (Euros) |
|---|---|---|---|---|
| 1 | Trabajo de un técnico TCP4 con titulación de Grado universitario (MECES2) con especialización en diseño electrónico | 300 | 16,35 | 4.905,24 |
| 2 | Horas de dirección del trabajo con titulación MECES 4 especialista en diseño electrónico | 30 | 23,27 | 698,05 |
| **Total** | | | | 5.603,28 |

# 4. Costes indirectos y otros gastos

Se han cuantificado los costes indirectos y otros gastos del proyecto en un 20 % de los costes de ejecución del proyecto[2]. Teniendo en cuenta que los costes de ejecución ascienden a 8.420,42 Euros, dichos costes ascienden a 1.684,08 Euros.

# 5. Resumen de costes del proyecto

En la tabla se muestra el resumen final de costes del proyecto, incluyendo los costes de ejecución (directos) los costes indirectos, el beneficio industrial, calculado sobre el 10 % de los costes totales[3]. Sobre dichos costes se calcula el IGIC aplicable (7 %) para determinar el presupuesto final después de impuestos.

[1] Ver https://www.ulpgc.es/sites/default/files/ArchivosULPGC/investigacion/actualizacion_tablas_salariales_2024.pdf
[2] En este enlace se dan detalles de los porcentajes a aplicar: https://contratos.gobierto.es/preguntas/62lbn903
[3] Ver https://contratos.gobierto.es/preguntas/b4pkj3hq

Tabla 6: Resumen de Costes del proyecto.

| N | Descripción | Coste (Euros) |
|---|---|---|
| 1 | Costes de los materiales del proyecto | 2.817,14 |
| 2 | Costes de recursos humanos | 5.603,28 |
| **Subtotal: Costes directos** | | **8.402,42** |
| 3 | Costes Indirectos y otros gastos (calculados sobre un 20 % de los costes directos) | 1.684,08 |
| **Subtotal: Costes totales** | | **10.104,50** |
| 4 | Beneficio (calculados sobre un 10 % de los costes totales) | 1.010,45 |
| **Subtotal: Costes + Beneficios** | | **11.114,95** |
| 5 | Impuestos (IGIC) (7 % de los costes + beneficios) | 778,05 |
| **Presupuesto total** | | **11.893,00** |

El presupuesto total del proyecto "Estudio e Implementación de extensiones dedicadas del juego de instrucciones de RISC-V" asciende a **once mil ochocientos noventa y tres** Euros.

Las Palmas de Gran Canaria, a 14 de julio de 2025

Firmado: René Rodríguez Paredes

# Objetivos de desarrollo sostenible

La realización de este TFG está alineado con los Objetivos y metas de desarrollo sostenible a través de las áreas mencionadas a continuación y justificado mediante las siguientes metas concretas indicadas:

| **ODS** | **Grado de relación con los ODS** | | | |
|---|---|---|---|---|
| | 0<br>No procede | 1<br>Bajo | 2<br>Medio | 3<br>Alto |
| ODS 1 Fin de la Pobreza | ● | ○ | ○ | ○ |
| ODS 2 Hambre cero | ● | ○ | ○ | ○ |
| ODS 3 Salud y Bienestar | ● | ○ | ○ | ○ |
| ODS 4 Educación de calidad | ○ | ○ | ● | ○ |
| ODS 5 Igualdad de género | ● | ○ | ○ | ○ |
| ODS 6 Agua limpia y saneamiento | ● | ○ | ○ | ○ |
| ODS 7 Energía Asequible y no contaminante | ● | ○ | ○ | ○ |
| ODS 8 Trabajo decente y crecimiento económico | ● | ○ | ○ | ○ |
| ODS 9 Industria, Innovación e Infraestructuras | ○ | ○ | ○ | ● |
| ODS 10 Reducción de las desigualdades | ● | ○ | ○ | ○ |
| ODS 11 Ciudades y comunidades sostenibles | ● | ○ | ○ | ○ |
| ODS 12 Producción y consumo sostenibles | ● | ○ | ○ | ○ |
| ODS 13 Acción por el clima | ● | ○ | ○ | ○ |
| ODS 14 Vida submarina | ● | ○ | ○ | ○ |
| ODS 15 Vida de ecosistemas terrestres | ● | ○ | ○ | ○ |
| ODS 16 Paz, justicia e instituciones sólidas | ● | ○ | ○ | ○ |
| ODS 17 Alianzas para lograr objetivos | ○ | ○ | ● | ○ |

En concreto, este TFG hace énfasis en los siguientes aspectos:

- Área 4, “Educación de calidad” (ODS 4): metas 4.3 y 4.4:
  - "4.3 De aquí a 2030, asegurar el acceso igualitario de todos los hombres y las mujeres a una formación técnica, profesional y superior de calidad, incluida la enseñanza universitaria."
  - "4.4 De aquí a 2030, aumentar considerablemente el número de jóvenes y adultos que tienen las competencias necesarias, en particular técnicas y profesionales, para acceder al empleo, el trabajo

decente y el emprendimiento."

- Área 9, "Industria, innovación e infraestructuras" (ODS 9): metas 9.3, 9.4 y 9.5.
  - "9.3 Aumentar el acceso de las pequeñas industrias y otras empresas, particularmente en los países en desarrollo, a los servicios financieros, incluidos créditos asequibles, y su integración en las cadenas de valor y los mercados."
  - "9.4 De aquí a 2030, modernizar la infraestructura y reconvertir las industrias para que sean sostenibles, utilizando los recursos con mayor eficacia y promoviendo la adopción de tecnologías y procesos industriales limpios y ambientalmente racionales, y logrando que todos los países tomen medidas de acuerdo con sus capacidades respectivas".
  - "9.5 Aumentar la investigación científica y mejorar la capacidad tecnológica de los sectores industriales de todos los países, en particular los países en desarrollo, entre otras cosas fomentando la innovación y aumentando considerablemente, de aquí a 2030, el número de personas que trabajan en investigación y desarrollo por millón de habitantes y los gastos de los sectores público y privado en investigación y desarrollo."
- Área 17, "Alianzas para lograr los objetivos" (ODS 17): metas, dentro del conjunto de tecnología, 17.6 y 17.7.
  - "17.6 Mejorar la cooperación regional e internacional Norte-Sur, Sur-Sur y triangular en materia de ciencia, tecnología e innovación y el acceso a estas, y aumentar el intercambio de conocimientos en condiciones mutuamente convenidas, incluso mejorando la coordinación entre los mecanismos existentes, en particular a nivel de las Naciones Unidas, y mediante un mecanismo mundial de facilitación de la tecnología"
  - "17.7 Promover el desarrollo de tecnologías ecológicamente racionales y su transferencia, divulgación y difusión a los países en desarrollo en condiciones favorables, incluso en condiciones concesionarias y preferenciales, según lo convenido de mutuo acuerdo."

# Anexos

## Anexo A. Código para la generación de las figuras relacionadas con la transformada de *wavelet*

Código adaptado de la documentación de la documentación de biblioteca de Python PyWavelets [56].

```
import matplotlib.pyplot as plt
import numpy as np
import pywt

def gaussian(x, x0, sigma):
    """Gaussian envelope."""
    return np.exp(-((x - x0) / sigma) ** 2 / 2)

def make_chirp(t, t0, a):
    frequency = (a * (t + t0)) ** 2
    chirp = np.sin(2 * np.pi * frequency * t)
    return chirp, frequency

# Suma de chirp
time = np.linspace(0, 1, 2000)
chirp1, frequency1 = make_chirp(time, 0.2, 9)
chirp2, frequency2 = make_chirp(time, 0.1, 5)
chirp = chirp1 + 0.6 * chirp2
chirp *= gaussian(time, 0.5, 0.2)

# Figura 0: Señal de entrada
fig0, axs0 = plt.subplots(2, 1, sharex=True, figsize=(9, 5))
axs0[0].plot(time, chirp, linewidth=1.2)
axs0[0].set_ylabel("Amplitud")
axs0[0].set_title("Señal de entrada")
```

Código fuente 21: Generación de figuras para la explicación de la transformada de *wavelet* (1).

```
axs0[1].plot(time, frequency1, label="Evolución en frecuencia del chirp1")
axs0[1].plot(time, frequency2, label="Evolución en frecuencia del chirp2")
axs0[1].set_yscale("log")
axs0[1].set_xlabel("Tiempo (s)")
axs0[1].set_ylabel("Frecuencia (Hz)")
axs0[1].legend(loc="upper right", frameon=False)
fig0.tight_layout()

# Transformada Continua de Wavelet
wavelet = "cmor1.5-1.0"
scales = np.geomspace(1, 1024, num=100)
sampling_period = np.diff(time).mean()
cwtmatr, freqs = pywt.cwt(chirp, scales, wavelet, sampling_period=sampling_period)
cwtmatr = np.abs(cwtmatr)

# Figure 1: Escalograma
fig1, ax1 = plt.subplots(figsize=(9, 6))
pcm = ax1.pcolormesh(
    time,
    freqs,
    cwtmatr,
    shading="auto",
    cmap="magma"
)
ax1.set_yscale("log")
ax1.set_xlabel("Tiempo (s)")
ax1.set_ylabel("Frecuencia (Hz)")
ax1.set_title("Transformada Continua de Wavelet (Escalograma)")
fig1.colorbar(pcm, ax=ax1, label="|cwt|")
fig1.tight_layout()

# Transformada de Fourier
from numpy.fft import rfft, rfftfreq

yf = rfft(chirp)
xf = rfftfreq(len(chirp), sampling_period)

# Figura 2: Espectro
fig2, ax2 = plt.subplots(figsize=(9, 4))
ax2.semilogx(xf, np.abs(yf), linewidth=1.2)
ax2.set_xlabel("Frecuencia (Hz)")
ax2.set_ylabel("Magnitud")
ax2.set_title("Transformada de Fourier")
ax2.grid(True, which="both", linestyle="--", linewidth=0.5)
fig2.tight_layout()
plt.show()
```

Código fuente 22: Generación de figuras para la explicación de la transformada de *wavelet* (2).