
UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

UNIVERSIDAD DE MÁLAGA

Grado en Ingeniería de Computadores

# Diseño de un acelerador en plataforma RISC-V para el algoritmo DTW

# Accelerator design for the DTW algorithm in RISC-V platform

*Realizado por*

Raúl Carriba Merino

*Tutorizado por*

Dr. Francisco Javier Hormigo Aguilar

*Departamento*

Arquitectura de Computadores

Málaga, Noviembre, 2025

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DE COMPUTADORES

# Diseño de un acelerador en plataforma RISC-V para el algoritmo DTW

# Accelerator design for the DTW algorithm in RISC-V platform


Realizado por

**Raúl Carriba Merino**

*Tutorizado por*

**Dr. Francisco Javier Hormigo Aguilar**

*Departamento*

**Arquitectura de Computadores**



UNIVERSIDAD DE MÁLAGA

MÁLAGA, NOVIEMBRE, 2025

Fecha defensa: Diciembre de 2025

# Resumen

Este trabajo de fin de grado tiene como objetivo principal desarrollar un acelerador hardware para el algoritmo *Dynamic Time Warping* compatible con la plataforma de desarrollo *X-Heep*, así como analizar su rendimiento en comparación con ejecutar el algoritmo en software a través del microcontrolador de propósito general del sistema. Para ello, primero se ha realizado la implementación del acelerador en SystemVerilog, para posteriormente adaptarlo como un periférico interno dentro de X-Heep. También se ha desarrollado un driver en C para poder controlarlo y así realizar las pruebas pertinentes. A mayores, se ha realizado la síntesis del hardware para validar su corrección, así como comprobar el consumo del conjunto. Los resultados obtenidos indican que con el uso del acelerador se obtiene una mejora de media de 23.16 veces más rendimiento con respecto a la ejecución de la versión software de igual características por parte del microcontrolador del sistema, además de mejorar la eficiencia energética del cálculo.

**Palabras clave:** Dynamic Time Warping, DTW, X-Heep, SystemVerilog, Acelerador hardware, Rendimiento, Síntesis

# Abstract

The main objective of this final degree thesis is to develop a hardware accelerator for the *Dynamic Time Warping* algorithm compatible with the *X-Heep* development platform, as well as to analyze its performance in comparison with running the algorithm via software through the system's general-purpose microcontroller. To this end, the accelerator was first implemented in SystemVerilog to later adapt it as an internal peripheral within X-Heep. A driver was also developed in C to control it and perform the pertinent tests. In addition, the hardware was synthesized to validate its correctness and check the overall power consumption. The results obtained reveal that using the accelerator improves performance by a factor of 23.16 on average compared to the execution of the software version with the same characteristics by the system's microcontroller, as well as improving the energy efficiency of the calculation.

**Keywords:** Dynamic Time Warping, DTW, X-Heep, SystemVerilog, Hardware Accelerator, Performance, Synthesis

# 1 Introducción

## 1.1 Motivación

En el ámbito de la medicina, una de las opciones que existen para detectar ciertas patologías, sobre todo las relacionadas con el sistema vascular y el sistema nervioso, es el análisis de señales biomédicas[17]. En los casos comentados antes, lo que se hace es analizar las señales y, en base a los resultados, poder actuar de la mejor manera posible para el paciente. Cuando el paciente se encuentra en observación, estos análisis se pueden realizar con máquinas hospitalarias de gran tamaño y potencia. Llevar a cabo este tipo de seguimiento en el día a día con ese tipo de maquinaria sería inviable debido a la complejidad que esto encierra. Por eso, existen dispositivos de bajo consumo y tamaño reducido, conocidos como "wearables", pensados para que el paciente los lleve en todo momento y permanezca monitorizado.

Para este tipo de análisis, uno de los métodos que se usan es comparar la señal proveniente del paciente con un patrón para poder determinar si existe dicha patología. Esto se hace hoy en día a través de algoritmos, y en este tipo de casos, uno de los que mejores resultados ha dado a la hora de detectar este tipo de patrones en señales fisiológicas es Dynamic Time Warping (DTW)[9]. Su capacidad para alinear señales desfasadas temporalmente, lo convierte en una opción atractiva para esta tarea. Sin embargo, su elevada complejidad computacional limita su uso en sistemas embebidos con recursos restringidos.

Por esta razón, surge la necesidad de desarrollar un acelerador específico para el cálculo del algoritmo DTW, de manera que se pueda ejecutar en tiempo real con las limitaciones de un dispositivo "wearable". Esto se integraría dentro de un proyecto cuyo objetivo es detectar ataques epilépticos, en el cual se usa X-Heep, como plataforma de desarrollo. Dicha plataforma permite implementar la mayor parte del algoritmo en tiempo real, excepto la DTW, que se ejecutará por parte del acelerador hardware desarrollado en este proyecto.

## 1.2 Objetivos

Este trabajo de fin de grado tiene como objetivo principal el diseño y desarrollo de un acelerador hardware para el cálculo del algoritmo DTW que sea compatible con la plataforma de desarrollo X-Heep, así como analizar su rendimiento en comparación con ejecutar el algoritmo en software a través del microcontrolador de propósito general del sistema. Todo el diseño del acelerador se desarrollará en SystemVerilog. Otro objetivo es que además sirva como guía para agregar el acelerador como un periférico interno al sistema completo de X-Heep y cómo utilizarlo. Con el fin de alcanzar dichos objetivos, se han seguido los siguientes pasos:

- Estudio del algoritmo Dynamic Time Warping.
- Estudio de las optimizaciones que se pueden aplicar al algoritmo.
- Definición de requisitos para adaptar el acelerador a X-Heep
- Diseño del acelerador básico y comprobación de su funcionamiento.
- Estudio en profundidad de la plataforma X-Heep.
- Adaptación e integración del acelerador en X-Heep.
- Sintetización del acelerador hardware y estimación del área.
- Validación del funcionamiento del acelerador dentro de la plataforma y estimación del rendimiento.

## 1.3 Estructura del documento

- **Sección 1. Introducción**: Contextualiza el proyecto, expresa los objetivos de este y la estructura de este documento. Además, se hace una explicación de algunos conceptos interesantes.
- **Sección 2. Estado del Arte**: Introduce y explica en profundidad el algoritmo a acelerar en este proyecto y la plataforma hardware en la que se va a trabajar.
- **Sección 3. Diseño e implementación**: Explica el proceso de diseño seguido para cada una de las partes de las que se compone el acelerador.
- **Sección 4. Integración en X-Heep**: Expone el proceso seguido para integrar el acelerador dentro de la plataforma X-Heep.

- **Sección 5. Síntesis, validación y verificación de resultados**: Muestra la creación de las pruebas y los resultados obtenidos de estas y del proceso de sintetización.
- **Sección 6. Conclusiones y trabajos futuros**: Enuncia las conclusiones deducidas a partir de los resultados obtenidos y, además, propone mejoras y ampliaciones futuras al proyecto.
- **Apéndice A. Guía de instalación**: Apartado que sirve de guía al usuario para poder acoplar el acelerador dentro de X-Heep.

## 1.4 Conocimientos previos

Este apartado resume algunos conceptos generales para ayudar a dar contexto al proyecto. A continuación, se presentan brevemente los más relevantes:

**Aceleradores hardware:** Son bloques diseñados específicamente para realizar una tarea en concreto con menor latencia y, a menudo, con menor consumo de lo que lo haría un procesador de propósito general. En este contexto, permiten cumplir las restricciones de energía y tiempo, lo es que fundamental para dispositivos tipo "wearable".

**Dispositivos wearable:** Son dispositivos electrónicos destinados a ser llevados sobre el cuerpo, capaces de capturar datos fisiológicos o de actividad. Operan con baterías pequeñas y memoria limitada, por lo que priorizan eficiencia energética, tamaño reducido y procesamiento en tiempo real.

**Arquitectura RISC-V:** RISC-V es una arquitectura de conjunto de instrucciones (ISA) de hardware libre. Este ISA fue diseñado para realizar implementaciones pequeñas, rápidas y de bajo consumo. Su enfoque en la simplicidad y escalabilidad la convierte en una plataforma atractiva para investigación y desarrollo de hardware [22].

**SystemVerilog:** SystemVerilog es un lenguaje de descripción y verificación de hardware que se utiliza habitualmente para modelar, diseñar, simular, probar e implementar sistemas electrónicos en la industria del diseño electrónico y de semiconductores. SystemVerilog es una extensión de Verilog, añadiendo todo el sistema de verificación.

Los fundamentos específicos del algoritmo y de la plataforma objetivo se desarrollan en la siguiente sección.

# 2

# Estado del Arte

En este capítulo se presentan los conceptos teóricos, tecnologías y trabajos previos relevantes para el desarrollo del acelerador diseñado en este proyecto. Se analizan distintas variantes del algoritmo DTW, sus optimizaciones y aplicaciones, así como estrategias de aceleración hardware documentadas en la literatura. Además, se hace un análisis de la plataforma hardware donde se acoplará dicho acelerador. El objetivo es situar el proyecto en su contexto técnico y justificar la elección de la arquitectura, el método de cálculo y el enfoque de diseño seguidos posteriormente.

## 2.1 Dynamic Time Warping (DTW)

El DTW es un algoritmo utilizado para medir la similitud entre dos secuencias temporales[1], las cuales pueden encontrarse desfasadas en el tiempo o tener diferentes velocidades. Su principal ventaja frente a otros algoritmos al comparar dichas series es que permite alinear las señales de forma no lineal. A pesar de que su nombre implica series temporales, no es necesario que sea así, ya que, mientras sean series de datos, se puede aplicar este algoritmo.

Para medir dicha similitud entre las dos series de entrada, este algoritmo busca, para cada muestra de una serie, su correspondencia más adecuada en la otra, de modo que pueda reconocer formas similares aunque estén desfasadas o deformadas en el tiempo. El procedimiento construye primero una matriz de distancias con todas las comparaciones punto a punto entre ambas series y, sobre esa matriz, selecciona un recorrido óptimo: la ruta cuya suma acumulada de costes es mínima. Ese recorrido debe cumplir estas restricciones:

- Todos los valores de una de las series deben ser comparado con uno o varios valores de la otra serie.
- El primer valor de una serie debe compararse con el primer valor de la otra serie, aunque no tiene que ser de manera exclusiva, también puede coincidir con más valores.

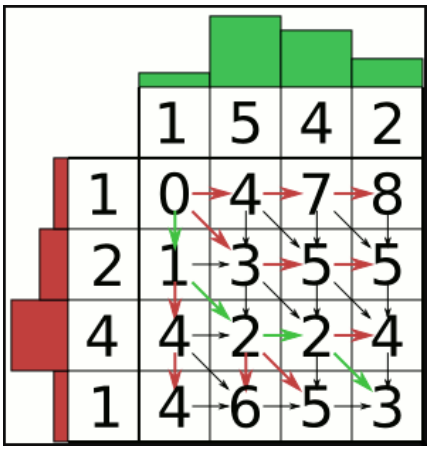


***Figura 2.1.*** *Ejemplo de matriz DTW y recorrido óptimo.*

Figura extraída de: [10].

- El último valor de una serie debe compararse, al menos, con el último valor de la otra serie y, al igual que en el punto anterior, puede haber más de una coincidencia.

- La función de alineamiento debe ser monótonamente creciente en ambas series, de manera que las correspondencias respeten el orden temporal y se mantenga la continuidad del recorrido.

El recorrido óptimo es, por tanto, la ruta que satisface todas esas condiciones minimizando el coste, donde el coste se define, de manera básica, como la suma de las diferencias absolutas entre los pares alineados. También se puede emplear la suma de la distancia euclídea al cuadrado. Esta sería la forma estándar de ejecutar el algoritmo, y podríamos decir que cada celda se calcula de la siguiente manera:

$$C(i,j) = d(a_i, b_j) + \min\{\, C(i-1,j),\ C(i,j-1),\ C(i-1,j-1)\,\},$$

donde $d(a_i, b_j)$ es la distancia entre los pares de datos de entrada para esa celda. En la Figura 2.1 se representa el resultado de aplicar el algoritmo a un par de series de 4 elementos, representando el camino óptimo de alineación entre las series donde, en este caso, el coste es la diferencia absoluta entre los pares.

Dado el esquema de cálculo sobre la matriz de costes acumulados, cada celda $(i, j)$ requiere computar su coste local y el mínimo entre tres predecesoras inmediatas. Por tanto, el número de operaciones crece con el número de celdas a visitar, por lo que para dos series de igual longitud $N$, queda una matriz de tamaño $N \times N$ y la complejidad temporal es $O(N^2)$ (en general $O(N \cdot M)$ si tenemos series de distinto tamaño). Ocurre de manera similar con la cantidad de memoria a utilizar, aunque solo en el caso de querer almacenar la matriz completa, pues usando *buffers* para almacenar

los resultados parciales se reduce el uso de memoria. Este elevado coste computacional tanto en tiempo como en memoria motiva la búsqueda de optimizaciones estructurales y restricciones de búsqueda.

Estas optimizaciones pueden consistir en limitar el espacio de búsqueda dentro de la matriz de costes mediante el uso de ventanas selectivas, como es el caso de la ventana de Sakoe–Chiba[19] o del paralelogramo de Itakura[6], cuyos objetivos son limitar la deformación máxima y reducir el número de celdas a calcular.

En el caso de la ventana de Sakoe–Chiba, se limita el cálculo a una banda de ancho constante alrededor de la diagonal principal de la matriz. Esto se hace fijando un radio $R$ y se exige evaluar solo las celdas $(i, j)$ de la matriz tales que $|i - j| \leq R$, asegurando de esta manera que se evalúan celdas cercanas a la diagonal. El resultado de su aplicación lo podemos ver en la imagen *a* de la Figura 2.2. El coste aproximado para series de entrada de igual longitud pasa de ser $O(N^2)$ a $O\big((2R + 1) \cdot N\big)$. Esta implementación tiene la ventaja de ser sencilla de implementar; sin embargo, si se aplica un radio demasiado pequeño, se corre el riesgo de excluir alineaciones válidas.

En el caso del paralelogramo de Itakura, se limita el cálculo a una región de ancho variable con forma de paralelogramo, siendo más estrecha en los extremos y más ancha en la zona central. Dicha forma la podemos ver en la imagen *b* de la Figura 2.2. Se formula acotando la razón de avance entre ejes mediante una pendiente máxima $s \geq 1$ tal que $\frac{1}{s} \leq \frac{\Delta j}{\Delta i} \leq s$, lo que restringe una región inicial y final que evita grandes desplazamientos y otra región central que tolera pequeñas desviaciones. Esta implementación tiene la ventaja de anclar mejor los extremos, evitando comportamientos extraños en estas zonas de la matriz, además de permitir flexibilidad en la zona central donde típicamente hay más diferencias entre las series; aunque presenta las desventajas de proporcionar una reducción no tan uniforme como Sakoe–Chiba y que, si se aplica una pendiente muy pequeña, puede ser demasiado restrictiva, y si se aplica una pendiente muy grande la ventana resultante es similar al caso sin restricción.

También existen versiones del algoritmo pensadas para ser aceleradas mediante cálculo por unidades de procesamiento gráfico (GPUs)[20], que permite aprovechar mucho mejor el paralelismo que estos dispositivos ofrecen. Estos algoritmos se basan en la realización del cálculo de la matriz por antidiagonales, ya que, al calcular las celdas de esta manera, cada una puede procesarse de manera independiente, pues las celdas necesarias para hallar el mínimo se encuentran en antidiagonales anteriormente calculadas, evitando dependencia de datos.

El recorrido que se lleva a cabo con este método difiere del tradicional, donde se va procesando la matriz por filas o por columnas, recorriendo todos los elementos de esta y una vez acaba, se pasa a

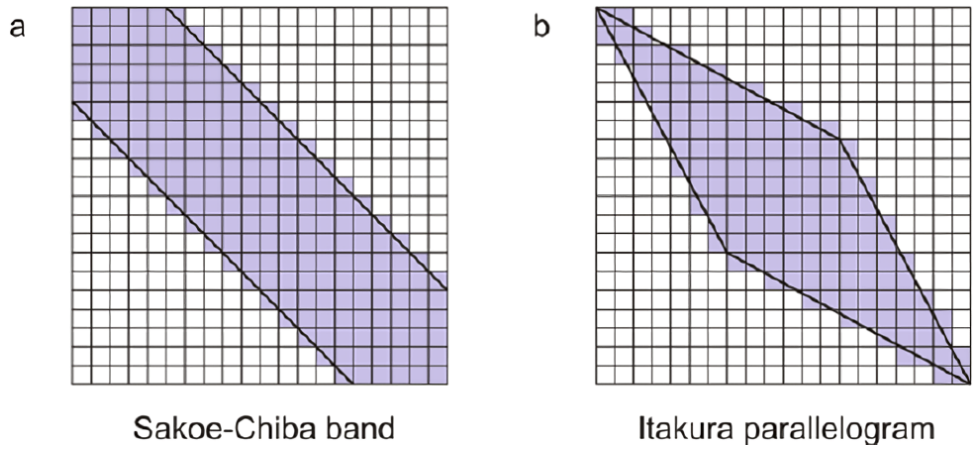


***Figura 2.2.*** *Restricciones aplicadas por la ventana de Sakoe-Chiba y paralelogramo de Itakura.*

Figura extraída de: [4].

la siguiente. En su lugar, lo que se hace es agrupar los elementos de la matriz según la suma de sus índices, es decir, todas las celdas que cumplan que $i + j = k$ pertenecen a la mista antidiagonal, donde $i$ es el índice de fila, $j$ el de columna y $k$ el índice que identifica cada antidiagonal de la matriz, siendo este valor como máximo $M + N - 1$, siendo $M$ y $N$ el ancho y alto de la matriz. Por este método se comienza desde la esquina superior izquierda *(0,0)* y avanza diagonalmente de arriba hacia abajo en la antidiagonal hasta la esquina inferior derecha *(M-1, N-1)*. Esta forma de recorrer la matriz la podemos ver en la Figura 2.3, donde la suma de los índices de las celdas coincide con la antidiagonal en la que se encuentra, empezando a contar desde el 0. Este tipo de aproximación también son usadas a la hora de implementar aceleradores hardware.

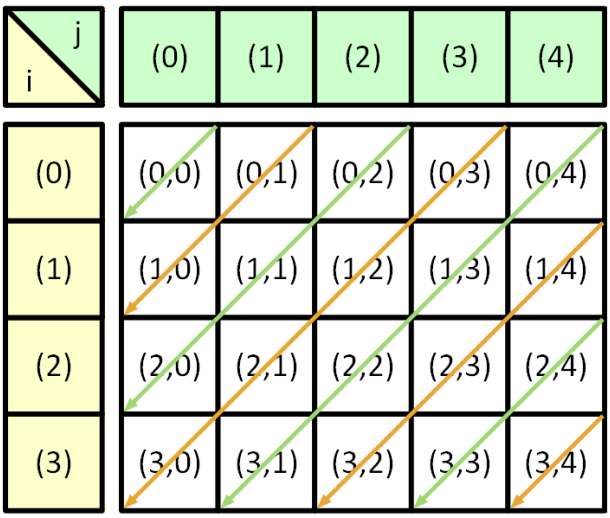


***Figura 2.3.*** *Recorrido de una matriz 5x4 mediante antidiagonales.*

Este algoritmo se ha usado ya en varias aplicaciones como el reconocimiento del habla [19] o de la

escritura a mano [3], además de la detección de anomalías médicas mediante el procesamiento de señales cardíacas y cerebrales [2, 23].

## 2.2 X-Heep

X-Heep es una plataforma de hardware abierto basada en RISC-V diseñada para el desarrollo y evaluación de sistemas embebidos de muy baja potencia [18]. Está orientada a la investigación y al prototipado rápido de arquitecturas heterogéneas, además de tener un uso enfocado en aplicaciones de *edge computing* o sistemas de ultra-bajo consumo, típicos en *Internet of Things* (IoT) o en sensores inteligentes embebidos. Se caracteriza por su filosofía completamente abierta, puesto que tanto el hardware como el software son accesibles, modificables y documentados. Además, permite la incorporación de hardware propio o aceleradores personalizados, lo que la hace ideal para experimentar con diseños. Está diseñada y escrita en SystemVerilog.

Esta plataforma puede dividirse en los siguientes subsistemas: subsistema de CPU, subsistema de memoria, subsistema de periféricos y subsistema de periféricos siempre activos. Los diferentes componentes se agrupan de esta manera, con el objetivo de maximizar el ahorro energético. Cuenta con diseños reutilizados de grandes proyectos de código abierto, tales como OpenHW, PULP y lowRISC [12, 16, 7].

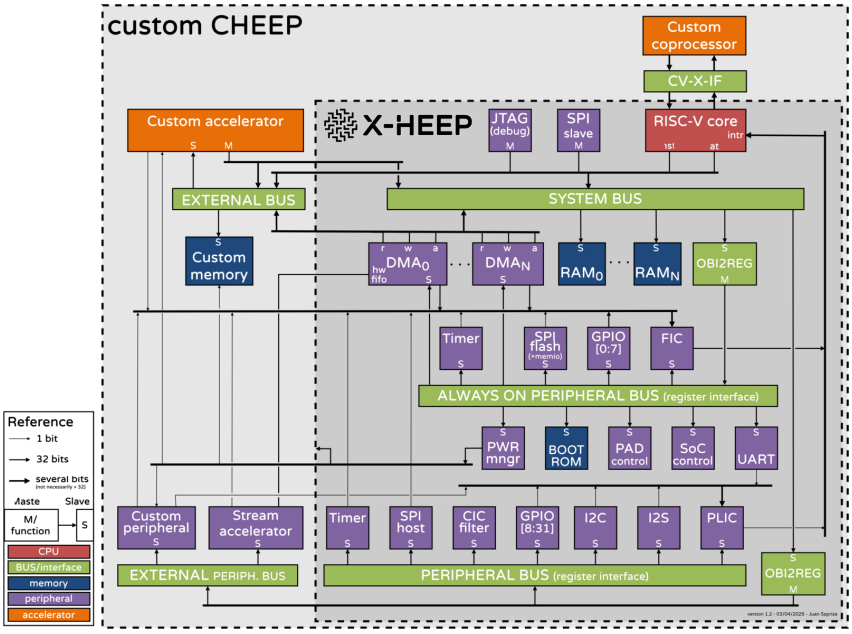


***Figura 2.4.*** *Diagrama de X-Heep.*

Figura extraída de: [25].

Desglosando cada subsistema, el de CPU se basa en núcleos RISC-V de 32 bits como CVE2, CV32E40P y sus variantes CV32E40PX y CV32E40X, que cuentan con una interfaz extra llamada

*Core-V eXtension InterFace* (CV-X-IF), explicada más adelante. Están diseñados con el bajo consumo y la simplicidad en mente, no cuentan con memoria caché y utilizan una arquitectura Harvard, donde existen dos buses diferenciados para instrucciones y datos, implementando este último el protocolo *Open Bus Interface* (OBI). Este subsistema puede ser apagado si esta sin uso por largos periodos de tiempo.

En cuanto al subsistema de memoria, está dividido en distintos bancos de distinto tamaño y pueden ser utilizados tanto por instrucciones como datos. Cada banco de memoria está conectado al sistema con un bus dedicado, lo que permite acceso simultaneo a varios bancos sin que esto genere conflicto. Al igual que en el subsistema anterior, para mejorar la eficiencia energética, se permite apagar o retener cada banco por separado.

El subsistema de periféricos contiene periféricos de uso general que no son usados continuamente durante el procesamiento. En concreto, incluye un temporizador de uso general, un controlador de interrupciones (PLIC), una interfaz de comunicación entre circuitos integrados ($I^2C$), una interfaz periférica serial (SPI) y 24 entradas-salidas de propósito general (GPIO). Este subsistema se conecta al bus por una única interfaz, aunque cuenta con una capa de decodificación para redirigir las peticiones al correspondiente periférico. También puede ser apagado o encendido a voluntad para minimizar el consumo.

Por último, el subsistema de periféricos siempre encendido contiene todos los periféricos que son indispensables para el funcionamiento de la plataforma. Aquí se encuentran el controlador del sistema, la ROM de arranque, el gestor de energía, un controlador rápido de interrupciones y la controladora de acceso directo a memoria (DMA). Existen algunos otros periféricos en este subsistema (otro temporizador de propósito general, un receptor-transmisor asíncrono universal [UART], dos SPIs y 8 GPIOs) importados de otros proyectos. Como indica su nombre, a este subsistema no se aplica ninguna estrategia de control de energía [25].

Como se mencionó anteriormente, la plataforma usa OBI como bus principal de datos, aunque también es utilizado para la comunicación punto a punto entre la CPU y los distintos periféricos del sistema. Es una interfaz síncrona y de tipo maestro–esclavo, que define señales para direccionamiento, control, datos y validación de transacciones. Su diseño busca ser ligero y de baja latencia, estando inspirada en AMBA AXI, pero simplificándola. En concreto, cuenta con dos canales de comunicación: el canal de dirección (A Channel) y el canal de respuesta (R Channel). Cada canal utiliza un mecanismo de "handshake" bidireccional que garantiza la transferencia segura de datos sin necesidad de arbitraje complejo [13].

- **Canal A (Address Channel)**: transporta la información de la solicitud (dirección, tipo de operación, datos de escritura, máscaras de byte, etc.). Su control se realiza mediante las señales *req* (request) y *gnt* (grant). El maestro inicia la transferencia cuando *req* está activo, y el esclavo la acepta al afirmar *gnt*.

- **Canal R (Response Channel)**: gestiona la fase de respuesta, donde el esclavo devuelve los datos de lectura (*rdata*) o códigos de error (*err*). El control del canal se realiza con *rvalid* y *rready*, que permiten al esclavo indicar que la respuesta está lista y al maestro confirmar que puede recibirla.

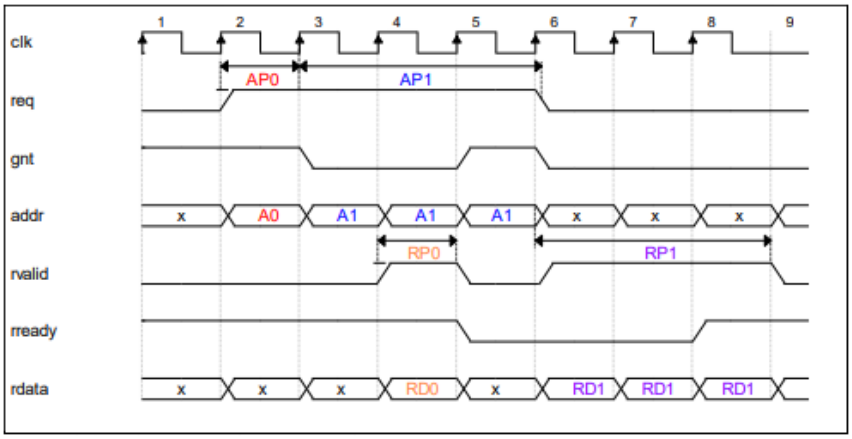


***Figura 2.5.*** *Fases de comunicación del bus OBI.*

Figura extraída de: [13].

Las transacciones OBI se completan en dos fases (dirección y respuesta), y siempre de forma *in order*, es decir, las respuestas se emiten en el mismo orden en que se solicitaron, aunque no necesariamente de manera inmediata, como se puede ver en la Figura 2.5, donde la respuesta *RP0* se realiza después de *AP0* existiendo entre ambas fases un ciclo de *AP1*.

Observando el diagrama de la Figura 2.4, se identifican dos bloques denominados *OBI2REG*, que actúan como interfaces puente entre el bus OBI y los periféricos. Su función es direccionar los registros internos de estos módulos a través del protocolo OBI. Esto resulta coherente, ya que los periféricos de la plataforma disponen de una interfaz denominada *Register Interface* (REG), cuyo propósito es permitir el acceso simplificado a los registros de control y estado. Los bloques *OBI2REG* se encargan, por tanto, de traducir las transacciones OBI en operaciones de lectura o escritura sobre dichos registros. Las señales de REG mantienen una estructura muy similar a las de OBI, aunque están especializadas para esta tarea concreta.

Cabe destacar que esta descripción se ha elaborado a partir del análisis de la arquitectura y los diagramas disponibles, ya que no existe documentación oficial que defina de forma explícita el funcionamiento interno de la interfaz REG. En la plataforma de X-Heep, esta interfaz se implementa a partir del repositorio público *register_interface* del ecosistema PULP/OpenHW [15].

Para añadir nuevos módulos hardware a la plataforma, esta ofrece varias opciones para dicho cometido. Existe la antes mencionada CV-X-IF, pensada para acoplar coprocesadores con instrucciones personalizadas y poder ampliar el conjunto de instrucciones de la CPU para realizar tareas especificas. Su manera de funcionar consiste en tener un canal de comunicación entre la CPU de X-Heep y el coprocesador, de manera que cuando se detecta una instrucción no reconocida por la CPU, esta es pasada al coprocesador que arranca con su tarea. Otra opción es utilizar *eXtendible Accelerator InterFace* (XAIF), que consiste en una interfaz conectada al bus del sistema por la cual la CPU, a través de registros mapeados por memoria, puede acceder al acelerador y configurarlo o leer su estado. Por este mismo bus, el acelerador puede tener acceso a la memoria del sistema o la DMA para poder recibir los datos y operar. Además, también cuenta con líneas de interrupción, de manera que pueda avisar a la CPU una vez acabado el procesamiento o en caso de que haya habido algún error. También se pueden añadir aceleradores o nuevos periféricos a los periféricos ya existentes dentro de la plataforma. Funcionaría de manera muy similar a XAIF, con la diferencia de que en esta última el canal es único para el acelerador añadido y al añadirlo como periférico interno, competiría por el canal de comunicación con el resto de periféricos del sistema. Esta última opción también implica modificar los ficheros de SystemVerilog que describen ese subsistema de la plataforma.

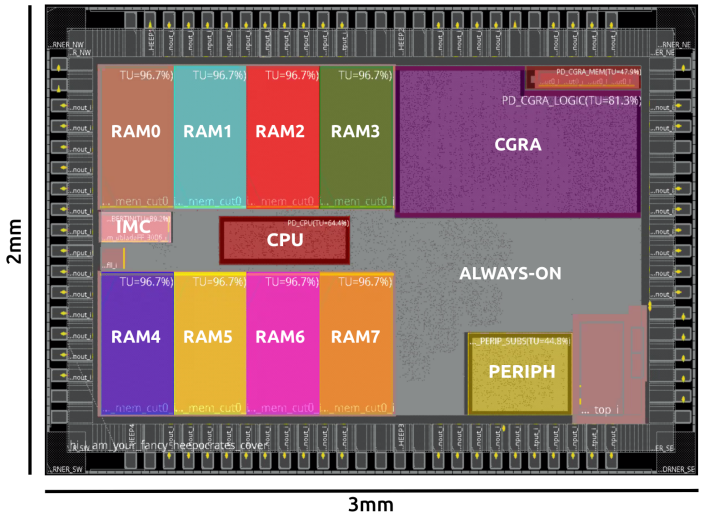


***Figura 2.6.*** *Diseño del chip de prueba HEEPocrates.*

Figura extraída de: [8].

Esta plataforma ya ha sido utilizada en proyectos como *HEEPocrates*, que fue la primera implementación en silicio de la plataforma. Su diseño físico se puede ver en la Figura 2.6, donde se ve la disposición de los diferentes elementos de lo que está compuesto el chip. A la plataforma se le añadió un *Coarse Grained Reconfigurable Array* (CGRA), que consiste en una matriz bidimensional de pequeños elementos procesadores (PE) interconectados entre sí, de manera que pueden obtener datos tanto de la memoria, sus propios registros o los registros de otro PE. Fue implementado en el proceso de fabricación de 65 nm LP CMOS de TSMC con el objetivo de ser usado para aplicaciones médicas de ultra bajo consumo [8].

# 3

# Diseño e implementación

Este capítulo describe el proceso seguido para diseñar e implementar el acelerador hardware del algoritmo DTW. Se detalla la arquitectura seleccionada, las decisiones de diseño adoptadas, los módulos que componen el hardware y su funcionamiento interno. El propósito es ofrecer una visión clara y completa del funcionamiento del acelerador, desde su concepción hasta su implementación final en SystemVerilog.

## 3.1 Diseño general

El acelerador diseñado para este proyecto tiene como objetivo principal realizar el cálculo completo del algoritmo DTW en hardware de forma eficiente, reduciendo el tiempo de ejecución y el consumo energético respecto a una implementación puramente software ejecutada sobre microprocesadores de propósito general. De manera general, el acelerador se compone de tres módulos principales:

- **Interfaz de registros**: recibe, decodifica y almacena las configuraciones enviadas desde la CPU para poder iniciar el cálculo del algoritmo. Además, genera interrupciones en caso de finalización o de fallo en la configuración.

- **Interfaz de memoria**: realiza las transacciones necesarias para leer las series de entrada con las que se calcula el algoritmo y escribir el resultado obtenido.

- **Unidad de cómputo**: almacena temporalmente las series leídas y ejecuta el algoritmo, devolviendo el resultado para su posterior escritura.

Estos tres módulos se encuentran interconectados mediante un módulo superior (*top*), cuya finalidad es redirigir las señales de entrada y salida de cada bloque, permitiendo una comunicación efectiva entre ellos. Además, este módulo proporciona la conexión con la plataforma X-Heep a través de los buses de datos y las líneas de interrupción, junto con las señales de *reset* y de reloj. Todo esto se puede ver representado en la Figura 3.1

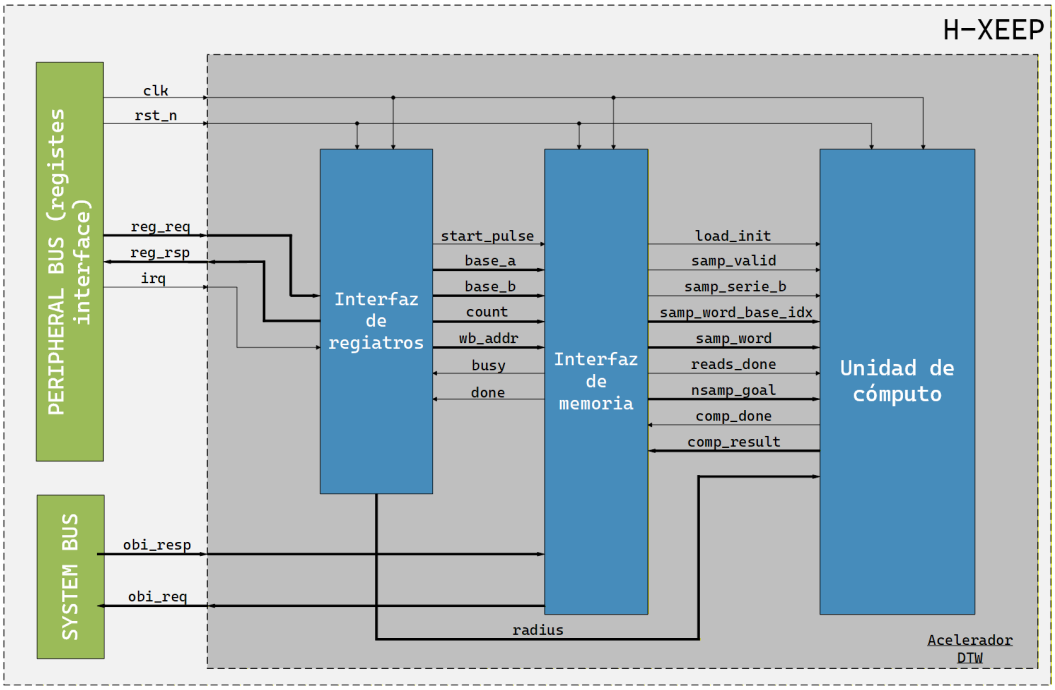


***Figura 3.1.*** *Diagrama del acelerador para DTW.*

Este acelerador opera con enteros con signo de 16 bits como datos de entrada, formato suficiente para representar las señales de entrada, aunque internamente se extienden en signo a 32 bits para los cálculos intermedios del algoritmo. Debido a que la plataforma X-Heep y su CPU son de 32 bits, se aplica una técnica de empaquetado de manera que en cada palabra de 32 bits contiene dos muestras de 16 bits útiles para el acelerador. De esta forma se mejora la eficiencia a la hora de realizar las lecturas a la memoria, reduciendo así la latencia de esta fase. Importante destacar que se optó por un diseño *little-endian*, donde el primer dato a leer se encuentra en los 16 bits menos significativos de la palabra y el segundo dato en los 16 más significativos. También es conveniente recalcar que el acelerador opera con un máximo de 1024 muestras por serie, siendo este un dato configurable a través de los registros del acelerador.

En cuanto al funcionamiento del cálculo, el acelerador implementa una estrategia basada en el procesamiento por antidiagonales, de forma que los elementos de la matriz de distancias puedan ser calculados de manera independiente sin que existan esperas por dependencias de datos. Adicionalmente,

se ha incorporado una restricción de tipo ventana, inspirada en la ventana de Sakoe-Chiba, que limita la región de la matriz en la que se realiza el cálculo a partir de un parámetro de radio (R), que se configura también a través de un registro. Dicha ventana llega a tener un ancho máximo de $2R + 1$ celdas. Con ello se reduce de manera significativa la cantidad de operaciones necesarias. Estas decisiones a la hora del diseño se tomaron con el objetivo de que el acelerador se pudiera adaptar correctamente a la plataforma de X-Heep y que funcionase de manera correcta y eficiente sin comprometer la precisión de los resultados que arroje, teniendo en cuenta las limitaciones del sistema.

El proceso de ejecución del algoritmo por parte del acelerador comienza con la configuración de sus registros por parte de la CPU de la plataforma. En el caso de que algún parámetro sea erróneo, se genera una interrupción para evitar una mala ejecución. Si todo está correcto, se espera a una señal de *START* por parte de la CPU. Una vez configurado y lanzado el acelerador, se comienza con la lectura de las series por parte de la interfaz de memoria y el almacenamiento de las mismas en la unidad de cómputo. Habiendo leído completamente las series, se comienza con la ejecución, y una vez ha acabado, se entrega el resultado a la interfaz de memoria para que realice su escritura. Posteriormente, se genera una interrupción para avisar a la CPU de que se ha concluido con la ejecución y que dispone del resultado. La participación y trabajo de los distintos bloques en este proceso de detalla en cada uno de sus apartados.

Teniendo ahora una descripción y visión general del acelerador, se pasa a continuación a la explicación en profundidad de los tres bloques relevantes de los que se compone.

## 3.2 Interfaz de registros

La interfaz de registros del acelerador está implementada como un conjunto de registros mapeados en memoria, accesibles desde la CPU. Esto significa que dichos registros están asociados a direcciones específicas dentro del espacio de memoria del sistema. De esta forma, cuando la CPU ejecuta una instrucción estándar de lectura o escritura sobre una de esas direcciones, la operación no se realiza sobre la memoria convencional, sino sobre el registro correspondiente del acelerador, permitiendo así su configuración o consulta mediante simples accesos de memoria. Esto se gestiona a través de la interfaz *REG*.

En concreto, este acelerador se ha diseñado con tan solo 8 registros para mantener el sistema lo más simple posible al mismo tiempo que se consigue todos los aspectos necesarios para su correcto funcionamiento. Cada registro tiene un modo de acceso definido: lectura y escritura (RW), solo lectura (RO) o solo escritura (WO)[1]. Los registros y su funcionamiento básico se encuentran representados en

[1] RW: Read/Write (lectura y escritura); RO: Read Only (solo lectura); WO: Write Only (solo escritura).

la tabla 3.1

| Nombre | Desplazamiento | Bits | Acceso | Descripción |
|---|---|---|---|---|
| RESERVED0 | 0x00 | 31:0 | RO | Reservado para ampliaciones |
| BASE_A | 0x04 | 31:0 | RW | Dirección base de la serie A |
| BASE_B | 0x08 | 31:0 | RW | Dirección base de la serie B |
| COUNT | 0x0C | 31:0 | RW | Palabras de 32 bits por serie |
| CONTROL | 0x10 | [0] START | WO | [0] Arrancar el acelerador |
| | | [1] CLR_ERR | | [1] Error corregido |
| STATUS | 0x14 | [0] DONE | RO | [0] Fin del cálculo |
| | | [1] BUSY | | [1] Ejecución en curso |
| | | [2] ERR_PARAM | | [2] Error en la configuración |
| WB_ADDR | 0x18 | 31:0 | RW | Dirección para el resultado |
| RADIUS | 0x1C | 31:0 | RW | Tamaño del radio a usar |

***Tabla 3.1.*** *Mapa de registros del acelerador.*

Para los registros que permiten la escritura, esta acción solo se puede realizar cuando el acelerador se encuentra inactivo, al contrario que la lectura, que siempre esta disponible.

Entrando más en detalle en cada registro, siguiendo el orden de la tabla 3.1, tenemos el registro *RESERVED0*. Este registro no se usa actualmente para nada y queda en la primera posición del espacio de direcciones a disposición de futuras ampliaciones.

Los siguientes registros, *BASE_A* y *BASE_B*, almacenan la dirección base de las series de entrada, es decir, la dirección de memoria que contiene el primer dato de cada una. Estos dos registros están alineados a 4 bytes para garantizar que se trata de una dirección de memoria válida.

Luego tenemos *COUNT*, que almacena al tamaño de las series en palabras de 32 bits (no confundir con muestras de 16 bits).

Seguidamente, tenemos el registro *CONTROL*. Este es un registro de solo escritura, en el que el bit 0 (*START*) sirve para que la CPU dé inicio al acelerador, y el bit 1 (*CLR_ERR*) sirve para que, tras un error en la configuración, la CPU indique que está corregido y es seguro arrancar de nuevo.

Después tenemos el registro *STATUS*, que es solo de lectura. En él, el acelerador refleja en qué estado se encuentra en cada momento. El bit 0 (*DONE*) se activa cuando el cálculo se ha completado, limpiándose cuando la CPU lee el registro. El bit 1 (*BUSY*) permanece activo mientras está en ejecución

y se desactiva cuando acaba. El bit 2 (*ERR_PARAM*) se activa cuando ha ocurrido un error a la hora de configurar algún registro, y se limpia cuando se escribe el bit *CLR_ERR* del registro *CONTROL*.

El siguiente registro en *WB_ADDR*, que funciona igual que los registros *BASE_A* y *BASE_B*, cambiando su finalidad a almacenar la dirección de memoria en la que se guardará el resultado.

Finalmente, tenemos *RADIUS*, que funciona igual que *COUNT*, solo que para almacenar el radio de la ventana que se ha de usar para ejecutar el algoritmo.

Por diseño del acelerador, los registros *RADIUS* y *COUNT* están acotados dentro de un rango. El primero tiene como valor mínimo permitido 1. Esto se debe a que poniendo un valor inferior eliminamos la posibilidad de desviaciones temporales, anulando el propósito del algoritmo DTW. Su valor máximo es 511. Esto está relacionado con el valor máximo de *COUNT*, que es 512 por diseño, ya que al tratarse de la cantidad palabras de 32 bits a leer, tiene que ser la mitad del número máximo de muestras que acepta el acelerador (1024 muestras de 16 bits). Entonces, como el ancho máximo de la ventana no puede superar al tamaño de las series, lo que produciría un error en el cálculo, y sabiendo que dicho ancho es $2R + 1$, el último valor de $R$ que permite eso es 511. Con respecto al valor mínimo de *COUNT*, este se fija en 2 debido a que, al tener un $R$ mínimo de 1 y para que no surjan errores, las series tienen que tener como mínimo 4 muestras cada una.

Si en el momento en que la CPU mande la señal *START* alguno de estos parámetros se encuentra fuera de rango, se activará el flag *ERR_PARAM* del registro *STATUS* y además se activará la línea de interrupción para avisar de este error a la CPU y que sea corregido. Esta misma línea de interrupción se activa también cuando se ha completado el cálculo del algoritmo para que atienda el resultado obtenido y se actúe en consecuencia. En caso de que en algún momento se active la señal de *RESET* desde el exterior, todos los valores de los registros se ponen a 0.

La secuencia de acciones específicas llevadas a cabo por esta interfaz es la siguiente: inicialmente, permanece a la espera de que se configuren los diferentes registros. El orden no es importante siempre y cuando se hagan antes de la llegada de la señal *START*. Mientras van llegando las configuraciones, se decodifican las direcciones para asignar el valor correspondiente a su registro. Una vez todos los registros han sido configurados, espera la llegada de la señal de *START*, que no es más que un bit del registro *CONTROL*. En el momento en que llega, se evalúa la validez de los valores introducidos tanto en *COUNT* como en *RADIUS*. Si todo es correcto, se da comienzo al proceso de lectura de las series y cálculo del algoritmo. Sin embargo, si alguno de los datos introducidos en esos registros es erróneo, la interfaz bloquea el arranque automáticamente, activa el bit *ERR_PARAM* del registro *STATUS* y manda una interrupción a la CPU para notificar el problema. Habiendo atendido la interrupción, la CPU

escribe de nuevo los valores corregidos de *COUNT* y *RADIUS*, además de activar el bit *CLR_ERR* del registro *CONTROL*, indicando de que se ha solventado el problema. Tras esto, se puede volver a activar *START* para dar comienzo al cómputo del algoritmo. Una vez el proceso ha terminado y el resultado ha sido guardado, la interfaz envía una interrupción para indicar dicho acontecimiento. A continuación, la CPU lee el registro *STATUS* confirmando que se ha acabo y limpiando el flag *DONE* de dicho registro que así lo indica. Con esto, la interfaz vuelve a su estado inicial, quedando lista para recibir una nueva configuración y repetir el proceso de cálculo.

## 3.3 Interfaz de memoria

La interfaz de memoria de este acelerador es el bloque encargado de realizar las transacciones con la memoria principal, tanto para leer las series necearías para la ejecución del algoritmo, almacenarlas en los buffers de la unidad de cómputo y, finalmente, escribir el valor resultante de aplicar el DTW. Todas estas operaciones, a excepción del almacenamiento en buffers, se gestionan a través del bus OBI del sistema, donde la interfaz actúa como maestro de bus, iniciando las transacciones, y la memoria como esclavo, recibiéndolas; siguiendo las fases estándar de dirección y respuesta. En todas las transacciones se trabaja con palabras de 32 bits, tanto para la lectura como para la escritura. Esto permite mantener compatibilidad con la plataforma además de que, en el caso de las lecturas, es lo más óptimo como se ha explicado por eficiencia.

En relación con los registros antes explicados, para esta interfaz son relevantes las direcciones de memoria (*BASE_A*, *BASE_B* y *WB_ADDR*), *COUNT*, el bit *START* del registro *CONTROL* y el registro *STATUS*. Los tres primero se usan para poder direccionar correctamente y acceder a las series y escribir el resultado final respectivamente. *COUNT* indica cuantas palabras de 32 bits leer para cada una de las series, además de que a partir de aquí, se le indica a la unidad de cómputo el número de muestras de 16 bits que debe de esperar; *START* indica cuando se empieza con el proceso y, desde esta interfaz, se indica el estado en el que se encuentra el acelerador con *STATUS*. Una vez configurados todos los valores y recibida la señal de *START*, la interfaz de memoria inicia el proceso, gestionando de manera ordenada cada fase mediante una máquina de estados finitos (FSM). Esta máquina controla la secuencia de operaciones necesarias para acceder a la memoria a través del bus OBI, el incremento y cambio de las direcciones cuando es necesario, y garantizar la correcta transferencia de datos.

Dicha maquina de estados se puede ver en la Figura 3.2. Cuenta con 9 estados y a continuación se va a explicar que acciones se realizan en cada uno de ellos, así como cómo se transiciona de unos a otros:

1. **M_IDLE**: Este es el estado en el que empieza la máquina. Inicialmente no hace nada hasta que

llega la señal *START*. En ese momento, prepara a la interfaz para comenzar a leer la serie A sin llegar a iniciar la transacción de bus, además de cambiar el estado del acelerador a *BUSY*. En ese mismo ciclo, al tener la señal de inicio, se transiciona hacia el estado *M_ADDR*. De no llegar dicha señal, se mantiene en este estado.

2. **M_ADDR**: En este estado si que se completa la petición a la memoria levantando la señal *req* del bus OBI. Al mismo tiempo, se transiciona hacia el siguiente estado, *M_WAIT_GNT*.

3. **M_WAIT_GNT**: Durante este estado, mantiene la señal *req* de manera que siga dentro de la misma transacción de la fase de direccionamiento. En el caso de que el esclavo levante su señal de *gnt*, indicando que aceptó la petición, transiciona al estado *M_WAIT_RVALID*. De no ser así se mantiene en este estado a la espera.

4. **M_WAIT_RVALID**: Mientras se está en este estado, se espera a que el esclavo active la señal *rvalid*. Si así ocurre, significa que se ha leído un dato con éxito, siendo reenviado a la unidad de cómputo junto con su índice para que se almacene. También se comprueba si se ha completado de manera total la fase de lectura de las series y se transiciona al estado *M_NEXT*. En el caso de que *rvalid* no llegue, se mantiene en este estado hasta que eso ocurra.

5. **M_NEXT**: En este estado, se actualizan los contadores, la dirección a leer y, en caso de que sea necesario, esta se cambia a la dirección base de la seria B para empezar a leerla. Si siguen quedando datos para leer, se vuelve al estado *M_ADDR* para repetir este ciclo. De lo contrario, se asume que se leído completamente todo y se transiciona al estado *M_WAIT_COMP*.

6. **M_WAIT_COMP**: Durante este estado, simplemente se espera a que la unidad de cómputo termine su cálculo. En el momento en que termine, se comprueba si se configuró una dirección en la que escribir el resultado. Si es así, transiciona hacia *M_WB_ADDR* y si no, se pasa a *M_FINISH*.

7. **M_WB_ADDR**: Al llegar a este estado, la interfaz se configura para mandar una petición de escritura con el resultado DTW a la dirección indicada. De aquí se transiciona directamente a *M_WB_WAIT_GNT*.

8. **M_WB_WAIT_GNT**: Aquí se mantiene las señales configuradas en el estado anterior a la espera de recibir la señal *gnt* por parte del esclavo, de manera similar que en *M_WAIT_GNT*. En el momento en que dicha señal llegue, se confirma que el dato ha sido escrito con éxito y se transiciona a *M_FINISH*.

9. **M_FINISH**: Al llegar a este estado, simplemente se cambia el estado del acelerador de *BUSY* a *DONE*, indicando que se ha completado el proceso. De aquí se transiciona a *IDLE* a la espera de repetir el proceso para un nuevo cálculo del algoritmo.

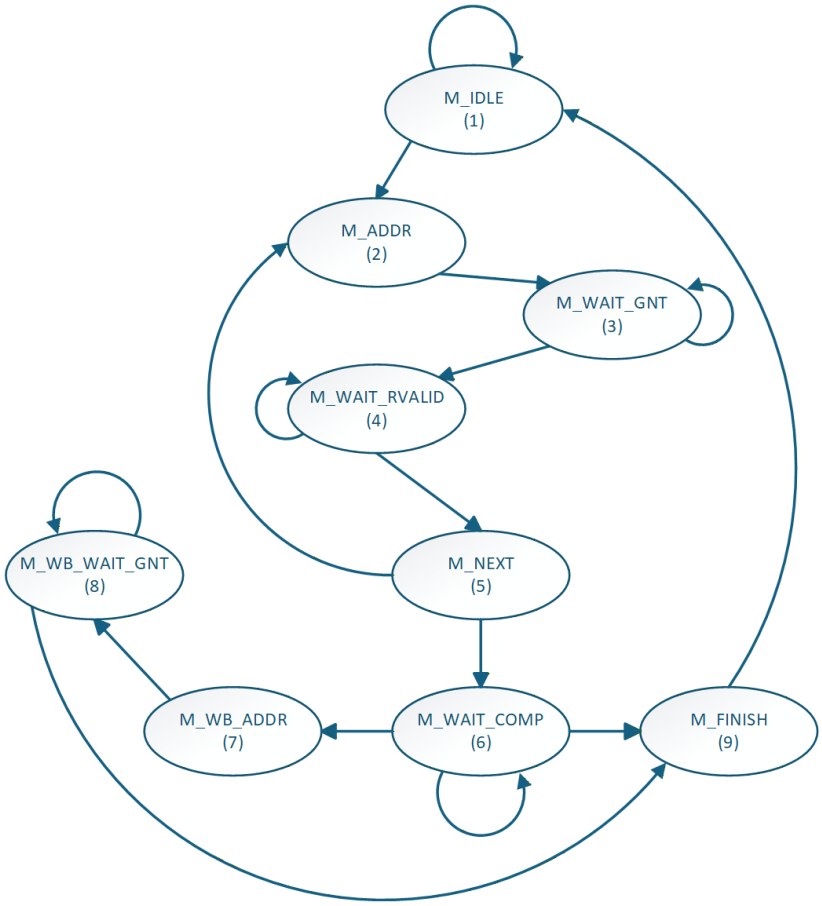


***Figura 3.2.*** *Máquina de estados finitos de la interfaz de memoria.*

Este enfoque asegura una comunicación eficiente y libre de conflictos con la memoria principal, garantizando que las operaciones se realicen en orden y con control total sobre cada fase. Con ello, la interfaz proporciona una base estable y fiable para el funcionamiento global del acelerador.

## 3.4 Unidad de cómputo

Este módulo es el encargado en realizar el cálculo del algoritmo DTW, siendo el módulo principal del acelerador. En relación con los registros configurables, para este bloque son relevantes los valores de *RADIUS* y *COUNT*. El primero es vital para configurar correctamente en ancho de la ventana a la hora de calcular. El segundo no se usa directamente, sino que nos llega el dato de cuantas muestras se han de leer a través de la interfaz de memoria. Esto se hace para asegurarnos de que tenemos

el número correcto de datos para ejecutar. Dichos datos se almacenan en sus respectivos registros dentro de esta unidad.

Como ya se ha comentado, para este acelerador, el procesamiento se hace por antidiagonales y aplicando una restricción tipo ventana de Sakoe-Chiba. Al combinar estas dos optimizaciones ocurre algo curioso con la matriz de distancias, y es que se diferencian tres zonas en dicha matriz: zona superior, donde la ventana va creciendo hasta llegar al tamaño máximo; zona central, donde la ventana mantiene su ancho; y zona inferior, donde la ventana va reduciendo su tamaño hasta quedar únicamente la última celda. Estas tres zonas las podemos ver resaltadas en la Figura 3.3 en amarillo, verde y rojo respectivamente, quedando en gris las celdas excluidas por la ventana.

A la hora de calcular el valor mínimo al que sumar la distancia, se presentan ciertas particularidades en la zona superior y central:

- En la primera zona (en amarillo en la Figura 3.3) ocurre que la primera celda de la matriz no tiene valores entre los que comparar, por lo que el mínimo se asume a 0. También ocurre que, en la primera fila de la matriz, el único valor mínimo de una celda es el que se encuentra a su izquierda. Esto ocurre de manera similar con la primera columna, solo que ahora el mínimo es aquel que se encuentra arriba. En cualquier otro caso dentro de esta zona sí que se dispone de los tres valores.

- En la zona central (en verde en la Figura 3.3) se dijo que el ancho de la ventana se mantiene constante. Esto no es del todo cierto, ya que aquellas antidiagonales que contengan un valor de la diagonal principal de la matriz tiene un dato más. Para esto se define el concepto de «paridad». En este contexto, se entiendo por antidiagonal «impar» aquella que contiene una celda de la diagonal principal, que al tener ese dato, tiene un número impar de elementos. En consecuencia, es «par» aquella que no contiene una celda de la diagonal principal. Esto se puede ver representado en la Figura 3.3, donde la quinta antidiagonal es «impar» y la sexta «par». En las diagonales que son «impares» ocurre que, tanto la primera como la última celda de la ventana solo dispone de dos valores entre los que hallar el mínimo: el superior y el que se encuentra en diagonal arriba a la izquierda si se está al principio; y el izquierdo y el que se encuentra en diagonal arriba a la izquierda si se encuentra al final de la ventana.

En cualquier otro caso, siempre se dispone de tres valores para encontrar el mínimo.

Con respeto a la arquitectura, que se puede ver en la Figura 3.4, podemos dividir este módulo en tres bloques funcionales, donde cada uno se encarga de una acción especifica:

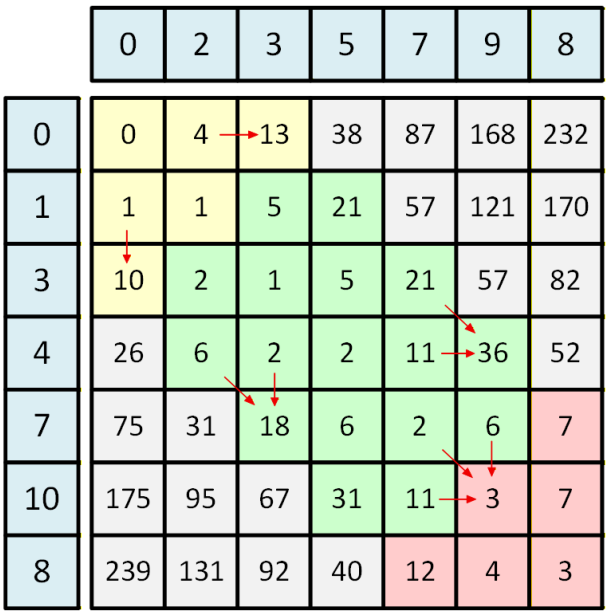


***Figura 3.3.*** *Ejemplo de matriz DTW calculada con ventana Sakoe-Chiba y antidiagonales.*

- **Bloque A**: Aquí se gestiona todo lo relacionado con las series de entrada. En concreto, está compuesto por los buffers para las series A y B y la unidad de distancia, También se realiza la división de la palabra de 32 bits de entrada en 2 muestras de 16 bits y se selecciona a qué serie pertenece. Todo este proceso de almacenamiento se gestiona mediante un *handshake* con la interfaz de memoria para garantizar sincronía y no perder datos. Además, a partir de los índices calculados en el bloque B, se direccionan las muestras necesarias en el cálculo para calcular la distancia entre ellas que, para este acelerador, se calcula como la distancia euclídea del cuadrado de las muestras, definida como $d(a, b) = (a - b)^2$. Esto implica extender los datos de 16 a 32 bits manteniendo el signo ya que al elevar al cuadrado un dato, el resultado puede necesitar el doble de bits para ser representado, además de que de esta forma el resultado final obtenido tiene el mismo formato que la memoria del sistema.

- **Bloque B**: En este bloque se encuentran los contadores, índices y registros necesarios para recorrer la matriz de distancias por antidiagonales. Exactamente, estos registros contienen el índice de la celda dentro de la antidiagonal actual, el índice de la antidiagonal dentro de la matriz, los valores máximo y mínimo que puede alcanzar el índice de la celda y el bit de paridad de la antidiagonal en la que se encuentra en ese momento. Con la combinación estos datos, se direccionan tanto a los buffers del bloque A para acceder a las series y calcular la distancia, como a los buffers del bloque C para encontrar el valor mínimo. Además, también se indica cuando hemos acabado con el recorrido.

- **Bloque C**: Dentro de este bloque se encuentran los tres buffers donde se almacenan las antidiagonales. Estas son la que está siendo calculada, la calculada en el paso anterior y la calculada dos pasos atrás. A partir de los índices almacenados en el bloque B, se direccionan los datos necesarios para encontrar el valor mínimo de los anteriormente calculados, teniendo en cuenta cada caso que se puede dar. También se realiza la rotación de los buffers. Esto consiste en que una vez completada la antidiagonal actual, esta pasa al buffer de la calculada en el paso anterior, y a su vez los de esta pasan al buffer de la calculada dos pasos atrás desechando los datos de este último buffer. Por último, a partir de lo calculado se pone el dato final en la salida del módulo para que sea escrito por la interfaz de memoria en la memoria principal del sistema.

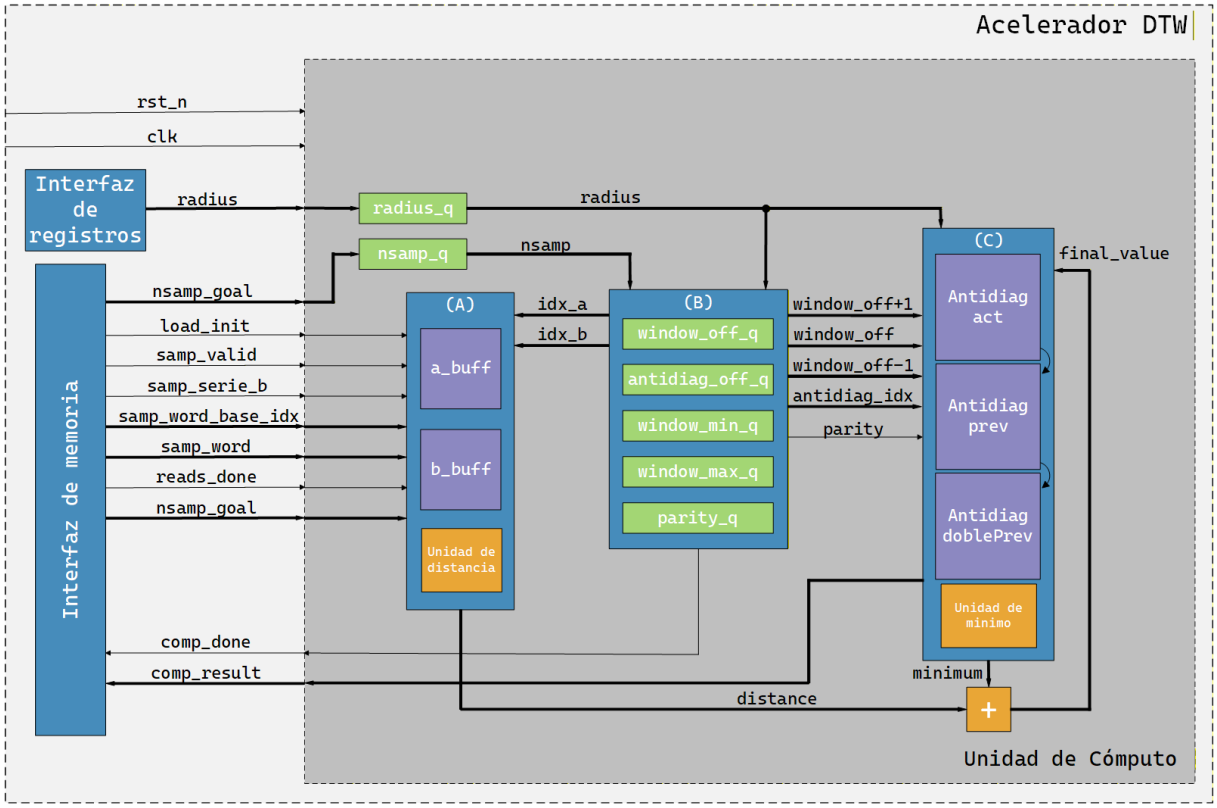


***Figura 3.4.*** *Diagrama de la unidad de cómputo del acelerador.*

Debido a la naturaleza secuencial del algoritmo DTW y a la necesidad de coordinar la lectura de datos, el cálculo de las distancias, búsqueda del mínimo y la entrega del resultado final, la unidad de cómputo organiza su funcionamiento interno mediante una FSM, de manera similar a la interfaz de memoria. En este caso, es una máquina de estados más simple que solo cuenta con 3 estados, como se puede ver en la Figura 3.5.

Estos estados son:

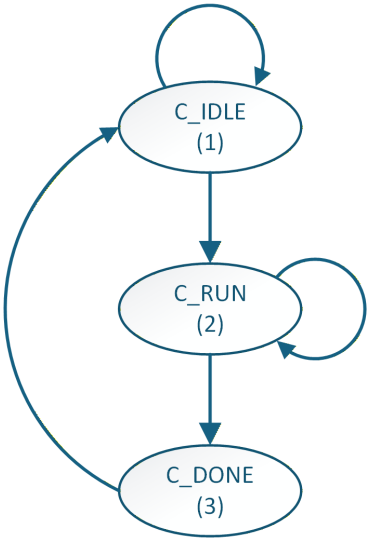


***Figura 3.5.*** *Máquina de estados finitos de la unidad de cómputo.*

1. **C_IDLE**: Este es el estado inicial de la máquina. Se espera a que llegue la señal que da inicio a las lecturas de las series, preparando el contador destinado a esto. Si llega un dato junto con su señal de validez, se almacena en su buffer. Una vez leídas completamente las series, prepara a la unidad para comenzar con la fase de ejecución del algoritmo. Mientras se está leyendo datos, permanece es este estado y, una vez leído todo, se pasa al estado *C_RUN*

2. **C_RUN**: Durante este estado se ejecuta todo el algoritmo DTW. Podemos diferenciar dos acciones simultaneas: por un lado, se calcula la distancia entre los dos datos de entrada, y por otro lado se encuentra el valor mínimo de los valores vecinos ya calculados. Teniendo ya estos dos datos, se suman y se almacenan en el buffer de la antidiagonal actual con su respectivo índice, para posteriormente actualizar todos los índices involucrados para el cálculo de esta celda. Una vez completada la antidiagonal actual, se realiza la rotación de los buffers. Todo este proceso de cálculo de celdas y actualización de índices, contadores y antidiagonales se va repitiendo de forma iterativa hasta cubrir la matriz completamente, y una vez llegado al final, se transiciona al estado *C_DONE*. Mientras se ejecuta el algoritmo, permanecemos en este estado.

3. **C_DONE**: Una vez acabado el cálculo, se llega a este estado, donde se pone el resultado calculado en la salida hacia la interfaz de memoria, junto con un flag indicando el fin del cómputo. Hecho esto, se transiciona hacia *C_IDLE*, quedando preparado para una nueva ejecución.

En conjunto, la unidad de cómputo constituye el núcleo funcional del acelerador, donde se realiza el procesamiento intensivo del algoritmo. Su diseño basado en buffers antidiagonales y controlado

mediante una FSM permite equilibrar eficiencia en el uso de recursos y rendimiento computacional.

# 4

# Integración en X-Heep

En este capítulo se explica el proceso de incorporación del acelerador en la plataforma X-Heep, analizando las posibles alternativas de integración y justificando la seleccionada. Se detallan las modificaciones realizadas sobre el hardware, el software y las herramientas de generación automática para garantizar la compatibilidad con la plataforma, así como el desarrollo del driver necesario para su utilización. Con ello, se demuestra cómo el acelerador pasa de ser un módulo independiente a convertirse en un periférico plenamente funcional dentro del sistema.

## 4.1 Métodos de integración

Para integrar hardware personalizado en la plataforma de X-Heep, ya se vio que existen tres posibles vías de ampliación: como coprocesador a través de la interfaz *CV-X-IF*, como periférico externo por medio de *XAIF* o como periférico interno conectado al bus del sistema.

La interfaz *CV-X-IF* está concebida para la incorporación de coprocesadores capaces de ejecutar extensiones del conjunto de instrucciones de RISC-V o instrucciones personalizadas. En otras palabras, permite añadir bloques hardware que amplían directamente las capacidades del procesador de la plataforma, integrándose de forma estrecha con él, de manera que cuando este encuentra una instrucción desconocida, se reenvía al coprocesador para su ejecución. Sin embargo, esta interfaz está orientada a coprocesadores de propósito más general o a módulos con un conjunto amplio de instrucciones. Puesto que el acelerador desarrollado para este proyecto se centra en la ejecución del algoritmo DTW, su implementación mediante esta interfaz supondría una infrautilización de la misma. Además, este método requeriría incorporar un decodificador de instrucciones en el nuevo hardware, incrementando la complejidad del diseño, siendo además innecesario para el caso de este acelerador, pues solamente serviría para notificar el arranque de la ejecución [14].

La segunda opción consiste en la conexión del acelerador como un periférico externo a través de la interfaz XAIF. Dicha interfaz permite incorporar hardware adicional sin modificar la estructura

interna de X-Heep, generando en la práctica una plataforma compuesta por el sistema base y el nuevo periférico. A nivel funcional, los resultados son comparables a los de una integración interna, aunque ofrece un mayor control del consumo energético aplicando técnicas como el control de la frecuencia o el apagado del módulo si este está sin uso. No obstante, el principal inconveniente radica en la escasa documentación sobre esta interfaz, lo que complica su uso y limita su reproducibilidad. Por ello, esta opción fue descartada [8].

Por último, está la opción de incorporar el acelerador como un periférico interno, conectado directamente al bus del sistema. Este método implica la modificación de ciertos archivos de descripción hardware y scripts de generación, pero presenta varias ventajas: mantiene la coherencia con el mapa de memoria de la plataforma y simplifica la gestión de interrupciones, al integrarlas en el mismo esquema general del sistema sin alterar su comportamiento y, sobre todo, se encuentra plenamente documentado dentro de los recursos oficiales de X-Heep. Aunque ofrece un menor grado de control sobre la eficiencia energética, pues se integra junto con otros periféricos, estas limitaciones no resultan relevantes en el contexto del proyecto y se compensan con una integración más limpia y estable [24].

En conclusión, las dos primeras alternativas resultan más complejas de implementar y contaban con menor soporte práctico, por lo que finalmente se optó por la integración como periférico interno. Esta solución ofrecía una incorporación directa, bien documentada y plenamente funcional, ajustándose de forma óptima a los objetivos del proyecto.

## 4.2 Modificación de archivos

Para llevar a cabo este proceso de modificación, se empieza con el repositorio oficial de X-Heep como base [5]. Aquí se encuentran todos los archivos de la plataforma, que a grandes rasgos son: las configuraciones, las descripciones hardware de sus elementos y periféricos, los códigos fuente de los drivers de estos y algunos ejemplos de su uso y herramientas de generación automática de archivos y scripts. Estos archivos se encuentran en sus respectivos directorios (*configs/*, *hw/*, *sw/*,*scripts/* y *util/* respectivamente). Es dentro de estos directorios donde se añadirán los archivos del proyecto, tanto del hardware como del software, además de modificar algunos de ellos para que la adición sea exitosa. Es cierto que el repositorio contiene más directorios, pero para este proyecto no son muy relevantes.

En cuanto a los archivos añadidos, dentro del directorio */hw* existe el subdirectorio */ip*, que contiene los distintos periféricos. Es aquí donde se añadió el nuevo directorio para el acelerador (*/dtw*), que contiene los archivos de descripción hardware en SystemVerilog, un archivo *CORE* que permite gestionar la construcción de la plataforma, listando los archivos necesarios, así como dependencias y parámetros de configuración, y un archivo *HJSON* que describe los registros del acelerador. Técnicamente, este

último tiene la finalidad de generar los registros automáticamente a través de un script *BAHS* también incluido.Sin embargo, durante el desarrollo del acelerador se optó por realizar esta tarea a mano para controlar a la perfección la finalidad y conectividad de estos con el resto del acelerador. Aun así, queda incluido por si en futuras implementaciones se quiere aplicar el método automático.

En el directorio */sw*, existe el subdirectorio que contiene los drivers escritos en C de los distintos periféricos. Dentro, se agregó un nuevo directorio para el acelerador con esta finalidad. Además, dentro de */sw*, también se encuentra el subdirectorio */applications* que contiene todos los programas de ejemplo para los distintos periféricos. Aquí se añadió tanto un ejemplo del uso del acelerador como un programa usado para medir el rendimiento en ciclos del mismo.

Por otro lado, también fue necesario realizar las modificaciones al hardware de la plataforma. Este proceso consistió en añadir al subsistema de periféricos el acelerador, además de las señales necesarias para su funcionamiento, que consisten en la señal de reloj y *RESET* necesarias para la sincronización, una interfaz de tipo *REG* como esclavo para que la CPU pueda configurarlo, un entrada-salida *OBI* como maestro para poder acceder a la memoria principal y una línea de interrupción propia. Esta última se conecta al PLIC que ya incorpora X-Heep para que gestione la interrupción cuando sea necesario. También es necesario modificar el bus del sistema para redirigir correctamente las señales OBI.

En cuanto a las modificaciones del software, solo fue necesario modificar el driver del PLIC para que sepa reconocer al nuevo periférico e incluir la gestión de la interrupción del acelerador y saber actuar cuando se dé el caso.

Por último, queda por modificar algunas configuraciones y scripts para que X-Heep se genere correctamente junto con el acelerador del algoritmo DTW diseñado en este proyecto. Para ello, primero se modifica un archivo *HJSON* dentro del directorio */configs* que contiene la configuración general de X-Heep. En él, simplemente se añade el acelerador a la lista de periféricos, indicando su espacio de direcciones y su índice de interrupción en la lista correspondiente. Posteriormente, se añade la dependencia al nuevo periférico en un archivo *CORE* junto con los demás periféricos. Finalmente, se modifican dos scripts de Python dentro de la carpeta */utils* que son usados para generar el sistema, de manera que sean capaces de localizar las configuraciones y los archivos del acelerador recién introducido.

Tras estas modificaciones y la regeneración del sistema, el acelerador queda incluido en la plataforma como un periférico más dentro de dicho subsistema, como se puede ver en la Figura 4.1. De esta manera, queda completamente integrado y operativo dentro del ecosistema de X-Heep,

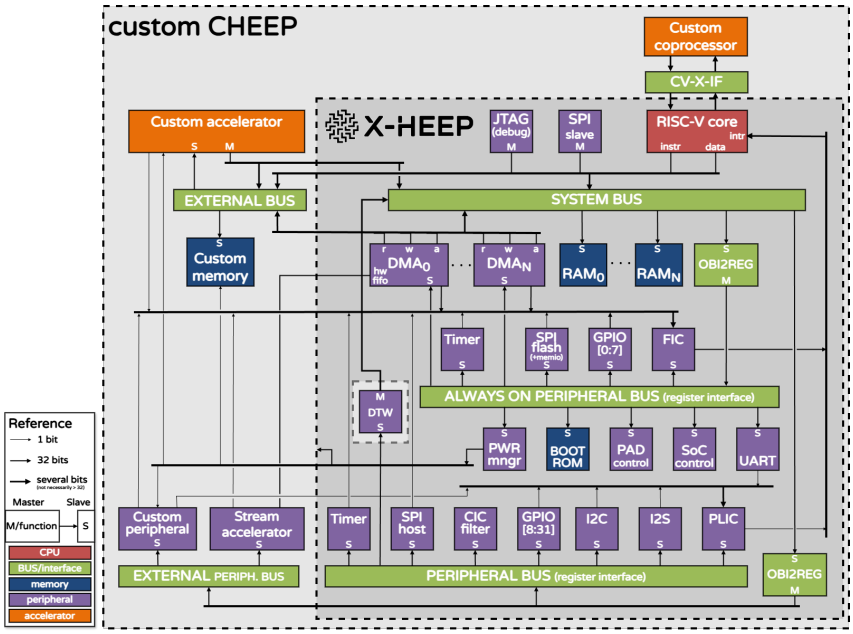


***Figura 4.1.*** *Diagrama de X-Heep junto con el acelerador introducido como periférico*

Modificación realizada sobre la Figura 2.4.

asegurando su correcta comunicación y funcionamiento con el resto de componentes del sistema.

## 4.3 Creación del driver

Como ya se ha comentado, junto con el nuevo hardware, fue necesario desarrollar un driver para poder controlarlo y realizar ejecuciones con él. En este apartado se describe qué funciones se diseñaron y con qué finalidad para poder hacer un uso directo y eficiente del acelerador. El driver está escrito en C y se organiza en dos archivos de cabecera y un archivo de implementación.

El primero de esos archivos de cabecera define la interfaz de registros del acelerador DTW utilizada por el software. Toma la dirección base del acelerador en el espacio de memoria y el identificador de interrupción desde la configuración generada por X-Heep, lo que garantiza coherencia entre la definición hardware y el software. Es a partir de esto que se especifican los offsets de cada uno de los registros que componen su interfaz de control mostrada en la Tabla 3.1. También define las máscaras y posiciones de bits asociadas a los diferentes flags de estado. Estos valores permiten al driver interactuar correctamente con el hardware mediante accesos de memoria mapeada.

El segundo archivo de cabecera define la interfaz pública del driver. En él, se declara la estructura que encapsula los parámetros para una ejecución (direcciones de las series y del resultado, tamaño y radio de la ventana) así como el tipo para una función *callback* opcional. Esto último es una función que no se invoca directamente, sino que es llamada por otra función cuando se da un evento concreto, como la llegada de una interrupción. Además, el fichero recoge las declaraciones de las funciones

necesarias para inicializar la gestión de interrupciones, registrar dicho *callback*, programar los registros del periférico, arrancar una ejecución, sincronizar la espera por su finalización y consultar su estado. Por último, incluye pequeñas utilidades *inline*, que son funciones que se llevan directamente al código donde se ejecutan en lugar de ser llamadas, para empaquetar y extraer muestras de 16 bits dentro de palabras de 32 bits y rutinas de soporte para habilitar las interrupciones a nivel de procesador.

Finalmente, está el fichero de código en C que contiene la implementación funcional del driver, es decir, el código encargado de ejecutar las operaciones que la cabecera declara. En él se materializa la interacción real con el acelerador mediante la escritura y lectura de sus registros, así como la gestión del estado interno del controlador software durante una ejecución. También incorpora la inicialización de la interrupción asociada, el tratamiento de la señal de fin de operación y la ejecución opcional de un *callback* registrado por el usuario. En conjunto, este módulo proporciona el comportamiento necesario para que la aplicación pueda configurar, arrancar y supervisar el acelerador DTW de forma sencilla y fiable.

En conjunto, la incorporación de estos ficheros de cabecera y del módulo de implementación proporciona una capa de abstracción completa y coherente entre el acelerador hardware y el software de aplicación. Gracias a esta estructura, el uso del periférico queda encapsulado en un conjunto claro de funciones y utilidades, facilitando su integración en cualquier programa que haga uso del DTW y garantizando una interacción segura, consistente y acorde con el diseño del hardware.

# 5

# Síntesis, verificación y rendimiento

En este capítulo se aborda el proceso de validación del acelerador DTW mediante verificación funcional, análisis de rendimiento y síntesis. En primer lugar, se explican las pruebas empleadas para verificar el correcto funcionamiento del acelerador y su interacción con la plataforma X-Heep. A continuación, se presentan los resultados de rendimiento obtenidos y se comparan con la ejecución equivalente en software, permitiendo valorar el impacto real de la aceleración propuesta. Por último, se describe la síntesis del hardware, cuyo objetivo es comprobar la viabilidad física del diseño y estimar su coste en recursos. Además, con los resultados obtenidos, se realiza un análisis del consumo del cálculo del algoritmo DTW.

## 5.1 Metodología de las pruebas

Para comprobar que el acelerador funciona correctamente y poder comparar la ejecución del algoritmo DTW frente a una versión software ejecutada por la CPU de X-Heep, se diseñaron una serie de pruebas orientadas a probar distintos aspectos de la ejecución. Estas pruebas se dividieron en tres escenarios:

- En el primer caso, se establece una relación entre el radio de la ventana ($R$) y el tamaño de las series de entrada ($N$) de 1/16, empezando con esos valores y aumentándolos en potencias de dos hasta llegar a un $R$ de 64 y un $N$ de 1024. El objetivo de esta prueba es evaluar el comportamiento general de ambas implementaciones y analizar cómo evolucionan los resultados conforme se aumenta el tamaño del problema a ejecutar. El número de celdas que se calculan en este caso sigue la siguiente función, simplificando $R$ como $\frac{N}{16}$:

$$C(N) = \frac{15}{64} \cdot N^2 + \frac{7}{8} \cdot N$$

- Para el segundo escenario, se deja un *R* constante de 4, únicamente aumentando *N* desde 16 hasta 1024, también en potencias de dos. De esta manera, se elimina el efecto que puede tener aumentar la banda de la ventana de datos a calcular, y así poder evaluar cómo afecta únicamente incrementar la cantidad de datos por serie. El número de celdas que se calculan en este caso sigue la siguiente función:

$$C(N) = 17 \cdot N - 72$$

- Para el último caso, se plantea el escenario contrario al anterior, ya que ahora se mantiene *N* fijo en 1024 y se va aumentando *R* desde 1 hasta 64 en potencias de 2, al igual que en las pruebas anteriores. En este caso se aísla *N* para poder estudiar cómo afecta aumentar el tamaño del radio entre las distintas pruebas en términos de rendimiento. El número de celdas que se calculan en este caso sigue la siguiente función:

$$C(R) = 4094 \cdot R + 1024 - 4 \cdot R^2$$

En total, para cada uno de los escenarios se realizaron siete ejecuciones, una por cada combinación de *R* y *N* antes mencionadas. Además, debido a la naturaleza del sistema al estar simulado, las pruebas son deterministas, lo que quiere decir que, si no se realiza ningún cambio entre las ejecuciones, los resultados obtenidos para la misma combinación de *R* y *N* en una prueba son idénticos sin importar en qué momento se hayan hecho. Por último, cabe comentar que, al igual que el driver, las pruebas se han ejecutado en C, tanto para la configuración, arranque y control de la interrupción del hardware del acelerador, como para la prueba en software ejecutada por la CPU. Por esto se decidió también aplicar las opciones de optimización del compilador, teniendo una pasada en cada escenario sin especificar la optimización, una con la opción más baja $(-O0)$ y una última con la opción más alta $(-O3)$, para así analizar su influencia tanto en la ejecución software como en la interacción con el hardware acelerado.

De estas pruebas, lo que se mide es la cantidad de celdas de la matriz de distancias que hay que calcular, cuantos ciclos totales ha empleado cada versión en calcular dichas celdas, y a partir de estos datos, se deducen los ciclos por celda de cada versión, así como el '*SpeedUp*' que consigue el acelerador frente a la ejecución software.

## 5.2 Comentario de los resultados

Una vez comentada la metodología de las pruebas a realizar en el apartado anterior, en este apartado se realiza el análisis y comentario de los resultados obtenidos tanto para la implementación software como para el acelerador hardware. Para ello, se comparan los ciclos necesarios en cada caso, así como los ciclos empleados en cada celda, identificando tendencias, diferencias de escalabilidad y posibles limitaciones.

Se comienza entonces comentando el primer escenario, que corresponden a las Tablas 5.1, 5.2 y 5.3, donde se mantiene una relación 1/16 entre el tamaño de la ventana ($R$) y el número de datos por serie de entrada ($N$) respectivamente:

| Casos (N/R) | Celdas | Ciclos (SW) | Ciclos (HW) | $\frac{Ciclos}{Celda}$ (SW) | $\frac{Ciclos}{Celda}$ (HW) | SpeedUp |
|---|---|---|---|---|---|---|
| 16/1 | 74 | 4930 | 304 | 66,62 | 4,11 | 16,22 |
| 32/2 | 268 | 15477 | 572 | 57,75 | 2,13 | 27,06 |
| 64/4 | 1016 | 53344 | 1464 | 52,50 | 1,44 | 36,44 |
| 128/8 | 3952 | 195978 | 4686 | 49,59 | 1,19 | 41,82 |
| 256/16 | 1558 | 750066 | 16896 | 48,13 | 1,08 | 44,39 |
| 512/32 | 61888 | 2932578 | 64352 | 47,39 | 1,04 | 45,57 |
| 1024/64 | 246656 | 11594946 | 251422 | 47,01 | 1,02 | 46,12 |

***Tabla 5.1.*** *Resultados del 1$^{er}$ escenario sin optimización explicita*

| Casos (N/R) | Celdas | Ciclos (SW) | Ciclos (HW) | $\frac{Ciclos}{Celda}$ (SW) | $\frac{Ciclos}{Celda}$ (HW) | SpeedUp |
|---|---|---|---|---|---|---|
| 16/1 | 74 | 12934 | 430 | 174,78 | 5,81 | 30,08 |
| 32/2 | 268 | 41345 | 698 | 154,27 | 2,60 | 59,23 |
| 64/4 | 1016 | 144695 | 1590 | 142,42 | 1,56 | 91,00 |
| 128/8 | 3952 | 537395 | 4814 | 135,98 | 1,22 | 111,63 |
| 256/16 | 1558 | 2066795 | 17022 | 132,62 | 1,09 | 121,42 |
| 512/32 | 61888 | 8101595 | 64478 | 130,91 | 1,04 | 125,65 |
| 1024/64 | 246656 | 32075195 | 251550 | 130,04 | 1,02 | 127,51 |

***Tabla 5.2.*** *Resultados del 1$^{er}$ escenario con optimización $-O0$*

| Casos (N/R) | Celdas | Ciclos (SW) | Ciclos (HW) | $\frac{Ciclos}{Celda}$ (SW) | $\frac{Ciclos}{Celda}$ (HW) | SpeedUp |
|---|---|---|---|---|---|---|
| 16/1 | 74 | 3110 | 294 | 42,03 | 3,97 | 10,58 |
| 32/2 | 268 | 9529 | 550 | 35,56 | 2,05 | 17,33 |
| 64/4 | 1016 | 32443 | 1427 | 31,93 | 1,40 | 22,74 |
| 128/8 | 3952 | 118627 | 4620 | 30,02 | 1,17 | 25,68 |
| 256/16 | 1558 | 452419 | 16762 | 29,03 | 1,08 | 26,99 |
| 512/32 | 61888 | 1765699 | 64091 | 28,53 | 1,04 | 27,55 |
| 1024/64 | 246656 | 6975043 | 250908 | 28,28 | 1,02 | 27,80 |

***Tabla 5.3.*** *Resultados del 1$^{er}$ escenario con optimización $-O3$*

En el caso del hardware, para los tres casos de optimización, la cantidad de ciclos que supone la configuración del acelerador, la lectura de los datos para el cálculo y el tratamiento de su interrupción tiene una gran influencia en el total de ciclos empleados para los casos pequeños. Esto se ve ya que en los casos sin optimización explicita y con la máxima optimización los ciclos por celda (ciclos/celda) empleados son 4,11 y 3,97 respectivamente, y en el caso con la optimización desactivada, este dato pasa a ser 5,81. Estos resultados distan del ciclo por celda teórico que debe de alcanzar el acelerador por diseñó. Sin embargo, a medida que el problema aumenta, los ciclos que suponen estas tres acciones pasan a ser cada vez más despreciables. Esto se puede confirmar ya que en los casos grandes sí que se llega a aproximadamente a 1 ciclo/celda para los tres casos de optimización. Podemos decir que, en general, aumentar la optimización a la hora de compilar no supone una gran diferencia en el hardware, sobre todo para casos más grandes, ya que las acciones que dependen de la optimización que hace el compilador, que son configurar y tratar la interrupción, acaban suponiendo muy pocos ciclos del total.

En el caso del software, como era de esperar, se tarda muchos más ciclos en procesar cada celda. Esto se debe a que, en los procesadores de propósito general, cada instrucción puede tardar varios ciclos en procesarse, además de tener que acceder a la memoria, gestionar condiciones y realizar saltos. Para los casos pequeños, el 'overhead' fijo que implican esas últimas acciones tiene que repartirse entre pocas celdas, de ahí que el número de ciclos/celda sea más elevado que para casos grandes, donde se amortiza hasta estabilizarse. Cabe destacar que, porcentualmente la diferencia entre casos pequeños y grandes es mucho menor para software que para el hardware, lo que se puede deber a que el 'overhead' por celda es más constante. En cuanto a la optimización, en este casi si que se ve una clara mejora para el caso con máxima optimización, reduciendo bastante los ciclos/celda y reduciendo la diferencia

con respecto al hardware. Este resultado se debe a que ahora, todo el cálculo si que depende del código en C y al optimizarlo, se mejoran bastante los resultados

En general, podemos ver que para esta prueba en la que se estudia el rendimiento general de ambos casos, usar el acelerador a igualdad de condición de frecuencia, es decir, usando la misma señal de reloj, supone una gran mejor en cuanto al rendimiento, siendo en el peor de los casos 10,58 veces mejor, y en el mejor de los casos 127,51 veces mejor.

Como se ha visto en este escenario, para el hardware la optimización que aplica el compilador supone una mejora, aunque ínfima, especialmente en los casos grandes. Sin embargo, en el caso del software ocurre todo lo contrario, donde la mejora es más que significativo. Por este motivo, aunque se incluyen las tablas completas correspondientes a la ejecución con las optimizaciones apagadas y sin opciones explícitas de compilación (Tablas 5.4, 5.5, 5.7 y 5.8), el análisis detallado de los siguientes escenarios se realiza tomando como referencia los resultados obtenidos con el nivel máximo de optimización disponible, correspondientes a las Tablas 5.6 y 5.9, al ser estos los que representan el comportamiento más eficiente y realista del software.

A continuación, se pasa a comentar los resultados del segundo escenario, representados por la Tabla 5.6, en el que se deja fijo el radio de la ventana variando el tamaño de las series:

| Casos (N) | Celdas | Ciclos (SW) | Ciclos (HW) | $\frac{Ciclos}{Celda}$ (SW) | $\frac{Ciclos}{Celda}$ (HW) | SpeedUp |
|---|---|---|---|---|---|---|
| 16 | 200 | 11968 | 426 | 59,84 | 2,13 | 28,0 |
| 32 | 472 | 25760 | 771 | 54,58 | 1,63 | 33,41 |
| 64 | 1016 | 53344 | 1464 | 52,50 | 1,44 | 36,44 |
| 128 | 2104 | 108512 | 2834 | 51,57 | 1,35 | 38,29 |
| 256 | 4280 | 218848 | 5587 | 51,13 | 1,31 | 39,17 |
| 512 | 8632 | 439520 | 11092 | 50,92 | 1,28 | 39,62 |
| 1024 | 17336 | 880864 | 22098 | 50,81 | 1,27 | 39,86 |

***Tabla 5.4.*** *Resultados del 2º escenario sin optimización explicita*

En el caso del hardware, vemos algo curioso, y es que a diferencia de la prueba anterior donde los ciclos/celda al principio eran mucho mayores que al final y tendían a estabilizarse en 1 ciclo/celda, en este caso vemos que, si bien es cierto que también existe diferencia entre los casos pequeños y los grandes, esta diferencia es menor (pasando de 3.97 ciclos/celda a 2,10 en el primer caso) y en casos grandes tiende a estabilizarse entre a 1,25 ciclos/celda. Esto se debe a que, en esta prueba, se realizan

| Casos (N) | Celdas | Ciclos (SW) | Ciclos (HW) | $\frac{Ciclos}{Celda}$ (SW) | $\frac{Ciclos}{Celda}$ (HW) | SpeedUp |
|---|---|---|---|---|---|---|
| 16 | 200 | 32327 | 558 | 161,64 | 2,79 | 57,93 |
| 32 | 472 | 69783 | 902 | 147,85 | 1,91 | 77,36 |
| 64 | 1016 | 144695 | 1590 | 142,42 | 1,56 | 91,00 |
| 128 | 2104 | 294519 | 2966 | 139,98 | 1,22 | 99,30 |
| 256 | 4280 | 594167 | 5718 | 138,82 | 1.41 | 103,91 |
| 512 | 8632 | 1193463 | 11222 | 138,26 | 1,34 | 106,35 |
| 1024 | 17336 | 2392055 | 22230 | 137,98 | 1,28 | 107,60 |

***Tabla 5.5.*** *Resultados del 2º escenario con optimización* $-O0$

| Casos (N) | Celdas | Ciclos (SW) | Ciclos (HW) | $\frac{Ciclos}{Celda}$ (SW) | $\frac{Ciclos}{Celda}$ (HW) | SpeedUp |
|---|---|---|---|---|---|---|
| 16 | 200 | 8275 | 420 | 41,38 | 2,10 | 19,70 |
| 32 | 472 | 16331 | 757 | 34,60 | 1,60 | 21,57 |
| 64 | 1016 | 32443 | 1427 | 31,93 | 1,40 | 22,74 |
| 128 | 2104 | 64667 | 2772 | 30,74 | 1,32 | 23,33 |
| 256 | 4280 | 129115 | 5461 | 30,17 | 1,28 | 23,64 |
| 512 | 8632 | 258011 | 10835 | 29,89 | 1,26 | 23,81 |
| 1024 | 17336 | 515803 | 21588 | 29,75 | 1,25 | 23,89 |

***Tabla 5.6.*** *Resultados del 2º escenario con optimización* $-O3$

en proporción muchas lecturas de datos para las celdas calculadas, y como resultado esto junto con los ciclos de configuración e interrupción, aportan bastantes ciclos de 'overhead' a lo largo de todos los casos, mitigándose en casos grandes al repartirse entre una mayor cantidad de celdas.

Para el software, ocurre que en comparación con el escenario anterior, los datos de ciclos/celda se mantienen bastante similares. Si bien es cierto que volvemos a ver que, para casos pequeños, el 'overhead' aumenta este dato, a medida que va creciendo el problema, vemos que la diferencia en ciclos/celda no es tan grande. Esto se puede deber a que, a pesar de tener bucles más cortos y realizar menos cálculos, el 'overhead' introducido por la lectura de los datos, que son la misma cantidad que en la prueba anterior, tiene un mayor peso, dejando los ciclos introducidos por las primaras dos acciones minoritarias.

Como conclusión de esta prueba, se puede observar que, debido a la lectura de datos desde la

memoria del sistema, la prueba software es más estable y no sufre de fluctuaciones grandes en la cantidad de ciclos/celda empleados entre los casos pequeños y grandes, aunque esto reduce la eficiencia. Del hardware podemos decir lo contrario, ya que en este caso no consigue llegar a ese cicla por celda teórico al tener que realizar tantos ciclos iniciales de lectura en comparación con los que emplea calculando, además de que los ciclos/celda entre los casos pequeños y los grandes fluctúan más.

Para este caso, se entiende por estabilidad que, entre los distintos casos propuestos para los escenarios, la cantidad de ciclos empleados con respecto al número de celdas totales a calcular se mantenga más o menos constante, es decir, sin mucha diferencia entre los primeros casos con *R* y *N* pequeños, y los últimos casos donde *R* y *N* son de mayor proporción.

Por último, se analizan los resultados arrojados al ejecutar el tercer escenario, representados por la Tabla 5.9, en el se fija el número de datos por serie, variando ahora el tamaño de la ventana de cálculo:

| Casos (R) | Celdas | Ciclos (SW) | Ciclos (HW) | $\frac{Ciclos}{Celda}$(SW) | $\frac{Ciclos}{Celda}$(HW) | SpeedUp |
|---|---|---|---|---|---|---|
| 1 | 5114 | 327490 | 9876 | 64,04 | 1,93 | 33,16 |
| 2 | 9196 | 512469 | 13961 | 55,73 | 1,52 | 36,71 |
| 4 | 17336 | 880864 | 22098 | 50,81 | 1,27 | 39,86 |
| 8 | 33520 | 1613450 | 38282 | 48,13 | 1,14 | 42,15 |
| 16 | 65504 | 3074034 | 70269 | 46,93 | 1,07 | 43,75 |
| 32 | 127936 | 5960546 | 132698 | 46,59 | 1,04 | 44,92 |
| 64 | 246656 | 11594946 | 251422 | 47,01 | 1,02 | 46,12 |

***Tabla 5.7.*** *Resultados del 3er escenario sin optimización explicita*

En este caso, al dejar *N* fijo en 1024 ocurre algo peculiar, y es que la fórmula contiene un $R^2$, luego cabría de esperar que el número de celdas *C(R)* creciera de manera cuadrática, sin embargo, en el rango de *R* en el que trabaja este acelerador, ese término de $R^2$ no ejerce tanta dominancia como $4094 \cdot R$, de modo que podemos decir que el número de celdas *C(R)* crece más bien de manera lineal a $4094 \cdot R$.

En el caso del hardware vemos que al principio ocurre parecido al escenario anterior con respecto a los casos pequeños, y es que empezamos teniendo unos ciclos/celda de aproximadamente 1.83, aunque en casos grandes ocurre más parecido al primer escenario, donde tienden a estabilizarse muy cerca de 1 ciclo/celda, como teóricamente debería de ser. Esto tiene sentido, pues el comienzo de esta prueba

| Casos (R) | Celdas | Ciclos (SW) | Ciclos (HW) | $\frac{Ciclos}{Celda}$(SW) | $\frac{Ciclos}{Celda}$(HW) | SpeedUp |
|---|---|---|---|---|---|---|
| 1 | 5114 | 861670 | 10006 | 168,49 | 1,96 | 86,12 |
| 2 | 9196 | 1371617 | 14090 | 149,15 | 1,53 | 97,35 |
| 4 | 17336 | 2392055 | 22230 | 137,98 | 1,28 | 107,60 |
| 8 | 33520 | 4426931 | 38414 | 132,07 | 1,15 | 115,24 |
| 16 | 65504 | 8472683 | 70398 | 129,35 | 1,07 | 120,35 |
| 32 | 127936 | 16468187 | 132830 | 128,72 | 1,04 | 123,98 |
| 64 | 246656 | 32075195 | 251550 | 130,04 | 1,02 | 127,51 |

***Tabla 5.8.*** *Resultados del 3er escenario con optimización* $-O0$

| Casos (R) | Celdas | Ciclos (SW) | Ciclos (HW) | $\frac{Ciclos}{Celda}$(SW) | $\frac{Ciclos}{Celda}$(HW) | SpeedUp |
|---|---|---|---|---|---|---|
| 1 | 5114 | 192614 | 9366 | 37,66 | 1,83 | 20,57 |
| 2 | 9196 | 300681 | 13448 | 32,70 | 1,46 | 22,36 |
| 4 | 17336 | 515803 | 21588 | 29,75 | 1,25 | 23,89 |
| 8 | 33520 | 946083 | 37772 | 28,22 | 1,13 | 25,05 |
| 16 | 65504 | 1806787 | 69756 | 27,58 | 1,06 | 25,90 |
| 32 | 127936 | 3528771 | 132188 | 27,58 | 1,03 | 26,70 |
| 64 | 246656 | 6975043 | 250908 | 28,28 | 1,02 | 27,80 |

***Tabla 5.9.*** *Resultados del 3er escenario con optimización* $-O3$

comienza teniendo muchos ciclos de lecturas en comparación con los de cómputo y a medida que se avanza, estos ciclos se amortizan mucho hasta considerarse despreciables.

Para el caso del software, se puede observar que el inicio de este escenario es algo mejor que el de los demás. Esto puede indicar que el 'overhead' por lecturas se mantiene constante sin importar el escenario y que el que se introduce por cálculos si que tiene algo más de impacto, pues a medida que aumenta la cantidad de celdas, el número de ciclos/celda se estabiliza en 28.28, muy similar a los escenarios anteriores. Cabe comentar que vuelve a mantener una gran estabilidad entre los ciclos/celda empleados en los casos pequeños y los casos grandes, indicando que su comportamiento no esta tan ligado a las variables como ocurre en el hardware

Como conclusión final, el acelerador demuestra que, a igual frecuencia de reloj, es la mejor opción en todos los casos. Sin embargo, al tratar con casos donde el radio es pequeño en comparación con la

cantidad de datos a leer, sufre de mucho 'overhead' por esa gran cantidad de lecturas, haciendo que para estos casos los ciclos/celda sean mayores que si el radio fuera más grande. Si se requiere de una estabilidad mayor, es decir, que entre los distintos casos no haya mucha diferencia de los ciclos empleados independientemente de las variables de entrada, la versión software es más indicada si no se requiere de tanta eficiencia temporal, pues de media es unas 23.16 veces más lento, aunque manteniendo unos ciclos/celda de media de 31.67, aunque en casos pequeños es algo mayor.

## 5.3 Síntesis del hardware

Se entiende por síntesis de hardware al proceso por el cual, a partir de una descripción funcional escrita en un lenguaje de descripción hardware, esta se transforma en una implementación lógica compuesta por elementos físicos disponibles en la tecnología destino, permitiendo su posterior integración o fabricación como parte de un sistema digital real. Es decir, tras este proceso se pasa de un código a un diseño real. Con esto se busca obtener una serie de datos como la potencia consumida por el hardware, el área que ocupa en un chip, o la frecuencia máxima a la que es capaz de funcionar.

Este proceso fue realizado tanto para la plataforma X-Heep por separado, como para la misma con el acelerador acoplado, esto con el objetivo adicional de validar que el conjunto es completamente funcional y se puede llevar a un diseño real. La síntesis fue realizada mediante la herramienta *OasysRTL release 2022.2.R1* [21], haciendo uso de librerías de tecnología a 45nm *FreePDK45* [11]. En concreto se realizaron cinco pasadas con diferentes frecuencias de reloj, que fueron 200 MHz, 250 MHz, $333,\overline{3}$ MHz, 500 MHz y 1000 MHz.

| | Frecuencia (MHz) | Potencia dinámica (mW) | Potencia interna (mW) | Potencia estática (mW) | Potencia total (mW) | Área ($\mu m^2$) |
|---|---|---|---|---|---|---|
| **X-Heep** | | | | | | |
| | 200 | 14,960 | 16,654 | 2,658 | 34,272 | 324161 |
| | 250 | 18,056 | 20,091 | 2,658 | 40,805 | 324161 |
| | 333 | 23,217 | 25,818 | 2,658 | 51,693 | 324161 |
| | 500 | 33,539 | 37,273 | 2,658 | 73,470 | 324161 |
| | 1000 | 64,504 | 71,638 | 2,658 | 138,800 | 324161 |
| **X-Heep+DTW** | | | | | | |
| | 200 | 20,302 | 22,932 | 14,275 | 57,509 | 1767968 |
| | 250 | 24,492 | 26,346 | 14,275 | 65,113 | 1767968 |
| | 333 | 31,476 | 32,037 | 14,275 | 77,788 | 1767968 |
| | 500 | 45,443 | 43,418 | 14,275 | 103,136 | 1767968 |
| | 1000 | 87,345 | 77,562 | 14,275 | 179,182 | 1767968 |

***Tabla 5.10.*** *Resultados de síntesis de X-Heep sin y con el acelerador acoplado a diferentes frecuencias*

En concreto lo que miden es la potencia dinámica, que es la energía consumida durante el funcionamiento activo del circuito, causada por los cambios de estado (transiciones de 0 a 1 o viceversa) de los nodos del diseño, la potencia interna, que es la energía disipada dentro de las celdas estándar durante la conmutación, independientemente de la carga de la red conectada, la potencia estática, que es la potencia consumida por estar en reposo, la potencia total, que es la suma de todas las anteriores, y el área, que es la cantidad de celdas estándar de las librerías que conforman el diseño.

Se comentan ahora los resultados obtenidos, que se pueden ver se pueden ver en la Tabla 5.10. En todos los casos de síntesis, se consigue un *slack* positivo, lo que indica que se cumple la restricción de temporización sin que el acelerador penalice el rendimiento de X-Heep y que los diseños funcionan correctamente a esas frecuencias,

En cuanto al área en celdas estándar de las librerías, como es de esperar, se mantiene constante para los dos diseños entre las distintas síntesis. Debido a esto, la potencia estática, que va directamente ligado a la cantidad de celdas que implementa un diseño, también se mantuvo constante. El número de celdas aumenta un 445% al añadir el acelerador. Esto se debe a los buffers que incorpora este en su unidad de cómputo para almacenar las series y los resultados parciales, al no haberse empleado macros para que se interpreten como memoria SRAM, lo más probable es que se hayan implementado como registros. Para la potencia estática tenemos un resultado similar, pues aumenta un 437%, que se puede deber a lo mismo, que los buffers al tener tanto tamaño producen mucha potencia estática simplemente por estar en el diseño

Con respecto a los otros datos de potencia, todos los componentes aumentan su valor a medida que se incrementa la frecuencia, como es de esperar. Sin embargo, lo hacen a ritmos diferentes. La potencia dinámica de X-Heep con el acelerador es un 35,56% más alto de media que su versión base para las distintas frecuencias evaluadas. Este incremento se mantiene relativamente constante porque el acelerador añade una cantidad fija de lógica activa cuyo consumo dinámico escala de manera proporcional con la frecuencia, del mismo modo que lo hace el resto de X-Heep. En consecuencia, el aumento porcentual se puede interpretar como el coste energético directo asociado a la inclusión del acelerador.

En cuanto a la potencia interna, se observa que también aumenta al añadir el acelerador y a medida que se incrementa la frecuencia, ya que depende simultáneamente del número de celdas presentes en el diseño y de la actividad interna de cada una de ellas. Sin embargo, la diferencia porcentual entre X-Heep base y al añadir el acelerador se reduce progresivamente a medida que aumenta la frecuencia, pasando de una diferencia del 37,69% a 200 MHz a solo de un 8,40% a 1000 MHz.

Esto se puede deber a que la potencia interna del acelerador apenas se incrementa al aumentar la frecuencia, al contrario que que ocurre en X-Heep. De este modo, la contribución del acelerador se vuelve proporcionalmente menos relevante a altas frecuencias y su impacto porcentual se diluye.

Por último, la potencia total, al ser la suma de las tres anteriormente comentadas, se comporta de manera similar, en concreto más similar a la potencia interna. Esto es por que la potencia estática es una constante, la potencia dinámica crece de manera constante y la que supone una diferencia en la forma en que aumenta el total es la potencia interna.

En conclusión, tanto el acelerador acoplado a X-Heep es sintetizable sin causar ningún tipo de interferencias, pudiendo funcionar a la misma frecuencia y manteniendo un consumo comedido con respecto al área que añade, aunque dicho consumo depende de la frecuencia a la que se ponga a trabajar al sistema. Esto confirma que se podrían llevar a un diseño físico real.

## 5.4 Análisis energético

Teniendo ya los datos en ciclos de reloj sobre la duración del cálculo de DTW, tanto en el acelerador como en la CPU, y disponiendo además de las estimaciones de potencia obtenidas para distintos valores de frecuencia, resulta interesante analizar no solo el tiempo real que tarda en completarse una ejecución del algoritmo DTW, sino también la energía consumida durante dicho proceso. Este análisis permite evaluar de forma más completa la eficiencia del acelerador frente a la implementación puramente software, ya que combina simultáneamente el coste temporal y el coste energético de cada solución.

Para dicho análisis, se usaran los valores en ciclos obtenidos de la prueba de rendimiento general con la máxima optimización expresados en la Tabla 5.3, ya que representan el caso de uso estándar del algoritmo DTW. Asimismo, se utilizan los valores de potencia total estimada para X-Heep sin y con el acelerador integrado, considerando todas las frecuencias sintetizadas, referenciados en la Tabla 5.10, lo que permite calcular la energía consumida por cada enfoque tanto en software como en hardware.

Los resultados calculados se encuentran en la Tabla 5.11, que recoge, para cada tamaño de prueba, el tiempo real de ejecución de una DTW y la energía total consumida tanto en la versión software como en la versión hardware del algoritmo. Para la potencia de la versión software, se uso la potencia total de la síntesis de solo X-Heep, ya que técnicamente con su CPU ya sería suficiente para poder ejecutar el algoritmo y no se requiere del acelerador para nada.

Para obtener el tiempo de ejecución, es suficiente con dividir la cantidad de ciclos empleados entre la frecuencia de cada caso. Una vez obtenido ese dato, podemos calcular la energía requerida para ejecutar una pasada del algoritmo DTW. Para esto, simplemente se multiplica el tiempo requerido por la potencia consumida para cada implementación.

Comentando los resultados obtenidos, se puede observar que el acelerador no solo reduce drásticamente el tiempo de ejecución de una operación DTW, sino que también disminuye de manera significativa la energía necesaria para completarla. Aunque la potencia total del sistema aumenta al incorporar el acelerador, su menor latencia compensa ampliamente este incremento, provocando que, incluso en los casos más desfavorables o en las frecuencias más altas, la energía requerida por la versión hardware sea siempre muy inferior a la de la implementación software. Se podría mejorar la eficiencia energética si se aplica alguna técnica de "gating" para mantener apagado el hardware del acelerador mientras no está en uso.

Como conclusión, a pesar de que al introducir el acelerador en X-Heep se aumenta de manera significativa la potencia total consumida, si se plantea dentro de un escenario donde el cálculo del algoritmo DTW sea la principal tarea, la integración de dicho acelerador está justificada ya que mejora simultáneamente el rendimiento y la eficiencia energética de manera sustancial.

| | Casos (N/R) | Ciclos (SW) | Ciclos (HW) | Tiempo (SW)($\mu$s) | Tiempo (HW)($\mu$s) | Consumo (SW)($\mu$J) | Consumo (HW)($\mu$J) |
|---|---|---|---|---|---|---|---|
| **200 MHz** | 16/1 | 3110 | 294 | 15,50 | 1,47 | 0,89 | 0,08 |
| **Potencia SW:** | 32/2 | 9529 | 550 | 47,65 | 2,75 | 1,63 | 0,16 |
| **34,272 mW** | 64/4 | 32443 | 1427 | 162,22 | 7,14 | 5,56 | 0,41 |
| | 128/8 | 118627 | 4620 | 593,14 | 23,10 | 20,33 | 1,33 |
| **Potencia HW:** | 256/16 | 452419 | 16762 | 2262,09 | 83,81 | 77,53 | 4,82 |
| **57,309 mW** | 512/32 | 1765699 | 64091 | 8828,49 | 320,45 | 302,57 | 18,43 |
| | 1024/64 | 6975043 | 250908 | 34875,22 | 1254,54 | 1195,20 | 72,15 |
| **250 MHz** | 16/1 | 3110 | 294 | 14,44 | 1,17 | 0,59 | 0,07 |
| **Potencia SW:** | 32/2 | 9529 | 550 | 38,12 | 2,20 | 1,55 | 0,14 |
| **40,805 mW** | 64/4 | 32443 | 1427 | 129,77 | 5,71 | 5,29 | 0,37 |
| | 128/8 | 118627 | 4620 | 474,51 | 18,48 | 19,36 | 1,20 |
| **Potencia HW:** | 256/16 | 452419 | 16762 | 1809,67 | 67,04 | 73,84 | 4,36 |
| **65,113** | 512/32 | 1765699 | 64091 | 7062,79 | 256,36 | 288,20 | 16,69 |
| | 1024/64 | 6975043 | 250908 | 27900,17 | 1003,63 | 1138,50 | 65,35 |
| **333 MHz** | 16/1 | 3110 | 294 | 9,34 | 0,88 | 0,48 | 0,07 |
| **Potencia SW:** | 32/2 | 9529 | 550 | 28,61 | 1,65 | 1,48 | 0,13 |
| **51,693** | 64/4 | 32443 | 1427 | 97,43 | 4,28 | 5,04 | 0,33 |
| | 128/8 | 118627 | 4620 | 356,24 | 13,87 | 18,41 | 1,08 |
| **Potencia HW:** | 256/16 | 452419 | 16762 | 1358,61 | 50,34 | 70,23 | 3,91 |
| **77,778 mW** | 512/32 | 1765699 | 64091 | 5302,40 | 192,46 | 274,10 | 14,97 |
| | 1024/64 | 6975043 | 250908 | 20946,07 | 753,48 | 1082,80 | 58,61 |
| **500 MHz** | 16/1 | 3110 | 294 | 6,22 | 0,59 | 0,46 | 0,06 |
| **Potencia SW:** | 32/2 | 9529 | 550 | 19,06 | 1,10 | 1,40 | 0,11 |
| **73,740 mW** | 64/4 | 32443 | 1427 | 64,88 | 2,85 | 4,77 | 0,29 |
| | 128/8 | 118627 | 4620 | 237,25 | 9,24 | 17,43 | 0,95 |
| **Potencia HW:** | 256/16 | 452419 | 16762 | 904,84 | 33,52 | 66,48 | 3,46 |
| **103,136 mW** | 512/32 | 1765699 | 64091 | 3531,40 | 128,18 | 259,45 | 13,22 |
| | 1024/64 | 6975043 | 250908 | 13950,08 | 501,81 | 1024,90 | 51,75 |
| **1000 MHz** | 16/1 | 3110 | 294 | 3,11 | 0,29 | 0,43 | 0,05 |
| **Potencia SW:** | 32/2 | 9529 | 550 | 9,53 | 0,55 | 1,32 | 0,10 |
| **138,800 mW** | 64/4 | 32443 | 1427 | 32,44 | 1,43 | 4,05 | 0,26 |
| | 128/8 | 118627 | 4620 | 118,67 | 4,62 | 16,47 | 0,83 |
| **Potencia HW:** | 256/16 | 452419 | 16762 | 452,42 | 16,76 | 62,80 | 3,01 |
| **179,182 mW** | 512/32 | 1765699 | 64091 | 1765,70 | 64,09 | 245,08 | 11,52 |
| | 1024/64 | 6975043 | 250908 | 6975,04 | 250,09 | 968,14 | 44,96 |

***Tabla 5.11.*** *Tiempos de ejecución y energía para un cálculo DTW a distintas frecuencias*

# 6

# Conclusiones y líneas futuras

## 6.1 Conclusiones

- **El algoritmo DTW es apropiado para ser acelerado en X-Heep**. Resulta ser un algoritmo computacionalmente costoso, con el que incluso aplicando optimizaciones para su ejecución, la versión software ejecutada por la CPU de X-Heep es más lenta que el acelerador desarrollado.

- **El acelerador se puede integrar perfectamente como periférico interno**. A pesar de existir más vías y más adecuadas para agregar este tipo de hardware a la plataforma X-Heep, como son las interfaces *CV-X-IF* y *XAIF*, este método ha demostrado ser suficiente para integrar el hardware sin suponer ningún cuello de botella y sin albergar una gran complejidad.

- **El acelerador mejora la eficiencia del calculo DTW**. A pesar de añadir consumo estático debido al área que ocupa el acelerador, al ejecutar el algoritmo en un menor tiempo y empleando menos ciclos, resulta mucho más eficiente en términos de energía.

- **El hardware desarrollado ha demostrado ser sintetizable**. Esto significa que, aparte de las simulaciones realizadas, se podría llevar a un diseño físico real y ser funcional en su tarea.

- **El proyecto ha desmostado ser un buen punto de partida para el aprendizaje del desarrollo hardware**. Se ha adquirido experiencia en SystemVerilog, en plataforma de hardware grandes como X-Heep así como el diseño de periféricos y creación de controladores de los mismos.

## 6.2 Líneas Futuras

- **Automatizar la generación de registros en X-Heep**. La plataforma cuenta con herramientas y script de generación que permiten realizar esta tarea de manera automática, permitido simplificar el diseño y mejor reproducibilidad. Además, es así como están diseñados los demás periféricos existentes en X-Heep.

- **Ampliar los tipos de datos que usa el acelerador**. Sería interesante realizar la implementación del acelerador con tipos de datos de punto fijo o flotante en lugar de unicamente enteros, de manera que se pueda representar números decimales y mejorar la precisión de los resultados.

- **Comparar el acelerador con GPU**. En este proyecto, solo se ha podido comparar con la CPU del sistema, la cual además cuenta con un solo núcleo lo que impide paralelizar el cálculo. Podría se interesante compararlo con arquitecturas de paralelismo masivo. En este caso, se podrían utilizar múltiples elementos procesadores dentro del acelerador.

- **Implementación del acelerador en FPGA**. Esto daría unos resultados más completos a los vistos a largo del proyecto, donde son todo se ha realizado a través de simulación.

- **Implementar el acelerador a través de los demás métodos que ofrece X-Heep**. Con esto se podría comparar cuál de ellos ofrece las mejores prestaciones para este acelerador a la hora de realizar la comunicación con la CPU de X-Heep, los periféricos del sistema y la memoria principal.

# Apéndice A. Guía de instalación y uso

Este apéndice sirve de manual para poder instalar la plataforma X-Heep para simulación, así como incluir el acelerador desarrollado en este proyecto en la mismo. Los pasos a seguir en esta guía se pueden encontrar en la documentación oficial de X-Heep [25], aunque para este caso, se requiere de cambiar algunos pasos, ya que se trabaja con versiones anteriores de algunos programas y del repositorio. Este proceso, según la guía oficial, esta probado para Ubuntu 22.04, aunque a lo largo del desarrollo de este proyecto, se ha podido comprobar que también funciona para Ubuntu 24.04 e incluso su versión en "Windows Subsystem for Linux" (WSL). Importante revisar correctamente las líneas pegadas en la terminal antes de ejecutarlas para evitar errores inesperados, así como los códigos.

También se comenta el proceso seguido para realizar la síntesis del hardware creado y utilizado a largo del proyectos. En el apartado 5.3 se hace referencia a la herramienta utilizada, así como las librerías y la tecnología.

## A.1 Montaje de X-Heep

En este apartado se explica el proceso a seguir para poder utilizar X-Heep en local para simulaciones. El primer paso es actualizar el sistema para instalar todos los paquetes y dependencias del sistema operativo. Para ello se ejecutan el siguiente comando:

```
    sudo apt update
    sudo apt upgrade
    sudo apt install autoconf automake autotools-dev curl python3 python3-
pip python3-tomli
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libto
patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-dev libslirp-
dev
help2man perl make g++ libfl2 libfl-dev zlib1g zlib1g-dev ccache mold libgoogle-
perftools
-dev numactl libelf-dev
```

En este caso, se ha añadido el paquete `libelf-dev`, ya que parece ser que es necesario más adelante a la hora compilar las aplicaciones de X-Heep, y se ha eliminado el paquete `zlibc` ya que da fallo. Una vez hecho esto, el siguiente paso es clonar el repositorio de X-Heep. Para ello, simplemente se ejecutará el comando:

```
git clone https://github.com/esl-epfl/x-heep
```

Tras esto, se tendrá el repositorio en local en su última versión. Para poder incluir el acelerador sin problemas, es necesario volver a un commit anterior, en concreto el identificado por el "hash" *2e175dc*. Esto se resuelve fácilmente ejecutando los siguientes comandos:

```
cd x-heep/
git checkout 2e175dc
```

Con esto lo que se hace es acceder al directorio creado tras clonar el repositorio y volver a esa versión concreta en la que vamos a trabajar. Tras esto, ya tenemos la base preparada, ahora falta por instalar las diferentes aplicaciones necesarias. La primera de ellas es *Miniconda*, que es una distribución ligera del gestor de entornos y paquetes *Conda*, diseñada para crear y administrar entornos de Python de forma aislada, evitando conflictos entre dependencias y versiones. Esta aplicación es necesaria ya que en X-Heep se usan scripts de Python para realizar la generación de algunos archivos tanto del hardware como del software de la plataforma. Para realizar su instalación, se ejecutarán los siguientes comandos:

```
mkdir -p ~/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-py38\_23.11.0-2-Linux-
x86\_64.sh -O
~/miniconda3/miniconda.sh
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
rm ~/miniconda3/miniconda.sh
source ~/miniconda3/bin/activate
conda init --all
```

Con esto queda *Miniconda* instalado en el sistema. Ahora solo queda activarlo dentro del entorno de X-Heep. Para ello entramos en el directorio de X-Heep y ejecutamos los siguientes comandos:

```
make conda
conda activate core-v-mini-mcu
```

El primer comando solo es necesario ejecutarlo la primera vez, ya que sirve para crear el entorno de *Conda* dentro del directorio. Sin embargo, el segundo habría que ejecutarlo al principio cada vez que se quiera trabajar con X-Heep. En este caso, se podría incluir el comando en el fichero `~/.bashrc`, que es donde el usuario puede colocar comandos y configuraciones que *Bash* carga automáticamente en cada sesión de terminal interactiva. Con esto queda instalado el entorno de Python necesario.

La siguiente herramienta a instalar es el compilador de RISC-V. Esto es lo que va a hacer que X-Heep sea capaz de ejecutar el código en C que se le indiquen. Para instalarlo, simplemente ejecutar los siguientes comandos:

```
    git clone https://github.com/riscv/riscv-gnu-toolchain
    cd riscv-gnu-toolchain
    git checkout 2023.01.03
    ./configure --prefix=/home/$USER/tools/riscv
--with-abi=ilp32 --with-arch=rv32imc --with-cmodel=medlow
    make -j $(nproc)
    cd ..
```

Para este caso, hay que definir la variable de entorno `RISC` de la siguiente manera:

```
    export RISCV=/home/$USER/tools/riscv
```

Esta también se puede incluir dentro del fichero `~/.bashrc`, pues si no es necesario redefinirla en cada sesión de terminal. Con esto ya queda instalado correctamente el compilador para RISC-V.

Se continua ahora con la instalación de Verilator, que es una herramienta de código abierto que traduce diseños hardware descritos en Verilog o SystemVerilog a código C++ o SystemC altamente optimizado para verificación hardware. Esto permitirá realizar las simulaciones de la plataforma y correr, en conjunto con el compilador, las distintas aplicaciones. Para este caso se necesita la versión 4.210, la cual no es compatible con las versiones 12.0 o superior de GCC. Para solucionar esto, primero instalamos las versiones anteriores con los siguientes comandos:

```
    sudo apt install gcc-11
    sudo apt install g++-11
```

Una vez hecho esto, se puede pasar a la instalación de Verilator en sí ejecutando lo siguiente en la terminal:

```
export VERILATOR_VERSION=4.210
git clone https://github.com/verilator/verilator.git
cd verilator
git checkout v$VERILATOR_VERSION
autoconf
CC=gcc-11 CXX=g++-11 ./configure --prefix=/home/$USER$/tools/verilator
/$VERILATOR_VERSION
CC=gcc-11 CXX=g++-11 makewh
sudo CC=gcc-11 CXX=g++-11 make install
cd ..
```

Para esta herramienta, también es necesario definir un par de variables de entorno, las cuales se recomienda incluir en `~/.bashrc`. Se definen de la siguiente manera:

```
export VERILATOR_VERSION=4.210
export PATH=/home/$USER/tools/verilator/$VERILATOR_VERSION/bin:$PATH
```

Por último, queda por instalar Verible. Esto es un conjunto de herramientas de código abierto diseñado para analizar, formatear y verificar código SystemVerilog, con el objetivo de mantener calidad, estilo y coherencia en proyectos hardware. Para instalarlo, simplemente se ejecuta lo siguiente en la terminal:

```
export VERIBLE_VERSION=v0.0-4023-gc1271a00
wget https://github.com/chipsalliance/verible/releases/download/${
VERIBLE_VERSION}/verible-${VERIBLE_VERSION}-linux-static-x86_64.tar.gz
tar -xf verible-${VERIBLE_VERSION}-linux-static-x86_64.tar.gz
mkdir -p /home/$USER/tools/verible/${VERIBLE_VERSION}/
mv verible-${VERIBLE_VERSION}/* /home/$USER/tools/verible/
${VERIBLE_VERSION}/
rm verible-${VERIBLE_VERSION}-linux-static-x86_64.tar.gz
rm -r verible-${VERIBLE_VERSION}
```

Después de la instalación, también se requiere definir un par de variables de entorno que se pueden incluir en `~/.bashrc` para simplificar. Se definen de la siguiente manera:

```
export VERIBLE_VERSION=v0.0-1824-ga3b5bedf
export PATH=/home/$USER/tools/verible/${VERIBLE_VERSION}/bin:$PATH
```

Con esto, quedan instaladas todas las herramientas y aplicaciones necesarias para trabajar con X-Heep. Lo siguiente ya es empezar a generar el sistema, compilar aplicaciones y comprobar que todo funciona correctamente. Para esto, tras haber accedido al directorio de X-Heep, haber activado el entorno de *Conda* y haber definido todas las variables de entorno, se puede realizar la primera generación del sistema ejecutando el siguiente comando:

```
make mcu-gen
```

A este comando se le pueden agregar una serie de argumentos para especificar que CPU se quiere utilizar (`CPU=<cpu>`), el tipo de bus del sistema (`BUS=<NtoM` o `onetoM>`), la cantidad de bancos de memoria (`MEMORY_BANKS=<X>`), y la cantidad de bancos de memoria intercalada (`MEMORY_BANKS_IL=<X>`), aunque para este caso con lo básico es suficiente. Ya por último queda compilar una aplicación y ejecutarla. Para esto existe el proyecto '*hello_world*', que imprime dicho mensaje por pantalla. Se compila con el comando:

```
make app
```

Y se ejecuta con el comando:

```
make run-helloworld
```

Tras esperar unos momentos y, si no existe ningún fallo, se deberá de ver por pantalla el mensaje en cuestión. Esta es la manera más básica de utilizar las aplicaciones que se encuentran en X-Heep. Si se quiere compilar y ejecutar otras, basta con añadir el argumento `PROJECT=<nombre_del_proyecto>` a la hora de compilar y ejecutarlo con el comando:

```
make run-app-verilator PROJECT=<nombre_del_proyecto>
```

Con esto, se da cierre a la explicación para montar X-Heep y su ecosistema, generar la plataforma y correr aplicaciones. Lo siguiente ya es añadir el acelerador desarrollado a lo largo de este proyecto.

## A.2 Inclusión del acelerador

En este apartado se comenta el proceso necesario a seguir para poder incluir el acelerador DTW desarrollado en la plataforma X-Heep. Para ello, es importante tener funcionando de manera óptima X-Heep en local. Lo primero a hacer es incluir los archivos nuevos a sus respectivos directorios.

Empezamos añadiendo las descripciones hardware del acelerador. Para ello, de entre las carpetas resultantes del proyecto, se copia `dtw_hw/` dentro de `hw/ip/`, para posterior mente renombrarla solo a `dtw/`.

A continuación, se añade el software controlador del acelerador. Basta con copiar la carpeta `dtw_sw` dentro de `sw/device /lib/drivers/` y renombrarla también unicamente a `dtw/`. Por último, se añaden las dos aplicaciones creadas para probar el acelerador y medir el rendimiento respectivamente. En este caso, se copian las carpetas `dtw_example` y `dtw_benchmark` dentro de `sw/applications`. Con esto estaría todo lo necesario añadido.

Lo siguiente es modificar los archivos del repositorio para que a la hora de generar la plataforma, el acelerador sea reconocido como un periférico más y se pueda utilizar en las aplicaciones sin ningún error. Este paso lo podemos dividir en cuatro secciones:

**Modificación de la configuración**: En esta sección se tratan los cambios realizados sobre los archivos de configuración de X-Heep. El primero de ellos es `general.hjson`, el cual se encuentra dentro del directorio `configs/`. En este archivo hay que modificar tres cosas. La primera de ellas es ampliar la cantidad de bancos de RAM del sistema, pues es necesario para poder almacenar sin problemas las series que usa el acelerador:

```
ram_address: 0
bus_type: "onetoM"
ram_banks: {
    code_and_data: {
        num: 4 // Pasa de 2 bancos a 4
        sizes: [32]
    }
}
// Resto del archivo
```

Lo siguiente es incluir el acelerador junto con su dirección base y el tamaño en memoria para ser mapeado a la lista de periféricos, de manera que se pueda localizar en el espacio de direcciones en el momento de configurarlo:

```
// Resto del archivo...
peripherals: {
```

```
    address: 0x30000000
    length:  0x00100000
    // perifericos existentes…
    <dtw>: {
        offset:  0x00080000
        length:  0x00001000
        is_included: "yes"
    }
    // perifericos existentes…
}
// Resto del archivo...
```

Por último, hay que incluir una linea de interrupción para el acelerador dentro del listado de interrupciones, de modo que quede constancia de que es necesaria para avisar de errores y cuando ha acabado su cálculo:

```
// Resto del archivo...
interrupts: {
    number: 64 // Do not change this number!
    list: {
        // First one is always zero
        null_intr:              0
        uart_intr_tx_watermark:  1
        // Otras señales de interupción...
        i2s_intr_event:         50
        // Nueva señal de interrupción:
         dtw_intr_event: 51
    }
}
```

Con esto, quedan realizadas las modificaciones de este archivo. El siguiente archivo a ser modificado es `core-v-mini-mcu .core`, que se encuentra en la raíz del repositorio. En el solamente hay que añadir la dependencia al nuevo periférico dentro de dicha lista, quedando tal que así:

```
filesets:
```

```
    files_rtl_generic:
        depend:
        - x-heep::packages
        - openhwgroup.org:ip:cv32e40p
        // Dependencias de otros perifericos...
        - x-heep:ip:pdm2pcm
        // Nueva dependencia
        - x-heep:ip:dtw
        - esl_epfl:ip:obi_spi_slave
// Resto del archivo...
```

Con esto, quedan concluidas las modificaciones relacionadas con la configuración de X-Heep.

**Modificación del hardware**: En esta sección se tratan las modificaciones realizadas sobre el hardware de X-Heep para poder conectar el acelerador al bus del sistema de manera que pueda ser configurado por la CPU como esclavo y que pueda acceder a la memoria como maestro. En este caso, lo que se modificaran son archivos plantilla *.tpl* que a la hora de generar el sistema, generan los archivos definitivos. Empezamos modificando los archivos dentro del directorio `hw/core-v-mini-mcu/`, que son `peripheral_subsystem.sv.tpl`, `system_bus.sv.tpl` y `core_v_mini_mcu.sv.tpl`. Además del archivo `core_v_mini_mcu_pkg.sv.tpl` dentro del subdirectorio `include/`.

En el primero de ellos, hay que añadir la entrada-salida OBI para el acelerador, la señal de interrupción especifica e incluirla en el vector de interrupciones y la instancia del nuevo módulo. Debería de quedar tal que así:

```
module peripheral_subsystem
    import obi_pkg::*;
    import reg_pkg::*;
#(
    //do not touch these parameters
    parameter NEXT_INT_RND = core_v_mini_mcu_pkg::NEXT_INT == 0 ? 1 :
core_v_mini_mcu_pkg::NEXT_INT
) (
    input logic clk_i,
    input logic rst_ni,
```

```
    // Otras señales...
    // PDM2PCM Interface
    output logic pdm2pcm_clk_o,
    output logic pdm2pcm_clk_en_o,
    input  logic pdm2pcm_pdm_i,
    // DTW Interface
    output obi_req_t  dtw_req_o,
    input  obi_resp_t dtw_resp_i
);

    // Señales internas del módulo...
    logic i2s_intr_event;
    logic dtw_intr_event;

    // this avoids lint errors
    assign unused_irq_id = irq_id;

    // Assign internal interrupts
    assign intr_vector[${interrupts["null_intr"]}] = 1'b0;
// ID [0] is a special case and must be tied to zero.
    assign intr_vector[${interrupts["uart_intr_tx_watermark"]}] =
uart_intr_tx_watermark_i;
    // Resto de asignaciones...
    assign intr_vector[${interrupts["i2s_intr_event"]}] = i2s_intr_event;
    assign intr_vector[${interrupts["dtw_intr_event"]}] = dtw_intr_event;

    // Instancias de otros módulo

    % if user_peripheral_domain.contains_peripheral('dtw'):
      dtw_top #(
          .reg_req_t(reg_pkg::reg_req_t),
          .reg_rsp_t(reg_pkg::reg_rsp_t),
```

```
            .obi_req_t(obi_pkg::obi_req_t),
            .obi_resp_t(obi_pkg::obi_resp_t)
        ) dtw_i (
            .clk_i(clk_cg),
            .rst_ni,
            .reg_req_i(peripheral_slv_req[core_v_mini_mcu_pkg::DTW_IDX]),
            .reg_rsp_o(peripheral_slv_rsp[core_v_mini_mcu_pkg::DTW_IDX]),
            .obi_req_o(dtw_req_o),
            .obi_resp_i(dtw_resp_i),
            .irq_o(dtw_intr_event)
        );
    % endif

endmodule : peripheral_subsystem
```

Con esas adiciones, queda el acelerador incluido en el subsistema de periféricos del sistema. A continuación, se realizan las modificaciones sobre `system_bus.sv.tpl`. En este archivo también hay que añadir una entrada-salido OBI, asignando esta al bus de maestros interno del sistema. Las modificaciones quedarían de esta manera:

```
module system_bus
    import obi_pkg::*;
    import addr_map_rule_pkg::*;
#(
    parameter NUM_BANKS = 2,
    parameter EXT_XBAR_NMASTER = 0,
    //do not touch these parameters
    parameter EXT_XBAR_NMASTER_RND = EXT_XBAR_NMASTER ==
0 ? 1 : EXT_XBAR_NMASTER
) (
    input logic clk_i,
    input logic rst_ni,

    // Internal master ports
```

```
    input  obi_req_t  core_instr_req_i,
    output obi_resp_t core_instr_resp_o,
    // Otros puertos maestro...
    input  obi_req_t  dtw_master_req_i,
    output obi_resp_t dtw_master_resp_o,
    // Otros puertos maestro...
)
    // Señales internas del modulo...

    // Internal master requests
    assign int_master_req[core_v_mini_mcu_pkg::CORE_INSTR_IDX]
= core_instr_req_i;
    assign int_master_req[core_v_mini_mcu_pkg::CORE_DATA_IDX]
= core_data_req_i;
    assign int_master_req[core_v_mini_mcu_pkg::DEBUG_MASTER_IDX]
= debug_master_req_i;
    assign int_master_req[core_v_mini_mcu_pkg::DTW_MASTER_IDX]
= dtw_master_req_i;

    //Resto de asignaciones...
    // Internal master responses
    assign core_instr_resp_o = int_master_resp[core_v_mini_mcu_pkg
::CORE_INSTR_IDX];
    assign core_data_resp_o = int_master_resp[core_v_mini_mcu_pkg
::CORE_DATA_IDX];
    assign debug_master_resp_o = int_master_resp[core_v_mini_mcu_pkg
::DEBUG_MASTER_IDX];
    assign dtw_master_resp_o = int_master_resp[core_v_mini_mcu_pkg
::DTW_MASTER_IDX];

    //Resto de asignaciones e instancias de módulos...
endmodule
```

Con esto, queda preparado el bus del sistema para que puedan pasar a través de él las transacciones OBI para el acelerador cuando se realzan lecturas o escrituras en la memoria. Toca pasar a las modificaciones de `core_v_mini_mcu.sv.tpl`. Aquí simplemente hay que añadir las señales OBI que sirven de puente entre las instancias de los anteriores dos módulos modificados. El resultado sería el siguiente:

```
module core_v_mini_mcu
    import obi_pkg::*;
    import reg_pkg::*;
    import fifo_pkg::*;
#(
    parameter COREV_PULP = 0,
    parameter FPU = 0,
    // Resto de parametros...
)(
    // Señales de entrada-salida del módulo...
)
    //  Más paramentros internos...

    // masters signals
    obi_req_t core_instr_req;
    obi_resp_t core_instr_resp;
    // Resto de señales de maestro...
    obi_req_t dtw_master_req;
    obi_resp_t dtw_master_resp;

    // Señales internas del módulo e instancias de otros módulos...

    system_bus #(
        .NUM_BANKS(core_v_mini_mcu_pkg::NUM_BANKS),
        .EXT_XBAR_NMASTER(EXT_XBAR_NMASTER)
    ) system_bus_i (
        .clk_i,
```

```
        .rst_ni(rst_ni && debug_reset_n),
        // Conexiones entre entradas-salida y señales internas...
        .dtw_master_req_i(dtw_master_req),
        .dtw_master_resp_o(dtw_master_resp),
        // Conexiones entre entradas-salida y señales internas...
    )

    // Instancias de otros módulos...

    peripheral_subsystem peripheral_subsystem_i (
        .clk_i,
        .rst_ni(peripheral_subsystem_rst_n && debug_reset_n),
        .clk_gate_en_ni(peripheral_subsystem_clkgate_en_n),
        // Conexiones entre entradas-salida y señales internas...
        .dtw_req_o(dtw_master_req),
        .dtw_resp_i(dtw_master_resp)
    );

    //Asignaciones de señales...
endmodule  // core_v_mini_mcu
```

Con esto, queda todo perfectamente interconectado.

Por último, queda por modificar `core_v_mini_mcu_pkg.sv.tpl`, donde solamente hay que asignar un índice al acelerador DTW y aumentar en uno el número de maestros de bus que hay en el sistema. Debería quedar algo así:

```
package core_v_mini_mcu_pkg;

    import addr_map_rule_pkg::*;
    import power_manager_pkg::*;
    // Código del paquete...
    //master idx
    localparam logic [31:0] CORE_INSTR_IDX = 0;
    localparam logic [31:0] CORE_DATA_IDX = 1;
```

```
    localparam logic [31:0] DEBUG_MASTER_IDX = 2;
    localparam logic [31:0] DMA_READ_P0_IDX = 3;
    localparam logic [31:0] DMA_WRITE_P0_IDX = 4;
    localparam logic [31:0] DMA_ADDR_P0_IDX = 5;

    localparam logic [31:0] DTW_MASTER_IDX = 9;

    localparam SYSTEM_XBAR_NMASTER = ${1 + 3 +
int(dma.get_num_master_ports())*3};
    // Resto del código del paquete...
endpackage
```

Tras todas estas modificaciones, el hardware de la plataforma queda preparado para aceptar correctamente al nuevo acelerador como un periférico más.

**Modificación del software**: En esta sección se tratan las modificaciones realizadas sobre el software ya existente en X-Heep. En concreto los cambios que hay que hacer están relacionados con el controlador de interrupciones de la plataforma, de manera que pueda identificar cuando el acelerador activa su línea de interrupción y poder actuar en consecuencia. Para ello tenemos que modificar los archivos `rv_plic.h` y `rv_plic.c`. Estos archivos se encuentran en `/sw/device/lib/drivers/rv_plic/`.

Empezando por el primero de ellos, hay que definir el *ID* del acelerador a través del *ID* de interrupción que se le asigna. Este archivo queda de la siguiente manera:

```
//Codigo de la cabecera...
/**
 * ID of the I2S interrupt request lines
 */
#define I2S_ID          I2S_INTR_EVENT

/**
 * ID of the DTW interrupt request lines
 */
#define DTW_ID          DTW_INTR_EVENT
```

```
/**
 * ID of the external interrupt request lines
*/
#define EXT_IRQ_START   EXT_INTR_0
// Resto del código de la cabecera...
```

Del segundo archivo, hay que incluir la cabecera del driver del acelerador DTW, además de añadir un puntero al gestor de la interrupción del acelerador dentro de la lista de estos. El resultado sería el siguiente:

```
// Diferentes "include" de otras librerias...
// Peripheral modules from where to obtain the irq handlers
// Cabeceras de otros módulos...
#include "spi_host.h"
#include "dtw.h"
// Código del driver...
void plic_reset_handlers_list(void)
{
    handlers[NULL_INTR] = &handler_irq_dummy;
    for( uint8_t i = NULL_INTR +1; i < QTY_INTR; i++ )
    {
        if ( i <= UART_ID_END)
        {
            handlers[i] = &handler_irq_uart;
        }
        // Resto de casos para otros periféricos...
        }
        else if ( i == DTW_ID)
        {
            handlers[i] = &handler_irq_dtw;
        }
        else
        {
            handlers[i] = &handler_irq_dummy;
```

```
        }
    }
}
// Resto del código del driver...
```

Tras dichas modificaciones, el sistema es capaz de interpretar correctamente la interrupción del acelerador y enrutarla hacia la CPU para su posterior tratamiento.

**Modificación de scripts**: En esta última sección, se explican las modificaciones realizadas sobre algunos de los scripts de Python usados para la generación de la plataforma. Dichos archivos son `laod_config.py` y `user_peripherals.py`. El primero se encuentra en `util/x_heep_gen/`, y dentro de este mismo directorio, en `peripheral/` está el segundo. Además, se modifica el `Makefile` para facilitar la ejecución de las aplicaciones antes añadidas para el uso del acelerador.

Para el primer archivo, se añade *DTW* a la lista de periféricos de usuario, y, en una función usada para cargar los periféricos, se añade la instancia de DTW para que sea detectado y configurado. Como resultado queda lo siguiente:

```
// Código de Python...
from .peripherals.user_peripherals import (
    UserPeripheralDomain,
    RV_plic,
    SPI_host,
    GPIO,
    I2C,
    RV_timer,
    SPI2,
    PDM2PCM,
    I2S,
    DTW,
)
// Código de Python...
def load_peripherals_config(system: XHeep, config_path: str):
    """
    Reads the whole peripherals configuration.
```

```
    :param XHeep system: the system object where the peripherals
should be added.
    :param str config_path: The path to the configuration file.
    :raise ValueError: If config file does not exist or if peripheral
name doesn't match a peripheral class.
    """
    // Resto de la función...
        elif peripheral_name == "i2s":
            peripheral = I2S(offset, length)
            peripheral.custom_configuration(peripheral_config["path"])
        elif peripheral_name == "dtw":
            peripheral = DTW(offset, length)
            if "path" in peripheral_config:
                peripheral.custom_configuration(peripheral_config["path"])
    // Resto del código Python...
```

Para el segundo archivo, se define la clase *DTW*. Se incluye de la siguiente manera después de la clase del periférico *I2S*:

```
class DTW(UserPeripheral, DataConfiguration):
    """
    Dynamic Time Warping Accelerator

    Default configuration file: ./hw/ip/dtw/data/dtw.hjson
    """

    _name = "dtw"
    _config_path = "./hw/ip/dtw/data/dtw.hjson"

    def __init__(self, address: int = None, length: int = None):
        super().__init__(address, length)
```

Por último, se modifica el `Makefile` añadiendo los siguientes dos conjuntos de comandos al final del archivo:

```
run-dtw-demo:
 $(MAKE) mcu-gen
 $(MAKE) verilator-sim
 $(MAKE) run-app-verilator PROJECT=dtw_example

run-dtw-benchmark:
 $(MAKE) mcu-gen
 $(MAKE) verilator-sim
 $(MAKE) run-app-verilator PROJECT=dtw_benchmark COMPILER_FLAGS=-O3
```

Con esta última modificación, la plataforma queda completa con el acelerador para el algoritmo DTW añadido como periférico. Para comprobar que funciona correctamente, solo haría falta ejecutar uno de los dos comandos '*make*' añadidos al Makefile y comprobar que no existe ningún error al generar los archivos de la plataforma ni al compilar. Tras unos momentos, verás la salida por pantalla la aplicación que hayas ejecutado.

## A.3 Proceso de síntesis

Para este proceso, lo primero es tener listados todos los archivos que se va a utilizar para sintetizar. Estos listados son los archivos `xheep_only_filelist.f`, que contiene la lista de archivos exclusivos de X-Heep, y `xheep_filelist.f`, que contiene el listado de archivos de X-Heep junto con los del acelerador.

Para conseguir la lista de archivos de X-Heep se aprovecho el comando `make asic` de su Makefile para sacar el listado (`openhwgroup.org_ systems_core-v-mini-mcu_0-read-sources.tcl`), pero con el estilo de la herramienta de síntesis que se usa en dicha plataforma. A ese listado bastó con aplicar el siguiente comando, además de completar las rutas de los archivos:

```
grep -oE '/[^ ]+\.(sv|v)' openhwgroup.org_systems_core-v-mini-mcu_0-read-sources.tcl
    > xheep_filelist.f
```

Con esto, ya la herramienta puede leer los archivos de descripción hardware para realizar la síntesis. Para el proceso de síntesis en sí, lo mejor es tener un script con todas las acciones necesarias, desde leer la librerías de la tecnología hasta aplicar las optimizaciones y escribir los reportes en un archivo. Dichos scripts son los archivos `DTW.tcl`, `X-Heep.tcl` y `X-Heeo+DTW.tcl`, cada uno pensado para el escenario correspondiente a su nombre. Ejecutando en la terminal el comando:

`oasys <script>.tcl`,

se lleva a cabo el proceso de ejecución de la síntesis del hardware especificado dentro del script.

# Bibliografía


[1] Donald J. Berndt and James Clifford. "Using Dynamic Time Warping to Find Patterns in Time Series". In: *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining (KDD Workshop)*. 1994, pp. 359–370.

[2] Imen Boulnemour, Bachir Boucheham, and Slimane Benloucif. "Improved Dynamic Time Warping for Abnormality Detection in ECG Time Series". In: *Bioinformatics and Biomedical Engineering*. Ed. by Francisco Ortuño and Ignacio Rojas. Cham: Springer International Publishing, 2016, pp. 242–253. ISBN: 978-3-319-31744-1.

[3] Heng-Da Cheng and King-Sun Fu. "VLSI architecture for dynamic time-warp recognition of handwritten symbols". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 34.3 (1986), pp. 603–613. DOI: 10.1109/TASSP.1986.1164836.

[4] Mohamed Dilmi et al. "Iterative Multiscale Dynamic Time Warping (IMs-DTW): A tool for rainfall time series comparison". In: *International Journal of Data Science and Analytics* 10 (June 2020). DOI: 10.1007/s41060-019-00193-1.

[5] ESL EPFL. *X-HEEP: eXtendable Heterogeneous Energy-Efficient Platform based on RISC-V*. https://github.com/esl-epfl/x-heep. Accedido el 3 de noviembre de 2025. 2025.

[6] Fumitada Itakura. "Minimum prediction residual principle applied to speech recognition". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 23.1 (1975), pp. 67–72. DOI: 10.1109/TASSP.1975.1162641.

[7] lowRISC Community. *lowRISC: Open Source Silicon*. https://www.lowrisc.org/. 2021.

[8] Simone Machetti et al. *X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators*. 2024. arXiv: 2401.05548 [cs.AR]. URL: https://arxiv.org/abs/2401.05548.

[9] Jonathan Martinez et al. "Boosted-SpringDTW for Comprehensive Feature Extraction of PPG Signals". In: *IEEE Open Journal of Engineering in Medicine and Biology* 3.1 (May 2022), pp. 78–85. DOI: 10.1109/OJEMB.2022.3174806. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC9299207/.

[10] Marion Moseby. *DTW matrix and pathway*. https://es.wikipedia.org/wiki/Archivo:DTW_matrix_and_pathway.gif. Licencia Creative Commons Attribution–Share Alike 4.0 International (CC BY-SA 4.0). 2020.

[11] North Carolina State University (NCSU) EDA Group. *FreePDK45: 45 nm variant of the FreePDK process design kit*. https://eda.ncsu.edu/freepdk/freepdk45/. Accedido el 20 de noviembre de 2025. 2025.

[12] OpenHW Group. *OpenHW Group: Open Source IP Cores and Platforms*. https://www.openhwgroup.org/. 2021.

[13] OpenHW Group. *Open Bus Interface (OBI) Specification, Version 1.6.0*. Accedido el 4 de noviembre de 2025. OpenHW Group. 2023. URL: https://github.com/openhwgroup/core-v-docs/blob/master/OBI-v1.6.0.pdf.

[14] OpenHW Group and contributing members. *Core-V eXtension interface (CV-X-IF) — Introduction*. https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/index.html. Revision f08d951d. Licensed under Apache-2.0 / SHL-2.1. Accedido el 12 de noviembre de 2025. 2024.

[15] PULP Platform. *Register Interface*. https://github.com/pulp-platform/register_interface. Accedido el 1 de noviembre de 2025. 2021.

[16] PULP Platform Team. *Parallel Ultra Low Power (PULP) Platform*. https://pulp-platform.org/. 2022.

[17] Rangaraj M. Rangayyan. *Biomedical Signal Analysis: A Case-Study Approach*. Hoboken, NJ: IEEE Press and Wiley, 2015. ISBN: 978-1-118-97961-8.

[18] Davide Rossi, Antonio Pullini, and Luca Benini. "X-HEEP: An Open Hardware Platform for Ultra-Low-Power RISC-V Research". In: *Proceedings of the 2023 Design, Automation & Test in Europe Conference*. Available at: https://github.com/pulp-platform/x-heep. IEEE. 2023. URL: https://github.com/pulp-platform/x-heep.

[19] Hiroaki Sakoe and Seibi Chiba. "Dynamic programming algorithm optimization for spoken word recognition". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26.1 (1978), pp. 43–49. DOI: 10.1109/TASSP.1978.1163055.

[20] Doruk Sart et al. "Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs". In: *2010 IEEE International Conference on Data Mining*. 2010, pp. 1001–1006. DOI: 10.1109/ICDM.2010.21.

[21] Siemens Digital Industries Software. *Oasys-RTL: Physical RTL Synthesis*. https://eda.sw.siemens.com/en-US/ic/oasys-rtl/. Accedido el 17 de noviembre de 2025. 2025.

[22] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. May 2011. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html.

[23] Sir-Lord Wiafe et al. "The dynamics of dynamic time warping in fMRI data: A method to capture inter-network stretching and shrinking via warp elasticity". In: *Imaging Neuroscience* 2 (June 2024), imag-2–00187. ISSN: 2837-6056. DOI: 10.1162/imag_a_00187. eprint: https://direct.mit.edu/imag/article-pdf/doi/10.1162/imag_a_00187/2380311/imag_a_00187.pdf.

[24] X-HEEP Platform Team. *Integrate a Peripheral IP — X-HEEP 1.0 documentation*. https://x-heep.readthedocs.io/en/latest/How_to/IntegratePeripheral.html. Accedido el 1 de noviembre de 2025. 2023.

[25] X-HEEP Platform Team. *X-HEEP 1.0 Documentation*. https://x-heep.readthedocs.io/en/latest/. Accedido el 1 de noviembre de 2025. 2023.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga