UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA
DEPARTAMENTO DE TECNOLOGÍAS Y SISTEMAS DE LA INFORMACIÓN

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS

Tareas hardware para FreeRTOS y X-Heep

Máster en Sistemas Microelectrónicos Basados en Arquitecturas Abiertas

Autor:
Antonio Fernando Mateo Francés

Director:
Fernando Rincón Calle

Ciudad Real, Albacete, 2025

Tareas hardware para FreeRTOS y X-Heep

Máster en Sistemas Microelectrónicos Basados en Arquitecturas Abiertas

Financiado por el Ministerio de Transformación Digital y Función Pública de España a través del programa PERTE Chip (Cátedra UCLM, TSI-069100-2023-0014).

ESCUELA SUPERIOR DE INFORMÁTICA
DEPARTAMENTO DE TECNOLOGÍAS Y SISTEMAS DE LA INFORMACIÓN

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS

UNIVERSIDAD DE CASTILLA-LA MANCHA

Tareas hardware para FreeRTOS y X-Heep

Máster en Sistemas Microelectrónicos Basados en Arquitecturas Abiertas

Realizada por:

ANTONIO FERNANDO MATEO FRANCÉS
Ingeniero Informático
Universidad de Castilla-La Mancha

Dirgida por:

FERNANDO RINCÓN CALLE
Profesor Titular de Universidad
Universidad de Castilla-La Mancha

Ciudad Real, Albacete, 2025

Antonio Fernando Mateo Francés

Escuela Superior de Informática
Edificio Fermín Caballero
Paseo de la Universidad, 4
13071 Ciudad Real, España

Escuela Superior de Ingeniería Informática
Edificio Infante Don Juan Manuel
Avda. de España s/n.
02071 Albacete, España

Teléfono: 648556069
E-mail: antoniofernando.mateo@alu.uclm.es
Web site:

Este documento fue realizado en LaTeX.

Tribunal:

Presidente: _____

Vocal: _____

Secretario(a): _____

Fecha de defensa: _____

Calificación: _____

Presidente	Vocal	Secretario(a)
Fdo.:	Fdo.:	Fdo.:

Agradecimientos

Aquí los agradecimientos (lo que solo entenderá la gente)

Aquí las dedicatorias

Aquí frase célebre

Resumen

El avance de la arquitectura abierta RISC-V, con su naturaleza modular y libre de licencias, se posiciona como una alternativa competitiva frente a arquitecturas propietarias como ARM. especialmente en entornos académicos y de investigación, donde la capacidad de personalización es clave. En este contexto, el uso de simulaciones *hardware* para validar arquitecturas, como las realizadas con el proyecto X-HEEP, permite reducir los tiempos de desarrollo y presentación al mercado al agilizar los ciclos de desarrollo. Para poder tener un control general de la plataforma en tiempo real, el uso de los RTOS es crucial actualmente, permitiendo una gestión global sencilla. En este trabajo se abordará el desarollo de un diseño *hardware* acelerador de una tarea de cómputo, validado mediante simulaciones *hardware* y gestionado en todo momento mediante un RTOS.

Abstract

The rising importance of the open RISC-V architecture, with its modular and license-free nature, is positioning itself as a competitive alternative to proprietary architectures usch as ARM, especially in academic and research environments where customization is key. In this context, hardware simulations are used to validate architectures' design validation, as the X-HEEP project does, reducing the time-to-market as they speed up the development cycles. To maintain general, real-time control of the platform, the use of RTOS is currently crucial, allowing designers a simple overall management. In this work, the development of a hardware accelerator design for a computational task will be adressed, validated through hardware simulations and managed globally by an RTOS.

Acrónimos

ACAP Adaptive Compute Acceleration Platforms

ARM Advanced RISC Machines

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

CAD Computer Aided Design

DSP Digital Signal Processor

DVCS Distributed Version Control System

DUT Device Under Test

EDA Electronic Design Automation

EPFL Escuela Politécnica Federal de Lausanne

FIFO First In First Out

FFT Fourier Fast Transform

FPGA Field Programmable Gate Array

FSM Finite State Machine

GCC GNU Compiler Collection

HDL Hardware Description Languages

HLS High-Level Synthesis

IA Inteligencia Artifical

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

IoT Internet of Things

IP Intellectual Property

ISA Instruction Set Architecture

MS Microsoft

MCU Micro Controller Unit

RISC-V Reduced Instruction Set Computing V

RTL Register Transfer Level

RTOS Real Time Operating Systems

OBI Open Bus Interface

PUD Procesos Unificados de Desarrollo

SoC System-on-Chip

TFM Trabajo Fin de Máster

UUT Unit Under Test

VHDL Very High Speed Integrated Circuit Hardware Description Language

X-HEEP eXtendable Heterogeneous Energy-Efficient Platform

Índice general

Índice de figuras

Capítulo 1

Introducción

«If we knew what we were doing, it wouldn't be called research, would it?»
Albert Einstein

Los Field Programmable Gate Array (FPGA) son dispositivos de hardware reconfigurables que permiten implementar circuitos digitales personalizados sin necesidad de fabricar un chip desde cero. Están compuestos por una matriz de bloques lógicos programables y una red de interconexiones que puede configurarse para realizar prácticamente cualquier función lógica. En la actualidad, l. Esto desemboca en que todas las fases necesarias en la realización del diseño tienen que estar optimizadas para no sufrir contratiempos inesperados. Esta flexibilidad los convierte en una herramienta fundamental tanto en investigación como en desarrollo industrial, ya que permiten probar y optimizar arquitecturas antes de su producción física, en un mercado cada vez más competitivo, donde las nuevas arquitecturas de *hardware* son lanzadas al mercado anualmente, a veces incluso reduciéndose este valor a la mitad. A diferencia de un microprocesador fijo, un FPGA puede adaptarse a distintos algoritmos, interfaces y arquitecturas, lo que lo hace ideal para prototipado rápido, procesamiento de señales, aplicaciones de alta velocidad y desarrollo de procesadores personalizados como los basados en Reduced Instruction Set Computing V (RISC-V). Esta arquitectura ha emergido como una alternativa abierta y flexible frente a arquitecturas propietarias como Advanced RISC Machines (ARM), especialmente en el desarrollo sobre FPGAs. Mientras que ARM ofrece núcleos muy optimizados pero bajo licencia y con limitaciones en la personalización, RISC-V permite a los diseñadores modificar y extender la arquitectura según las necesidades específicas del proyecto. En entornos de FPGA, esta apertura es especialmente valiosa, ya que facilita la experimentación con extensiones personalizadas, la integración con periféricos específicos y la adaptación a aplicaciones concretas. Además, la creciente comunidad y ecosistema de herramientas libres en torno a RISC-V reduce barreras de entrada y fomenta la innovación, donde la experimentación y la optimización del procesador forman parte del objetivo principal. RISC-V es aún una arquitectura en proceso de adaptación a

nivel global. A pesar de ello, son muchas las empresas que están realizando la transición hacia esta arquitectura, como la conocida Nvidia [A25], anunciando hace un mes que CUDA tendrá soporte a RISC-V. Sin embargo, no son solo las empresas las que participan en su desarrollo: actualmente, hay gran cantidad de proyectos sobre esta arquitectura, como es el simulador de núcleos RISC-V: el proyeto eXtendable Heterogeneous Energy-Efficient Platform (X-HEEP) [MSM+24] desarrollado por el Escuela Politécnica Federal de Lausanne (EPFL) en Lausanne, Suiza. Será precisamente este simulador el empleado en este Trabajo Fin de Máster (TFM) para realizar validaciones y mediciones de rendimiento sobre el diseño *hardware* realizado.

Figura 1.1: Logo de RISC-V. Wikimedia Commons

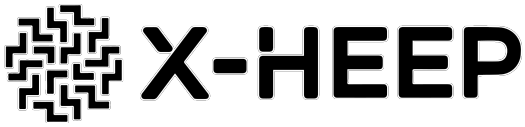

Figura 1.2: Logo de X-HEEP. Fuente: [EPF25]

Figura 1.3: Título común para ambas figuras.

Debido a la gran complejidad que supone el diseño *hardware*, se han propuesto una gran cantidad de metodologías [Sch02] Entre las metodologías más utilizadas se encuentra el diseño a nivel de registro Register Transfer Level (RTL), que utiliza Hardware Description Languages (HDL)s como Very High Speed Integrated Circuit Hardware Description Language (VHDL) o Verilog para especificar el comportamiento y la estructura del circuito de forma detallada. Esta aproximación ofrece un control total sobre la implementación, pero requiere un conocimiento profundo de la arquitectura y de las restricciones de temporización. Otra metodología relevante es la síntesis de alto nivel High-Level Synthesis (HLS), que permite describir el hardware a partir de lenguajes de programación como C, C++ o SystemC, reduciendo el tiempo de desarrollo y facilitando la exploración de arquitecturas, aunque con un posible coste en optimización a grano fino. El diseño basado en Intellectual Property (IP) Cores reutilizables es también ampliamente empleado, permitiendo integrar bloques funcionales ya probados para acelerar la creación de sistemas complejos. A estos enfoques se suman metodologías híbridas que combinan RTL, HLS y IPs, junto con el uso de herramientas de verificación y simulación avanzadas, formando un flujo de trabajo iterativo que va desde el modelado funcional hasta la implementación física. Por ello, han surgido herramientas que simplifican todo el proceso con el objetivo de hacer el diseño *hardware* algo sencillo y apto para un público más amplio, como diseñadores de *software*.

Desde hace tiempo, se han empleado herramientas Computer Aided Design (CAD) con el objetivo de acelerar y automatizar la mayor cantidad de los procesos necesarios en el diseño *hardware*. En este contexto, destacan las herramientas de la empresa Xilinx, actualmente propiedad de AMD [Ray22]. Su plataforma principal, Vivado Design Suite [AMD25c], ofrece un entorno unificado que abarca desde la descripción RTL tradicional hasta el diseño a través de HLS con Vitis HLS, así como la integración de IPs mediante su IP Integrator. Este enfoque fomenta un desarrollo modular, donde se combinan bloques predefinidos con lógica personalizada, optimizando el uso de recursos del FPGA y acelerando la validación del sistema. La metodología de Xilinx también pone especial énfasis en la verificación temprana mediante simulación, análisis de temporización y estimación de consumo, permitiendo corregir problemas antes de la implementación física. En proyectos orientados a System-on-Chip (SoC), se integra además el flujo de diseño para procesadores embebidos como MicroBlaze o núcleos ARM (en FPGAs con SoC integrado), así como el soporte a arquitecturas abiertas como RISC-V. Esta filosofía de diseño, ahora reforzada por la integración con AMD, apunta a ofrecer un entorno flexible y escalable, capaz de cubrir desde prototipado rápido hasta despliegues en producción de alto rendimiento.

Dentro del campo del diseño de *hardware*, destaca el apartado de las simulaciones. Esto es así debido a que el coste de un prototipo es, por lo general, demasiado elevado como para realizar pruebas sobre una plataforma física, lo cual incurriría en un coste extra de desarrollo. Por tanto, la simulación es clave, pues permite validar el diseño antes de su implementación física, ahorrando tiempo y costos asociados a posibles errores. Mediante simuladores de ciclo preciso o de alto nivel, los desarrolladores pueden verificar el comportamiento funcional, analizar el rendimiento y detectar problemas de sincronización sin necesidad de programar repetidamente el FPGA. En proyectos como X-HEEP, es posible evaluar diferentes configuraciones del núcleo RISC-V, probar periféricos personalizados y garantizar la correcta integración del sistema completo. De este modo, se minimizan riesgos y se acelera el ciclo de desarrollo, haciendo posible que las fases de depuración y optimización se realicen con mayor precisión y menor dependencia de hardware físico.

Precisamente en la ejecución de simulaciones, a la hora de simular un conjunto indefinido de tareas, es necesario el uso de herramientas que permitan una correcta gestión de los recursos y que simplifiquen el proceso de ejecución de cada tarea. La solución planteada actualmente a esta problemática es el uso de los conocidos como RTOS, sistemas operativos livianos que, con un conjunto reducido de librerías son capaces de gestionar un diseño de forma global, algo especialmente útil en entornos embebidos, donde los plazos de tiempo generalmente son críticos.

Capítulo 2

Estado del arte

«Cita o lo que quieras»
Author

En el siguiente capítulo de este TFM se presentarán los conceptos y herramientas que es necesario comprender para poder seguir correctamente el flujo de este trabajo.

2.1. Conceptos Fundamentales

2.1.1. RISC-V

RISC-V es un Instruction Set Architecture (ISA) abierta y libre de *royalties*, desarrollada originalmente en la Universidad de Berkeley. Desde sus inicios en 2010, ha sido diseñada para permitir implementaciones ligeras, de bajo consumo y fácilmente adaptables para diversos dispositivos. Además, su flexibilidad, bajo costo y capacidad de personalización lo posicionan como un serio competidor frente a arquitecturas más tradicionales como ARM, con el principal objetivo de desbancarlo en términos de rendimiento y consumo, siendo precisamente éstos los más importantes para el proyecto actualmente. Si bien es posible, gracias a la capacidad de modularización, estar presente en diferentes escenarios, RISC-V tiene su mayor relevancia en sistemas embebidos y sensores Internet of Things (IoT), estimándose que miles de millones de unidades equipadas con RISC-V ya están en circulación [Int21].

RISC-V potencia la colaboración e innovación al ser una plataforma abierta que no requiere revelar propiedad intelectual, lo cual permitiría a, por ejemplo, Europa, a diseñar sus propios procesadores sin depender de terceros, siendo prueba de ello la fuerte inversión que está realizando en potenciar esta industria [Eur24]. De hecho, fruto de esta inversión es precisamente este programa de máster, enmarcado en el programa PERTE-CHIP [SER22]. Un ejemplo de procesador RISC-V se presenta en 2.1.

Figura 2.1: El primer chip RISC-V europeo, EPAC. Fuente: [Xat21]

2.1.2. Cola FIFO

Una cola First In First Out (FIFO) es una estructura de datos en la que el primer elemento que entra es el primero que sale, siguiendo el principio de una fila en la vida real. En hardware, las colas FIFO suelen implementarse como bloques de memoria con dos punteros (lectura y escritura) y lógica de control que garantiza que los datos se lean en el mismo orden en que fueron escritos. En el diseño digital, las FIFO se usan como *buffers* entre módulos que operan a velocidades o relojes diferentes, o para absorber variaciones en el flujo de datos. Pueden ser síncronas (un solo dominio de reloj) o asíncronas (diferentes dominios de reloj en lectura y escritura, con circuitos de sincronización). En el contexto de este trabajo, las FIFO utilizadas son de carácter síncrono, a modo de simplificar el diseño final.

Las colas FIFO presentan un considerable número de ventajas, destacando:

1. Aislamiento entre productores y consumidores
2. Simplificación de la lógica de control
3. Facilidad para integrarse en arquitecturas *pipeline*
4. Versatilidad en IP cores
5. Evitan pérdidas de datos, sirviendo de almacenamiento temporal

2.1.3. FPGA

Los FPGA surgieron en la década de 1980 como una solución intermedia entre circuitos integrados de propósito general y los Application Specific Integrated Circuit (ASIC), chips de propósito específico, con el objetivo de tener una plataforma con capacidad suficiente para desplegar prototipos *hardware* y agilizar los procesos de desarrollo. Fue precisamente la empresa Xilinx, actualmente AMD, la que introdujo en 1985 el primer FPGA comercial, permitiendo a los diseñadores configurar y reconfigurar la lógica interna del chip un número casi ilimitado de veces, perfecto para pruebas [Rom19]. En la actualidad, los FPGAs han pasado de ser componentes para lógica reconfigurable a Adaptive Compute Acceleration Platforms (ACAP)s [AMD25d], plataformas heterogéneas de aceleración que integran bloques de CPU, motores para Inteligencia Artifical (IA) y Digital Signal Processor (DSP), memoria de alta velocidad y conectividad coherente. Además, la adopción en la nube y el empuje por herramientas (propietarias y open-source), los ponen como opción clave para aceleración en centros de datos y computación en el *edge*. En este contexto, destaca Vivado Design Suite [AMD25c], la herramienta de desarrollo empleada en este trabajo.

Los retos actuales incluyen la reducción de la complejidad del flujo de diseño, el desarrollo de estándares abiertos para operar entre plataformas de fabricantes distintos, y el equilibrio entre programabilidad y rendimiento. A pesar de que el uso de lenguajes de alto nivel gracias a la adopción del HLS ha simplificado el desarrollo, todavía existe una curva de aprendizaje pronunciada. En el futuro, se espera que los FPGA integren cada vez más interfaces coherentes como CXL, se adapten mejor a los flujos de trabajo de IA y expandan su papel en computación en el *edge* y aplicaciones de carácter científico. En la figura 2.2 puede verse la placa FPGA Nexys A7, de Diligent, empleada a lo largo del máster.

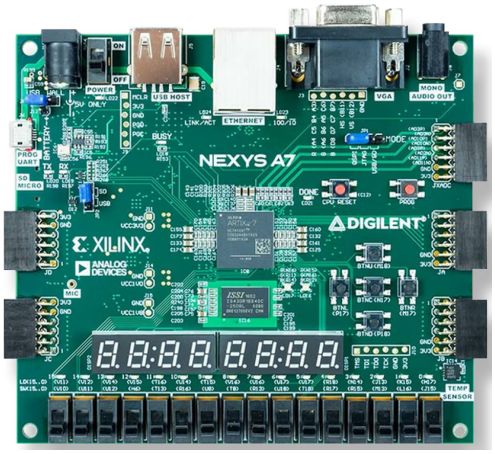

Figura 2.2: Diligent Nexys A7. Fuente: [AMD22]

2.1.4. Diseño *hardware*

Es el proceso de concebir, describir y realizar un sistema directamente en términos de sus componentes físicos y su comportamiento electrónico, sin abstraerse totalmente al software o a un nivel puramente algorítmico [Jim23], lo que implica definir cómo estará estructurado y cómo funcionará el circuito que implemente la funcionalidad deseada.

Diferencias con el diseño *software*

A diferencia del diseño *software*, el diseño *hardware* no trata de escribir un programa que un procesador interpretará, sino de definir el propio procesador y/o el circuito que ejecutará una tarea planteada. Una comparativa puede verse en 2.1.

Tabla 2.1: Comparación entre diseño a nivel de hardware y diseño a nivel de software

Aspecto	**Diseño a nivel de hardware**	**Diseño a nivel de software**
Descripción	Definición de la funcionalidad de un sistema mediante componentes físicos y lógica digital.	Definición de instrucciones y algoritmos que serán ejecutados por un procesador o máquina virtual.
Lenguajes	HDL (VHDL, Verilog, SystemVerilog) y herramientas de síntesis/implementación.	Lenguajes de alto o medio nivel (C, C++, Python, Java, etc.).
Unidad de trabajo	Señales, registros, puertas lógicas, bloques IP, rutas de datos, temporización.	Variables, estructuras de datos, funciones, hilos, procesos.
Ejecución	El comportamiento está "fijo" en el hardware (aunque pueda ser reconfigurable en FPGA).	Se ejecuta secuencial o concurrentemente sobre hardware ya existente.
Paralelismo	Intrínseco: múltiples operaciones ocurren en paralelo por diseño físico.	Generalmente limitado al paralelismo gestionado por hilos/procesos del software.
Tiempo de desarrollo	Mayor complejidad y validación más costosa, incluye simulación, síntesis, implementación y pruebas en hardware real.	Más rápido de iterar y depurar; la validación se hace en un entorno controlado (IDE, simuladores de software).
Optimización	Área, consumo de energía, latencia y frecuencia de operación.	Tiempo de ejecución, uso de memoria, escalabilidad del código.

Principales puntos de vista

Los principales puntos de vista del diseño a nivel *hardware* son los siguientes:

- **Diseño lógico**: descripción del comportamiento con HDLs como VHDL o Verilog, especificando registros, puertas lógicas, Finite State Machine (FSM)s, y protocolos de interconexión a utilizar.
- **Diseño estructural**: organización y conexionado de los módulos o IPs del diseño, integrándolos en FPGAs, ASICs o dispositivos similares mediante el uso de líneas de comunicación y/o protocolos de interconexión planteados en el diseño lógico.
- **Diseño físico**: es la fase del diseño donde se implementa el diseño de forma física en el silicio, incluyendo ubicación de los módulos/IPs, enrutado de las conexiones y optimizaciones de área y consumo.
- **Diseño eléctrico**: definición de niveles de voltaje y limitaciones de corriente máxima dentro del circuito, comprobaciones temporales de las señales y su estabilidad, así como compatibilidad con los buses/protocolos de interconexión utilizados.

Conceptos clave

Metodología EDA

A grandes rasgos, pueden verse dos tipos principales de metodologías: las de gestión de proyectos, y las técnicas, centradas en la optimización del producto final. Será precisamente en ésta última categoía donde entran las metodologías del diseño de *hardware*. En este contexto, se entiende por *metodología técnica* al conjunto de enfoques, procedimientos y herramientas utilizadas para diseñar, desarrollar, verificar y optimizar un producto desde el punto de vista técnico o ingenieril. [Sch02]. En este contexto, se emplean metodologías variadas, como *Top-Down*, que descompone el sistema en módulos jerárquicos hasta llegar a los módulos básicos, y *Bottom-Up*, que consiste en construir el sistema a partir de módulos básicos hasta llegar a abarcar completamente el sistema a desarrollar.

Sin embargo, la metodología relevante dentro de este proyecto es la basada en herramientas Electronic Design Automation (EDA), siendo la metodología base para las herramientas utilizadas en este trabajo, como es la Vivado Design Suite, ya mencionada. Esta metodología adopta un enfoque estructurado e iterativo que se extiende desde la definición de requisitos hasta la fabricación física. El flujo de la metodología EDA se basa en dos grandes

bloques: Front-End y Back-End, con verificación continua en todas las etapas. [Rin24]. Este flujo se detalla en 2.2.

Etapa	**Sub-etapas**
Especificación y planificación	Requisitos funcionales y no funcionales; Arquitectura y selección de IP cores; Modelado de alto nivel: SystemC, Matlab
Front-End	Diseño algorítmico y HLS; Diseño lógico RTL: Verilog, VHDL, SystemVerilog; Reutilización de IP (Soft y Hard); Síntesis lógica a netlist
Back-End	Planificación física, placement y routing; Síntesis y distribución de reloj; Verificación física: DRC y LVS
Verificación	Funcional y formal; Análisis de tiempo estático; Simulación post-layout
Sign-off y fabricación	Validaciones finales y análisis de consumo; Generación de vectores de test (DFT, BIST); Creación de archivos GDSII y tape-out

Tabla 2.2: Resumen de las actividades en cada etapa del flujo de la metodología EDA

Gracias a esta metodología, el diseño de ASICs y sistemas complejos integra fases iterativas, reutilización de IPs, modelado jerárquico y verificación exhaustiva, buscando optimizar coste, tiempo y fiabilidad.

Front-end y Back-end

El Front-End se refiere a la fase inicial de planteamiento del diseño *hardware*, centrada en la funcionalidad y la lógica del circuito, antes de realizar implementación física alguna. Por otra parte, el Back-end va después del Front-end, y se centra en la implementación física del diseño planteado y en preparar el circuito para la fabricación en silicio [Rin24]. En 2.3 se presentan detalladamente las principales actividades de estas dos fases.

Hardware Description Language (HDL)

Los HDLs son lenguajes de programación especializados utilizados para describir, modelar y simular el comportamiento y la estructura de circuitos electrónicos digitales. A diferencia de los lenguajes de programación tradicionales (como C o Python), los HDLs permiten especificar tanto el comportamiento lógico como la implementación física de un sistema, posibilitando su síntesis en dispositivos de hardware como FPGAs o ASICs. Su aparición en la segunda mitad del siglo XX revolucionó la forma en que ingenieros y diseñadores desarrollan hardware, permitiendo ciclos de diseño más rápidos y complejos. [Wik23]. Entre los HDLs más conocidos se encuentran VHDL, Verilog y Chisel.

Etapa	Descripción
Front-End	
Diseño algorítmico	Modelado funcional de los bloques o sistema completo sin preocuparse por la implementación física (ej. algoritmos en SystemC o Matlab).
Síntesis de alto nivel (HLS)	Traducción de modelos funcionales a RTL considerando restricciones de frecuencia, área y consumo.
Diseño lógico RTL	Descripción detallada de la lógica interna, rutas de datos y control usando Verilog, VHDL o SystemVerilog, lista para síntesis.
Reutilización de IP	Integración de bloques ya diseñados: Soft IP (modificable, en RTL) o Hard IP (cercano a silicio, optimizado).
Síntesis lógica	Conversión final de RTL a netlist de puertas lógicas adaptada a la tecnología seleccionada.
Back-End	
Planificación física	Distribución de bloques y componentes en el chip para optimizar área y conexiones.
Placement	Ubicación óptima de celdas para minimizar retardos y conexiones.
Routing	Trazado de interconexiones físicas de celdas en el chip o configuración de recursos en FPGA.
Síntesis y distribución de reloj	Creación de la red de sincronización para que todos los bloques funcionen coordinadamente.
Verificación física	Comprobación de reglas de diseño (DRC) y equivalencia entre esquemático y layout (LVS).

Tabla 2.3: Resumen de etapas del Front-End y Back-End en diseño digital

De todos ellos, el más representativo en el contexto de este trabajo es Verilog, el cual será detallado en posteriores apartados.

Hardware Level Synthesis (HLS)

HLS, también conocido como *behavioral synthesis* o *algorithmic synthesis*, es un proceso automatizado que transforma una descripción funcional de alto nivel, como escrita en C, C++, SystemC, MATLAB o similar en una representación de hardware en HDLs como Verilog [Eng22]

A la hora de diseñar *hardware* para probar en FPGAs, HLS ofrece ventajas como un desarrollo más rápido, una mayor abstracción y portabilidad entre plataformas. Permite a los diseñadores trabajar con lenguajes de alto nivel como C o C++, algo especialmente interesante para diseñadores de *software*. La depuración del código generado por HLS puede ser un desafío y

la calidad de salida de la herramienta depende de la experiencia del diseñador, el cual, para alcanzar una mayor optimización del diseño generado, deberá revisar la generación obtenida. [Tos22]

Las señales Clock y Reset

En procesadores, microcontroladores o circuitos implementados en FPGA, las señales Clock y Reset son fundamentales para que el hardware funcione de manera ordenada y predecible.

La señal de *Clock* (reloj) es una onda periódica, normalmente cuadrática, que marca el ritmo al que un circuito digital ejecuta operaciones. [WH15] [HH21]. Su principal función es sincronizar el funcionamiento de todos los módulos donde esté conectada, como flip-flops, registros y contadores, de manera que los cambios de estado ocurran únicamente en momentos definidos, normalmente en los bordes de subida (*active-high*) o bajada (*active-low*) de la señal. Asimismo, el término *frecuencia de clock*, se refiere a la velocidad con las que se producen estas subidas y bajadas de la señal, siendo a mayor frecuencia, mayor el número de transiciones por unidad de tiempo, lo que desemboca en una mayor capacidad de procesamiento, mientras que su ciclo de trabajo y estabilidad en las transiciones de subida y bajada influyen en la fiabilidad de las operaciones de los módulos conectados a dicha señal.

Por otro lado, La señal de *Reset* tiene la función de inicializar el sistema a un estado conocido y seguro [Cor21]. Al activarse, fuerza a todos los registros y elementos secuenciales a tomar valores iniciales, garantizando que el sistema comience siempre en condiciones predecibles. Existen dos tipos principales: el reset síncrono, que actúa junto con el clock, y el reset asíncrono, que se activa inmediatamente sin esperar un ciclo de reloj. Esta señal es crucial al encender un dispositivo, para evitar estados indeterminados, y también se utiliza para reiniciar el sistema en caso de errores o bloqueos.

La importancia conjunta de ambas señales es clara: el *clock* determina cuándo ocurren las acciones dentro del hardware, mientras que el *reset* define desde dónde y en qué condiciones se empieza a trabajar. Sin un clock estable, el sistema pierde el orden en sus operaciones; sin un reset confiable, el arranque puede ser inestable, por tanto, poco seguro, lo que podría desembocar en graves accidentes allá donde esté desplegado el circuito en cuestión.

Módulo RTL y módulo IP

Un módulo RTL es un bloque de hardware descrito en un lenguaje de descripción de hardware como Verilog, del cual se hablará posteriormente, y donde se especifica el comportamiento y la estructura del circuito a nivel de registros,

señales y operaciones lógicas. El diseñador puede escribir este código manualmente o generarlo con herramientas automatizadas, y normalmente se tiene acceso al código fuente, lo que permite modificarlo, optimizarlo o adaptarlo a distintas plataformas. Un módulo RTL describe con detalle cómo funciona internamente el hardware y es portable. Por tanto, el módulo RTL se centra en el diseño detallado y editable del hardware. [ISD23]

Por otro lado, un módulo IP es un bloque de hardware ya diseñado, probado y empaquetado para ser reutilizado en múltiples proyectos. Generalmente es proporcionado por un fabricante o un tercero (por ejemplo, Xilinx o Intel/Altera) y puede venir como un diseño cerrado (*black-box*) o con opciones limitadas de configuración. [Awa22]. Su uso suele ser más rápido porque evita diseñar el hardware desde cero, pero muchas veces no se puede ver ni modificar su implementación interna, ya que forma parte de la propiedad intelectual del proveedor. Por tanto, el módulo IP es una unidad reutilizable, e inmutable, que puede ser integrada en un diseño para agregarle funcionalidades.

La diferencia principal entre ambos es que el módulo RTL es un diseño abierto y editable que describe el funcionamiento interno del hardware en detalle, de tipo *white-box*, mientras que un módulo IP es un bloque ya terminado y optimizado que normalmente se usa como componente prefabricado, a menudo sin acceso a su código interno, pues, por norma general, está protegido por licencias.

Módulo Top

El módulo top (conocido también por *top-level module*) es la unidad más elevada en la jerarquía de la arquitectura del sistema *hardware* diseñado. Este módulo no es instanciado dentro de otro, y sirve como contenedor de todos los submódulos necesarios para materializar el diseño completo, actuando como el punto de integración final antes de su implementación física [Chi22]. Es en el módulo Top donde se define y organiza la interfaz global con el exterior, como los pines I/O de un FPGA, a la vez que sincroniza la interconexión entre bloques internos que tiene instanciados.

Wrapper

Por otra parte, el *wrapper* desempeña un rol igualmente crucial como intermediario entre un módulo RTL o IP ya existente con el resto del sistema. En plataformas como Vivado Design Suite, herramienta empleada en este trabajo, el *wrapper* se genera para conectar los puertos de entrada y salida del diseño con los pines físicos definidos en el archivo de restricciones. Asimismo, el *wrapper* también facilita la adaptación entre jerarquías, ajustes en nombres de señales o protocolos, y la integración de módulos que de otra forma serían

incompatibles [For06]. El uso de *wrappers* permite mantener la integridad del módulo original (por ejemplo, un IP de terceros), mientras se ajusta a requerimientos externos sin modificar su código fuente. El objetivo general de un *wrapper* es, en múltiples ocasiones, facilitar la conectividad exclusivamente, lo que hace que casi siempre estén carentes de lógica específica.

Llegados a este punto, puede parecer que el módulo Top y un *Wrapper* son muy parecidos. En realidad, desempeñan funciones complementarias. El primero constituye la instancia principal que integra el diseño, mientras que el segundo actúa como un mediador que permite integrar módulos de cualquier tipo, como IPs de terceros, los cuales, en numerosos casos, presentan especificaciones técnicas distintas respecto al resto del diseño. [For06].

Block Design

En el diseño de *hardware* digital, el concepto de *Block Design* se refiere a la construcción de sistemas a partir de bloques modulares que poseen funcionalidades concretas [Xil23]. Cada bloque representa un módulo, que puede ser desde un procesador, un controlador de memoria, o un módulo personalizado. Estos bloques se conectan entre sí a través de interfaces, como el bus Advanced eXtensible Interface (AXI).

Una de las principales ventajas del enfoque de Block Design es la modularidad, permitiéndose el reuso de IPs de terceros, así como módulos personalizados, dentro de un mismo desarrollo. De esta manera, no se necesita trabajar directamente a nivel de puertas lógicas o escribir RTL desde cero, basta con importar cada componente dentro del diseño y generar un *wrapper* que contenga información de todos los presentes dentro del mismo. Así, se acelera el proceso de desarrollo y mejora la escalabilidad del diseño, permitiendo a la vez una visualización clara y jerárquica del sistema, útil para entender y depurar arquitecturas complejas [Xil23] [MK04].

En la práctica, este método es ampliamente utilizado en el desarrollo *hardware*. En el contexto de este trabajo, se ha realizado el desarrollo del diseño a nivel *hardware* precisamente sobre un Block Design, por la buena integración y facilidad de uso dentro de Vivado Design Suite.En la figura 2.3 se presenta un ejemplo de Block Design realizado con Vivado Design Suite

Testbench

En el diseño de circuitos digitales, un *testbench* o banco de pruebas es un archivo escrito generalmente en lenguajes HDL como Verilog, y su función principal es aplicar estímulos al circuito bajo prueba, el Device Under Test (DUT), (o el Unit Under Test (UUT), si se quiere probar un módulo y no el conjunto completo), para finalmente observar las respuestas que se producen, y

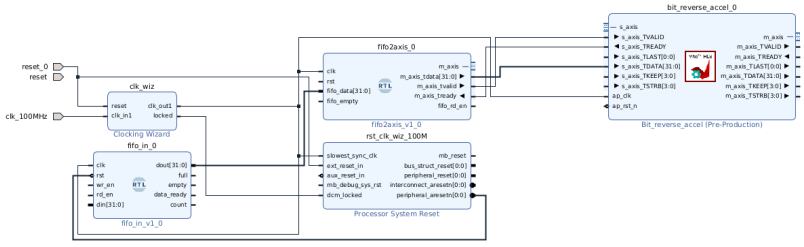

Figura 2.3: Block Design del diseño realizado en Vivado

así comprobar el cumplimiento de los requisitos. El testbench no es un módulo más del diseño; es una herramienta de simulación. Dentro de él, se generan señales de entrada, se definen patrones de prueba (como impulsos de reloj, valores de datos, resets, etc.) y se supervisan las salidas del DUT por medio de equivalencias con lo esperado y aserciones.

Existen distintos niveles de sofisticación en los testbenchs: desde versiones simples que solo aplican algunos vectores de entrada manualmente, hasta entornos más complejos con generación automática de estímulos, verificación dirigida por cobertura (coverage-driven verification), e incluso entornos basados en metodologías formales como UVM (Universal Verification Methodology). En el máster se ha trabajado con UVM, sin embargo, los testbenchs realizados no lo emplean principalmente por errores incomprensibles al intentar integrarlo en el testbench utilizado para las simulaciones en Vivado.

2.1.5. Herramientas utilizadas

X-HEEP

X-HEEP [MSM+24] [EE24] es una plataforma RISC-V *open-source*, diseñada como microcontrolador configurable y extensible, desarrollada en SystemVerilog por el EPFL. Su arquitectura permite personalización en niveles como *cores*, periféricos, memorias y topologías para habilitar aceleradores especializados sin modificar el Micro Controller Unit (MCU) base. Además, soporta múltiples flujos de simulación para pruebas y verificación, destacando Verilator, así como despliegue físico en FPGAs y ASICs, dando resultado a implementaciones como HEEPocrates o X-TRELA [EE25a]. A modo de resumen, se puede decir que X-HEEP es una plataforma de prototipado y simulación a nivel de System-on-Chip (SoC) RISC-V con capacidad de ir desde simulación RTL hasta emulación en FPGA y implementación en ASIC. Sin embargo, aunque se ha hablado sobre X-HEEP, lo cierto es que, en realidad, se haá uso de un *fork* de X-HEEP llamado GR-HEEP [Tor24], preparado para el uso de periféricos y aceleradores externos, como el desarrollado en este proyecto.

En lo que respecta a su arquitectura, el diseño modular de X-HEEP está dividido en distintos dominios de energía (CPU, periféricos, memoria, always-on), optimizando el consumo y permitiendo habilitar/deshabilitar componentes según necesidad. Emplea cores del OpenHW Group como CVE2, CV32E40P(X), todos RISC-V de 32 bits estilo Harvard y sin cache, ideales para escenarios ultra-low-power y RTOS integrados. [EE25b]. En 2.4 se presentan los principales dominios en detalle. Finalmente, en la figura 2.4 se presenta un diagrama de la arquitectura en cuestión.

Tabla 2.4: Dominios del SoC X-HEEP

Dominio	**Descripción resumida**
CPU Subsystem	Núcleos RISC-V OpenHW de 32 bits (CVE2, CV32E40P, CV32E40PX, CV32E40X), arquitectura Harvard sin caché, protocolo OBI, diseñados para bajo consumo y con capacidad de apagado completo o por clock-gating.
Memory Banks	Múltiples bancos de memoria independientes para instrucciones y datos, accesibles en paralelo sin conflictos, con control granular de energía mediante clock-gating, retención o apagado individual.
Peripheral Subsystem	Conjunto de periféricos generales (timer, PLIC, I2C, SPI, 24 GPIO) conectados por una única interfaz con decodificación interna, con posibilidad de apagado o reducción de consumo cuando están inactivos.
Always-On Peripherals	Periféricos e IPs críticos que permanecen activos todo el tiempo (controlador SoC, boot ROM, gestor de energía, fast interrupt, DMA, timer, UART, 2 SPI, 8 GPIO), sin aplicar técnicas de apagado de energía.

Verilog

Creado en 1984 por Phil Moorby y Prabhu Goel en Gateway Design Automation, su diseño buscaba una sintaxis familiar a programadores en C, lo que contribuyó a su rápida adopción. Fue estandarizado por primera vez como IEEE 1364-1995 ("Verilog-95"), seguido por revisiones en 2001 y 2005. En 2009, Verilog fue fusionado dentro de SystemVerilog bajo el estándar IEEE 1800-2009 y actualmente forma parte del estándar unificado IEEE 1800-2023. [IEE01] Verilog ocupa un lugar central en el diseño digital moderno debido a su equilibrio entre legibilidad, flexibilidad y potencia. Fue originalmente un lenguaje propietario en los años 80, diseñado para modelado de hardware, y en la década de los 90 se abrió al dominio público a través de Open Verilog International (OVI), que lo llevó a estandarización del Institute of Electrical and Elec-

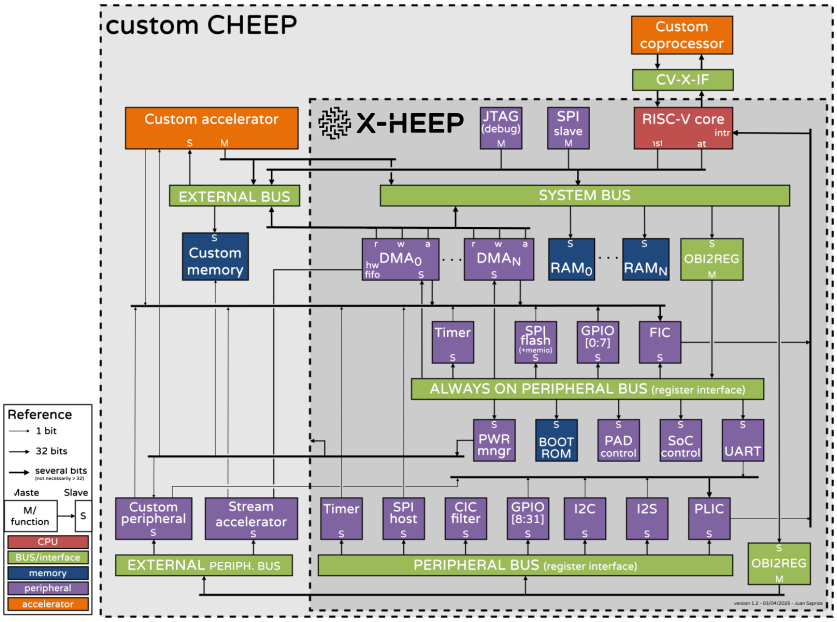

Figura 2.4: Arquitectura de X-HEEP. Fuente: [EE24]

tronics Engineers (IEEE) [Rea20]. Su adopción global y la abundancia de herramientas de simulación y síntesis compatibles lo han consolidado como uno de los pilares del diseño digital moderno. En 2.1 puede verse un ejemplo de un sumador desarrollado en Verilog.

Listado 2.1: Código fuente en Verilog de un sumador

```
module adder_4bit(
    input  [3:0] a,
    input  [3:0] b,
    input        cin,
    output [3:0] sum,
    output       cout
);

    // Salidas del sumador
    assign {cout, sum} = a + b + cin;

endmodule
```

FreeRTOS

Antes de entrar en detalle con la herramienta FreeRTOS, es necesario explicar en qué consiste un Real Time Operating Systems (RTOS). Un RTOS es un tipo

especializado de sistema operativo diseñado para gestionar tareas que deben ejecutarse dentro de límites de tiempo estrictos y predecibles. A diferencia de los sistemas operativos de propósito general, como Windows, que están orientados a la multitarea y la gestión eficiente de recursos, un RTOS se centra en garantizar que las tareas críticas se completen dentro de plazos específicos [Tec24] [Win23].

Si bien hay multitud de opciones a día de hoy, destacan, entre otros, Zephyr [Pro25] y FreeRTOS [Fre25], siendo este último el utilizado en el proyecto. Originalmente desarrollado por Richard Barry en 2003, y mantenido por Amazon desde 2017, FreeRTOS es un kernel de sistema operativo de tiempo real de código abierto, diseñado específicamente para sistemas embebidos con recursos limitados. Además, recientemente ha recibido soporte para RISC-V [Auf19]. Las características principales de FreeRTOS son:

- **Pequeño y eficiente**: diseñado para tener una huella de memoria mínima, lo que lo hace adecuado para microcontroladores con recursos limitados.
- **Multiplataforma**: compatible con más de 40 arquitecturas, incluyendo ARM, AVR, RISC-V, entre otras.
- **Planificación preemptiva**: implementa un planificador de tareas que permite la ejecución de tareas con prioridades, garantizando que las tareas críticas se ejecuten a tiempo.
- **Soporte para multitarea**: permite la crear y gestionar múltiples tareas, semáforos, colas y temporizadores, facilitando la programación concurrente.
- **Licencia MIT**: es de código abierto, lo que permite su uso y modificación sin restricciones.

Para finalizar la sección, se presenta en 2.5 el logo.

Figura 2.5: Logo de FreeRTOS. Fuente: [Com25]

Vivado Design Suite

Vivado Design Suite [AMD25c], desarrollado originalmente por Xilinx en 2012, y ahora parte de AMD, es una plataforma integrada en la metodología basada en

herramientas EDA. Sustituta del antiguo ISE Design Suite, fue creada desde cero para abordar las crecientes exigencias de diseño en FPGAs y SoCs modernos [Mic24].

Vivado está construido sobre un modelo de datos compartido y escalable, que permite una integración fluida entre diseño, síntesis, análisis, simulación, depuración y cierre de tiempos (*timing closure*) Su arquitectura favorece la visibilidad continua de métricas clave como utilización de recursos, potencia, congestión o tiempos, en todas las etapas del flujo de diseño. [Mic14]. Será, por tanto, Vivado Design Suite la empleada para la realización del diseño *hardware* desarrollado en este trabajo.

Las principales herramientas presentes en Vivado Design Suite son:

- **Vivado IDE**: Interfaz gráfica principal para capturar el diseño, gestionar proyectos, correr síntesis, implementación y generar el bitstream. Incluye consola Tcl para automatización y scripting.
- **Vivado Synthesis**: Motor de síntesis lógica para VHDL, Verilog y SystemVerilog, que convierte el código HDL en una netlist optimizada para la FPGA.
- **Vivado Implementation**: Realiza place & route, optimiza para timing closure y genera el bitstream final del diseño.
- **Vivado Simulator (XSim)**: Simulador HDL integrado y multilenguaje para depurar el diseño antes de implementarlo en hardware.
- **Vivado IP Integrator**: Entorno gráfico para ensamblar IPs predefinidos y crear subsistemas usando interfaces como AXI de forma automática.
- **Vivado High-Level Synthesis (HLS)**: Herramienta para diseñar en C, C++ o SystemC y generar hardware a partir de código de alto nivel.
- **Vivado Logic Analyzer**: Permite la depuración en hardware mediante analizadores lógicos integrados (ILA) y monitorización de señales internas en tiempo real.
- **Vivado Power Analysis**: Estimación y análisis de consumo de potencia del diseño en distintas fases del flujo.
- **Vivado ECO Editor**: Permite aplicar Engineering Change Orders al diseño implementado sin necesidad de recompilar todo.

Para finalizar, se presenta en 2.6 un proyecto abierto en Vivado Design Suite.

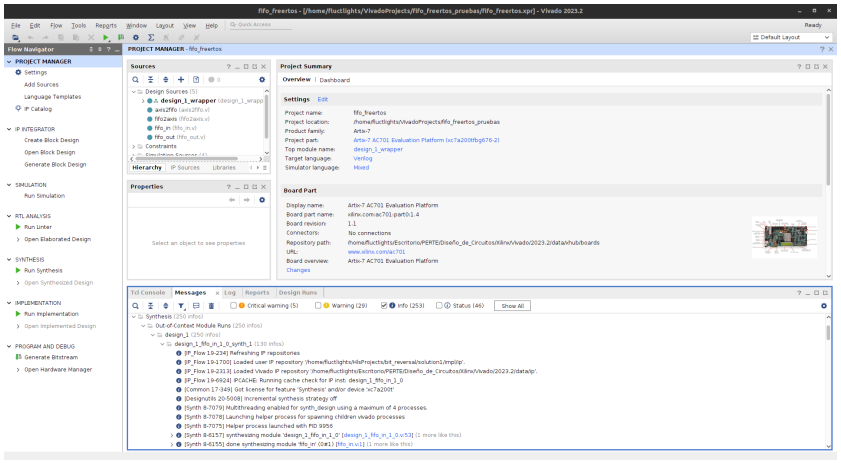

Figura 2.6: Proyecto abierto

Vitis HLS

Vitis HLS es una herramienta de síntesis de alto nivel desarrollada por Xilinx (ahora propiedad de AMD) que permite la conversión de código en C/C++ a RTL sintetizable para dispositivos FPGAs [AMD25a] Esta herramienta facilita la creación de algoritmos complejos para *hardware* sin necesidad de escribir código a lenguajes de descripción de *hardware* tradicionales como VHDL o Verilog. Una de las principales ventajas de Vitis HLS es su capacidad para abstraer el diseño de hardware, permitiendo a los diseñadores centrarse en la funcionalidad del algoritmo en lugar de los detalles de implementación del hardware. La herramienta soporta directivas de síntesis que permiten optimizar aspectos como la paralelización, la unroll de bucles y la partición de arrays, lo que resulta en una implementación más eficiente en términos de recursos y rendimiento. Además, Vitis HLS ofrece simulación en C para validar la funcionalidad del diseño antes de la síntesis, lo que acelera el ciclo de desarrollo y mejora la productividad [AMD25b].

Vitis HLS está estrechamente integrado con Vivado Design Suite [Sha23], permitiendo exportar el diseño sintetizado como un bloque IP, que puede ser importado directamente en Vivado para su integración en un diseño más amplio. Este flujo de trabajo facilita la creación de sistemas complejos que combinan lógica personalizada generada por HLS con bloques IP preexistentes. De esta forma, permite una transición fluida desde el desarrollo de algoritmos hasta los primeros pasos para la implementación en *hardware*.

Nuevamente, para terminar este apartado, se presenta en 2.7 un proyecto abierto en Vitis HLS.

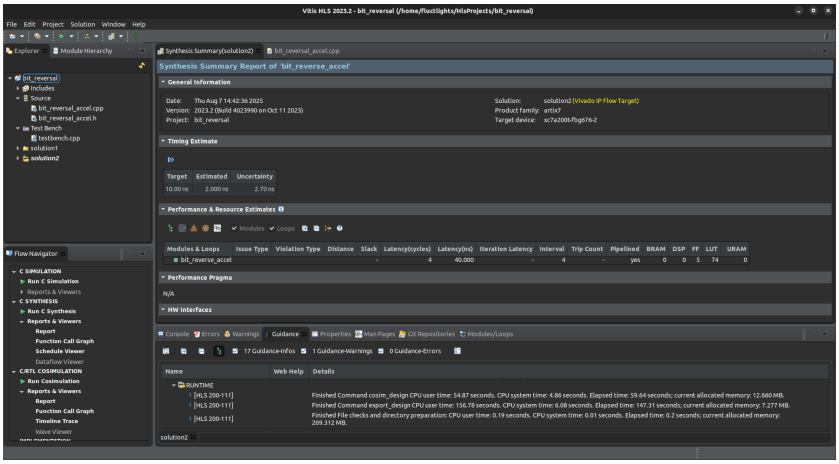

Figura 2.7: Proyecto abierto

EdaPlayground

EDAPlayground es una plataforma en línea gratuita que permite escribir, simular y compartir código de diseño digital utilizando lenguajes de descripción de hardware como Verilog, SystemVerilog y VHDL. Esta herramienta está orientada a estudiantes e ingenieros que quieren realizar simulaciones y compilaciones de diseños hardware sin tener que instalar software especializado. EdaPlayground tiene además, la posibilidad de compartir proyectos fácilmente mediante enlaces, lo convirtiéndolo en una herramienta ideal para tareas de equipo y docencia. Es precisamente por estas cualidades que ha sido la herramienta elegida como alternativa a la Vivado Suite para realizar pruebas de simulación, pudiendo verse el Playground en cuestión en el siguiente enlace.

La plataforma ofrece un Integrated Development Environment (IDE) web, donde los usuarios pueden crear módulos de diseño y *testbenchs*, además de compilarlos y simularlos utilizando diferentes herramientas de simulación proporcionadas por proveedores como Synopsys. Entre los simuladores disponibles se encuentran ModelSim, o el utilizado en el contexto de este trabajo, QuestaSim, de Siemens, en su versión 2023.2. Además, EdaPlayground permite visualizar resultados de simulación, ya sea en forma de texto EPWave, lo que facilita el análisis del comportamiento del circuito. Sin embargo, debido a la complejidad de la simulación, se ha dependido de otra herramienta de visualización llamada *GTKwave*, explicada posteriormente.

GTKwave

GTKwave es un software de visualización de señales de onda, al igual que EPWave de EdaPlayground. Éste es, sin embargo, mucho más potente, pues permite

ver un mayor intervalo de tiempo, entre otras cuestiones y al estar instalado en local (es una instalación muy ligera y fácil de instalar con Aptitude) posee una mayor integración con el entorno gráfico del ordenador en cuestión. Para los diagramas de onda de EdaPlayground se ha utilizado GTKWave por su gran ligereza y potencia, así como su fácil usabilidad.

Figura 2.8: Logo de GTKWave

Capítulo 3

Diseño de la solución

«Cita o lo que quieras»
Author

En este capítulo se expondrán la metodología seguida en el trabajo, los requisitos capturados del mismo, y presentación de la solución desarrollada.

3.1. Metodología seguida

Se define metodología [Esp], como los métodos y pasos seguidos con el objetivo de desarrollar o fabricar algo, de manera que el resultado final sea acorde a los resultados y calidad esperados del mismo. De esta forma, es posible cumplir con los plazos de entrega del producto o servicio, a la vez que se obtiene un resultado acorde a lo establecido.

3.1.1. Procesos Unificados de Desarrollo

En la actualidad, es posible encontrar innumerables metodologías para gestionar un proyecto de cualquier tipo. Si bien es imposible o muy difícil adecuarse por

completo a una sola metodología, es posible decantarse por un conjunto reducido o incluso una sola en función del objetivo del proyecto a desarrollar. En este trabajo se ha emplear una metodología basada en Procesos Unificados de Desarrollo (PUD) [Jac00] [AF15], centrada en el enfoque iterativo e incremental con el objetivo de aportar un mayor valor conforme avanza el proyecto a realizar. Como principales razones para esta elección destacan las siguientes:

- El proyecto se organiza de manera estructurada en iteraciones
- Cada iteración puede verse como un micro-proyecto, permitiendo, si es necesario, la aplicación de metodologías distintas, dando gran flexibilidad.
- Los resultados de cada iteración sirven como base para la siguiente, aumentando progresivamente el valor del proyecto a medida que éste avanza.
- No se necesita definir todos los requisitos al comienzo, es posible realizar ajustes en iteraciones previas sin dificultad.
- Agiliza el desarrollo del proyecto al generar resultados a corto plazo

En la metodología PUD, el esfuerzo se divide en fases, cada una tomando unas entradas, y generando unas salidas, las cuales serán la entrada de la siguiente fase a realizar. El conjunto de todas las fases es lo que se conoce como ciclo. Así pues, un ciclo está compuesto de las siguientes fases:

1. **Inicio:** Se describe del resultado esperado al fin de la iteración.
2. **Elaboración:** Se detallan los casos de uso que a considerar durante la iteración.
3. **Construcción:** Se modela el resultado de la iteración a partir de los casos de uso. En caso de fallo de los requisitos, se realimenta la fase hasta obtener un resultado correcto.
4. **Transición:** Se valida el resultado generado en la iteración. En caso de fallo de los requisitos, se realimenta la fase hasta obtener un resultado correcto.

En este trabajo este ciclo se repetirá las veces necesarias para cumplir con los objetivos planteados en el mismo.

3.1.2. Iteraciones realizadas

A modo de esquema, se presenta en 3.1 una tabla relacionando las iteraciones realizadas en este proyecto, fechas de inicio/fin, descripciones, dentro del único ciclo de PUD realizado. Para finalizar, se presenta en 3.1 el diagrama de Gantt equivalente.

FASE	ITERACIÓN	DESCRIPCIÓN	INICIO	FINAL
Inicio	1	Modelo del sistema	31/05/2025	15/06/2025
Elaboracion	2	Desarrollo del diseño hardware	10/06/2025	31/07/2025
	3	Ejecución de simulaciones funcionales	01/08/2025	10/08/2025
Construccion	4	Integración del diseño en X-HEEP	11/08/2025	25/08/2025
	5	Ejecución del diseño en X-HEEP sobre FreeRTOS	16/08/2025	01/09/2025
Transicion	6	Comparativa de resultados frente a versión software	30/08/2025	03/09/2025
	7	Documentación del proyecto	01/08/2025	05/09/2025

Tabla 3.1: Iteraciones identificadas en el proyecto

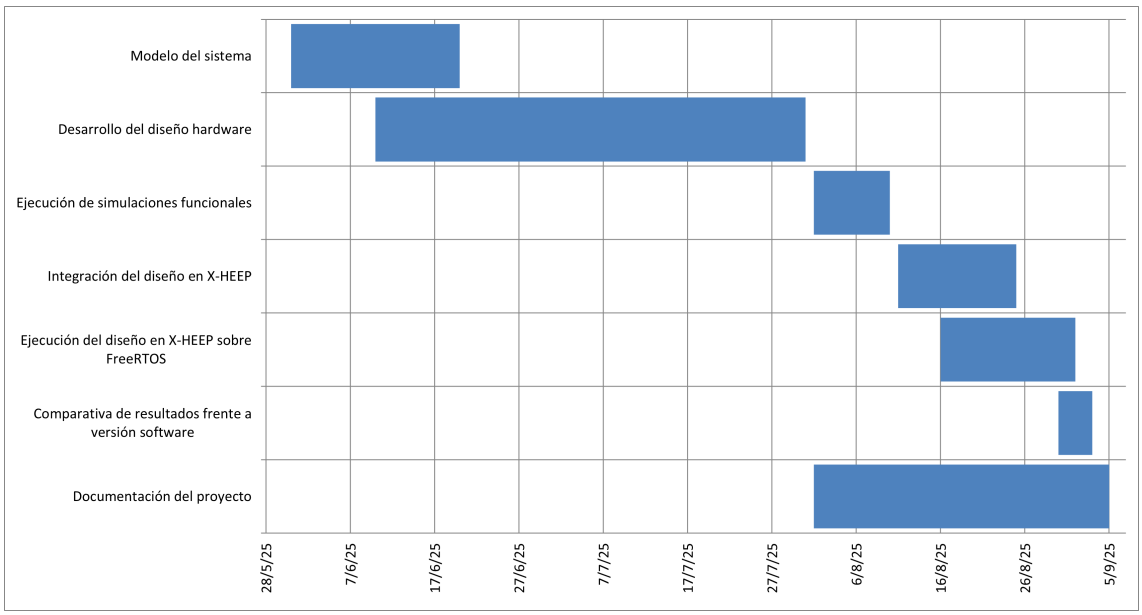

Figura 3.1: Diagrama de Gantt

3.2. Análisis de requisitos

Los requisitos identificados en el proyecto son:

- Desarrollar una aplicación en *hardware*
- Emplear metodologías basadas en herramientas EDA

- Hacer uso del protocolo de comunicaciones AXI-Stream
- Validar la funcionalidad del diseño realizado
- Integración del diseño en X-HEEP
- Ejecutar el diseño en X-HEEP sobre FreeRTOS
- Comparar el rendimiento con una versión *software*

El cumplimiento (completo o parcial) o incumplimiento de los requisitos tenidos en cuenta en este trabajo se valorará posteriormente, en el capítulo 5.

3.3. Arquitectura de la solución

3.3.1. Componentes funcionales

En el proyecto se ha llevado a cabo una solución basada en diagramas de bloques (*block design*) dde carácter modular, donde cada bloque presenta una funcionalidad específica, y su agrupación permite realizar funciones de major complejidad. Concretamente, se han realizado los siguientes bloques:

1. Una cola de entrada de datos, que permite la entrada de enteros de 32 bits, así como una señal de comienzo (*start*) que arranca el diseño globalmente.
2. Una versión acelerada por *hardware* del algoritmo de reverso de bits (*bit reversal*) realizada con el programa Vitis HLS, e importado a Vivado
3. Una cola de salida de datos, que permite obtener el resultado final tras aplicar el reverso, así como una señal de terminado (*done*) para saber en qué momento se tienen resultados.

Estos elementos se han integrado en un Block Design en Vivado, que es el que instancia a todos ellos, tal y como puede verse en 2.6.

3.3.2. Interfaces utilizadas

Para la interconexión de los módulos de las colas con el acelerador en cuestión, se ha hecho uso del conocido protocolo AXI Stream, donde se ha realizado una implementación reducida, teniendo en cuenta, por simplicidad, el menor número de señales

posible: *datos, ready, valid, last*. En la figura 2.6 puede apreciarse estas conexiones citadas.

Además de AXI, es necesario utilizar otro protocolo para poder integrar el diseño en X-HEEP: Open Bus Interface (OBI), protocolo libre de licencias y utilizado en los SoCs RISC-V, RI5CY y LowRISC-Ibex *cores* [Gro21], como el CV32E40P. Este protocolo es similar a AXI, y se basa igualmente en un esquema *request-response*.

3.3.3. Componentes técnicos

Si bien las herramientas principales, empleadas para el diseño de la solución, ya han sido detalladas:

- **Vivado Design Suite**: en 2.1.5
- **X-HEEP**: en 2.1.5
- **FreeRTOS**: en 2.1.5
- **EdaPlayground**: en 2.1.5

Se ddebe, como mínimo, presentar una serie de herramientas extra, clave para el despliegue técnico del proyecto, destacando:

- **Vitis HLS**: [AMD25a] herramienta que convierte un algoritmo en C a código HDL como Verilog, clave para poder integrarlo con otros módulos en *hardware*.
- **Make**[Sta99]: herramienta para compilar la plataforma de X-HEEP
- **RISC-V GCC-Toolchain**[Fou25]: para compilar programas RISC-V con GNU Compiler Collection (GCC)
- **Visual Studio Code**[**?**]: como el IDE empleado para el desarrollo técnico del proyecto

Hay que presentar también herramientas utilizadas en la gestión del proyecto a nivel global:

- **Git**: es un Distributed Version Control System (DVCS), sistema de control de versiones distribuido, que permite mantener un historial del código desarrollado en cada momento, pudiendo volver atrás si es necesario. En Git, un proyecto está compuesto por un *repositorio*, o un conjunto de éstos.

- **GitHub**: plataforma donde los usuarios pueden alojar repositorios Git y registrar los cambios realizados. Para este trabajo, se ha creado un repositorio, al cual puede accederse en este enlace
- **ClickUp**: herramienta web de gestión de proyectos. Permite conocer el estado en todo momento y a planificar diagramas de Gantt cómodamente, entre otros.

Y finalmente, las herramientas utilizadas en la redacción de este proyecto son:

- **LaTeX**: sistema de procesamiento de documentos basado en TeX, muy popular en el ámbito académico, principalmente por la alta calidad de los documentos finales, lo que facilita la lectura.
- **Latex Workshop**: extensión para Visual Studio Code que proporciona soporte a características y compilación para documentos LaTeX.
- **Drawio**: editor web que sirve, entre otros, para haber realizado los diagramas presentes en el trabajo, con gran facilidad y flexibilidad de uso.
- **Tables Generator**: editor web empleado para realizar las tablas presentes en el trabajo.
- **Microsoft (MS) Excel**: gestor de hojas de cálculo polivalente, empleado en este trabajo para ordenar datos y generar las tablas comparativas de rendimiento entre diseño desarrollado y versión puramente software.

3.3.4. Implementación e infraestructura

La implementación del proyecto se ha realizado en tres fases bien diferenciadas:

1. **Desarrollo del algoritmo *bit reversal***: mediante la herramienta Vitis HLS, se ha generado el código RTL necesario para realizar la implementación en *hardware*.
2. **Creación del diseño basado en *block design***: mediante Vivado Design Suite se ha realizado y simulado con *testbenchs* un diseño basado en colas que utiliza AXI-Stream para comunicarse con el acelerador desarrollado.
3. **Integración del diseño en X-HEEP sobre FreeRTOS**: se hace uso de un *fork* de X-HEEP llamado GR-HEEP [Tor24] para ejecutar software en C con las librerías de FreeRTOS y haciendo uso del diseño creado. Será precisamente esta integración la intraestructura final sobre la que se sostenga el trabajo.

Capítulo 4

Resultados

«Cita o lo que quieras»
Author

En el siguiente capítulo se detallarán los desarrollos y se presentarán los resultados. Para una mejor comprensión, se dividirá en diversas secciones, una por cada una de las iteraciones realizadas en este trabajo.

4.1. Iteraciones

4.1.1. Iteración 1 - Modelo del sistema

Esta iteración se centra en la descripción del problema planteado y en la elección de las herramientas a utilizar para el desarrollo de un diseño *hardware* capaz de cumplir con los requisitos expuestos en 3.2. En este punto, se ha analizado en detalle el conjunto de dichos requisitos, además de realizar diversas preguntas, para entender mejor la problemática.

Con el objetivo de cumplir dichos requisitos, se pretende desarrollar un diseño *hardware* capaz de realizar una operación concreta, siendo, en el contexto de este trabajo, el reverso de bits de un número entero de 32 bits. Para ello, se ha optado por un esquema basado en colas FIFO, de las cuales ya se detalló tanto su concepto como sus ventajas en 2.1.2. La idea es, por tanto, hacer sencillo el paso de datos al

acelerador, así como la recepción de los resultados. Las colas FIFO, al tener memoria interna, proporcionan un mecanismo sencillo y relativamente seguro de transmisión.

Para realizar este diseño, es necesario realizarlo en base a una metodología. En el contexto de este trabajo, se ha optado por alinear el esfuerzo a las metodologías basadas en herramientas EDA, ello implicando el uso de programas software que estén adecuados al flujo de este tipo de metodologías. Por ello, se ha optado por seleccionar Vitis HLS y Vivado como programas para el diseño de este hardware. Ambas son herramientas de la misma compañía, Xilinx, lo que repercute en, teóricamente, una mejor integración entre ellas, lo que agiliza el desarrollo. Además, han sido presentadas en el máster, lo que las hace más fáciles de utilizar que herramientas de la competencia, como las de la empresa Cadence. Para pruebas, ya con un SoC simulado, se ha optado por el uso de X-HEEP, que permite la creación de núcleos RISC-V de múltiples tipos, así como simulaciones a nivel *hardware* gracias a Verilator, y con soporte para FreeRTOS, lo que proporciona flexibilidad a la hora de realizar cualquier tipo de despliegue.

Por tanto, a modo de resumen, el resultado de esta iteración ha sido:

1. Se han analizado los requisitos
2. Se ha proporcionado un marco de trabajo para el desarrollo técnico del proyecto
3. Se ha dado respuesta a cómo comunicarse a la entrada y a la salida del módulo acelerador
4. Se han seleccionado e instalado las herramientas a utilizar tras valorar diversas opciones

4.1.2. Iteración 2 - Desarrollo del diseño hardware

En esta fase se ha realizado el esquema correspondiente al un diseño hardware que sea capaz de satisfacer los requisitos y cumplir con el problema planteado. Si bien se estudiaron varios mecanismos, tal y como se expone en 4.1.1, se ha optado finalmente por un esquema basado en colas FIFO.

Diseño original

Para lograr el diseño final, han sido necesario un refinamiento del planteamiento inicial. Originalmente, se ideó un esquema de 5 módulos, imposible de mostrar, debido a que los archivos del proyecto fueron sobreescritos durante la optimización del mismo. Estos módulos son los siguientes:

1. **Cola FIFO de entrada**: cuyo objetivo es guardar datos para, llegados a un *threshold*, comenzar a mandarlos al siguiente. Mediante una señal de escritura, se escriben los datos. A partir de entonces, es posible leerlos por la salida.
2. **Cola FIFO de conversión a AXI-Stream**: este módulo se comunica con el acelerador con señales AXI-Stream de entrada. Recibe los datos guardados por el módulo anterior, y, recibidos todos (ese *threshold*) el módulo activa las señales AXI-Stream para mandar los datos al módulo acelerador
3. **Acelerador generado con Vitis HLS**: este módulo se basa en el protocolo AXI-Stream, de forma que, al activarse las señales de su entrada, comienza a recibir los datos y a procesarlos uno a uno hacia el siguiente módulo y activa señales de salida.
4. **Cola FIFO de conversión AXI al formato de origen**: este módulo se comunica con el acelerador con señales AXI-Stream de salida. Recibe los resultados y, al activarse sus entradas AXI-Stream, comienza a mandarlos al siguiente.
5. **Cola FIFO de salida**: este módulo obtiene los datos resultado del módulo conversos de AXI-Stream a datos puros. Una vez recibidos todos, los devuelve por su salida de datos.

Este esquema no ha sido el definitivo, por varias razones:

- **Demasiados componentes para la tarea a realizar**: se puede agruparlos sin aumentar en exceso la lógica de control del diseño global
- **Inicialización del acelerador inconsistente**: no se tiene lógica alguna para activar el módulo HLS, activándose ésta al inicio, siendo necesario un tiempo extra para un correcto funcionamiento
- **Falta de señales de control**: solo se devuelven los datos del acelerador, sin tener señales de intermedias que indiquen el estado de la ejecución

Diseño final

Descripción

Finalmente, se ha optimizado el diseño original, el cual se presenta en 4.1, añadiendo señalizaciones de control y estado, y reduciendo a tres el número de módulos, que son:

1. **Cola FIFO de conversión a AXI-Stream**: este módulo se comunica con el acelerador con señales AXI-Stream de entrada. Una vez que la señal

de *reset* deja de estar activa, se activa la señal de control *start_accel* que activa el módulo acelerador, el cual requiere de una cantidad reducida de ciclos para inicializarse. Los datos fluyen de la siguiente forma: al activarse la señal de control *start*, se comienza a guardar datos en *buffer*. Llegados al *threshold* de 4 valores, el módulo activa las señales AXI-Stream y manda los datos al módulo acelerador.

2. **Acelerador generado con Vitis HLS**: este módulo emplea en el protocolo AXI-Stream para comunicarse; ya inicializado gracias a *start_accel*, al activarse las señales de su entrada y a recibir datos, comienza a procesarlos uno a uno y los manda hacia el siguiente módulo activando sus señales AXI-Stream de salida.

3. **Cola FIFO de conversión AXI al formato de origen**: este módulo se comunica con el acelerador con señales AXI-Stream de salida. Recibe los resultados y, al activarse sus entradas AXI-Stream, comienza a devolverlos por su salida, a la vez que activa la señal de estado *done*.

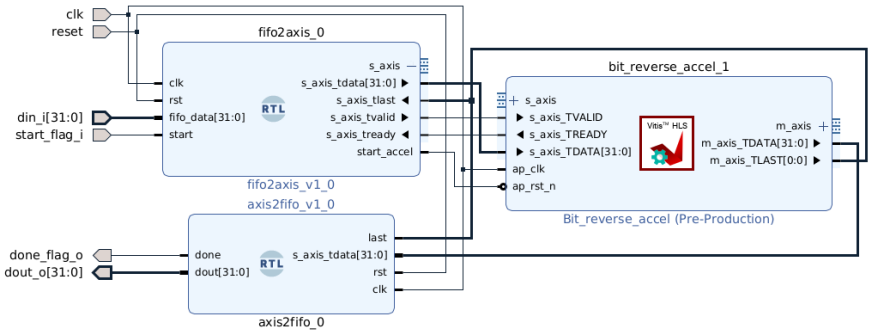

Figura 4.1: Digrama de bloques final

Señales principales del diseño

En esta sección se presentan en **??** las señales más importantes del diseño hardware realizado en este trabajo.

Señal	Descripción
clk	Señal de reloj principal del sistema.
reset / rst	Señal de reset (activa en alto), reinicia los módulos.
din_i[31:0]	Entrada de datos de 32 bits hacia el sistema.
start_flag_i	Señal de inicio para habilitar el envío de datos.
fifo_data[31:0]	Datos provenientes del FIFO hacia el bus AXI-Stream.
s_axis_tdata[31:0]	Bus de datos de entrada (AXI-Stream).
s_axis_tvalid	Indica que los datos en `s_axis_tdata` son válidos.
s_axis_tready	Señal de ready"del esclavo, indica que puede aceptar datos.
s_axis_tlast	Indica la última transferencia de un paquete AXI-Stream.
m_axis_tdata[31:0]	Bus de datos de salida (AXI-Stream).
m_axis_tlast	Señal de fin de paquete de datos en la salida.
done_flag_o	Señal que indica finalización del procesamiento.
dout_o[31:0]	Datos de salida del sistema (32 bits).
ap_clk	Reloj para el acelerador HLS.
ap_rst_n	Reset del acelerador HLS (activo en bajo).

Tabla 4.1: Principales señales del diseño

Con estas señales, el sistema comienza recibiendo los datos de entrada a través de la señal *din_i[31:0]*, los cuales son almacenados temporalmente en un FIFO. Cuando la señal *start_flag_i* habilita el proceso, el bloque *fifo2axis* toma los datos del FIFO y los coloca en la interfaz AXI-Stream, usando *s_axis_tdata* junto con las señales de control *s_axis_tvalid*, *s_axis_tready* y *s_axis_tlast* para garantizar la correcta transmisión. Estos datos entran al acelerador *bit_reverse_accel*, implementado en HLS, donde se realiza el procesamiento de inversión de bits. Una vez procesados, los resultados son enviados a través de la interfaz de salida AXI-Stream (*m_axis_tdata* y *m_axis_tlast*) hacia el bloque *axis2fifo*, que convierte nuevamente la transmisión en datos almacenados en FIFO. Finalmente, los resultados procesados se entregan en la salida *dout_o[31:0]*, mientras que la señal *done_flag_o* indica la finalización del procesamiento.

Código acelerador en C/C++ para HLS

El código relativo al acelerador del reverso de bits en C/C++ empleado para HLS se presenta en esta sección.

Listado 4.1: Código fuente en C/C++ del acelerador reverso de bits

```
1  #ifndef BIT_REVERSAL_ACCEL_H
2  #define BIT_REVERSAL_ACCEL_H
3
4  #include <hls_stream.h>
5  #include <ap_int.h>
6  #include <ap_axi_sdata.h>
7
8  // AXI Stream width: 32 bits of data, no side channels
9  typedef ap_axiu<32, 0, 0, 0> axis_t;
10
```

```
11 void bit_reverse_accel(hls::stream<axis_t>& s_axis,
12                        hls::stream<axis_t>& m_axis);
13
14 #endif
15 // Top function
16 void bit_reverse_accel(hls::stream<axis_t>& s_axis,
17             hls::stream<axis_t>& m_axis) {
18
19 #pragma HLS INTERFACE axis port=s_axis
20 #pragma HLS INTERFACE axis port=m_axis
21 #pragma HLS INTERFACE ap_ctrl_none port=return
22 #pragma HLS PIPELINE II=3
23
24     axis_t input_word, output_word;
25     ap_uint<32> reversed;
26     int i,j = 0;
27
28     // Read 4 words
29     for (i = 0; i < 4; i++) {
30         input_word = s_axis.read();
31
32         for (j = 0; j < 32; j++) {
33 #pragma HLS UNROLL
34       reversed[31 - j] = input_word.data[j];
35     }
36
37         output_word.data = reversed;
38         output_word.last = (i == 3);         // signal
              last word in the stream
39         m_axis.write(output_word);
40     }
41 }
```

En 4.1 se presenta el `bit_reverse_accel`, que implementa un acelerador de hardware para realizar una inversión de bits (*bit-reversal*) sobre palabras de 32 bits, utilizando la interfaz AXI Stream. La función toma como entrada un flujo (*stream*) de datos de tipo `axis_t`, y produce un flujo de salida del mismo tipo. El tipo `axis_t` se define como una estructura AXI Stream de 32 bits (`ap_axiu<32,0,0,0>`), sin canales adicionales (como `keep`, `id`, `dest`, etc.).

La función es decorada con directivas específicas para Vitis HLS. En particular, las interfaces `s_axis` y `m_axis` se definen como puertos AXI4-Stream mediante la directiva `#pragma HLS INTERFACE axis`, lo que permite que el acelerador se comunique directamente con otros módulos de hardware o

con un procesador. También se usa `#pragma HLS INTERFACE ap_ctrl_none port=return`, lo cual indica que no hay control explícito de inicio o fin (es decir, no se utilizan señales como `ap_start` ni `ap_done`), y la función opera de forma continua. Además, se emplea la directiva `#pragma HLS PIPELINE II=3`, solicitando un *Initiation Interval (II)* de 3 ciclos de reloj, permitiendo así procesar un nuevo dato cada 3 ciclos. Si bien, podría hacerse en uno solo, a modo de prueba, se ha decidido dejarlo así.

Dentro del cuerpo de la función, se procesan 4 palabras de 32 bits. En cada iteración del bucle principal (`for (i = 0; i <4; i++)`), se lee una palabra del stream de entrada, y se aplica un bucle anidado que recorre los 32 bits de la palabra, invirtiendo su orden. Esta operación se realiza bit a bit: el bit en la posición `j` del dato de entrada se coloca en la posición `31 - j` de la variable `reversed`, logrando así la inversión. Este bucle está completamente desenrollado con la directiva `#pragma HLS UNROLL`, permitiendo que todos los bits se procesen en paralelo, en un solo ciclo.

Una vez calculada la palabra invertida, se empaqueta en una nueva estructura `output_word`, incluyendo el campo `last`. Este campo indica si la palabra es la última del flujo, lo cual ocurre cuando `i == 3`. Finalmente, el dato transformado se escribe en el flujo de salida `m_axis`.

Código RTL

El código relativo a cada uno de los módulos RTL se presenta en esta sección.

Listado 4.2: Código fuente de la cola FIFO conversora a AXI-Stream

```
module fifo2axis #(
    parameter DATA_WIDTH = 32,
    parameter THRESHOLD = 4,
    parameter DEPTH = 4
)(
    input  wire        clk,
    input  wire        rst,

    // FIFO interface
    input  wire [31:0] fifo_data,
    input  wire        start,

    // AXI Stream master
    output reg [31:0]  s_axis_tdata,
    output reg         s_axis_tvalid,
    input  wire        s_axis_tready,
    output wire        start_accel,
    input wire         s_axis_tlast
```

```
19 );
20
21     // States
22     localparam IDLE       = 3'd0,
23                WRITE_FIFO = 3'd1,
24                WAIT_START = 3'd2,
25                SEND       = 3'd3,
26                DONE       = 3'd4;
27
28     reg [2:0] state, next_state;
29
30     // Internal buffer
31     reg [31:0] buffer [0:3];
32     reg [1:0]  buffer_index;
33     reg [1:0]  send_index;
34     reg start_hls_module;
35
36     // When RESET signal is 0 and module starts, also
          start the HLS module
37     assign start_accel = ~rst;
38
39     // FSM Sequential
40     always @(posedge clk or posedge rst) begin
41         if (rst) begin
42             state <= IDLE;
43             buffer_index <= 2'd0;
44             send_index <= 2'd0;
45             s_axis_tvalid <= 0;
46             s_axis_tdata  <= 32'd0; //s_axis_tdata =
                  {32{1'bx}};
47
48         end else begin
49             state = next_state;
50
51             case (state)
52                 IDLE: begin
53                     if (start) begin
54                         next_state <= WRITE_FIFO;
55                     end else begin
56                       next_state <= IDLE;
57                         buffer_index <= 0;
58                     end
59                 end
60
61                 WRITE_FIFO: begin
62                     buffer[buffer_index] <= fifo_data;
63
```

```
64                      if (buffer_index == 2'd3) begin
65                          next_state <= WAIT_START;
66
67                      end else begin
68                          buffer_index <= buffer_index + 1;
69                          next_state <= WRITE_FIFO;
70                      end
71                  end
72
73                  WAIT_START: begin
74                      if (s_axis_tready) begin
75                          send_index <= 0;
76                          next_state <= SEND;
77                      end else begin
78                          next_state <= WAIT_START;
79                      end
80                  end
81
82                  SEND: begin
83
84                      #5; //wait a bit for AXISTREAM MODULE
85
86                      if (s_axis_tready) begin
87                          s_axis_tdata  <= buffer[send_index
                                ];
88                          s_axis_tvalid <= 1;
89                          #10
90                          s_axis_tvalid <= 0;
91
92                          if (send_index == 2'd3) begin
93                              next_state = DONE;
94                          end else begin
95                              send_index <= send_index + 1;
96                              next_state <= SEND;
97                          end
98
99                      end else begin
100                         next_state <= SEND;
101                     end
102
103                 end
104
105                 DONE: begin
106                     if (s_axis_tlast) begin
107                         s_axis_tdata = {32{1'bx}};
108                         next_state <= IDLE;
109                     end else begin
```

```
110                             next_state <= DONE;
111                         end
112                     end
113
114                     default: begin
115                         next_state <= IDLE;
116                     end
117                 endcase
118             end
119         end
120
121 endmodule
```

En 4.2 se presenta un módulo en Verilog llamado fifo2axis, correspondiente a la cola FIFO de conversión a AXI-Stream. Está parametrizado por el ancho de datos DATA_WIDTH a 32 bits, además de un umbral y la profundidad del buffer interno DEPTH, fijos a 4. El módulo se sincroniza con una señal de reloj clk y un reinicio asincrónico activo alto rst, el cual pasado un mínimo de tiempo, se pone a 0. Además, destacar que el uso de retardos como #5 y #10 dentro del bloque secuencial (always (posedge clk ...)) no es sintácticamente correcto para síntesis en hardware, ya que los retardos solo son válidos en simulación. Por lo tanto, este código necesitaría ajustes si se pretende utilizar en un diseño sintetizable para FPGA o ASIC, algo que, en el contexto de este trabajo, no se tendrá en cuenta. Internamente, el módulo utiliza una FSM que gestiona cinco estados: IDLE, WRITE_FIFO, WAIT_START, SEND y DONE.

- En el estado IDLE, el sistema espera a que se active la señal start, la cual inicia el proceso de escritura de datos en un buffer interno de 4 posiciones (*buffer[0:3]*). Una vez en el estado WRITE_FIFO, el módulo copia los datos de entrada (*fifo_data*) secuencialmente en el buffer, incrementando un índice (*buffer_index*) hasta que se llenan las cuatro posiciones, momento en el cual transiciona al estado WAIT_START.
- El estado WAIT_START espera a que el módulo esclavo conectado a la interfaz AXI esté listo para recibir (s_axis_tready en alto). Cuando esta condición se cumple, se prepara para enviar los datos mediante el estado SEND. En SEND, se transmiten secuencialmente los datos almacenados en el buffer hacia la salida AXI-Stream (*s_axis_tdata*), activándose la señal *s_axis_tvalid* en cada ciclo válido. Este proceso se controla mediante un índice de envío (*send_index*). Una vez que se han enviado todos los datos, el sistema transiciona al estado DONE.
- En el estado DONE, el módulo espera a que se reciba una señal de fin de transmisión desde el receptor AXI (*s_axis_tlast*). Cuando esta señal está

activa, se limpian las salidas y el módulo retorna al estado IDLE, listo para comenzar una nueva transferencia. Además, el módulo expone una señal *start_accel* que se mantiene activa siempre que el sistema no está en reset, y que servirá como activador para acelerador HLS del reverso de bits.

Listado 4.3: Código fuente en Verilog del módulo conversor de AXI al formato de datos origen

```
module axis2fifo #
(
    parameter DATA_WIDTH = 32,
    parameter THRESHOLD = 4,
    parameter DEPTH = 10
)(
    input  wire        clk,
    input  wire        rst,

    // AXI Stream master
    input  wire [31:0] s_axis_tdata,
    input  wire        last,

    // To FIFO
    output reg [31:0]  dout,
    output reg         done
);

    // States
    localparam IDLE         = 3'd0,
               READ_STREAM  = 3'd1,
               SEND_OUTSIDE = 3'd2,
               DONE         = 3'd4;

    reg [2:0] state, next_state;
  integer i;
    // Internal buffer
    reg [31:0] buffer [0:3];
    reg [1:0]  buffer_index;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= IDLE;
            buffer_index <= 2'd0;
            i <= 0;
            done <= 0;

        end else begin
```

```
39              state = next_state;
40
41              case (state)
42                  IDLE: begin
43
44                      if (s_axis_tdata !== {32{1'bx}} &&
                           s_axis_tdata !== {32{1'b0}}) begin
45                          buffer[buffer_index] <=
                               s_axis_tdata;
46                          buffer_index <= buffer_index + 1;
47                          next_state <= READ_STREAM;
48                      end else begin
49                          next_state <= IDLE;
50                      end
51
52                  end
53
54                READ_STREAM: begin
55                      if(last == 1'b0) begin
56                          buffer[buffer_index] <=
                               s_axis_tdata;
57                          buffer_index <= buffer_index + 1;
58                          next_state <= READ_STREAM;
59
60                      end else if (last == 1'b1) begin
61                          buffer[buffer_index] <=
                               s_axis_tdata;
62                          buffer_index <= 0;
63                          next_state <= SEND_OUTSIDE;
64
65                      end else begin
66                          next_state <= READ_STREAM;
67                      end
68                  end
69
70                  SEND_OUTSIDE: begin
71                      dout <= buffer[buffer_index];
72                      done <= 1;
73                      if (buffer_index == THRESHOLD -1)
                           begin
74                          next_state <= DONE;
75
76                      end else begin
77                  buffer_index <= buffer_index + 1;
78                      end
79                  end
80
```

```
81                  DONE: begin
82                      // Set output to X, all data
                           transfered
83                      dout <= {32{1'bX}};
84                      done <= 0;
85
86                      // Set all bits to X into the internal
                            memory FIFO
87                      for (i = 0; i < THRESHOLD; i = i+1)
                           begin
88                          buffer[i] = {32{1'bX}};
89                      end
90
91                      next_state <= IDLE;
92                  end
93
94                  default: begin
95                      next_state <= IDLE;
96                  end
97              endcase
98          end
99      end
100 endmodule
```

En 4.3 se presenta el código fuente del módulo `axis2fifo`, que hace la función de conversor del protocolo AXI al formato original de envío de los datos. Este módulo es complementario al módulo `fifo2axis`, ya que permite recibir datos serializados de un acelerador o componente AXI y almacenarlos temporalmente en un buffer interno antes de enviarlos como los datos originales, pero con el resultado de revertir bits. El diseño está parametrizado mediante DATA_WIDTH a 32 bits, THRESHOLD (cantidad de palabras a procesar, 4) y DEPTH, definido a 10, aunque este último no se usa explícitamente, es únicamente el límite del buffer implementado.

La lógica del módulo está controlada por una FSM con cuatro estados: IDLE, READ_STREAM, SEND_OUTSIDE y DONE. En el estado IDLE, el sistema espera la llegada de un dato válido en `s_axis_tdata`. Para evitar procesar datos no válidos o indeterminados, se verifica que la entrada no sea todo ceros ni indeterminada (`X (32{1'bx})`). Cuando se detecta un dato válido, se almacena en el buffer interno *buffer[0:3]* y se transiciona al estado READ_STREAM.

En el estado READ_STREAM, el módulo continúa leyendo datos de la interfaz AXI Stream. Cada palabra se almacena en el buffer y se incrementa el índice `buffer_index`. Este proceso continúa hasta que se detecta la última

palabra del flujo, señalada por la señal `last`. Una vez leída la última palabra, el índice se reinicia y se cambia al estado SEND_OUTSIDE.

En SEND_OUTSIDE, el módulo comienza a transferir los datos almacenados en el buffer hacia la salida `dout`. Cada palabra se envía una por una y se activa la señal `done` para indicar que un dato válido está disponible. El envío continúa hasta que se han enviado todas las palabras del buffer (determinadas por el parámetro THRESHOLD). Al completarse, el sistema entra al estado DONE.

En el estado DONE, se marca el final de la transmisión: se pone `dout` en estado indefinido (`32{1'bX}`) y se desactiva `done`. Además, se recorren todas las posiciones del buffer para asignarles el valor X, limpiando así la memoria temporal. Finalmente, el módulo retorna al estado IDLE.

Listado 4.4: Definición del módulo y puertos

```
module bit_reverse_accel (
    ap_clk,
    ap_rst_n,
    s_axis_TDATA,
    s_axis_TVALID,
    s_axis_TREADY,
    s_axis_TKEEP,
    s_axis_TSTRB,
    s_axis_TLAST,
    m_axis_TDATA,
    m_axis_TVALID,
    m_axis_TREADY,
    m_axis_TKEEP,
    m_axis_TSTRB,
    m_axis_TLAST
);

//...

parameter ap_ST_fsm_pp0_stage0 = 4'd1;
parameter ap_ST_fsm_pp0_stage1 = 4'd2;
parameter ap_ST_fsm_pp0_stage2 = 4'd4;
parameter ap_ST_fsm_pp0_stage3 = 4'd8;

(* fsm_encoding = "none" *) reg [3:0] ap_CS_fsm;

//...

reg ap_rst_n_inv;
```

```
30 reg s_axis_TDATA_blk_n;
31 reg m_axis_TDATA_blk_n;
32 reg ap_enable_reg_pp0_iter1;
33
34 //...
35
36 assign m_axis_TDATA = {
37     s_axis_TDATA[0],  s_axis_TDATA[1],  s_axis_TDATA[2],
          s_axis_TDATA[3],
38     s_axis_TDATA[4],  s_axis_TDATA[5],  s_axis_TDATA[6],
          s_axis_TDATA[7],
39     ...
40     s_axis_TDATA[31]
41 };
42
43 //...
```

En 4.4 se presenta de forma resumida el código fuente generado por Vitis HLS de la función top del acelerador reverso de bits. Este módulo organiza la inversión de bits como un bloque *pipeline* con interfaz AXI-Stream, listo para integrarse en un sistema mayor. La estructura del código refleja las fases típicas de un diseño de hardware acelerado: interfaz, FSM, control de flujo y operación principal. Se ha dividido en diversas partes separadas por puntos cada uno de los apartados más importantes.

En la primera parte se definen los puertos del módulo:

- `ap_clk` y `ap_rst_n` son reloj y reset.
- Las señales `s_axis_*` representan la entrada AXI-Stream.
- Las señales `m_axis_*` representan la salida AXI-Stream.

En la segunda parte, se define la máquina de estados presente en el módulo, a juicio de la herramienta Vitis HLS; El diseño está dividido en cuatro etapas de *pipeline*, representadas como estados de la FSM. Esto permite procesar varios datos en paralelo, aprovechando el flujo continuo que ofrece AXI-Stream.

En la tercera parte, se definen las señales de control internas al módulo; estas señales sirven para manejar el control del flujo de datos:

- `ap_rst_n_inv` es el reset invertido.
- `s_axis_TDATA_blk_n` y `m_axis_TDATA_blk_n` indican bloqueo de entrada y salida.

- `ap_enable_reg_pp0_iter1` habilita la siguiente iteración en el pipeline.

En la úiltima parte, se implementa la inversión de bits: el valor de entrada de 32 bits (`s_axis_TDATA`) se reordena de manera que el bit menos significativo pase a ser el más significativo, y viceversa. Este patrón es muy usado en algoritmos como la Fourier Fast Transform (FFT), por ejemplo.

4.1.3. Iteración 3 - Ejecución de simulaciones funcionales

En esta sección se detallarán las simulaciones realizadas sobre el diseño.

Para realizar estas pruebas, ha sido necesario el desarrollo de archivos *testbench*, siendo este concepto explicado en **??**. La sección de simulaciones puede dividirse en dos apartados, las ejecuciones realizadas con Vivado y las realizadas con EdaPlayground, este último detallado en 2.1.5.

Simulaciones en Vivado

La herramienta Vivado, ya detallada previamente en 2.1.5, se ha empleado para realizar además simulaciones sobre el diseño, el cual fue creado dentro de esta herramienta.

Testbench desarrollado

El programa desarrollado para realizar las simulaciones en Vivado se presenta en 4.5.

Listado 4.5: Código fuente del testtbench para simular en Vivado

```
1  `timescale 1ns / 1ps
2
3  module tb_pipeline_blockdesign;
4
5      // Señales de testbench
6      reg clk = 1;
7      reg start_flag_i = 0;
8      reg [31:0] din_i = {32{1'bx}}; //dont care, dont enter
           into the fifo mem
9      wire [31:0] dout_o;            //we just want to see
          the output
10     wire done_flag_o;
11     integer i;
```

```
12
13      // Instancia del diseño completo
14      design_1_wrapper uut (
15          .start_flag_i(start_flag_i),
16          .din_i(din_i),
17          .dout_o(dout_o),
18          .done_flag_o(done_flag_o)
19      );
20
21      // Reloj 100 MHz
22      always #5 clk = ~clk;
23
24      initial begin
25          #20 //2 cycles
26          start_flag_i = 0;
27          #10;
28
29          // PUSHING DATA TO FIFO_IN AND THEN GETTING IT
                AGAIN
30          repeat(4) begin
31
32              // Escritura
33              @(posedge clk);
34              start_flag_i = 1;
35
36              for (i = 0; i < 4; i = i + 1) begin
37                @(posedge clk);
38                din_i = $random; // put random data into din
                      wire
39              end
40
41              @(posedge clk);
42              start_flag_i = 0;
43              wait(done_flag_o); //espero a que empiece a
                   haber datos
44
45              for (i = 0; i < 4; i = i + 1) begin
46                  @(posedge clk);
47                  $display("Reversed = %h", dout_o); // put
                       random data into din wire
48              end
49              wait(!done_flag_o); //termina la transmision
                   de datos
50          end
51          #100 $finish;
52       end
53  endmodule
```

El programa testbench funciona de la siguiente forma:

1. Al iniciar la simulación, el primer paso es la generación de un reloj. El reloj es una señal periódica (`clk`) que alterna su valor cada 5 unidades de tiempo, lo que da lugar a un ciclo de reloj de 10 unidades de tiempo, o 10 ns, correspondiente a una frecuencia de 100 MHz. Este reloj es fundamental para la sincronización de las señales dentro del diseño y el testbench. Una vez que el reloj está funcionando, el testbench pasa a ejecutar el bloque inicial (`initial`), donde se define la secuencia de pruebas. Primero, se realiza una espera de 20 unidades de tiempo para permitir que el sistema se estabilice antes de comenzar las pruebas. Durante este tiempo, el diseño bajo prueba no está recibiendo ninguna señal de entrada, para evitar inconsistencias.
2. Las pruebas comienzan como tal al activarse la señal `start_flag_i`. En cada ciclo de reloj, se envían 4 valores aleatorios a través de la señal de entrada `din_i`, simulando la escritura de datos en el diseño. Durante este proceso, el valor de la señal `din_i` se actualiza con datos generados aleatoriamente por el testbench. Después de enviar los 4 datos, se desactiva la señal `start_flag_i`, indicando que el proceso de escritura ha finalizado. Una vez que los datos se han enviado, el testbench espera que el diseño termine de procesarlos. Esto se logra mediante la señal `done_flag_o`, la cual es utilizada para indicar cuando el diseño ha completado su operación. El testbench espera a que esta señal se active para proceder a la siguiente fase.
3. Cuando la señal `done_flag_o` indica que el diseño ha terminado de procesar los datos y va a comenzar a mandar los resultados de vuelta, el testbench lee los resultados de la salida `dout_o`. Estos resultados, que son los datos procesados por el diseño, se muestran en la consola en formato hexadecimal utilizando la función `$display`. Este paso permite verificar si el diseño ha procesado correctamente los datos de acuerdo a lo esperado.
4. Después de mostrar los resultados, el testbench espera a que la señal `done_flag_o` se desactive, lo que indica que el diseño ha terminado completamente con la transmisión de los datos. Una vez cumplido este requisito, el testbench repite el ciclo de prueba 4 veces, lo que garantiza que el diseño se evalúe en diferentes condiciones y que su comportamiento sea consistente. Finalmente, después de completar todas las pruebas, el testbench espera 100 unidades de tiempo antes de finalizar la simulación utilizando `$finish`.

Resultados

El diagrama de onda resultado se presenta en la figura 4.2.

Figura 4.2: Simulación de Vivado vista con su visor integrado

Como puede apreciarse, el comportamiento es acorde a lo esperado en el programa *testbench* y en el diseño a probar: en la simulación se observa la señal de reloj `clk`, que marca el ritmo de todo el sistema. Al inicio, entre 0 ns y 40 ns, la señal de inicio `start_flag_i` está en cero, lo que significa que no se ha dado ninguna orden para comenzar. En ese mismo intervalo, la entrada de datos `din_i` y la salida `dout_o` permanecen en valores indefinidos (X), mientras que la señal de fin `done_flag_o` también está en cero. Después, la entrada `din_i` empieza a cargarse con valores concretos como `121...`, `c08...` y `648...`. El testbench está aplicando datos a la entrada para que el *pipeline* comience a procesarlos. En este punto, la salida `dout_o` sigue sin tener un valor válido. Esta latencia se debe precisamente a los ciclos necesarios para poder obtener los enteros con sus bits ya invertidos.

Más adelante, `done_flag_o` se pone a 1, y comienzan a aparecer cambios en la salida `dout_o`, que muestra los valores con la inversión aplicada, como `24a...`, `817...`, `906...` y `c66...`. Este comportamiento demuestra que ya está entregando resultados y que los valores aplicados previamente en la entrada están propagándose hasta la salida. Finalmente, la salida `dout_o` se estabiliza en el valor `89375212`, pues no se resetea ni se vuelve X. Este momento marca la finalización del procesamiento, y se queda a la espera de la siguiente vez que se suceda el ciclo, algo que puede verse al final en la propia figura. Esto ocurrirá otras dos veces más, siendo cuatro el total, tal y como se especifica en el testbench.

Simulaciones en EdaPlayground

EdaPlayground, herramienta ya detallada en 2.1.5, también ha sido empleada para realizar pruebas sobre el diseño.

Testbench desarrollado

El programa desarrollado para realizar las simulaciones en EdaPlayground es idéntico al de Vivado, presentado en 4.5, salvo por la introducción de *$dumpvars(0, tb_pipeline_blockdesign);* al inicio de la ejecución del *testbench*

Resultados

Tras aplicar dichas modificaciones, se ha realizado una simulación y descargado los resultados, para poder ver el diagrama de onda en la aplicación GTKwave, mencionada en 2.1.5. El resultado se presenta en la figura 4.3.

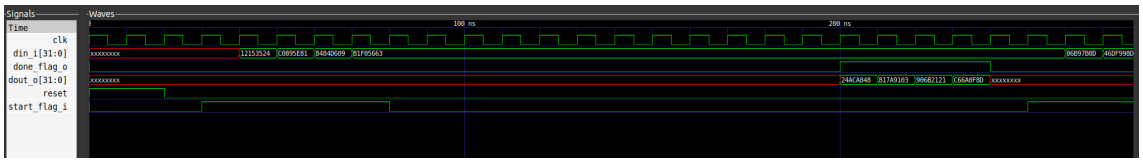

Figura 4.3: Simulación de EdaPlayground vista con GTKWave

Como puede apreciarse, el comportamiento es equivalente al del resultado realizado con Vivado; al inicio, durante el *reset*, las señales de entrada y salida están en estado indefinido (X). Una vez que el *reset* baja, se activa `start_flag_i` y el módulo comienza a capturar valores en `din_i`. Tras varios ciclos de reloj de procesamiento, aparecen datos válidos en `dout_o`, acompañados por la activación de `done_flag_o`, que indica que los resultados ya están listos. Finalmente, `done_flag_o` vuelve a cero y regresa a un estado indefinido (X). Más adelante, tal y como se puede ver al final del diagrama, se repite de nuevo todo el ciclo de operación, algo que ocurrirá otras dos veces más hasta llegar a cuatro, pues así se definió en el *testbench*.

4.1.4. Iteración 4 - Integración del diseño en X-HEEP

4.1.5. Iteración 5 - Ejecución del diseño en X-HEEP sobre FreeRTOS

4.1.6. Iteración 6 - Comparativa de resultados frente a versión software

Capítulo 5

Conclusiones y trabajo futuro

«Cita o lo que quieras»
Author

Resume los principales hallazgos y cómo responden a tus objetivos. Analiza si se han cumplido y si han existido limitaciones o problemas que han impedido alcanzar las metas planteadas inicialmente.

5.1. Líneas de trabajo futuro

Apéndice A

Apéndice A

Este es un ejemplo de Apéndice.

Referencias bibliográficas

[AF15] Suliman Al Freidi. A unified project management methodology (upmm) based on pmbok and prince2 protocols: foundations, principles, structures and benefits of the integrated approach. *International Journal of Business Policy and Strategy Management*, 2:27–38, 11 2015.

[AMD22] AMD. Digilent nexys a7. Recurso web: `https://www.amd.com/es/corporate/university-program/aup-boards/digilent-nexys-a7.html`, 2022. Accedido: 11-08-2025.

[AMD25a] AMD. Amd vitis hls empowers rtl designers with faster verification and design iteration. Recurso web: `https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html`, 2025. Página principal Vitis HLS. Accedido: 14-08-2025.

[AMD25b] AMD. Amd vitis hls user guide (ug1399). Recurso web: `https://docs.amd.com/r/en-US/ug1399-vitis-hls`, 2025. Accedido: 14-08-2025.

[AMD25c] AMD. Amd vivado design suite. Recurso web: `https://www.amd.com/es/products/software/adaptive-socs-and-fpgas/vivado.html`, 2025. Página de descarga. Accedido: 10-08-2025.

[AMD25d] AMD. Versal adaptive soc design guide (ug1273). Recurso web: `https://docs.amd.com/r/en-US/ug1273-versal-acap-design/System-Architecture`, 2025. Accedido: 11-08-2025.

[Auf19] Jean-Luc Aufranc. Freertos kernel now supports risc-v architecture. Recurso web: `https://www.cnx-software.com/2019/03/06/amazon-freertos-risc-v/`, 2019. Accedido: 10-08-2025.

[Awa22] Rahul Awati. What is an intellectual property core (ip core)? Recurso web: `https://www.techtarget.com/whatis/definition/IP-core-intellectual-property-core`, 2022. TechTarget. Accedido: 14-08-2025.

[Chi22] ChipVerify. What are top-level modules? Recurso web: `https://www.chipverify.com/verilog/verilog-modules`, 2022. Accedido: 15-08-2025.

[Com25] Wikipedia Community. Logo de freertos. Recurso web: `https://es.wikipedia.org/wiki/FreeRTOS#/media/Archivo:Logo_freeRTOS.png`, 2025. Accedido: 10-08-2025.

[Cor21] Intel Corporation. An 917: Reset design techniques for hyperflex architecture fpgas. Recurso web: `https://www.intel.com/programmable/technical-pdfs/683539.pdf`, 2021. Documento PDF. Accedido: 14-08-2025.

[EE24] ESL-EPFL. X-heep. Recurso web: `https://github.com/esl-epfl/x-heep`, 2024. Repositorio GitHub. Accedido: 13-08-2025.

[EE25a] ESL-EPFL. X-heep asic implementations. Recurso web: `https://x-heep.readthedocs.io/en/latest/ASIC/asic.html`, 2025. Accedido: 13-08-2025.

[EE25b] ESL-EPFL. X-heep documentation. Recurso web: `https://x-heep.readthedocs.io/en/latest/index.html`, 2025. Documentación de X-HEEP. Accedido: 13-08-2025.

[Eng22] Semiconductor Engineering. High-level synthesis (hls). Recurso web: `https://semiengineering.com/knowledge_centers/eda-design/verification/high-level-synthesis`, 2022. Accedido: 14-08-2025.

[EPF25] EPFL. Logo de x-heep. Recurso web: `https://github.com/esl-epfl/x-heep/blob/main/docs/source/images/x-heep-outline.png`, 2025. Accedido: 10-08-2025.

[Esp] Real Academia Española. Definición de metodología. Recurso web: `https://dle.rae.es/metodología`. Página web de la Real Academia Española. Accedido 18-08-2025.

[Eur24] Comisión Europea. Propuesta de modificación del reglamento (ue) 2021/2085. Recurso web: `https://eur-lex.europa.eu/legal-content/ES/TXT/HTML/?uri=CELEX:52022PC0047`, 2024. Accedido: 12-08-2025.

[For06] Edaboard Forum. What does ”wrapper”stand for? Recurso web: `https://www.edaboard.com/threads/what-does-wrapper-stand-for.63932/`, 2006. Accedido: 15-08-2025.

[Fou25] RISC-V Foundation. Risc-v gnu toolchain. Recurso web: `https://github.com/riscv-collab/riscv-gnu-toolchain`, 2025. Repositorio de GCC para RISC-V. Accedido: 18-08-2025.

[Fre25] FreeRTOS. Página web de freertos. Recurso web: `https://www.freertos.org/`, 2025. Accedido: 10-08-2025.

[Gro21] OpenHW Group. Obi v1.2. Recurso web: `https://raw.githubusercontent.com/openhwgroup/obi/188c87089975a59c56338949f5c187c1f8841332/OBI-v1.2.pdf`, 2021. Documento PDF. Accedido: 18-08-2025.

[HH21] S. Harris and D. Harris. *Digital Design and Computer Architecture, RISC-V Edition*. Elsevier Science, 2021.

[IEE01] IEEE. Ieee standard verilog hardware description language. Recurso web: `https://standards.ieee.org/ieee/1364/2052/`, 2001. Accedido: 14-08-2025.

[Int21] RISC-V International. About risc-v. Recurso web: `https://riscv.org/about/`, 2021. Accedido: 12-08-2025.

[ISD23] ISDI. Rtl verilog. Recurso web: `https://www.doulos.com/knowhow/verilog/rtl-verilog/`, 2023. Accedido: 14-08-2025.

[Jac00] Rumbaugh Jacobson, Booch. *El proceso unificado de desarrollo de software*. Addison Wesley, second edition, July 2000. ISBN: 978-8478290369.

[Jim23] Ormarys Jimenez. ¿qué es diseño de software y hardware? Recurso web: `https://www.iutepi.edu/que-es-diseno-de-software-y-hardware/#:~:text=En%20pocas%20palabras%2C%20se%20refiere%20al%20proceso,PCB%2C%20cables%2C%20resistencias%2C%20capacitores%20y%20mucho%20m%C3%A1s`, 2023. Accedido: 12-08-2025.

[Mic14] Microchip. 9 reasons why the vivado design suite accelerates design productivity. Recurso web: `https://www.xilinx.com/publications/prod_mktg/vivado/Vivado_9_Reasons_Backgrounder.pdf`, 2014. Accedido: 14-08-2025.

[Mic24] Microchip. Amd vivado design suite: Redefining fpga and soc development. Recurso web: `https://www.microchipusa.com/articles/amd/amd-vivado-design-suite-redefining-fpga-and-soc-development`, 2024. Accedido: 14-08-2025.

[MK04] C. Maxfield and I. Kerszenbaum. *The design warrior's guide to FPGAs*. Elsevier, 2004.

[MSM+24] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Miguel Peón-Quirós, and David Atienza. X-heep: An open-source, configurable and extendible risc-v microcontroller for the exploration of ultra-low-power edge accelerators, 2024.

[Pro25] Zephyr Project. Página web de zephyr. Recurso web: `https://www.zephyrproject.org/`, 2025. Accedido: 10-08-2025.

[Ray22] Adrián Raya. Amd completa la mayor adquisición del sector de los semiconductores con la compra de xilinx. Recurso web: https://www.eleconomista.es/tecnologia/noticias/11619399/02/22/AMD-completa-la-mayor-adquisicion-del-sector-de-los-semiconductores-con-la-co
html, 03 2022. elEconomista. Accedido: 09-08-2025.

[Rea20] RealDigital. A brief history of verilog. Recurso web: https://www.realdigital.org/doc/1946d210d1411b214203a6673322a61f, 2020. Accedido: 14-08-2025.

[Rin24] Fernando Rincón. Microsof stream - metodología. Recurso web: https://pruebasaluuclm-my.sharepoint.com/personal/fernando_rincon_uclm_es/_layouts/15/stream.aspx?id=%2Fpersonal%2Ffernando%5Frincon%5Fuclm%5Fes%2FDocuments%2Fmaster%2Fmetodologia%2Emp4&ga=1&referrer=StreamWebApp%2EWeb&referrerScenario=AddressBarCopied%2Eview%2E121e8790%2Daeb0%2D4659%2Da548%2Dc3808491aa3a, 2024. Vídeo de metodologías EDA. Accedido: 14-08-2025.

[Rom19] David Romano. A brief history of fpga. Recurso web: https://makezine.com/article/maker-news/a-brief-history-of-fpga/, 2019. Accedido: 11-08-2025.

[Sch02] R.J Schweers. Metodologías de diseño de hardware. Recurso web: https://sedici.unlp.edu.ar/bitstream/handle/10915/3835/2_-_Metodolog%C3%ADas_de_dise%C3%B1o_de_hardware.pdf?sequence=4&isAllowed=y, 2002. Documento PDF, Universidad de la Plata. Accedido: 10-08-2025.

[SER22] Cadena SER. La cátedra perte-chip llega a la uclm para la formación de expertos en microelectrónica y semiconductores. Recurso web: https://cadenaser.com/castillalamancha/2024/10/04/la-catedra-perte-chip-llega-a-la-uclm-para-la-formacion-de-expertos-en-microe
2022. Accedido: 12-08-2025.

[Sha23] Vishal Sharma. Fpga development made easy: A comprehensive guide to vivado and vitis hls. Recurso web: https://medium.com/@the_daft_introvert/fpga-development-made-easy-a-comprehensive-guide-to-vivado-and-vitis-hls-5f30
2023. Accedido: 14-08-2025.

[Sta99] Richard Stallman. make - gnu make utility to maintain groups of programs. Recurso web: https://linux.die.net/man/1/make, 1999. Manual de make. Accedido: 18-08-2025.

[Tec24] TechTarget. What is a real-time operating system (rtos)? Recurso web: https://www.techtarget.com/searchdatacenter/definition/real-time-operating-system, 2024. Accedido: 14-08-2025.

[Tor24] VLSI Lab Politecnico Di Torino. Gr-heep, a fork of x-heep. Recurso web: ttps://github.com/esl-epfl/x-heep, 2024. Repositorio de GR-HEEP. Accedido: 13-08-2025.

[Tos22] Frank Toshioka. Beneficios del hls. Recurso web: https://es.linkedin.com/advice/1/what-advantages-disadvantages-using-high-level?lang=es&lang=es&contributionUrn=urn%3Ali%3Acomment%3A%28articleSegment%3A%28urn%3Ali%3AlinkedInArticle%3A7046538086146580481%2C7046538089065787392%29%2C7169942824606535681%29&articleSegmentUrn=urn%3Ali%3AarticleSegment%3A%28urn%3Ali%3AlinkedInArticle%3A7046538086146580481%2C7046538089065787392%29&dashContributionUrn=urn%3Ali%3Afsd_comment%3A%287169942824606535681%2CarticleSegment%3A%28urn%3Ali%3AlinkedInArticle%3A7046538086146580481%2C7046538089065787392%29%29, 2022. Accedido: 14-08-2025.

[WH15] N.H.E. Weste and D. Harris. *CMOS VLSI Design : A circuits and systems perspective*. Pearson, 2015.

[Wik23] Wikipedia. Hardware description language. Recurso web: https://en.wikipedia.org/wiki/Hardware_description_language, 2023. Accedido: 14-08-2025.

[Win23] Windriver. What is a real-time operating system (rtos)? Recurso web: https://www.windriver.com/solutions/learning/rtos, 2023. Accedido: 14-08-2025.

[Xat21] Xataka. El primer chip risc-v europeo cobra vida: Epac está destinado a supercomputadoras, pero esto es solo el principio. Recurso web: https://www.xataka.com/componentes/primer-chip-risc-v-europeo-cobra-vida-epac-esta-destinado-a-supercompu 2021. Accedido: 12-08-2025.

[Xil23] Xilinx. Vivado design suite user guide: Designing with ip (ug896). Recurso web: https://docs.amd.com/r/2023.2-English/ug896-vivado-ip/IP-Integrator, 2023. Accedido: 16-08-2025.

[A25] Ángel Aller. Nvidia cuda soportará la arquitectura risc-v, ¿se viene una revolución? Recurso web: https://www.profesionalreview.com/2025/07/21/nvidia-cuda-risc-v/, 2025. Accedido: 10-08-2025.