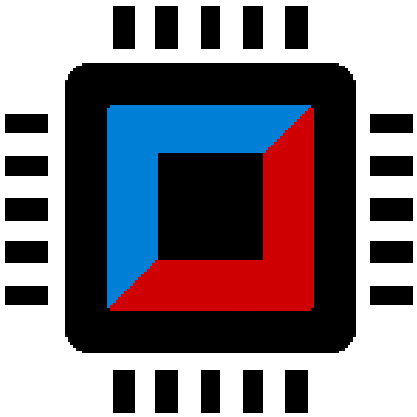
Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN SISTEMAS ELECTRÓNICOS AVANZADOS

TRABAJO FIN DE MÁSTER

INTEGRACIÓN Y CARACTERIZACIÓN DEL RENDIMIENTO DE UNIDADES COPROCESADORAS EN NÚCLEO RISC-V PARA APLICACIONES DE IA



Estudiante: Sainz Estebanez, Unai

Director/Directora: Basterretxea Oyarzabal, Koldo

Curso: 2023-2024

Fecha: Bilbao, 30, 09, 2024

UNIVERSIDAD DEL PAÍS VASCO

TRABAJO FIN DE MÁSTER

Integración y caracterización del rendimiento de unidades coprocesadoras en núcleo RISC-V para aplicaciones de IA

Autor:
Unai SAINZ ESTEBANEZ

Director:
Dr. Koldo BASTERRETXEA OYARZABAL

El presente Trabajo Fin de Máster

se ha realizado en el

Grupo de Investigación de Diseño en Electrónica Digital

30 de septiembre de 2024

«L'umanità è a un bivio. O torna a credere di avere una natura diversa rispetto alle macchine o sarà ridotta a macchina tra le macchine.»

Federico Faggin

Resumen

Para llevar a cabo la inferencia de IA de manera generalizada, se requieren dispositivos de procesamiento que sean energéticamente eficientes, compactos y altamente fiables. En este sentido, las arquitecturas de procesamiento heterogéneo, que combinan CPUs personalizadas con coprocesadores de aplicación específica, proporcionan un buen equilibrio entre eficiencia computacional y flexibilidad, resultando adecuadas para desplegar en ellas IA en el borde. Además, estas arquitecturas permiten reducir los tiempos de desarrollo en comparación con los procesadores completamente personalizados. Siguiendo la iniciativa de promover la soberanía europea en el ámbito de la microelectrónica, en este trabajo se propone el uso de una plataforma hardware de código abierto basada en RISC-V, así como de herramientas de Automatización del Diseño Electrónico (EDA) Free/Libres y/o de Código Abierto (FLOS) para evaluar el rendimiento de diferentes opciones de integración de coprocesadores en un Sistema-en-Chip (SoC) prototipado sobre FPGA. Se evalúan cuatro opciones de integración (Stream, XBUS, CFS y CFU) con objeto de obtener datos que permitan tomar decisiones de diseño precisas para el desarrollo futuro de dispositivos integrados destinados al procesamiento de alto rendimiento de IA en el borde. Adicionalmente, se verifica el beneficio de este enfoque comparando, en términos de latencia computacional, el cálculo de la función de activación sigmoide mediante la arquitectura heterogénea propuesta frente al uso exclusivo de la CPU. Para este propósito, se ha integrado un acelerador basado en el método de Interpolación Recursiva Centrada (CRI) a través de una instrucción personalizada de la extensión ISA Zxcfu, logrando un ratio de aceleración promedio y máximo de 15,25:1 y 21,53:1, respectivamente. Estos resultados confirman que la arquitectura distribuida propuesta es capaz de mejorar significativamente el rendimiento en el cálculo de las funciones de activación dentro del contexto de las redes neuronales artificiales (RNAs).

Laburpena

AAren inferentzia modu orokorrean aurrera eramateko, energetikoki eraginkorrak, estrukutralki trinkoak eta erabat fidagarriak diren gailuak ezinbestekoak dira. Ildo horretatik, prozesamendu heterogeneoko arkitekturek, PUZ pertsonalizatuak aplikazio espezifikoko koprozesadoreekin konbinatzen dituztenek, oreka egokia lortzen dute eraginkortasun konputazionalaren eta malgutasunaren artean, hortaz, egokiak izanez bertan AA ertzean zabaltzeko. Halaber, arkitektura horiek garapen-denborak murriztea ahalbidetzen dute, prozesadore erabat pertsonalizatuekin alderatuta. Mikroelektronikaren esparruan Europaren subiranotasuna sustatzeko ekimenari jarraituz, lan honetan RISC-V-an oinarritutako kode irekiko hardware plataforma baten erabilera proposatzen da, baita Diseinu Elektronikoa Automatizatzeko (EDA) tresna Free/Libre eta/edo Kode Irekikoa (FLOS) ere, FPGA-an prototipatutako Txip bidezko Sistema (SoC) batean, koprozesadoreak integratzeko aukera ezberdinen errendimendua ebaluatzeko. Lau integrazio aukera (Stream, XBUS, CFS y CFU) ebaluatzen dira datuak eskuratzeko asmoz, AAertzean-aren errendimendu handiko prozesamendurako zuzenduta dauden gailu integratuen etorkizuneko garapenerako diseinu-erabaki zehatzak hartzea ahalbidetzen dituztenak. Horrez gain, ikuspegi horren onura egiaztatzen da, proposatutako arkitektura heterogeneoaren bidezko sigmoide aktibazio-funtzioaren kalkuluaren, eta PUZ-aren erabilera esklusiboaren latentzia konputazionala konparatuz. Helburu honetarako, Interpolazio Errekurtsibo Zentratua (CRI) metodoan oinarritutako azeleratzailea integratu da, ISA Zxcfu hedapenaren agindu pertsonalizatu baten bidez, 15,25:1 eta 21,53:1 -ko batazbesteko azelerazio ratioa eta ratio maximoa lortuz hurrenez hurren. Emaitza horiek berresten dute proposatutako arkitektura banatua gai dela neurona-sare artifizialen (RNA) testuinguruan aktibazio-funtzioen kalkuloetan errendimendua nabarmen hobetzeko.

Abstract

Performing AI inference ubiquitously requires energy-efficient, small footprint and highly reliable processing devices. Heterogeneous processing architectures combining customized CPUs with domain specific coprocessors can provide a good tradeoff between computational efficiency and application flexibility for edge AI deployments while shortening development times compared to full custom application-specific processor designs. Following the impulse for the European sovereignty in the microelectronics field, in this work we propose the use of a RISC-V based open-source hardware platform and Free/Libre and/or Open Source (FLOS) Electronic Design Automation (EDA) tools to evaluate the performance of different coprocessor integration options in a System-on-Chip (SoC) prototyped on FPGA. We tested four integration options (XBUS, Stream, CFS and CFU) to obtain precise data that will allow making the correct design decisions for the future development of integrated devices for high-performance AI at the edge. Additionally, the benefit of this approach was verified by comparing the computational latency of the sigmoid activation function calculation using the proposed heterogeneous architecture versus the exclusive use of the CPU. For this purpose, an accelerator based on the Centered Recursive Interpolation (CRI) method was integrated through a custom instruction of the Zxcfu ISA extension, achieving an average and maximum acceleration ratio of 15.25:1 and 21.53:1, respectively. These results confirmed that the proposed distributed architecture can significantly improve the performance of activation function calculations within the context of artificial neural networks (ANNs).

Índice general

Índice de figuras

Índice de Tablas

Lista de acrónimos

AI	**A**rtificial **I**ntelligence
ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
AXI	**A**dvanced e**X**tensible **I**nterface
CFU	**C**ustom **F**unction **U**nit
CFS	**C**ustom **F**unction **S**ubsystem
CI	**C**ontinuous **I**ntegration
CLI	**C**ommand-**L**ine **I**nterface
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
CRI	**C**entered **R**ecursive **I**nterpolation
CSR	**C**ontrol and **S**tatus **R**egisters
CSV	**C**omma-**S**eparated **V**alues
CXU	**C**omposable e**X**tension **U**nit
DDR	**D**ouble **D**ata **R**ate
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
EDA	**E**lectronic **D**esign **A**utomation
ESA	**E**uropean **S**pace **A**gency
FA	**F**unción de **A**ctivación
FIFO	**F**irst **I**n **F**irst **O**ut
FLOS	**F**ree/**L**ibre and/or **O**pen **S**ource
FOSS	**F**ree and **O**pen **S**ource **S**oftware
FPGA	**F**ield **P**rogrammable **G**ate **A**rrays
FPGAIF	**FPGA I**nterchange **F**ormat
FPU	**F**loating **P**oint **U**nit
GCC	**G**NU **C**ompiler **C**ollection
GDB	**G**NU **D**e**B**ugger
HDL	**H**ardware **D**escription **L**anguage
ILA	**I**ntegrated **L**ogic **A**nalyzer
IMEM	**I**nstruction **MEM**ory
ISA	**I**nstruction **S**et **A**rchitecture
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
JTAG	**J**oint **T**est **A**ction **G**roup
IA	**I**nteligencia **A**rtificial
MAC	**M**ultiply–**AC**cumulate
MLO	**M**odified **L**attice **O**perators
MSB	**M**ost **S**ignificant **B**it
ODS	**O**bjetivos de **D**esarrollo **S**ostenible
OSVVM	**O**pen **S**ource **V**HDL **V**erification **M**ethodology
PCI	**P**eripheral **C**omponent **I**nterconnect
POWER	**P**erformance **O**ptimization **W**ith **E**nhanced **R**ISC

RAM	**R**andom **A**ccess **M**emory
ReLU	**Re**ctified **L**inear **U**nit
RNA	**R**edes **N**euronales **A**rtificiales
ROM	**R**ead **O**nly **M**emory
RTOS	**R**eal **T**ime **O**perating **S**ystem
SATA	**S**erial **A**dvanced **T**echnology **A**ttachment
SDK	**S**oftware **D**evelopment **K**it
SLINK	**S**tream **LINK** Interface
SoC	**S**ystem **o**n a **C**hip
SPARC	**S**calable **P**rocessor **ARC**hitecture
SPI	**S**erial **P**eripheral **I**nterface
TCL	**T**ool **C**ommand **L**anguage
UART	**U**niversal **A**synchronous **R**eceiver-**T**ransmitter
UVVM	**U**niversal **V**HDL **V**erification **M**ethodology
VHDL	**VHSIC** **H**ardware **D**escription **L**anguage
VHSIC	**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
XBUS	Processor-E**X**ternal **BUS** Interface
XIP	e**X**ecute **I**n-**P**lace
YML	**Y**AML Ain't **M**arkup **L**anguage

En agradecimiento a Jon, Koldo, Oscar y Unai miembros de GDED y a Stephan Nolting por su contribución al hardware libre.

Capítulo 1

Memoria

1.1. Introducción

Atendiendo al contexto socioeconómico actual es razonable señalar que el conjunto de conocimientos, tanto teóricos como tecnológicos, asociados al desarrollo de la IA (Inteligencia Artificial) suponen un ámbito científico estratégico. En este sentido, la investigación actúa como herramienta vertebradora de su progreso. De esta manera, se faculta a los ingenieros/as para materializar y perfeccionar las innovaciones que definirán su evolución.

En términos generales, la IA está basada en modelos de redes neuronales. Existen varias formas de enfocar la gestión computacional de estos modelos. La tendencia general es emplear unidades de procesamiento gráfico en combinación con procesadores complejos administrados por sistemas operativos. De este modo, se consigue gestionar grandes volúmenes de datos, lo que conlleva un entrenamiento óptimo y en consecuencia un aprendizaje exitoso. No obstante, la infraestructura necesaria para tal fin ocupa un tamaño considerable. Además, se requiere de una gran cantidad de energía. Hay ocasiones en las que el tamaño y/o la potencia consumida son factores determinantes, por ejemplo, en unidades autónomas con capacidad de operar aisladas como satélites, vehículos autónomos etc. En estos casos la IA es necesaria para el desempeño de sus funciones, pero su procesamiento debe estar descentralizado en su propio hardware. Este procesamiento de algoritmos de *Machine Learning* a nivel local y sin necesidad de estar conectados a internet es lo que se conoce como IA en el borde [1] (*AI at the edge*) y surge en contraposición a los modelos conectados a la nube. Además, cabe destacar que realizar por completo el creciente número de procesos de IA en la nube no sólo no es deseable, sino que es insostenible [2].

Para llevar a cabo la inferencia de IA en el borde, se sugiere un enfoque de gestión computacional distribuido. A este respecto, se plantea el uso de FPGAs. Se propone implementar en estos dispositivos una arquitectura SoC heterogénea con un microcontrolador acoplado a coprocesadores de aplicación específica. De esta manera, se consigue distribuir las tareas con objeto de ganar flexibilidad y lograr una mayor eficiencia. Por un lado, el microcontrolador permite programabilidad y capacidad de gestionar los datos de entrada/salida. Por otro lado, los coprocesadores embebidos permiten realizar las operaciones asociadas a las redes neuronales con un alto rendimiento. Existen varios modos de acoplar estos dos elementos. A lo largo del desarrollo de este trabajo, se realiza una caracterización del rendimiento de los diferentes modos de conexión que ofrece el microcontrolador RISC-V seleccionado. Mediante esta caracterización se pretende concluir el método de acoplamiento más adecuado en términos de latencia. Como se ha mencionado, la sugerencia de distribución computacional para llevar a cabo IA en el borde conlleva externalizar los

cálculos asociados a las redes neuronales del micro a coprocesadores embebidos. En lo referente a este trabajo, se comienza por evaluar un acelerador relativo al cálculo de funciones de activación, particularmente la función sigmoide. Se escoge esta función ya que requiere de operaciones matemáticas que no son directas para la ALU. Esto le supone al microcontrolador numerosos ciclos de reloj para calcularla. Por lo tanto, emplear un coprocesador embebido para acelerar este proceso resulta en un ahorro significativo del tiempo de computación. Este acelerador, se integra al microcontrolador mediante el método previamente concluido como el más adecuado. Finalmente, se corroboran los beneficios de emplear este tipo de enfoque distribuido.

La investigación presentada en este trabajo ha sido desarrollada en el Grupo de Investigación de Diseño en Electrónica Digital (GDED). El grupo está familiarizado con el diseño hardware, así como con su descripción empleando principalmente el lenguaje VHDL. Asimismo, fomenta el uso de herramientas FLOS, así como la metodología de integración continua (CI), que en conjunción con programas/plataformas de control de versiones, han sido esenciales para la gestión eficaz de todo el código empleado. En definitiva, la colaboración con el grupo ha sido determinante para la correcta realización de este Trabajo Fin de Máster.

1.2. Contexto

A la hora de implementar una arquitectura heterogénea, es preciso que los elementos que la componen se caractericen por su flexibilidad. Uno de los pilares fundamentales en lo referente a una solución distribuida, es la parte relativa al procesador por instrucciones. En este sentido, hay ocasiones en las que se requiere modificar sus fuentes descriptivas, con objeto de personalizar su funcionamiento. De este modo, se logra adaptar la CPU a la coyuntura objetivo, por ejemplo, integrando en ella los coprocesadores específicos necesarios. En consecuencia, se consigue amoldar en un diseño Sistema-en-Chip (SoC) la resolución relativa al ámbito de aplicabilidad concreto. Para conseguir la plasticidad descrita, se requiere interactuar con los diseños mediante las descripciones conceptuales de los mismos. A este respecto, se propone trabajar con las fuentes HDL de procesadores basados en una ISA libre de derechos. Esta decisión no es de corte dogmática, sino que está fundamentada en las posibilidades que ella ofrece, tales como: la fabricación en silicio libre de regalías (*royalty-free*) de soluciones SoC basados en estas ISAs, la difusión sin restricciones en plataformas públicas de las descripciones de CPUs y derivados, la publicación de comparativas de rendimiento etc. Efectuando un repaso por las principales ISAs abiertas/libres, existe una relativamente moderna que destaca sobre el resto, esta es RISC-V [3]. Nativa de Berkeley, esta ISA ha definido, en gran medida, el panorama actual de implementaciones de procesadores de hardware libre. De este modo, numerosos usuarios, así como organizaciones, han distribuido sus propias soluciones de CPUs y microcontroladores basados en esta ISA. Como resultado, la oferta de procesadores *soft-core* se ha multiplicado en los últimos años, disponiendo de una accesibilidad de estas descripciones nunca antes vista. Además, en lo que respecta a las investigaciones relativas a la aceleración de aplicaciones de IA mediante coprocesadores acoplados a CPUs RISC-V, se ha observado un aumento considerable en los últimos 3 años. Adicionalmente, cabe destacar que la trascendencia del proyecto RISC-V ha sido tan significativa que la industria privada relacionada con el sector se ha visto obligada a ofertar sus propias soluciones basadas en esta ISA o incluso a abrir alguna de sus ISAs cerradas para acomodarse al nuevo paradigma actual.

1.2.1. ISAs libres

A grandes rasgos, la CPU o núcleo del microcontrolador, es la unidad encargada de decodificar y ejecutar secuencialmente las instrucciones que previamente extrae de un programa almacenado en memoria. Estas instrucciones codifican operaciones y dependen de la arquitectura que las procesa para aplicarlas. En otras palabras, el mecanismo secuencial de extracción/decodificación/ejecución de instrucciones transforma sucesivamente una operación (aritmeticológica, de movimiento de datos etc.) de un plano conceptual, codificada en un programa, a un plano material, es decir, físicamente haciendo uso de los recursos electrónicos de la arquitectura digital. Es por ello que el set de instrucciones está íntimamente ligado a la arquitectura y definirlo condiciona de manera abstracta la topología de la misma. En este sentido, una arquitectura de conjunto de instrucciones (ISA) es una especificación de instrucciones que permite generar arquitecturas de procesadores a partir de ella. Además, este concepto debe estar complementado con un soporte para compilar lenguajes de alto nivel, como por ejemplo C, que mediante las herramientas y librerías necesarias traduzcan su sintaxis a instrucciones para una ISA en concreto.

En el siglo pasado, el desarrollo de microarquitecturas estaba completamente monopolizado por el ámbito privado. Esto se debía a que el esfuerzo de definir una ISA y dar soporte para la compilación de su set de instrucciones, además de verificar todo su ecosistema, era demasiado elevado para afrontarlo desde un punto de vista *Open Source*. Es por ello que a comienzos del siglo XXI eran pocos los proyectos de hardware libre de este tipo. Caben mencionar los procesadores OpenRISC [4] basados en su propia ISA y desarrollados por la comunidad OpenCores [5] y los procesadores OpenSPARC [6] y LEON [7] (proyecto referente de la ESA), basados en la ISA SPARC. No obstante, entre mediados de los 2000 y de la década de 2010, han aumentado las especificaciones de ISAs libres de derechos que se han puesto a disposición de los usuarios. Esto se ha debido principalmente a tres motivos. El primero de ellos es porque las patentes han expirado. Este es el caso de la ISA SuperH [8]. Durante la crisis económica asiática de 1997 Hitachi se asoció con Mitsubishi y escindió su división de microcontroladores en una nueva compañía llamada Renesas [9]. Esta compañía no heredó los ingenieros de Hitachi que habían desarrollado SuperH, por lo que con el tiempo Renesas perdió el interés por ella y expiraron sus patentes. Un ejemplo de diseño de código abierto que utiliza este set de instrucciones es el procesador J-core [10]. El segundo motivo es porque se han dado las circunstancias materiales necesarias para llevar a cabo un proyecto de ISA abierto concebido desde el primer momento como *Open Source*. Este es el caso de RISC-V [3]. El tercer motivo es porque a consecuencia de la enorme relevancia del proyecto anterior, las compañías intencionalmente han decidido abrir las especificaciones. Este es el caso de POWER ISA [11]. A principios de la década de 1990, IBM desarrolló una ISA llamada IBM POWER (*Performance Optimization With Enhanced RISC*) para la familia de procesadores POWER. Más tarde, a finales de los 90, esta arquitectura se abandonó y lo que se había convertido en PowerPC evolucionó hasta que en 2006 se estableció como POWER ISA. En 2019, IBM decidió hacer esta ISA *Open Source* y desde entonces es mantenida por la fundación OpenPower [12]. Tres ejemplos de diseños de código abierto que utilizan este set de instrucciones son Microwatt [13], Chiselwatt [14] y Libre-SOC [15].

1.2.2. Ecosistema RISC-V

En mayo de 2010, como parte del Laboratorio de Computación Paralela (Par Lab) de la universidad de Berkeley, el profesor Krste Asanović y los estudiantes de posgrado Yunsup Lee y Andrew Waterman iniciaron la especificación de un set de instrucciones denominado RISC-V. El Par Lab fue un proyecto dirigido por David Patterson y subvencionado con capital privado proveniente de empresas como Intel y Microsoft, además de con fondos públicos del estado de California. Cabe destacar que todos los proyectos del Par Lab eran de código abierto y utilizaban la licencia *Berkeley Software Distribution* (BSD), incluido RISC-V. El 13 de mayo de 2011 se publicó el primer manual del set de instrucciones de RISC-V bajo el nombre *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA* [16]. Ese mismo año se realizó el primer *tapeout* de un chip RISC-V, producido en 28nm. Durante los siguientes años se realizaron publicaciones y talleres. A este respecto, una publicación interesante es la realizada en 2014 por Asanović y Patterson titulada *Instruction Sets Should Be Free: The Case For RISC-V* [17] donde se plantean argumentos que defienden una ISA libre y abierta. En concreto, RISC-V pasó a ser de dominio público en el momento en el que se publicaron los informes técnicos que definen la ISA, aunque el texto de dichos informes está bajo una licencia de *Creative Commons* para permitir su mejora por colaboradores externos. Además, no se ha registrado ninguna patente relativa a RISC-V, puesto que esta ISA per se no representa ninguna tecnología novedosa. En 2015, se creó la fundación RISC-V con el objetivo de construir una comunidad abierta e internacionalizar el proyecto. En 2018, la fundación RISC-V anunció una colaboración conjunta con la Fundación Linux en términos de cooperación técnica y estratégica, lo que incluye apoyo operativo, gestión de miembros, contabilidad, divulgación comunitaria etc. En marzo de 2020, la Asociación Internacional RISC-V se constituyó en Suiza debido a factores geopolíticos externos que defienden la continuidad del modelo de propiedad intelectual. El interés de la Asociación Internacional RISC-V es animar a organizaciones, particulares y entusiastas a hacer posible una nueva era de innovación en procesadores a través de la colaboración en estándares abiertos y *Open Source*.

En este sentido, las plataformas de control de versiones, como GitHub y GitLab, cumplen un papel crucial a la hora de compartir entre los usuarios sus diseños de núcleos y microcontroladores SoC basados en esta ISA libre. En particular, el primer proyecto de un procesador RISC-V se escribió en Chisel [18]. No obstante, en la actualidad hay multitud de proyectos en todo tipo de lenguajes de descripción de hardware. Existe una lista de implementaciones RISC-V de código abierto [19] donde se pueden consultar los proyectos *Open Source* más relevantes. Además, a cada uno de ellos se le asigna un identificador (ID). Algunos de estos ejemplos son:

- En Scala:
  - Rocket Chip [20] ID: 1
  - VexRiscv [21] ID: 33
- En Verilog:
  - PicoRV32 [22] ID: none [1]
  - Hummingbirdv2 E203 [23] ID: 26
  - Steel Core [24] ID: 24
  - SERV [25] ID: 18

[1] A pesar de no tener ID se considera un proyecto *Open Source* de gran relevancia.

- En VHDL:
  - NEORV32 [26] ID: 19
  - ORCA [27] ID: 7

CFUs/CXUs

Una de las finalidades de este proyecto de investigación es realizar una caracterización del rendimiento de los métodos de acoplamiento de coprocesadores en núcleos RISC-V. A este respecto, un procesador basado en RISC-V no solo se limita a ofrecer la posibilidad de dar soporte para conexiones mapeadas en memoria o comunicaciones mediante interfaces *stream*. Un concepto destacable que posibilitan los núcleos basados en esta ISA es el de gestionar instrucciones personalizadas (*custom*). Haciendo uso de este concepto surge la oportunidad de realizar extensiones de instrucciones personalizadas y asociarlas con aceleradores embebidos. Esta vinculación resulta muy ventajosa, ya que permite a la CPU resolver aplicaciones especificas decodificando instrucciones *custom*. De esta modo, se deriva la información directamente al acelerador embebido de manera similar a como lo haría una instrucción de suma con la ALU. Es decir, se consigue una integración completa del acelerador dentro del *pipeline* de la CPU, lo que significa una forma muy eficiente de acoplar micro y coprocesador. A priori, lo que resulta como una ventaja evidente puede conllevar algunos inconvenientes. Esto es debido a que las librerías de software que utilizan estas extensiones y los núcleos que las implementan son creados por diferentes organizaciones. Comúnmente estas utilizan herramientas diferentes y sus desarrollos, aunque aislados son operativos, al integrarlos para genera un nuevo sistema podrían no funcionar. Es por ello que se requiere de cierta estandarización que permita la reutilización robusta de las extensiones y librerías, además de proporcionar un modelo de programación uniforme para todas ellas.

En este sentido, en 2019, Google propuso el concepto *Custom Function Unit* (CFU) [28] y lo integró en la CPU VexRiscv. A partir de entonces esta extensión ha ganado popularidad y otras microarquitecturas RISC-V han añadido esta funcionalidad. Derivado de este trabajo, existe un borrador que especifica la integración hardware-software de estas unidades de funcionalidades personalizadas [29]. En este texto se propone la idea de *Composable eXtension Unit* (CXU), que plantea una generalización al concepto de CFU.

1.2.3. NEORV32

El NEORV32 es un microcontrolador SoC personalizable *Open Source* construido entorno a la CPU RISC-V homónima NEORV32. El repositorio principal del proyecto comenzó el 23 de junio de 2020 con un primer *commit* de su creador y máximo contribuidor Stephan Nolting.

Para la elaboración de este proyecto de investigación resulta de gran importancia conocer los métodos de conexión disponibles en el NEORV32. A este respecto, los principales modos de acoplamiento de coprocesadores con los que cuenta este microcontrolador son los siguientes:

- *Stream Link Interface* (SLINK)
- *Processor-External Bus Interface* (XBUS)
- *Custom Functions Subsystem* (CFS)
- *Custom Functions Unit* (CFU)

Se procede a hacer un breve descripción de cada uno de ellos. En primer lugar, *Stream Link* es una interfaz para realizar transmisiones *stream* compatible con un subconjunto de AXI4-Stream [30]. Esta interfaz proporciona canales RX y TX unidireccionales e independientes para enviar y recibir un flujo de datos. Cada canal dispone de una FIFO interna configurable para almacenar los datos provenientes del *stream*. En segundo lugar, *Processor-External Bus* es una interfaz de bus general para acoplar aceleradores mapeados en memoria compatible con Wishbone [31] y AXI4-Lite [32]. Esta interfaz cuenta con un módulo de caché opcional denominado *X-CACHE*. En tercer lugar, *Custom Functions Subsystem* es un *template* vacío que proporciona hasta 64 registros de lectura/escritura de 32 bits mapeados en memoria a los que la CPU puede acceder mediante operaciones normales de carga/almacenamiento. En cuarto lugar, *Custom Functions Unit* es una unidad funcional que se integra directamente en el *pipeline* de la CPU. Fue añadida al NEORV32 basándose en el borrador de la especificación CFU/CXU. Esto permitió implementar instrucciones RISC-V personalizadas. Los formatos de instrucción soportados por el NEORV32 son *R3-Type*, *R4-Type* y *R5-Type* [2]. Los dos primeros formatos son un estándar de RISC-V y el último es exclusivo de NEORV32. En concreto, el primer y el segundo formato permiten direccionar dos y tres registros de entrada de 32 bits, respectivamente. Además, la función se seleccionada a través de *funct7* y/o *funct3* en el primer caso y a través de *funct3* en el segundo caso. El tercer formato permite direccionar cuatro registros de entrada de 32 bits. Al no disponer del campo *funct3* y/o *funct7* solo se pueden realizar dos funciones personalizadas a través de este formato, *A format* y *B format*. La figura 1.1 muestra los tipos de instrucciones personalizadas de 32 bits soportadas por NEORV32 a través de la extensión ISA específica Zxcfu.

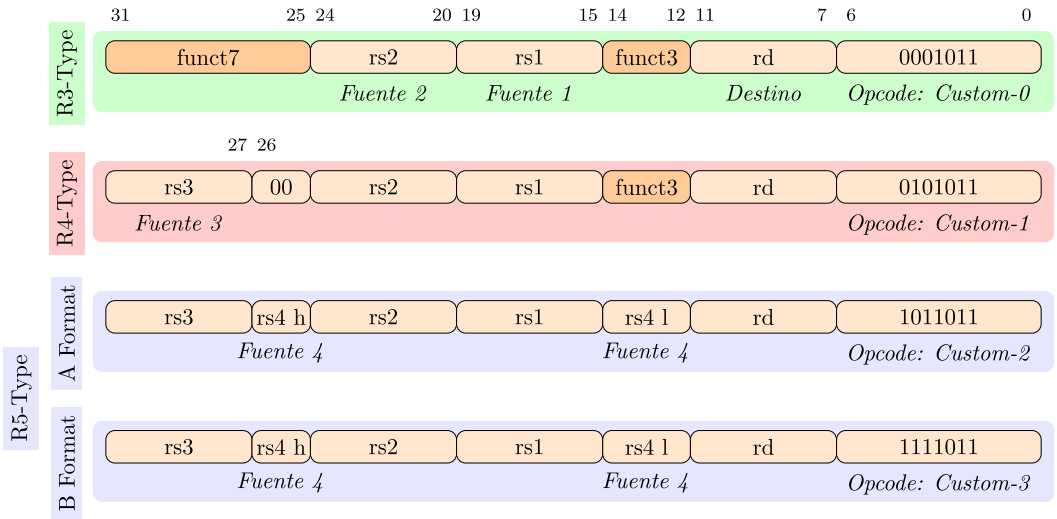


FIGURA 1.1: Tipos de instrucciones CFU *custom* para el caso del NEORV32.

A lo largo de estos cuatro años el proyecto ha estado en constante evolución. A este respecto, son destacables las contribuciones que realizó el miembro de GDED Unai Martinez Corral (*umarcor*) a lo largo del 2021 y 2022 entorno al tema de la integración continua del proyecto, además de llevar acabo su sugerencia de separar del repositorio principal el repositorio *neorv32-setups*. En el transcurso del desempeño

[2] Cabe destacar que recientemente Stephan a eliminado el soporte para la instrucción *R5-Type* #971

de este proyecto de investigación, se han visto aplicar numerosos cambios. Con respecto a los modos de conexión se destacan las siguientes actualizaciones: añadir a la interfaz SLINK la señal AXI-stream-compatible *tlast* #815, renombrar la interfaz de bus externa de *Processor-External Memory Interface (WISHBONE)* a *Processor-External Bus Interface (XBUS)* #846, reajustar la interfaz CFU #932. Estas actualizaciones ejemplifican la constante transformación del proyecto. En el apartado 2.1 se detallan los argumentos por los que se ha escogido este microcontrolador.

1.2.4. CRI

Desde finales del siglo pasado, investigadores de multitud de universidades, incluyendo la universidad del País Vasco, han indagado sobre la posibilidad de externalizar cálculos, de procesadores de propósito general a aceleradores hardware específicos. En 1999, el profesor Koldo Basterretxea, presentó un nuevo método recursivo para la aproximación de funciones basado en el uso de operadores reticulares modificados (MLO) [33] [34]. El objetivo era utilizar este método como un marco general para realizar futuras implementaciones, especialmente enfocadas a los sistemas difusos (*fuzzy*), así como a las redes neuronales artificiales (RNA). Este método, denominado Interpolación Recursiva Centrada (CRI), es un algoritmo que opera nuevas funciones afines de interpolación a cada nivel de recursión. Para hacerlo de manera sencilla y tratar de minimizar los parámetros del algoritmo, CRI genera funciones de interpolación centradas.

$$h_1 = \frac{1}{2}(y_1 + y_2 - \Delta) \tag{*}$$

Atendiendo a la ecuación (*) encontramos el parámetro Δ, denominado profundidad de interpolación. CRI utiliza valores de Δ optimizados en cada nivel de recursión, de manera que se minimiza el error máximo de la aproximación. En este sentido, para la aproximación mediante CRI de funciones paramétricas (con pendiente y saturación ajustable), se demuestra que el Δ optimizado es constante cuando variamos la pendiente. Para el caso de la función paramétrica sigmoide (**), el Δ optimizado toma los siguientes valores para las 3 primeras recursiones:

TABLA 1.1: Tres primeras recursiones del Δopt (función sigmoide).

Sigmoide	Δopt
q=1	0.3091
q=2	0.2811
q=3	0.2654

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{**}$$

Desde un punto de vista hardware, Basterretxea ha realizao descripciones de diseños empleando este algoritmo desde el inicio de su planteamiento teórico. A este respecto, en el repositorio del grupo de investigación, hay disponible un acelerador configurable y programable de funciones de activación basado en CRI. Dicho acelerador soporta 7 tipos de FAs: sigmoide, tangente, lineal, ReLu, Hardlimit, satlin y leakyReLu. Para el presente trabajo de investigación, se ha utilizado exclusivamente la lógica referente a la función sigmoide.

1.3. Anteproyecto

Cabe destacar que este proyecto se ha desempeñado en el marco de unas prácticas remuneradas tituladas *Desarrollo de Custom Functional Units de IA para RISC-V*. La duración de estas prácticas ha sido de 5 meses (594 horas) y por realizarlas se ha recibido una bolsa de ayuda total de 2000€. No obstante, previas a estas prácticas remuneradas, se realizaron otras prácticas obligatorias relativas al *practicum* del Máster. Estas también se llevaron a cabo en el grupo de investigación GDED bajo el título *Diseño e integración de CFU para RISC-V en FPGA*. Tuvieron una duración de 2 meses y 20 días (225 horas) y fueron calificadas con una matrícula de honor (10). Las tareas a desarrollar fueron las siguientes:

- Realizar un *set-up* de herramientas FOSS (*Free and Open-Source Software*) para la implementación y simulación del proyecto.
- Implementar el *soft-core* NEORV32 en FPGA (Arty A7 35t y 100t) y verificarlo.
- Experimentar con distintas opciones de integración de CFUs para NEORV32 y evaluar.
- Generar documentación.

El desarrollo de estas labores sirvió para tomar contacto con el ecosistema RISC-V. En concreto, con el micropocesador NEORV32. A este respecto, se comenzó a experimentar con los diferentes modos de acoplamiento de coprocesadores con los que cuenta este micro. Además, se realizaron los primeros ensayos de integración de este *soft-core* en FPGA. Con relación a esto, se comenzó a investigar sobre herramientas libres (FOSS) para realizar la implementación en placa de los diseños hardware. Esto fue motivado porque hasta entonces únicamente se utilizaban las herramienta privativas que ofrece Xilinx. Asimismo, se documentó los avances relativos al proyecto en forma de *issues* en el repositorio de GitLab asociado a este proyecto de investigación. Más tarde se trasladó parte de esta información a la pagina web vinculada a dicho repositorio.

En definitiva, las tareas desarrolladas en estas prácticas obligatorias se consideran el anteproyecto de esta investigación.

1.4. Objetivos y alcance del proyecto

El objetivo principal de este proyecto es integrar unidades coprocesadoras para IA en un núcleo RISC-V, en concreto en el *soft-core* NEORV32. En este contexto, se realizará la caracterización del rendimiento de cada uno de los modos de conexión con objeto de determinar la forma óptima de integración. Además, se deberá verificar su operatividad mediante la implementación en FPGA.

Los objetivos secundarios son:

- Realizar aportaciones al estado del arte referente al *soft-core* NEORV32 en forma de contribuciones a su repositorio oficial de GitHub.
- Realizar aportaciones al estado de la investigación sobre el uso de procesadores RISC-V realizada por GDED. Específicamente, mediante contribuciones de código, *pipelines* de integración continua y documentación en el repositorio del grupo relativo al NEORV32. Pretendiendo que este hecho facilite en un futuro la investigación de otros alumnos/as.

Respecto al alcance del proyecto, se definen con precisión los elementos incluidos dentro de este proyecto de investigación:

- La carectirazación mediante simulación, en términos de rendimiento y throughput, de los siguientes métodos de acoplamineto disponibles en el NEORV32:
  - *Stream Link Interface*
  - *Processor-External Bus Interface*
  - *Custom Functions Subsystem*
  - *Custom Functions Unit*
- La integración mediante *Custom Functions Unit* de un coprocesador hardware embebido para aplicaciones de IA.
  - El coprocesador integrado únicamente acelera los cálculos de la activación a través de la función sigmoide.
- La comparación mediante simulación de los ciclos de reloj necesarios para calcular la activación mediante la integración micro más acelerador versus micro haciendo uso exclusivamente de sus funcionalidades por defecto.
- La implementación en FPGA de todos los ensayos realizados en simulación para la verificación de su correcta operatividad.

Queda excluido del alcance del proyecto la integración del resto de funciones de activación que ofrece el acelerador configurable y programable basado en CRI. También queda excluida la última etapa definida en el informe de viabilidad de la propuesta de TFM que pretendía comparar en términos de rendimiento un modelo completo de RNA computado por software con el mismo modelo computado en un NEORV32 incorporando las instrucciones de aceleración del cálculo de las FAs.

1.5. Beneficios que aporta el trabajo

Se consideran beneficios aportados por el presente proyecto de investigación las siguientes contribuciones al repositorio principal del NEORV32:

- *Pull request #717*: *Fix bug in neorv32_slink_available() function*
- *Pull request #722*: *Fix-up the litex wrapper*
- *Pull request #727*: *Fix comment mistake*
- *Pull request #891*: *Fix UART receiver*

En relación a estas aportaciones, el autor de este TFM (bajo el alias *Unike267*) se encuentra como el contribuidor número 16 de 37 del repositorio principal del NEORV32, a fecha 11 de septiembre de 2024. Este hecho se refleja en la figura 1.2.

Además, cabe destacar que los resultados obtenidos mediante la caracterización de los diferentes métodos de acoplamiento de coprocesadores con los que cuenta el NEORV32 se han plasmado en un **artículo de congreso**. Este artículo se ha presentado en el marco del congreso DCIS 2024. La información relativa a este congreso, así como el propio artículo, se proporciona en el apéndice A. Puesto que dicho artículo ha sido aceptado, se publicará en el IEEEXplore. Se considera un beneficio aportado por este trabajo la contribución en dicha base de datos.

También, se considera un beneficio aportado por este trabajo la publicación del contenedor [35] desarrollado para realizar la síntesis, implementación y generación

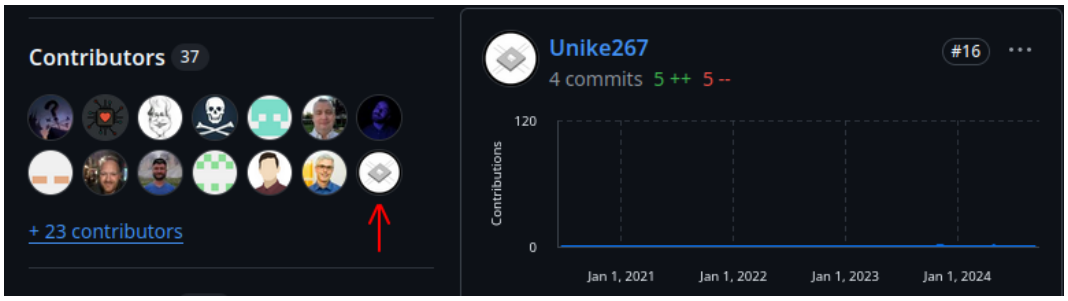


FIGURA 1.2: *Unike267* como contribuidor del repositorio principal del NEORV32.

de *bitstream* para las FPGAs Arty A7 35T y 100T mediante las herramientas *Open Source*: GHDL + yosys + GHDL yosys plugin + nextpnr-xilinx + prjxray.

En lo que respecta a los Objetivos de Desarrollo Sostenible (ODS), este trabajo de investigación pretende aportar beneficios en el marco del objetivo *Industria, innovación e infraestructura* (ODS 9) y del objetivo *Reducción de las desigualdades* (ODS 10). Respecto al ODS 9, se considera que la investigación entorno al enfoque distribuido para realizar IA en el borde supone fomentar la innovación en lo que respecta a este ámbito. Respecto al ODS 10, se considera que distribuir contenedores para facilitar el uso de herramientas *Open Source*, así como contribuir a proyectos de hardware libre que puedan derivar a la fabricación de chips *royalty-free*, fomenta la reducción de las desigualdades.

1.6. Análisis del estado del arte

Haciendo un repaso por las principales bases de datos de ámbito electrónico, se observan varios proyectos con una filosofía similar a la presentada en este trabajo de investigación. Un ejemplo muy interesante es el encontrado en una publicación titulada *Tiny Neuron Network System based on RISC-V Processor: A Decentralized Approach for IoT Applications* [36]. En dicho artículo, se presenta una investigación sobre un pequeño acelerador de redes neuronales en un SoC basado en RISC-V para acelerar una IA empleada en aplicaciones de IoT. Este coprocesador implementa una MAC (multiplicador y acumulador) de precisión variable en bits o una MAC estocástica para reducir el área de hardware y el consumo de energía. Es curioso el hecho de que se emplea la misma tecnología FPGA que en el presente proyecto, una Arty A7 100T. Los resultados presentados en este trabajo son destacables, consiguiendo realizar con precisión de 8 bits redes neuronales convolucionales (CNN) con una precisión del 98,55 %. En su caso, el método optado para acoplar los aceleradores ha sido una interfaz AXI.

Otro ejemplo muy interesante es el expuesto en una publicación titulada *CNN Specific ISA Extensions Based on RISC-V Processors* [37]. En ella, se presenta una extensión de instrucciones basada en la ISA RISC-V destinada a aumentar la eficiencia computacional de las CNN en dispositivos en el borde. En su caso emplean el core RISC-V *Open Source* Zero-riscy [38] [39]. Con objeto de evaluar el efecto de la extensión de instrucciones propuesta, se realiza una serie de cargas de trabajo en el núcleo de referencia y en el ampliado. Se obtinen un ratio de aceleración de 1,5× cuando se ejecuta una CNN, y alcanza 2,48×-2,82× cuando sólo se realizan los cálculos de convolución. Los resultados obtenidos en este artículo demuestran que las extensiones de ISA propuestas pueden mejorar eficazmente el rendimiento de las CNN.

Los casos mencionados, así como otros de gran interés, se encuentran recogidos en una publicación titulada *A Review of Edge Intelligence Applications Based on RISC-V* [40]. En ella se resume el uso de RISC-V en aplicaciones de IA en el borde desde un punto de vista hardware y software. Además, a lo largo de la revisión se analizan varios aceleradores, coprocesadores, compiladores y *toolchains* basados en RISC-V, así como las principales aplicaciones software de la IA en el borde.

A lo largo del transcurso de este proyecto se ha preferenciado, en la medida de lo posible, el uso de herramientas libres. En este sentido, las herramientas EDA FLOS y las propietarias/comerciales no son ecosistemas aislados. Al contrario, en los últimos años se han visto colaboraciones de proyectos *Open Source* con iniciativas privativas. Un ejemplo de ello es la integración de la herramienta RapidWright [41] a la *Suite* de diseño Vivado. En concreto, este proyeto *Open Source* desarrollado por *AMD Research and Advanced Development* tiene como objetivo permitir a los usuarios avanzados una mayor flexibilidad a la hora de personalizar sus soluciones mediante una metodología de diseño utilizando módulos pre-implementados. Además, se han realizado concursos [42] patrocinados por AMD con el objetivo de promover y demostrar que el Formato de Intercambio de FPGA (FPGAIF - *FPGA Interchange Format*) [43] es una representación intermediaria eficiente y robusta para trabajar en problemas de *backends* de FPGAs, incluso a escala industrial. Asimismo, este tipo de iniciativas también tratan de fomentar la innovación de algoritmos de enrutamiento de FPGAs que den prioridad al tiempo de ejecución, con objeto de posibilitar su aplicación en la emulación de ASICs. Cabe destacar que el FPGAIF es un estándar de formato de intercambio diseñado para proporcionar toda la información necesaria mediante la cual realizar el *place and route* en un contexto *Open Source*. En la misma línea, Siemens ha observado un crecimiento saludable entorno a la Metodología de Verificación VHDL de Código Abierto (OSVVM - *Open Source VHDL Verification Methodology*) [44] y la Metodología de Verificación Universal VHDL (UVVM - *Universal VHDL Verification Methodology*) [45] desde 2018, lo que en sus propias palabras «es alentador» [46]. Por lo tanto, a la vista de estos ejemplos, podemos afirmar que los comerciales tradicionales de herramientas EDA están empezando a facilitar el uso de herramientas FLOS e incluso a integrar parte o la totalidad de las mismas en sus propuestas comerciales. Este hecho refleja un futuro híbrido en lo referente al ecosistema de herramientas para FPGAs.

1.7. Análisis de alternativas

Alternativamente, la selección del microcontrolador se puede afrontar desde dos puntos de vista. El primero de ellos es seguir defendiendo la elección desde una perspectiva *Open Source*. En este sentido, se puede elegir otra de las opciones RISC-V propuestas en la sección 1.2.2. Sin embargo, también se puede optar por un microcontrolador basado en otra ISA libre, como alguno de las nombrados en la sección 1.2.1. El segundo punto de vista es claudicar con la iniciativa de emplear un proyecto *royalty-free*. Es decir, trasladar la búsqueda a un *soft-core* privativo sujeto al modelo de propiedad intelectual. A este respecto, se puede optar por MicroBlaze™ [47], ya que la entidad desarrolladora de su diseño es Xilinx (ahora AMD), se ve adecuado utilizar un *soft-core* realizado por la misma empresa encargada de fabricar las FPGAs utilizadas en este proyecto de investigación. Como curiosidad, cabe mencionar que debido a la enorme importancia que ha supuesto la entrada de RISC-V al contexto de ISAs actuales, AMD ofrece un *soft-core* RISC-V llamado MicroBlaze™ V [48]. Esta tesitura sirve para ejemplificar que un proyecto por el hecho de ser *Open Source*,

no impide que se desarrollen iniciativas comerciales que lo exploten. No obstante, es labor de las personas que defendemos la filosofía del software/hardware libre aportar opciones que supongan una competencia real frente a estas iniciativas propietarias. Además, este punto de vista privativo supondría un cambio de paradigma que incurriría en dejar de fomentar el ODS 10, defendido en el apartado 1.5.

Asimismo, existe la alternativa de emplear el *framework* LiteX [49] para llevar a cabo la implementación del NEORV32. Entre otras cosas, LiteX ofrece un conjunto de periféricos, como controladores DRAM, PCI, Ethernet, SATA etcétera, que junto a la descripción hardware del propio microcontrolador permiten multiplicar sus posibilidades de aplicación. Cabe destacar que la lógica digital de los complementos que ofrece LiteX están descritos en Migen [50]. Este hecho no impide integrar en él código en otros HDLs. De igual modo, es común generar diseños de LiteX en HDLs más tradicionales, como verilog.

Por último, se encuentra la alternativa de comenzar la aproximación de IA a en el borde mediante la externalización a hardware específico de otro cálculo relativo a las RNAs. Esta alternativa es bastante diversa y podría ir desde acelerar la activación mediante otra función disponible en el coprocesador configurable y programable basado en CRI hasta elegir otra operación para ser acelerada. Un ejemplo de esta última propuesta es la detallada en una publicación titulada *A Soft RISC-V Vector Processor for Edge-AI* [51]. En ella se presenta una unidad vectorial basada en un *array* sistólico que está estrechamente integrada en el pipeline de un núcleo RISC-V de 32 bits. Tras evaluar el rendimiento de este enfoque distribuido en una FPGA Xilinx Virtex 7, se concluye un aumento de velocidad de hasta 40,7 veces con respecto al núcleo escalar RISC-V en tareas de reconocimiento de imágenes, a costa de un aumento en el consumo energético y en los recursos hardaware de 1,2 y 1,8 veces respectivamente.

1.8. Descripción de la solución propuesta

La principal tesis que defiende este TFM es externalizar la computación referentes a las RNA, del procesador a hardware específico. Este hecho tiene como objetivo distribuir la gestión computacional en pos de implementar una primera aproximación de IA en el borde. En este sentido, se propone utilizar un coprocesador basado en el método CRI y acoplarlo a un procesador RISC-V. Mediante los argumentos descritos en la sección 2.1 se elige el microcontrolador NEORV32. Con objeto de generar un criterio de selección del modo de acoplamiento, se propone realizar una caracterización del rendimiento de los principales métodos de conexión con los que cuenta este microcontrolador. Esta caracterización se detalla en la sección 2.3. Por último, se acopla el coprocesador encargado de acelerar el cálculo de la FA sigmoide basado en CRI al NEORV32. Asimismo, se verifica el beneficio de emplear este tipo de enfoque distribuido comparándolo con realizar los mismos cálculos utilizando únicamente los recursos predeterminados del microcontrolador. Esta integración se detalla en la sección 2.4. Se procede a concretar los pasos seguidos a lo largo del desarrollo 2.

El primer paso para llevar a cabo este proyecto es realizar la selección del microcontrolador. Se requiere de su implementación en FPGA, por lo que se necesita la descripción en HDL de su arquitectura. Es interesante destacar que el término empleado para referirse a este formato es *soft-core* (núcleo blando). A diferencia de los micros *hard-core* (núcleo duro), es decir, los impresos en silicio, los *soft-cores* están pensados para ser implementados en dispositivos lógicos programables. Además, no solo se precisa de la descripción hardware del micro sino también de las

herramientas asociadas para compilar software interpretable por el mismo. En este sentido, la búsqueda se ha centrado en un proyecto *royalty-free*.

El siguiente paso es evaluar los diferentes modos de acoplamiento de coprocesadores con los que cuenta el micro seleccionado. Con objeto de caracterizar el rendimiento de cada uno de ellos y concretar el más adecuado para nuestro propósito. Para ello, se testean varios aceleradores acoplados al micro mediante todos los modos de los que este dispone. De esta manera, se obtiene un conjunto de bancos de prueba para ensayar en simulación y así proporcionar las lecturas de latencia y throughput de cada uno de los métodos de conexión. En concreto, se han acoplado 3 multiplicadores con diferentes características. Además, todos los ensayos se han implementados en FPGA para verificar su correcta operatividad.

El último paso es comenzar a abordar un enfoque distribuido para realizar IA en el borde. A este respecto, se requiere de aceleradores embebidos que realicen los cálculos asociados a las redes neuronales. Atendiendo a la caracterización realizada en el paso anterior, se establece un criterio mediante el cual decidir que modo de conexión emplear para acoplar estos diseños embebidos. Una vez acoplados, se corrobora el beneficio de utilizar coprocesadores frente a realizar los cálculos íntegramente mediante las funcionalidades predeterminadas del micro. Para ello, se realiza un banco de pruebas que compare en simulación y en implementación los ciclos de ejecución de ambos planteamientos. Dado el limitado tiempo de investigación tan solo se ha externalizado el cálculo de la activación a través de la función sigmoide. Para lo cual, se ha utilizado un acelerador embebido CRI diseñado por Koldo Basterretxea.

Capítulo 2

Desarrollo

2.1. Selección del microcontrolador

Resulta imprescindible que el microcontrolador seleccionado para este proyecto de investigación cumpla con los siguientes requisitos:

- Estar basado en una ISA RISC-V, ya que el contexto del proyecto está orientado a acoplar coprocesadores en núcleos RISC-V.
- Estar descrito en el lenguaje de descripción de hardware VHDL. En ocasiones, se han de modificar las fuentes del microcontrolador con el fin de acoplarle coprocesadores. En este sentido, resulta más oportuno integrar todo en un mismo lenguaje. A pesar de que los sintetizadores soportan varios lenguajes, ciertos elementos pueden suponer incompatibilidades. Por ejemplo, los tipos de datos varían dependiendo de cada lenguaje. En consecuencia, se tendría que dedicar tiempo extra a realizar las adaptaciones necesarias. Debido a que el coprocesador para aplicaciones de IA CRI está descrito en VHDL, se decide buscar un proyecto de microcontrolador descrito en este lenguaje.
- Contar con extensión de instrucciones para conectar coprocesadores mediante CFU, además de con soporte para comunicaciones mapeadas en memoria e interfaces *stream*. Debido a que se necesita una variedad de métodos de conexión para realizar la caracterización del rendimiento, el microcontrolador seleccionado debe contar con al menos los mencionados.

En este sentido, el NEORV32 [26] cumple con todos estos requerimientos. Además, en la plataforma de desarrollo colaborativo donde está alojado, cuenta con una comunidad muy activa. Es por ello que se encuentra bajo una revisión constante de fallos, tanto por parte del autor como de los usuarios. De esta manera, se asegura en gran medida la correcta operatividad del mismo. Además, el autor se dedica a realizar actualizaciones periódicas de sus funcionalidades. Por si fuera poco, tanto el autor como la comunidad tienen una gran disponibilidad para responder dudas sobre temas relacionados con el proyecto, lo que resulta de gran ayuda. Con respecto a la compilación de lenguajes de alto nivel, el proyecto ofrece *toolchains* precompiladas de RISC-V para GCC. Estas herramientas permiten hacer compilación cruzada de C/C++ a instrucciones de RISC-V en un entorno Linux [52]. Cabe destacar que también se facilita un contenedor para realizar esta tarea [53]. Además, cuenta con un soporte de librerías para compilar funciones software específicas de NEORV32. Asimismo, el repositorio ofrece una variedad de ejemplos de aplicación software de todos los recursos con los que cuenta el micro. Además de todo lo mencionado, este microcontrolador cuenta con una hoja de características [54] y una guía de usuario [55] realizadas por el autor y actualizadas a la par que el código del proyecto, las

cuales destacan por su calidad. Teniendo en cuenta todas estas consideraciones, el NEORV32 es el procesador seleccionado para este proyecto.

2.2. Flujo de trabajo

Atendiendo al paradigma híbrido de herramientas expuesto en la sección 1.6, en el presente trabajo se propone el uso tanto de herramientas FLOS como privativas. El esquema general de flujo de trabajo que se ha aplicado a lo largo de los ensayos realizados en este proyecto se ilustra en la figura 2.1.

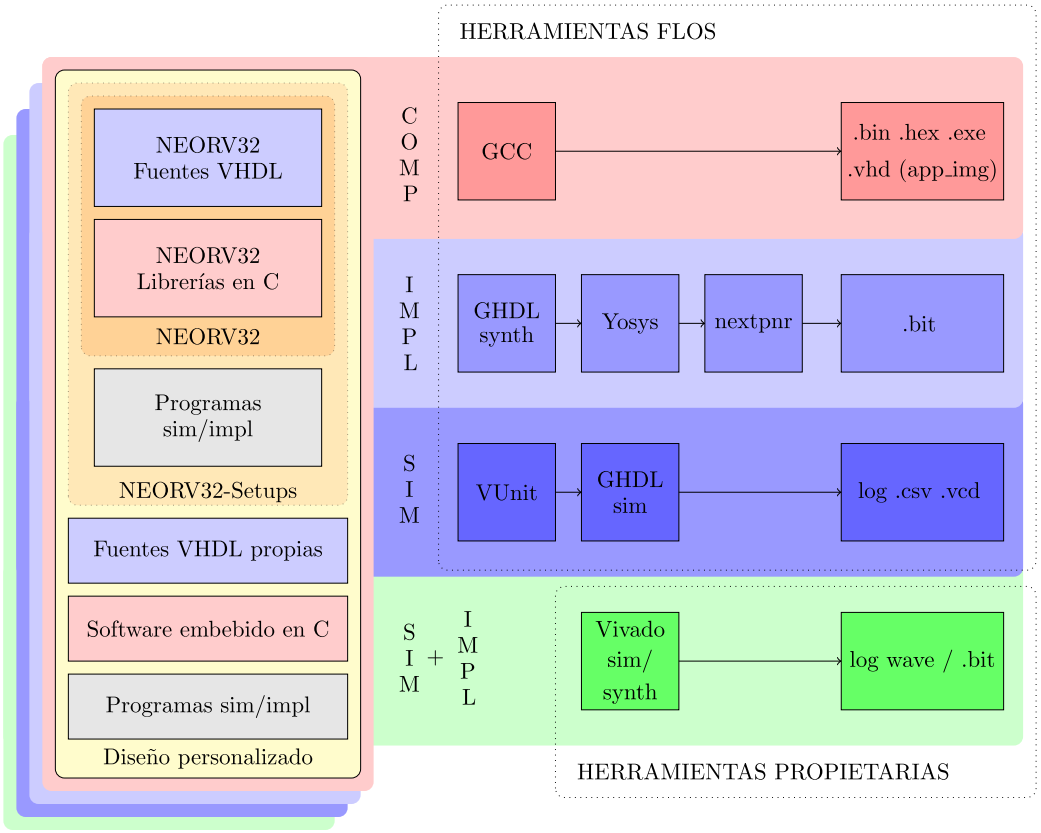


FIGURA 2.1: *Workflow* del *Setup* personalizado.

- **Compilación**: respecto a la compilación cruzada de software en C, de x86 (64 bits) a RISC-V (32 bits), se emplea la herramienta FLOS GCC.
- **Implementación**: respecto a las implementaciones en placa de los diseños, se efectúan mediante dos vías paralelas:
  - Haciendo uso del conjunto de herramientas FLOS: GHDL [56], yosys [57], GHDL yosys plugin [58], nextpnr-xilinx [59] y prjxray [60] para realizar la elaboración, la síntesis y el *place and route* y de la herramienta openFPGALoader [61] para cargar el *bitstream* en la placa.
  - Haciendo uso de la *Suite* de diseño privativa Vivado.

  Esto se debe a que la generación automática de *bitstreams* mediante la integración continua de repositorios públicos está limitada al uso de herramientas FLOS. Asimismo, la implementación más eficiente de los diseños, en términos de gestión de recursos de la Arty, está condicionada al uso de herramientas

privativas. Por ello, se eligen la generación mediante ambas vías. Además, para realizar una verificación doblemente redundante, se implementa tanto en la Arty A7 35T como en la 100T.

- **Simulación**: respecto a las simulaciones realizadas a lo largo de las secciones 2.3 y 2.4 se emplea principalmente el *framework* FLOS VUnit [62], con el cual se realizan todas ellas. No obstante, también se utiliza Vivado en ciertas ocasiones. En concreto, en los ensayos en los que se hace uso de la funcionalidad ILA. Esta funcionalidad se ha utilizado, por ejemplo, para testear la correcta operatividad de uno de los *wrappers* de Wishbone.

Las herramientas descritas en la explicación de este flujo de trabajo no solo se utilizan a nivel local, sino que también se emplean, todas o parte de ellas, en la **integración continua (CI)** de repositorios *online*, tanto en el GitLab del grupo de investigación como en el GitHub propio. Para ello, se utilizan varios **contenedores**. Para la generación de *bitstream* mediante herramientas FLOS, se utiliza el contenedor mencionado en la sección 1.5, el cual es generado a su vez en CI. Este contenedor se utiliza en la integración continua tanto del repositorio de GitLab como de GitHub. Para la generación de *bitstream* mediante Vivado, se utiliza un contenedor que solamente es accesible por los ordenadores del laboratorio del grupo de investigación, el cual está alojado en nuestro servidor Orion. Esto es debido a que al ser un programa privativo no se pueden distribuir públicamente contenedores con este software. A consecuencia de ello, la generación de *bitstream* mediante esta vía solo está disponible en la integración continua del repositorio del grupo (GitLab). Para realizar los ensayos en simulación, se utilizan principalmente dos contenedores de VUnit. El gcr.io/hdl-containers/sim/osvb:latest con GHDL compilado con llvm como *backend* y el docker.io/ghdl/vunit:mcode-master con GHDL compilado con mcode como *backend*. Esto se debe a que la funcionalidad *external names*, para capturar señales de jerarquías inferiores, solo es soportada en GHDL si este está compilado con el *backend* mcode. En definitiva, el uso de estos recursos mediante la metodología de integración continua posibilita automatizar todas las simulaciones, visualizando y gestionando sus resultados, así como la generación de todos los *bitstreams*, cada vez que se hace un *push* al repositorio. Cabe destacar que la compilación de software solo se realiza en local, aunque también se utiliza un contenedor [53], no está automatizada en integración continua.

2.2.1. Cargar software en el NEORV32

Antes de entrar en los detalles del acoplamiento de periféricos *custom*, se procede a realizar un repaso de cómo cargar un software en C al *softcore* NEORV32. Como se ha mencionado, el proyecto NEORV32 proporciona *toolchains* de RISC-V para GCC con las que realizar la compilación cruzada desde Linux a la arquitectura RISC-V. Estas herramientas están acompañadas de archivos *Makefiles* mediante los cuales se permiten añadir argumentos al comando *Make*, con objeto de, entre otras cosas, proporcionar el programa compilado en diferentes formatos de salida. A lo largo de esta sección, nos centraremos en tres de estos formatos:

- Ejecutable, *exe* (.bin)
- app_image (.vhd)
- Hexadecimal (.hex)

Cada una de estas salidas tiene la misma información, el programa compilado. No obstante, cada una de ellas puede utilizarse para cargar el software en la IMEM (memoria de instrucciones) en diferentes puntos del flujo de trabajo:

- El *exe* se puede cargar en el NEORV32 una vez que esté ejecutándose en la FPGA. Esta transferencia se realiza a través del *bootloader*.
- La app_image reemplaza el contenido por defecto de una de las fuentes RTL del diseño del NEORV32, de modo que su contenido se codifica cuando este se sintetiza.
- El archivo .hex se lee durante la síntesis, por lo que es equivalente a la solución de la app_image, pero no requiere modificar las fuentes RTL cada vez que se actualiza el software a cargar.

Estas opciones se resumen en la tabla 2.1.

TABLA 2.1: Tres formas de introducir software en la IMEM.

Formato	Comando	Descripción	Bootloader
.bin	make exe	Después de la implementación, cargar el exe mediante la CMD	Habilitado
.vhd	make image	Antes de la síntesis, sustituir la app_image por defecto	Deshabilitado
.hex	make hex	Durante la síntesis, leer del .hex	Deshabilitado

Bootloader

El NEORV32 viene por defecto con un *bootloader* que se encarga de establecer la comunicación serie vía UART y generar una CMD visible desde terminales como CuteCom [63], cu, o screen en GNU/Linux. En este sentido, hay tres formas posibles de proceder:

- Deshabilitar el *bootloader* y cargar/iniciar un programa desde la app_image o desde un archivo hexadecimal.
  - No se utiliza el *bootloader*.
- Habilitar el *bootloader* y cargar/iniciar un programa a través del *Autoboot*.
  - Después del *reset*, cuando el *bootloader* está habilitado, la primera secuencia que ocurre es el *Autoboot*. Esta secuencia intenta obtener una imagen de arranque válida desde la flash SPI externa. Si se encuentra una imagen válida que se pueda transferir correctamente a la IMEM (memoria de instrucciones), se inicia automáticamente la aplicación. No obstante, si han pasado 8 segundos y no se ha detectado ninguna flash SPI o no se encuentra ninguna imagen de arranque válida, se mostrará el código de error «ERR EXE», bloqueando la ejecución. Sin embargo, durante esos 8 segundos, se puede detener la secuencia del *Autoboot* pulsando cualquier tecla. De esta manera, se pone a disposición una CMD lista para recibir comandos.
- Habilitar el *bootloader* y cargar/iniciar un programa a través de comandos en la CMD. Los comandos soportados son los siguientes:
  - «h» - Muestra el texto de ayuda.
  - «r» - Reiniciar el *bootloader*.

- «u» - Cargar un programa en formato ejecutable (*neorv32_exe.bin*) a la IMEM.
- «s» - Almacenar un ejecutable en flash SPI.
- «l» - Cargar un ejecutable desde flash SPI.
- «x» - Arrancar un programa desde flash a través de XIP.
- «e» - Iniciar un programa almacenado en la IMEM.

Para elegir una de estas tres formas de proceder, se debe entender que el *bootloader* es útil/necesario cuando:

- La FPGA utilizada no permite inicializar la memoria en el *bitstream*. En consecuencia, no es posible cargar/arrancar programas a través de la app_image. Este es el caso de las FPGAs con SPRAM, como la Lattice ICE40 (UP3K, UP5K).
- Múltiples programas deben ser cargados/arrancados durante el desarrollo, sin resintetizar el diseño.

En la figura 2.2 se muestra como cargar/iniciar un programa ejecutable (.exe) al NEORV32 mediante la CMD proporcionada por el *bootloader*. Concretamente, se utiliza la terminal CuteCom [1], en ella se emplean sucesivamente los comandos «u» (*upload* - cargar) y «e» (*execute* - ejecutar).

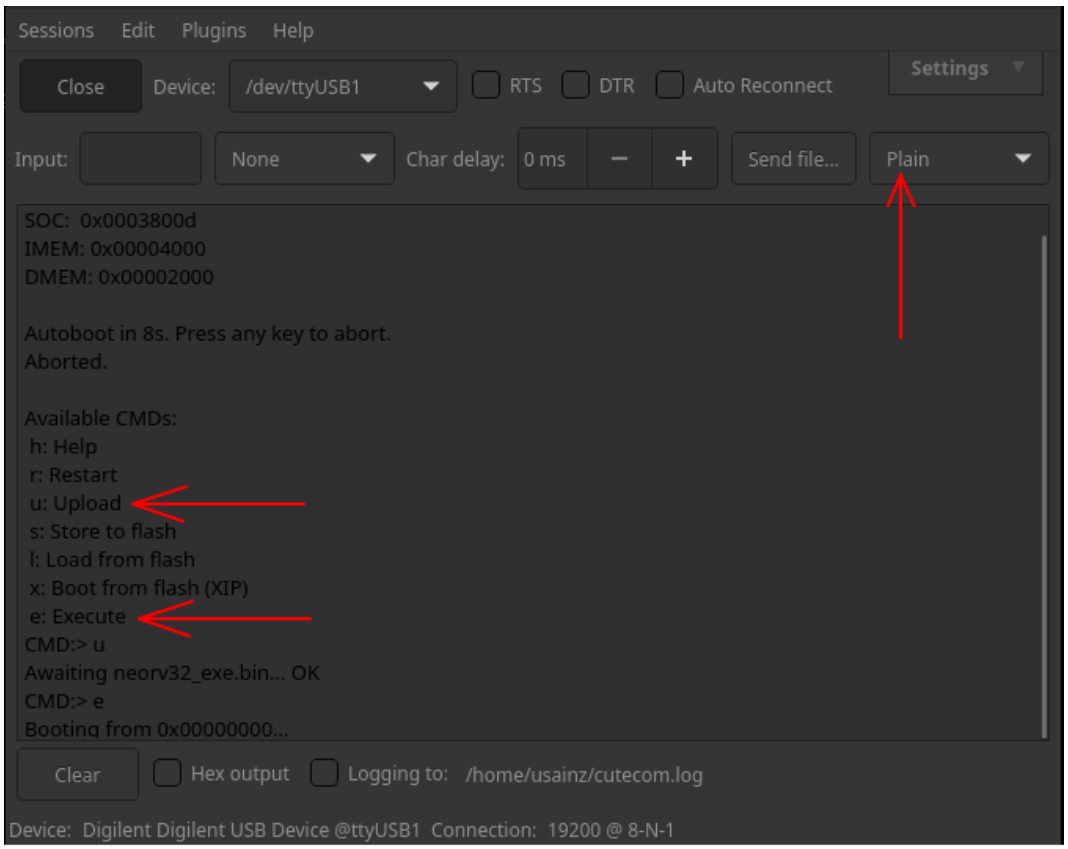


FIGURA 2.2: Cargar un *exe* a través del *bootloader* de NEORV32 (terminal CuteCom).

[1] En CuteCom, el archivo que se carga a la terminal debe ser de tipo *Plain* (como se muestra en la figura 2.2), de lo contrario se dará el error «ERR EXE».

Habilitar/Deshabilitar el *Bootloader*

Si el *bootloader* no es útil/necesario para nuestra aplicación tendremos que considerar lo siguiente. La IMEM se puede implementar de dos formas, como una RAM vacía o como una ROM inicializada a través del archivo que contiene el programa compilado, ya sea la *neorv32_application_image.vhd* o el hexadecimal. Con el genérico `IMEM_AS_IROM` se selecciona la implementación de la IMEM mediante una de estas dos opciones. Este genérico es

```
IMEM_AS_IROM => imem_as_rom_c
```

y se define como

```
imem_as_rom_c : boolean := not INT_BOOTLOADER_EN;
```

Por lo tanto, para cargar un programa desde la *neorv32_application_image.vhd* (o desde el hexadecimal), la IMEM debe implementarse como una ROM inicializada mediante ese archivo, por lo que el *bootloader* **debe estar deshabilitado**. Se discutió con Stephan (#824) acerca de por qué la IMEM se inicializa como una RAM vacía cuando el *bootloader* está activado. Y según el diseñador del NEORV32, «si la IMEM se implementara como una RAM preinicializada, entonces la imagen podría corromperse durante el tiempo de ejecución (imagina algún puntero deshonesto escribiendo en la IMEM), lo que requeriría volver a cargar el programa original. Por lo tanto, la carga del *bootloader* se requeriría de todos modos.»

El proceso para deshabilitar el *bootloader* es sencillo, en el TOP del diseño del NEORV32, se debe cambiar la constante `INT_BOOTLOADER_EN` de *true* a *false*, como se muestra en el extracto de código 2.1.

```
neorv32_top_inst : neorv32_top
generic map(
-----------------------------------
INT_BOOTLOADER_EN         => false,
-----------------------------------
)
```

CÓDIGO 2.1: Constante para deshabilitar el *bootloader*.

Cargar un programa compilado desde un archivo hexadecimal

Como se ha mencionado, en vez de cargar un programa compilado desde el archivo *neorv32_application_image.vhd*, es posible cargar el programa compilado desde un archivo hexadecimal (.hex). Para ello, se necesitan hacer unas pequeñas modificaciones en el código HDL del NEORV32. En particular, se debe añadir una nueva función en el paquete *neorv32_package.vhd*. Esta función se encargará de leer el archivo hexadecimal usando la librería *std.textio.all*. [2] La función en cuestión es la descrita en el extracto de código 2.2.

[2] Esta librería está soportada desde la versión VHDL 2008.

```
-- Initialize mem32_t from hex
-- MEMORY_SIZE is IMEM_SIZE/4, see neorv32_imem.default.vhd

impure function mem32_init_hex(name : STRING; MEMORY_SIZE : natural) return mem32_t is
    file rom_file : text open read_mode is name;
    variable rom_line : line;
    variable temp_word : std_ulogic_vector(31 downto 0);
    variable temp_rom : mem32_t(0 to MEMORY_SIZE-1) := (others => (others => '0'));
begin
    for i in 0 to MEMORY_SIZE - 1 loop
        exit when endfile(rom_file);
        readline(rom_file, rom_line);
        hread(rom_line, temp_word);
        temp_rom(i) := temp_word;
    end loop;

    return temp_rom;
end function;
```

CÓDIGO 2.2: Función a añadir al *neorv32_package.vhd* para leer un software compilado en formato hexadecimal.

Además, se debe modificar el archivo *neorv32_imem.default.vhd* [3] para cargar el contenido del archivo hexadecimal (*neorv32_raw_exe.hex*) a la memoria de instrucciones, usando la función definida en el extracto de código 2.2. Para ello se debe añadir el extracto de código 2.3.

```
constant ROM_INIT_FILE : string := "neorv32_raw_exe.hex";
-- ROM - initialized with hex code --
constant mem_rom_c : mem32_t(0 to IMEM_SIZE/4-1) := mem32_init_hex(ROM_INIT_FILE,
IMEM_SIZE/4);
```

CÓDIGO 2.3: Modificación del archivo *neorv32_imem.default.vhd* para cargar la IMEM mediante la función descrita en el extracto de código 2.2.

Este método propone leer desde VHDL un formato hexadecimal, el cual es una salida nativa del compilador, en lugar de autogenerar código HDL con el programa compilado como pasa cuando utilizamos la opción de la app_image. Ambas opciones cargan la IMEM cuando se sintetiza el diseño. Sin embargo, con la opción de lectura del archivo .hex conseguimos dos cosas: no autogenerar código HDL tras la compilación y no modificar el código HDL existente cada vez que se actualiza el software a cargar.

Por último, cabe destacar que a lo largo del desarrollo de este proyecto se ha cargado software compilado al NEORV32 mediante los tres formatos expuestos. No obstante, mayoritariamente se ha utilizado el formato .vhd generando una *neorv32_application_image.vhd* para cada software empleado.

[3] En el archivo *neorv32_imem.default.vhd* se debe comentar el código relacionado con cargar la ROM desde la app_image.

2.3. Caracterización del rendimiento de los métodos de conexión

Para caracterizar el rendimiento de los diferentes modos de conexión con los que cuenta el NEORV32, se propone acoplarle 3 tipos de multiplicadores con diferentes características. Debido a que el objetivo de esta sección es caracterizar los métodos de acoplamiento, no son relevantes las funciones del coprocesador en sí. En este sentido, se ha pretendido emular en ellos cualidades de rendimiento y tiempo de respuesta propias de coprocesadores de IA. Para ello, se han establecido ciertos patrones que comparten los aceleradores que se pretenden integrar en aplicaciones futuras, ver 4.2. De esta manera, se ha determinado que las características de almacenamiento a la entrada/salida, así como de segmentación de señales internas, son cualidades a imitar. Además, se decide realizar dos tipos de ensayos, con objeto de evaluar los métodos de conexión en términos de latencia y *throughput*. Cabe destacar que para el caso de los métodos *Stream Link Interface* (SLINK) y *Processor-External Bus Interface* (XBUS), se realiza una caracterización adicional acoplando los 3 multiplicadores a *Verification Components*[4] de AXI-Stream y Wishbone respectivamente. De esta manera, se realiza por cada multiplicador acoplado mediante cada método de conexión un ensayo de latencia y si es posible de *throughput*. Esto es debido a que para realizar una caracterización de *throughput*, el acelerador o el modo de conexión debe disponer de un *buffer* de datos. Teniendo en cuenta estas consideraciones, se han llevado a cabo un total de 29 ensayos de simulación con éxito, todos ellos realizados mediante el *framework* VUnit. Dichos ensayos se resumen en la tabla 2.2. Asimismo, se han implementado en FPGA todos los diseños realizados referentes al conjunto NEORV32 más multiplicador, con objeto de verificar en placa su correcta operatividad.

TABLA 2.2: Ensayos de latencia y *throughput* realizados: VC, el multiplicador individual acoplado a *Verification Components*; C, el SoC completo incluyendo el NEORV32, el multiplicador y la ejecución de software.

Modo / Tipo	SLINK		XBUS		CFU	CFS
	VC	**C**	**VC**	**C**	**C**	**C**
Mult-B	Ambos[5]	Ambos	Ambos	Ambos	Latencia	Ambos
Mult-BP	Ambos	Ambos	Ambos	Ambos	Latencia	Ambos
Mult-UBP	Latencia	Ambos	Latencia	Latencia	Latencia	Latencia

2.3.1. Descripción y conexión de los multiplicadores

La figura 2.3 ilustra las posibles combinaciones de acoplamiento de los 3 diferentes multiplicadores mediante los modos de conexión SLINK, XBUS, CFU y CFS. Las características que definen a cada tipo de multiplicador son las siguientes:

[4] Herramienta de verificación funcional que ofrece el *framework* VUnit.

[5] «Ambos» se refiere a que se han realizado los ensayos tanto de latencia como de *throughput*.

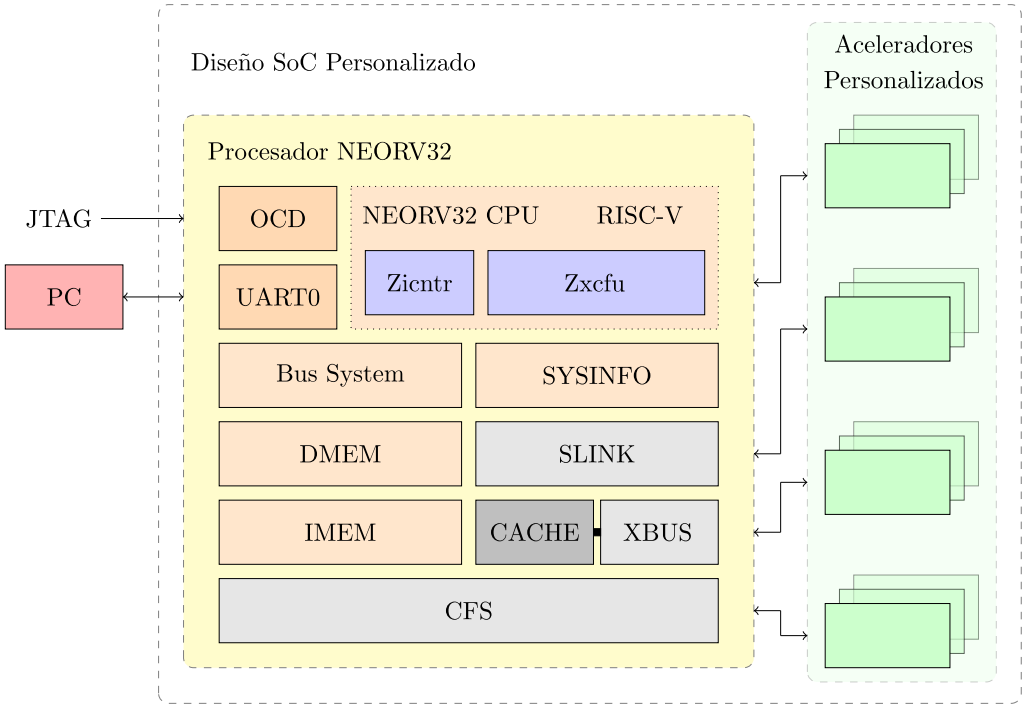


FIGURA 2.3: Esquema de las posibles combinaciones del SoC personalizado.

- **Mult-B** (**Mult**iplicador-***B****uffered***), en el código HDL del proyecto referido como *Mult_wfifos*, es un multiplicador D.1 al que se le han añadido dos FIFOs D.2, una a la entrada y otra a la salida. Además, las señales internas se gestionan mediante una máquina de estados. Es configurable en número de bits de entrada/salida y en profundidad de la FIFO, aunque normalmente se ha configurado en 32 y 4 bits respectivamente. Puede recibir hasta tres relojes diferentes, uno en la FIFO de entrada, otro en el multiplicador y otro en la FIFO de salida, aunque comúnmente se han ajustado los tres a la misma frecuencia. Su topología interna se puede observar en la figura 2.4. Además, su descripción hardware se encuentra en el apéndice D, código D.3.
- **Mult-BP** (**Mult**iplicador-***B****uffered*** y ***P****ipelined*), en el código HDL del proyecto referido como *Multp_wfifos*, es un multiplicador al que se le han añadido dos FIFOs, una a la entrada y otra a la salida, pero al contrario que Mult-B, las señales internas se gestionan segmentadas. Es configurable en número de bits de entrada/salida y en profundidad de la FIFO, aunque normalmente se ha configurado en 32 y 4 bits respectivamente. Puede recibir hasta tres relojes diferentes, uno en la FIFO de entrada, otro en el multiplicador y otro en la FIFO de salida, aunque comúnmente se han ajustado a la misma frecuencia. Cabe destacar que su descripción hardware se encuentra en el apéndice D, código D.7.
- **Mult-UBP** (**Mult**iplicador-***UnB****uffered*** y ***P****ipelined*), en el código HDL del proyecto referido como *Multp*, es un multiplicador sin *buffers* de entrada/salida, además las señales internas se gestionan segmentadas. Es configurable en número de bits de entrada/salida, aunque normalmente se ha configurado en 32 bits. Cabe destacar que su descripción hardware se encuentra en el apéndice D, código D.6.

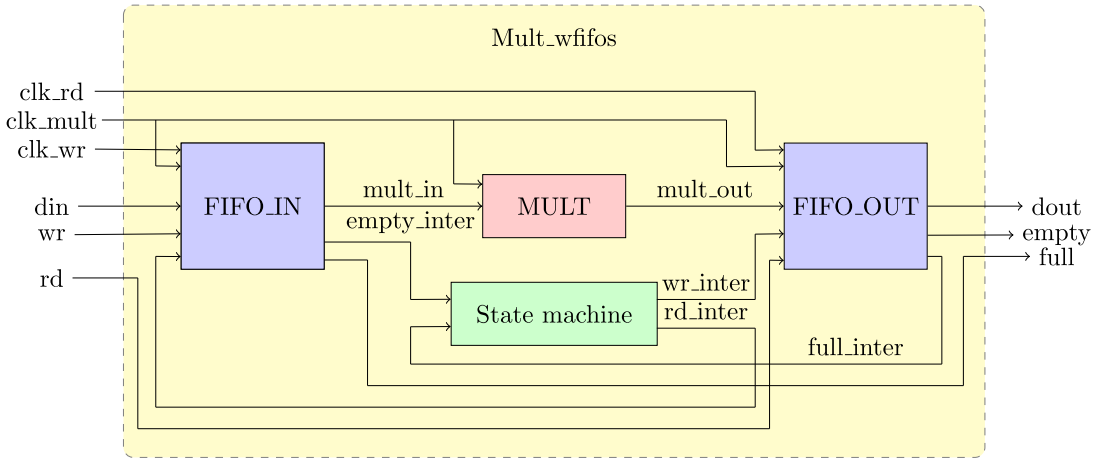


FIGURA 2.4: Plano del multiplicador tipo MULT-B.

Para realizar el acoplamiento de los multiplicadores con las interfaces SLINK (AXI-Stream) y XBUS (Wishbone), se han realizado *wrappers* definidos en el apéndice D. En concreto, para el caso del Mult-B la descripción HDL del *wrapper* se encuentra en el código D.4 y D.5, para AXI y Wishbone respectivamente. Para el caso del Mult-BP, en el código D.8 y D.9, para AXI y Wishbone respectivamente. Para el caso del Mult-UBP las señales AXI están autocontenidas en la descripción de su diseño D.6, así como el *wrapper* para Wishbone está descrito en D.10. En lo que respecta al software, se han empleado funciones de C propias del NEORV32 para interactuar con cada uno de los modos de conexión. Mediante las librerías aportadas por Stephan, tras compilarse, estas funciones se transforman a instrucciones interpretables por RISC-V. Dichas funciones son las siguientes:

- SLINK:
  - Para enviar datos del NEORV32 al coprocesador: `neorv32_slink_put(dato)`
  - Para recibir datos del coprocesador al NEORV32: `neorv32_slink_get()`
- XBUS:
  - Para enviar datos del NEORV32 al coprocesador: `neorv32_cpu_store_unsigned_word(dirección,dato)`
  - Para recibir datos del coprocesador al NEORV32: `neorv32_cpu_load_unsigned_word(dirección)`
- CFU:
  - Para enviar datos del NEORV32 al coprocesador y recibir, en función de dichos datos, la salida del coprocesador en la misma ejecución de la instrucción: `neorv32_cfu_r3_instr(funct7,funct3, rs1, rs2)`
- CFS:
  - Para enviar datos del NEORV32 al registro mapeado en memoria con el coprocesador: `NEORV32_CFS->REG[0] = dato_de_salida;` [6]
  - Para recibir datos del registro mapeado en memoria con el coprocesador al NEORV32: `dato_de_entrada = NEORV32_CFS->REG[0];`

[6] Los registros disponibles asociados a CFS son del REG[0] al REG[63].

En este sentido, en el apéndice D se muestran los programas en C realizados para llevar a cabo los diferentes métodos de conexión. Se observa que en todos ellos se emplea un `#ifdef`. El objetivo de esto es separar la compilación de software orientado a ser simulado, con muy pocos `printf` para agilizar la simulación y la del software orientado a ser implementado en FPGA. En concreto, en el código D.11 se muestra el main.c para SLINK, en D.12 para XBUS, en D.13 para CFU y en D.14 y D.15 para CFS. Respecto a este último, se realiza esta división porque el primer programa está orientado a multiplicadores *buffered* y el segundo al *unbuffered*. Esto es debido a que para el primer caso, se destina un registro mapeado en memoria para gestionar las señales de lectura/escritura, además de otro registro para las entradas/salidas. No obstante, para el caso del multiplicador *unbuffered* solo se necesita un registro para gestionar las entradas/salidas, por lo que el software varía. Cabe destacar que se genera una app_image por cada programa compilado, teniendo en cuenta que los `#define` se van comentando para obtener un programa destinado a simulación, de latencia o throughput o a implementación.

Con respecto a la CFU, se realiza una instrucción *custom* para operar cada multiplicador. En concreto, se utilizan tres instrucciones *R3-Type* de la extensión Zxcfu. Se emplea el registro *rs1* para la entrada de la multiplicación, los 16 primeros bits para el primer factor y los 16 últimos para el segundo. El segundo registro fuente *rs2* no se utiliza y el resultado de la multiplicación se guarda en el registro de destino *rd*. Para especificar la instrucción de cada multiplicador se emplea el campo *funct3*, también se puede emplear el campo *funct7* pero en este caso no se tiene en cuenta. En concreto, `funct3=000` se asocia a Mult-B, `funct3=001` a Mult-BP y `funct3=010` a Mult-UBP. En el código D.16, se observa cómo están integrados los multiplicadores en el *core* del NEORV32. Además, se aprecia cómo se evalúa el campo *funct3* para dirigir la información de entrada/salida a cada uno de los multiplicadores, así como se observan los recursos lógicos empleados para iniciar las operaciones e indicar cuándo estas han terminado.

Con respecto al subsistema CFS, como se ha mencionado, se distinguen dos casos. Por un lado, para el caso de los multiplicadores *buffered*, se mapean 2 de los 64 registros asociados, el registro REG[0] para los datos de entrada/salida y el registro REG[1] para las señales de control. Además, puesto que las señales de entrada/salida del subsistema CFS son ajustables, se emplean 34 bits para su salida. En concreto, se utilizan los 2 bits MSB para realizar el control del multiplicador, es decir, gestionar sus señales de lectura/escritura. Los otros 32 bits se emplean para representar el dato de entrada al multiplicador. En lo que respecta a la entrada al subsistema CFS, se emplean 32 bits, lo que equivale al tamaño de la salida del multiplicador. En el código D.17, se muestra el subsistema CFS para el caso de los multiplicadores *buffered*. Por otro lado, para el multiplicador *unbuffered*, tan solo se mapea el registro REG[0] con objeto de gestionar los datos de entrada/salida. En este caso, las señales del subsistema CFS se ajustan a 32 bits tanto para la entrada como para la salida, ya que no es necesario administrar señales de control. En el código D.18, se muestra el subsistema CFS para el caso del multiplicador *unbuffered*.

Cabe destacar que para los casos de SLINK y XBUS, simplemente se enlazan los *wrappers* con el NEORV32 en el TOP del diseño. Es decir, no es necesario modificar archivos internos del NEORV32, como en el caso de CFU/CFS. En el caso de XBUS, es de interés señalar que se asocia la dirección de memoria *0x90000000* con dicha interfaz.

2.3.2. Metodología de medición mediante el registro CSR(mcycle)

La metodología de medición seguida para caracterizar procesos referentes al NEORV32 o al conjunto NEORV32 más coprocesador, se ha generalizado para todos los ensayos de simulación. A continuación, se procede a explicar dicha metodología.

Cada operación realizada por el NEORV32 está asociada a una o varias instrucciones de RISC-V. Con objeto de caracterizar una operación, se propone medir el tiempo de ejecución dedicado a computar las instrucciones que conllevan aplicar dicha operación. Para ello, se decide emplear el registro CSR *mcycle*. Este registro se incrementa con cada ciclo de reloj activo de la CPU. El acceso de este registro es tanto de lectura como de escritura. Este hecho permite inicializar el CSR *mcycle* a cero justo antes de comenzar el proceso a medir y leerlo en el momento que este finalice. De esta manera, se caracteriza de forma precisa los ciclos de reloj que emplea la CPU para llevar a cabo un proceso concreto. A pesar de que este método podría emplearse a nivel ensamblador, existen funciones en C que permiten la lectura y escritura de este registro desde un programa de alto nivel. Estas funciones son `neorv32_cpu_csr_write(CSR_MCYCLE, 0)` para inicializar a cero el registro y `neorv32_cpu_csr_read(CSR_MCYCLE)` para leer su contenido. De esta manera, se da la posibilidad de generar un programa en C que permita caracterizar el tiempo de ejecución de una función o funciones en C, simplemente aplicando el esquema mostrado en el extracto de código 2.4.

```
neorv32_cpu_csr_write(CSR_MCYCLE, 0)
//Ubicar aquí la función (o funciones) a caracterizar
neorv32_cpu_csr_read(CSR_MCYCLE)
```

CÓDIGO 2.4: Código para caracterizar el tiempo de ejecución de una función (o funciones) en C.

De este modo, si tenemos un programa que utilice esta metodología, compilado y cargado dentro de la IMEM de un NEORV32 corriendo en una simulación, se puede extraer el valor de la medición (contenido en el registro CSR *mcycle*) y visualizarlo como resultado de la misma. Para ello, se propone añadir el extracto de código VHDL 2.5 en un test bench de VUnit.

```
for x in 0 to test_items-1 loop
    wait until rising_edge(clk) and csr_we = '0' and csr_valid = '1' and csr_addr =
    ↪  x"B00" and csr_rdata_o /= x"00000000"; -- CSR MCYCLE ADDR IS 0xB00
    info(logger, "Data " & to_string(x+1) & "/" & to_string(test_items) & " latency
    ↪  is " & to_string(to_integer(unsigned(csr_rdata_o))-1) & " cycles");
end loop;
```

CÓDIGO 2.5: Código VHDL para extraer en simulación el contenido del CSR(*mcycle*).

Atendiendo al código 2.5, se observa que al extraer el valor del registro CSR *mcycle* se le resta un ciclo. Esto es debido a que la ejecución de la instrucción de lectura del CSR (*csrr* en ensamblador), añade un ciclo extra a la medida. Esta situación se discutió y verificó con Stephan en una *issue* titulada *Latency measurement through CSR(MCYCLE) adds one extra cycle #897*. Además, el hecho de utilizar la función de VUnit `info()` permite exportar los resultados en formato CSV para su posterior procesamiento. Cabe destacar que para emplear esta metodología debe estar activada la extensión Zicntr:

```
CPU_EXTENSION_RISCV_Zicntr => true
```

Para ejemplificar esta explicación se dispone de uno de los *test bench* de VUnit utilizados que aplican esta metodología, en el apéndice D código D.19. En él se puede observar como se utiliza el recurso *external names* para acceder a las señales de jerarquía inferior, entre ellas a las referentes a los registros CSR, con objeto de realizar su evaluación. En concreto, este código está diseñado para caracterizar en términos de latencia el rendimiento del método de conexión CFU. No obstante, se ha empleado este esquema para caracterizar todos los métodos de acoplamiento.

Descripción de los ensayos realizados

El objetivo de esta sección 2.3 es caracterizar el rendimiento de los cuatro métodos principales de conexión que ofrece el NEORV32. Todos estos métodos están asociados a una o varias funciones en C para realizar una transmisión, como se ha descrito en la subsección 2.3.1. Haciendo uso de esta metodología descrita en 2.3.2, se proponen dos tipos de ensayos. El primero de ellos es un ensayo de latencia. En él se realizan 4 operaciones consecutivas de envío/recepción de datos entre el NEORV32 y el multiplicador mediante cada método de conexión y se mide el tiempo de ejecución de cada una de estas operaciones en ciclos de reloj del sistema. El segundo ensayo propuesto es de *throughput*. En él se realizan 4 operaciones de envío de datos consecutivas del NEORV32 al multiplicador, después se realizan 4 operaciones de recepción consecutivas desde el multiplicador al NEORV32 y se mide cuántos datos por ciclo de reloj se reciben. Con objeto de almacenar los primeros 4 datos enviados, el multiplicador o el método de conexión debe contar con un *buffer* de datos. Es por ello que para el multiplicador Mult-UBP solo se puede realizar el ensayo de *throughput* para el método SLINK, debido a que esta interfaz cuenta con FIFOs asociadas. Para el caso del método CFU, solo se puede realizar la medición de latencia. Debido a las características internas de la instrucción personalizada, la operación de envío y recepción se realiza en un único paso. En la figura 2.5 se aclara gráficamente estos dos tipos de ensayos.

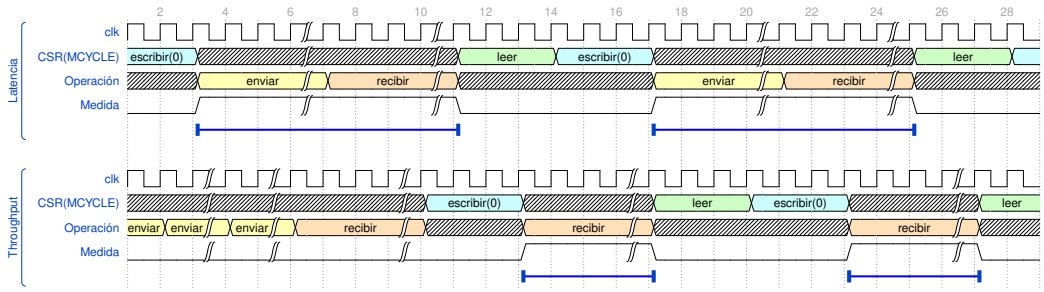


FIGURA 2.5: Aclaración gráfica de los dos tipos de ensayos: latencia y *throughput*.

A este respecto, la parte de código destinada a simulación descrita en D.11, D.12, D.13 y D.14 para SLINK, XBUS, CFU Y CFS respectivamente, así como los respectivos *test bench* de VUnit, se encargan de llevar a cabo estos ensayos.

Por último, cabe destacar que la evaluación mediante este método solo es posible para los ensayos del SoC completo, NEORV32 más multiplicador. Para los ensayos mediante *Verification Componets* se han empleado simulaciones de VUnit en conjunto con programas de Python encargados de gestionar los CSV de salida producidos por la función `info()` y así hacer los cálculos necesarios para obtener la

latencia/*throughput*. En concreto, en el apéndice D se ejemplifica este proceso para la latencia de Mult-B acoplado mediante AXI-Stream. Para ello se muestra el archivo que importa los VCs y extrae la información D.20, el *test bench* de VUnit D.21 y el *script* de Python D.22 encargado de calcular a latencia.

2.3.3. Resultados de los ensayos de simulación

Tal como se ha comentado en el apartado 2.2, las simulaciones se han automatizado mediante la integración continua del repositorio. En este sentido, los 29 ensayos de simulación recogidos en la tabla 2.2 se han realizado con éxito y sus resultados se muestran en el apéndice C, desde la figura C.1 hasta la C.23. En el apartado 2.3.5, se procede a hacer un análisis exhaustivo de los resultados obtenidos en estos ensayos. Asimismo, cabe destacar que en el caso de los ensayos de latencia C.10, C.11, C.12, C.19, C.20 y C.21, se observa que los resultados de los cuatro datos de entrada para XBUS y CFS no son constantes. El primer cálculo varía en 2 ciclos para XBUS y en 8 ciclos para CFS con respecto al resto de casos. Esto se debe a que el compilador no genera la misma secuencia de instrucciones en todos los casos. Como se observa en D.12 y D.14, cada transmisión de dato se mide de forma aislada. En este contexto, cuando el compilador construye el programa emplea una única vez la instrucción para apuntar al registro inmediato donde se cargará la entrada. Para el resto de los cálculos posteriores, debido al nivel de optimización, el compilador omite esta instrucción. En el caso de XBUS, el ahorro de esta instrucción se traduce en una ejecución 2 ciclos más rápida. En el caso de CFS, suponen más ciclos, ya que se emplean/ahorran más direccionamientos a registros inmediatos. Este hecho se discutió con Stephan en la *issue* #888. Es relevante destacar que para realizar la caracterización de los métodos de conexión se han empleado las medidas más restrictivas. Con objeto de facilitar la lectura de estos resultados, se agrupan en la tabla 2.3, así como se visualizan mediante gráficos de barras en las figuras 2.6 y 2.7 para los ensayos de latencia y *throughput*, respectivamente.

TABLA 2.3: Resultados de los ensayos de latencia y *throughput*: VC, el multiplicador individual acoplado a *Verification Components*; C, el SoC completo incluyendo el NEORV32, el multiplicador y la ejecución de software.

Mult		**SLINK**		**XBUS**		**CFU**	**CFS**
		VC	**C**	**VC**	**C**	**C**	**C**
Latencia	**Mult-B**	6	45	5	16	13	37
	Mult-BP	4	45	3	16	11	37
	Mult-UBP	1	45	2	16	8	18
Throughput	**Mult-B**	1/4	1/20	1/2	1/5	X	1/15
	Mult-BP	1	1/20	1/2	1/5	X	1/15
	Mult-UBP	X	1/20	X	X	X	X

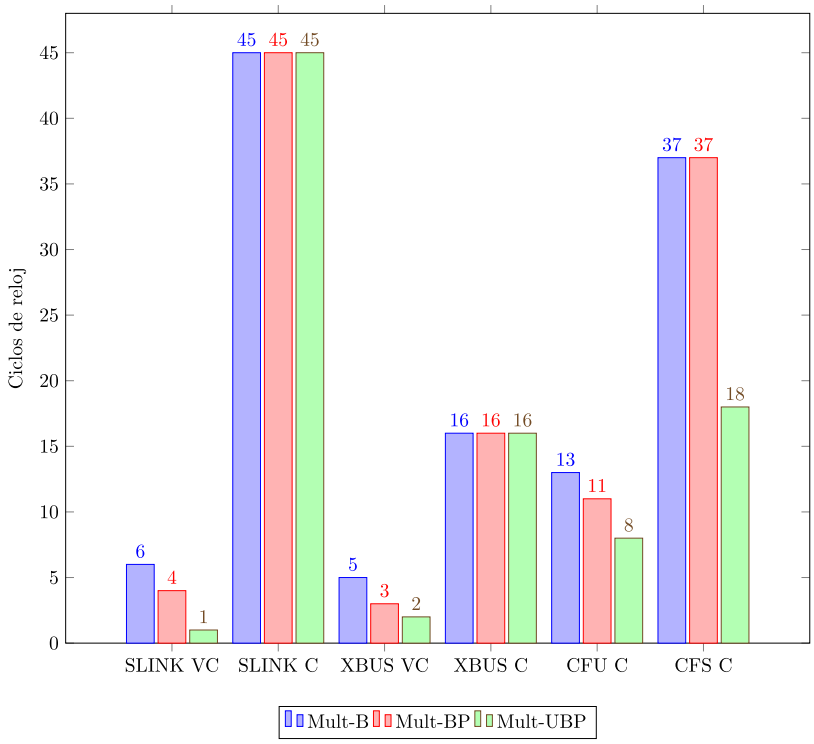


FIGURA 2.6: Resultados del ensayo de latencia: VC, el multiplicador individual acoplado a *Verification Components*; C, el SoC completo incluyendo el NEORV32, el multiplicador y la ejecución de software.

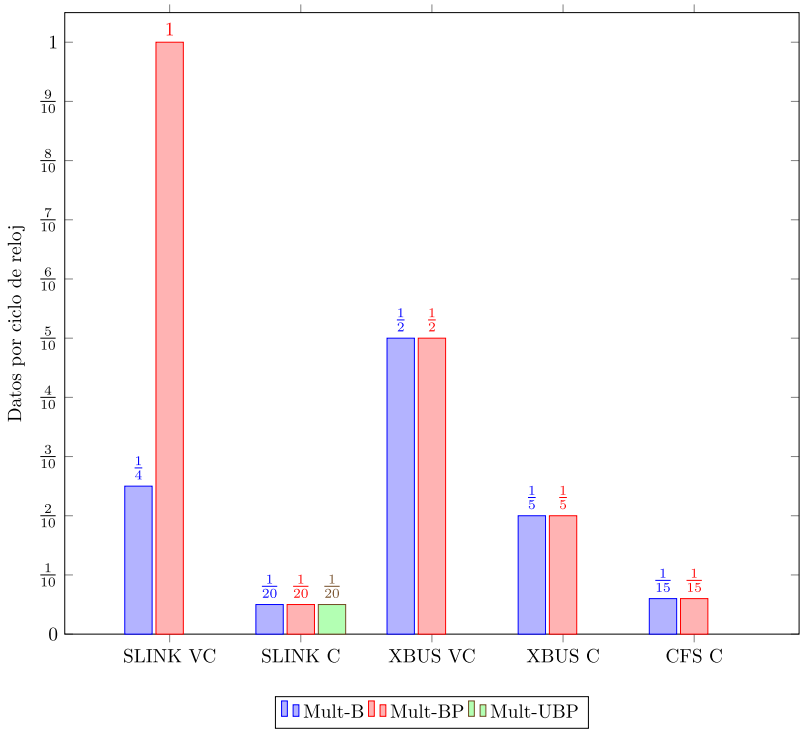


FIGURA 2.7: Resultados del ensayo de *throughput*: VC, el multiplicador individual acoplado a *Verification Components*; C, el SoC completo incluyendo el NEORV32, el multiplicador y la ejecución de software.

Cabe destacar que por cada ensayo de simulación se ha producido un archivo .vcd que contiene las formas de onda producidas. En este sentido, la integración continua se encarga de gestionar estos archivos y subirlos como artefactos. En el apéndice B, se expone la forma de onda referente al ensayo de *throughput* para el NEORV32 más Mult-BP acoplado mediante XBUS B.1, así como la referente al ensayo de latencia para el NEORV32 más Mult-B acoplado mediante CFU B.2.

2.3.4. Implementación en FPGA

Todas las combinaciones del SoC, reflejadas en la figura 2.3, se han verificado tanto en la placa Arty A7 35T como en la 100T, lo que ha supuesto la generación duplicada de los *bitstreams*. Como se ha mencionado, el hecho de duplicar las placas permite asegurar una doble redundancia en la verificación. Esta generación se ha realizado mediante las dos vías expuestas en la sección 2.2: herramientas FLOS y Vivado. Con objeto de ilustrar cómo se aprecia la integración continua, en la imagen 2.8 se muestran los procesos para generar de forma automatizada los *bitstreams* mediante herramientas FLOS en el repositorio de GitHub. En ella, se observa que se producen 18 artefactos, que contienen los *bitstreams* y estarán almacenados en GitHub durante 90 días. También se observa que la ejecución de los procesos es paralela, a diferencia de la relativa a Vivado en el CI de GitLab. Debido a las características de gestión de procesos de Orion, el *runner* asignado a Vivado solo permite la generación de *bitstreams* de forma secuencial. Este hecho aporta otro argumento que revalida la decisión de utilizar tanto herramientas FLOS como privativas. Además, en el apendice D código D.23, se muestra un ejemplo de un archivo bash realizado para generar automáticamente el *bitstream* de la implementación CFU mediante herramientas FLOS. En él, se observa un matiz destacable, al realizar la síntesis con yosys se añaden los argumentos `-nodsp` y `-nolutram`. Este hecho es debido a que la síntesis para la Arty mediante yosys no está del todo pulida, así como para Lattice sí, para Xilinix temas como la gestión de DSPs y de RAM distribuida todavía no están soportados. Además, para obtener un correcto funcionamiento de los *bitstreams* generados con herramientas FLOS, se ha tenido que rebajar la capacidad de la IMEM a:

```
MEM_INT_IMEM_SIZE : natural := 6*1024
```

Como se ha mencionado, además de la generación, el proceso de integración continua también automatiza la subida de los *bitstreams* como artefactos. En consecuencia, se han descargado y testeado todos ellos en ambas placas para comprobar su correcta operatividad. En este sentido, se ha obtenido un resultado satisfactorio.

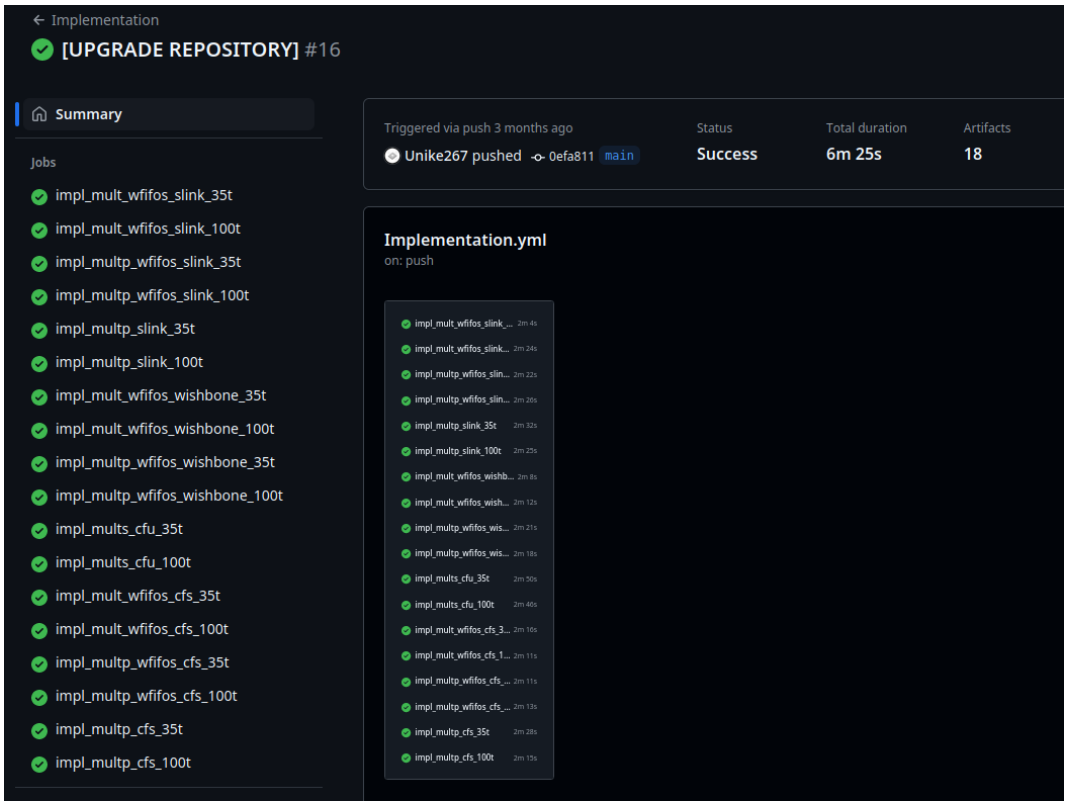


FIGURA 2.8: Procesos de integración continua para generar los *bitstreams* de los ensayos llevados a cabo mediante herramientas FLOS.

Los ensayos de implementación siguen un esquema similar a los ensayos de simulación. No obstante, la finalidad no es realizar la caracterización del rendimiento, sino verificar la correcta operatividad. En este sentido, todos los ensayos operan los mismos datos mediante el siguiente procedimiento. Un programa de la IMEM, resultado de la compilación de la parte de código C destinada a implementación descrita en D.11, D.12, D.13 y D.12, lanza 4 datos de entrada[7] a los multiplicadores mediante cada método de acoplamiento, el acelerador los multiplica y los devuelve al NEORV32. Además, las implementaciones muestran datos por UART, con objeto de visualizar en el ordenador la entrada y el resultado de cada operación, así como el tipo de ensayo. De todos los ensayos en implementación realizados, se muestran cuatro con objeto de ejemplificar la correcta operatividad de los diseños. Los resultados que se proceden a mostrar se han visualizado mediante la terminal CuteCom. En concreto, la figura 2.9 refiere al multiplicador Mult-B acoplado al NEORV32 mediante SLINK, la figura 2.10 refiere al multiplicador Mult-BP acoplado mediante XBUS, la figura 2.11 refiere los tres multiplicadores acoplados mediante CFU y la figura 2.12 refiere al multiplicador Mult-UBP acoplado mediante CFS.

[7]Los datos de entrada son 1 x 1, 2 x 2, 4 x 4 y 8 x 8 (al igual que en simulación); por lo que se debe obtener en hexadecimal 0x1, 0x4, 0x10 (16) y 0x40 (64).

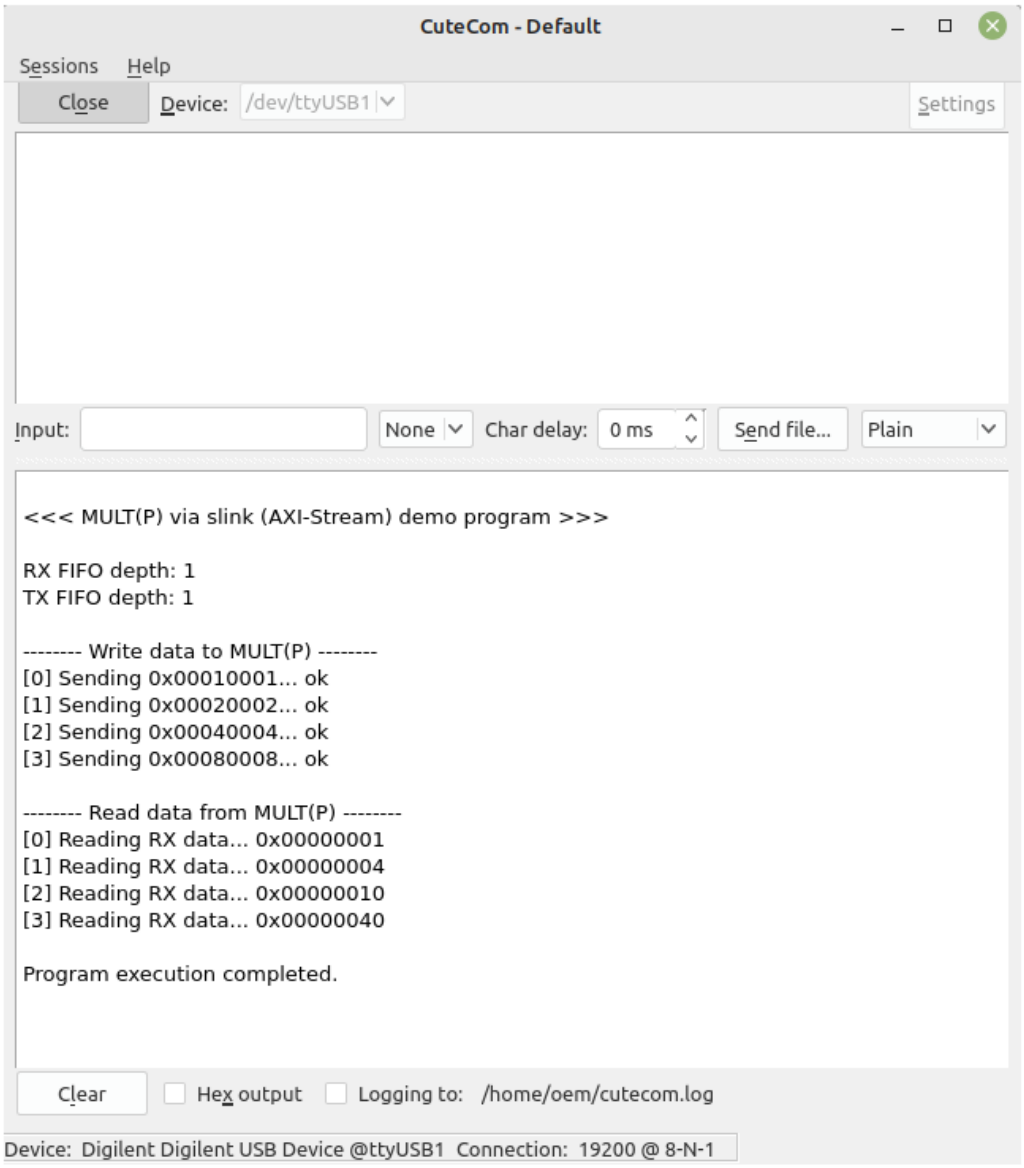


FIGURA 2.9: Ensayo de implementación de Mult-B acoplado al NEORV32 mediante SLINK.

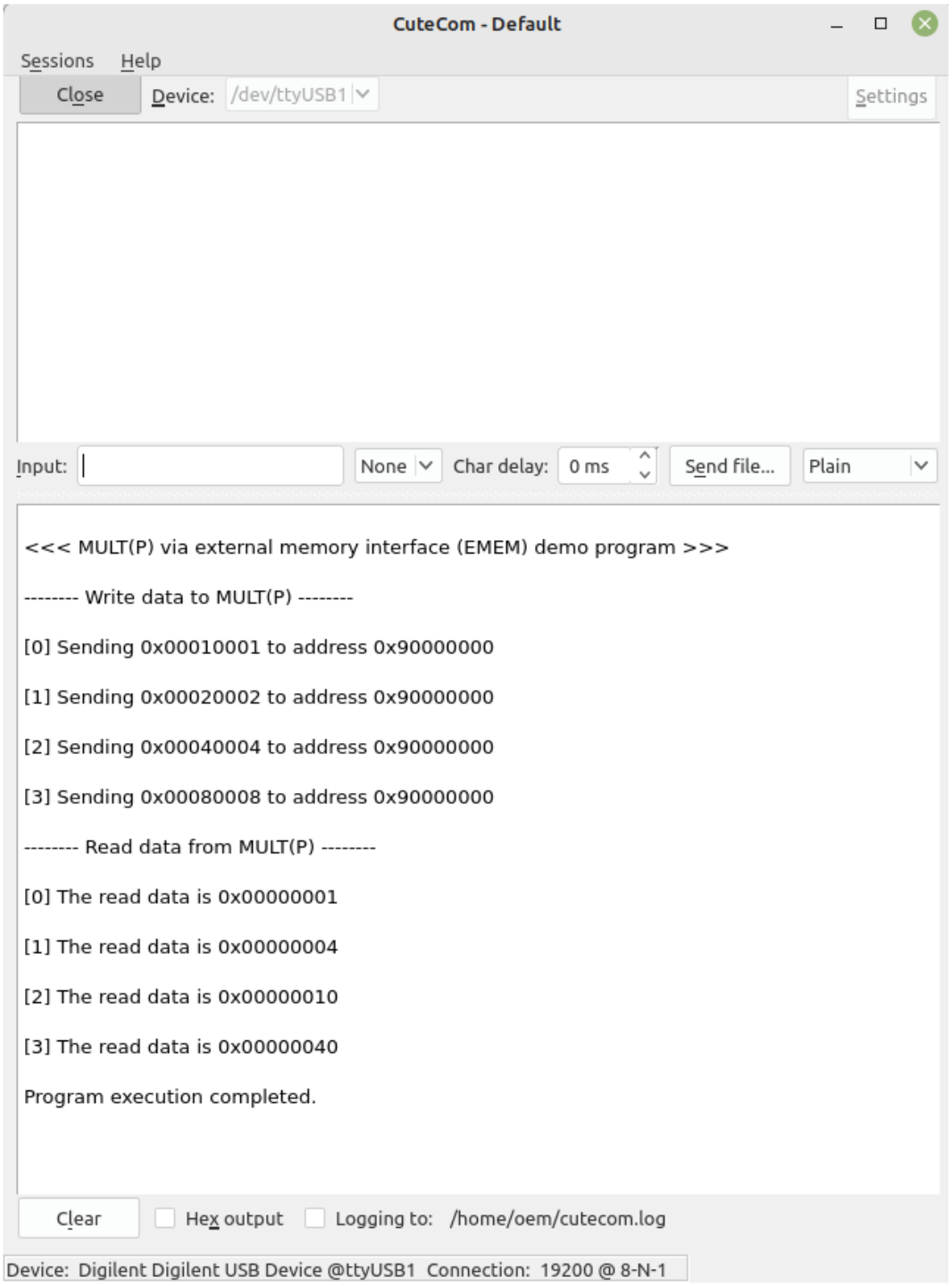


FIGURA 2.10: Ensayo de implementación de Mult-BP acoplado al NEORV32 mediante XBUS.

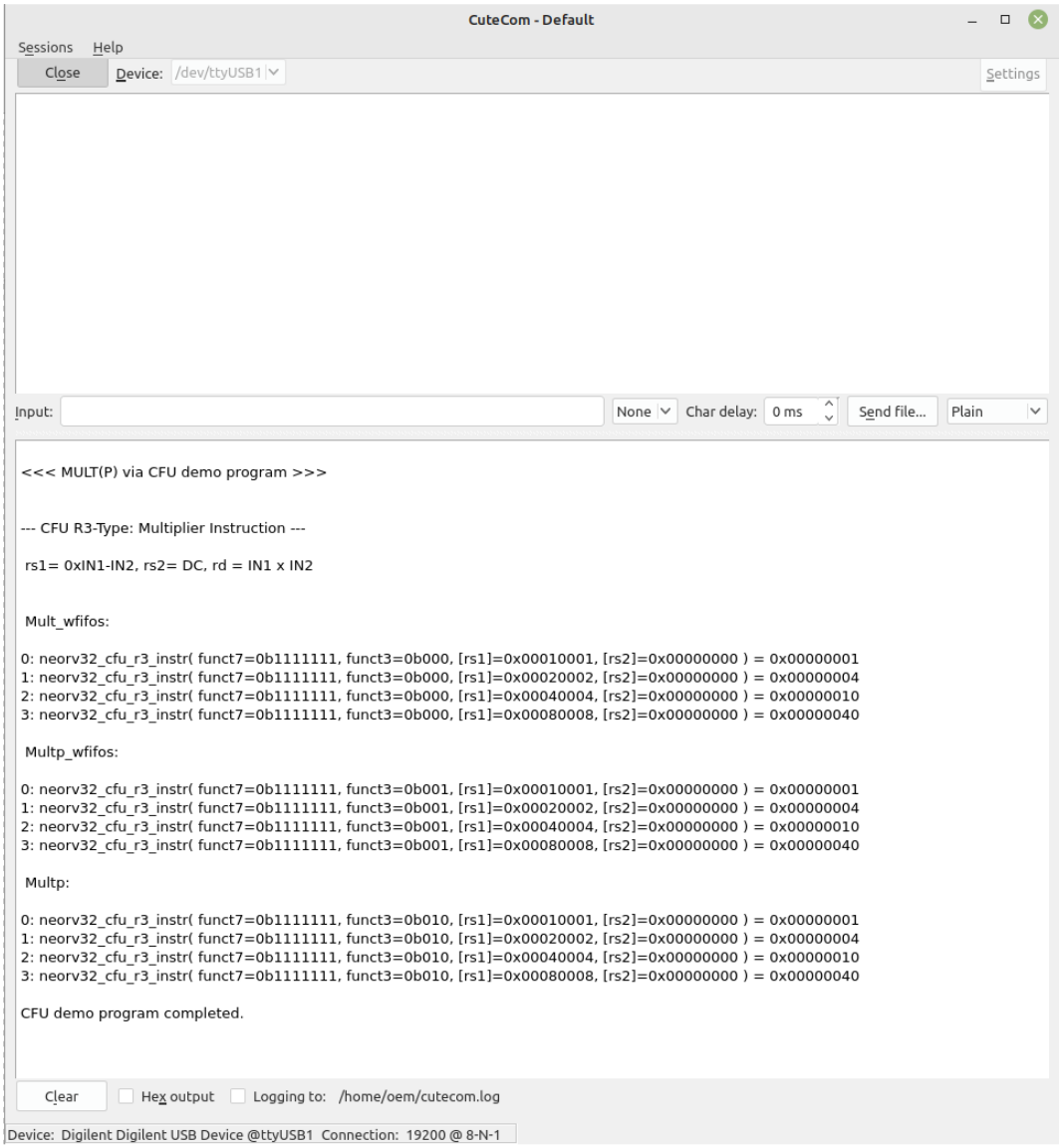


FIGURA 2.11: Ensayo de implementación de Mult-B, Mult-BP y Mult-UBP acoplados al NEORV32 mediante CFU.

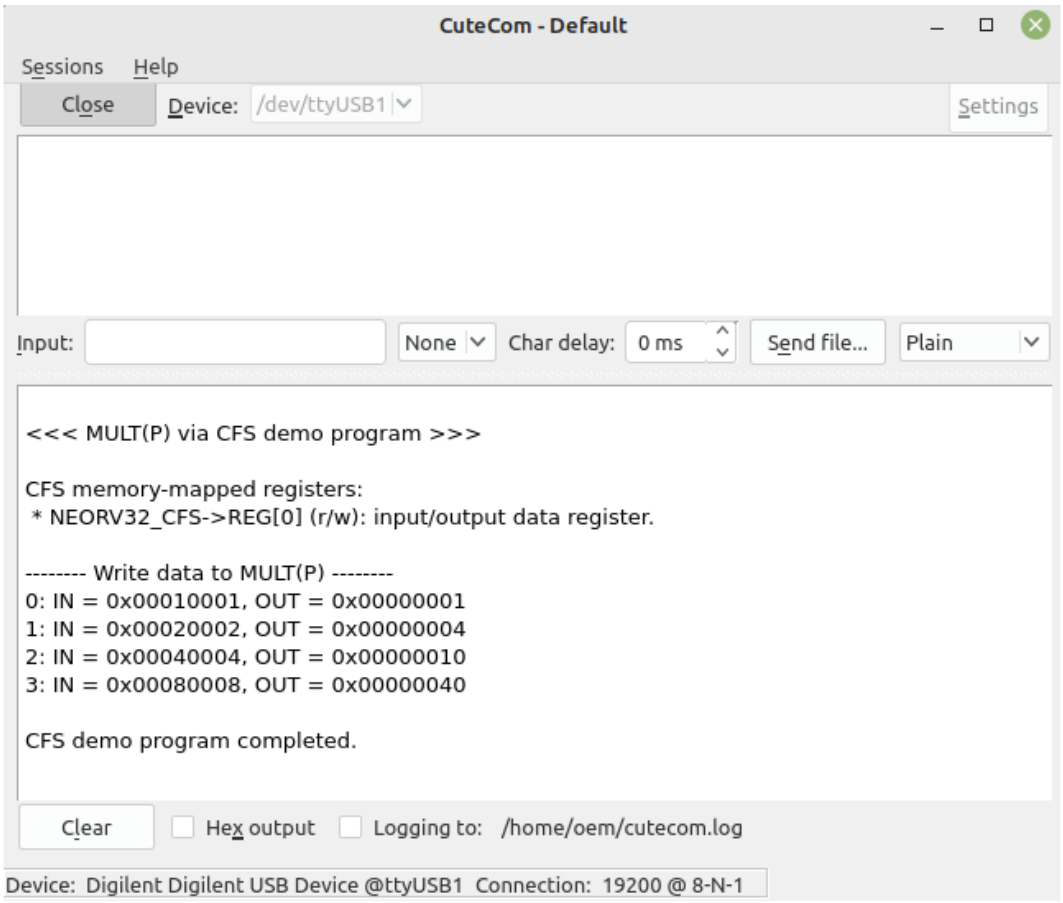


FIGURA 2.12: Ensayo de implementación de Mult-UBP acoplado al NEORV32 mediante CFS.

2.3.5. Análisis de los resultados

A la vista de estos resultados, podemos concluir que el método de integrar coprocesadores mediante Custom Functions Unit (CFU) es el modo de conexión que ofrece la latencia más baja, entre ocho y trece ciclos de reloj del sistema dependiendo del tipo de acelerador. Además, el rendimiento más bajo se obtiene mediante la interfaz External Bus (XBUS), un dato por cada cinco ciclos de reloj del sistema ($\frac{1}{5}$) para ambos aceleradores. Es importante señalar que dependiendo del modo de conexión, la arquitectura interna del acelerador puede o no afectar a la medición de latencia/*throughput*. Este hecho puede ser decisivo a la hora de seleccionar el modo de conexión, ya que según las características internas del acelerador personalizado el rendimiento de la transmisión puede verse afectado. Por lo tanto, en el caso de coprocesadores con baja latencia interna, se mejora la eficiencia mediante el modo de conexión CFU. No obstante, una vez que aumenta la latencia interna del coprocesador, el rendimiento del modo de conexión CFU se acerca más al de otros modos. Además, si el coprocesador a acoplar cuenta con *buffer* y nos interesa recibir el mayor *throughput* de datos posible, el modo de conexión óptimo sería a través de XBUS. Por otra parte, la interfaz *stream* destaca por la alta latencia que conlleva realizar una transmisión mediante este método de conexión. Este hecho es bastante significativo, ya que esta interfaz suele estar asociada a grandes velocidades de transmisión de datos. Sin embargo, la implementación *stream* (SLINK) que ofrece el NEORV32 no

muestra resultados que destaquen por su baja latencia. Este comportamiento puede deberse a diversos factores. Por un lado, al tipo de optimización llevada a cabo por el compilador. Ya que las funciones en C asociadas a este modo se deben de convertir a instrucciones específicas de RISC-V, el compilador puede realizar diferentes programas dependiendo del nivel de optimización ajustado. En este caso, el nivel ajustado es el predeterminado «EFFORT=-0s», pero se podría ajustar a «EFFORT=-03» (máximo nivel de optimización). Dado que todos los ensayos se han realizado con el nivel de optimización predeterminado, se ha decidido mantener esta coherencia para permitir una comparativa justa entre todos ellos. Por otro lado, a la gestión del movimiento de datos que realiza internamente este modo de conexión con sus *buffers*. Debido a que la interfaz SLINK cuenta con memorias FIFO (TX/RX) asociadas a las líneas de envío y recepción, la gestión del movimiento de los datos de entrada/salida con estas memorias puede aumentar la latencia general del método. Por último, respecto a la transmisión mediante Custom Functions Subsystem (CFS), se observa una diferencia clara entre las implementaciones de aceleradores con y sin *buffers*. Esto se debe a que para el caso de los aceleradores con *buffer* las señales de control de las memorias se deben gestionar mediante un registro extra, lo que implica añadir latencia al método. Por otra parte, sorprende que el *throughput* de datos del método general de implementación mediante registros mapeados en memoria sea significativamente mayor que CFS, que se supone un subsistema específico más eficiente. Esto es debido a que, en el caso de XBUS, la gestión de las señales de control de las memorias asociadas a los aceleradores con *buffers* se realiza mediante el *wrapper*. En este sentido, dicho *wrapper* escribe y lee de la FIFO utilizando las señales de control asociadas al propio estándar de conexión.

Cabe destacar que los resultados previamente analizados son los referentes al conjunto microprocesador más NEORV32. Los resultados relativos a los *Verification Components* sirven para caracterizar la latencia/*throughput* interna de cada acelerador aislado para el caso de los métodos AXI-Stream y Wishbone. En este sentido, se observa en ambos métodos un mayor rendimiento para los aceleradores que manejan las señales internas segmentadas. Además, para el caso de Wishbone, se observa una latencia de un ciclo menor para los aceleradores que incluyen memoria. Esto se debe a que para este modo de conexión, la medida se realiza entre el *acknowledge* de envío y el *acknowledge* de recibo. Normalmente, la interfaz Wishbone y también su implementación XBUS, tarda dos ciclos entre el establecimiento de la señal *strobe* y la recepción del *acknowledge*. En este contexto, el acelerador comienza a funcionar cuando se recibe el *strobe*, enmascarando un ciclo en la medida en comparación con el test de AXI-Stream *Verification Component*.

2.4. Integración de coprocesador para aplicaciones de IA

Una vez generada la caracterización de todos los métodos de conexión, se cuenta con un criterio objetivo para escoger un modo de acoplamiento y conectar de forma óptima un coprocesador para acelerar aplicaciones de IA. Atendiendo al análisis realizado en 2.3.5, así como a las características internas del acelerador CRI que destacan por su baja latencia, se decide realizar su integración en el NEORV32 mediante el método de conexión Custom Functions Unit (CFU). Como se ha mencionado, el coprocesador implementado únicamente se encarga de calcular la activación por medio de la función sigmoide. El procedimiento seguido para realizar la verificación, integración y comparación de este coprocesador se ha dividido en tres etapas. En la primera etapa, se realiza un ensayo de simulación del acelerador en solitario.

Además, se acopla al NEORV32 mediante el método CFU con objeto de verificar en placa la correcta operatividad del mismo. En la segunda etapa, se busca una manera de calcular la función de activación sigmoide haciendo uso de la unidad de coma flotante que ofrece el NEORV32. En la tercera etapa, se realiza un ensayo comparativo, tanto en simulación como en implementación, para verificar las ventajas de realizar el cálculo de la FA mediante un enfoque distribuido en términos de latencia.

2.4.1. Verificación de la operatividad del acelerador

El acelerador ha sido proporcionado por el grupo de investigación. Como se ha mencionado en 1.2.4, este coprocesador acelera los cálculos mediante el método de interpolación recursiva centrada. Cabe destacar que de las 7 funciones de activación disponibles solo se ha empleado la lógica de la función sigmoide. Además, se han debido de hacer las debidas adaptaciones para conseguir la correcta operatividad del circuito en un entorno de VUnit.

El objetivo de la verificación es sencillo, se deben recrear los resultados obtenidos por Basterretxea para los 9 datos de entrada recogidos en la tabla 2.4. Para ello se comienza con una simulación del coprocesador en solitario en el entorno de VUnit. Después, se acopla al NEORV32 mediante CFU y se realiza una implementación en placa. En este sentido, el NEORV32 se encargará de enviar los 9 datos de entrada al coprocesador, asimismo este calculará y devolverá el resultado de la FA de vuelta al microcontrolador. Tanto el ensayo en simulación del coprocesador en solitario como la generación del *bitstream* del ensayo en implementación del coprocesador acoplado al NEORV32 mediante CFU, se realizan mediante tareas automatizadas en la integración continua del repositorio de GitLab. Estos ensayos no se encuentran disponibles en el repositorio propio de GitHub.

TABLA 2.4: Resultados de la función sigmoide a verificar.

Dato de entrada	**Resultado de la función sigmoide**		
	Matlab		**Modelsim**
	Coma fija	**Hexadecimal**	**Coma fija**
-10	0	0000	0
-7,5	0,0078125	0002	0,0078125
-5	0,15625	0028	0,15625
-2,5	0,484375	007C	0,484375
0	1	0100	1
2,5	1,515625	0184	1,515625
5	1,84375	01D8	1,84375
7,5	1,9921875	01FE	1,9921875
10	2	0200	2

Los resultados obtenidos en simulación del coprocesador en solitario se muestran en la figura 2.13, así como los obtenidos en implementación para el coprocesador acoplado al NEORV32 vía CFU se muestran en la figura 2.14. Asimismo, el código que describe el acelerador utilizado se encuentra en el apéndice D código D.24, así como la integración en CFU se muestra en el código D.25. Además, se muestra la forma de onda resultante del cálculo de dos sigmoides por el acelerador CRI mediante CFU en el apéndice, B forma de onda B.3. En esta forma de onda se observa algo interesante, las señales de la CFU varían con respecto a la forma de onda B.2 (implementación del multiplicador). Esto es debido a que en el transcurso de la elaboración

de esta investigación la integración mediante CFU cambió ligeramente, como se ha mencionado en el apartado 1.2.3. Además, se observa que la latencia propia de la ejecución de la lógica del CRI es de 6 ciclos. Esta latencia es variable según el nivel de recursión escogido. Para este caso y en lo referente al resto de ensayos, se ha ajustado el parámetro de recursión «q» a 3. Este parámetro permite al acelerador CRI una gran flexibilidad, por ejemplo si se ajusta a 0, se obtiene una única interpolación, lo que resulta en una sigmoide dura (*hard-sigmoid*). Adicionalmente, el CRI cuenta con parámetros de saturación y pendiente ajustables. En lo referente a todos los ensayos llevados a cabo, estos parámetros se han ajustado a 2 y 0,5, respectivamente. Al igual que para el caso del multiplicador, se emplea una instrucción personalizada *R3-Type* de la extensión Zxcfu. Se emplea el registro fuente *rs1* para la entrada del acelerador. Ya que los datos a verificar se generaron con una longitud de palabra WL = 16 bits, se ha querido preservar el acelerador a este tamaño de entrada/salida para todos los ensayos realizados en este capítulo. En este sentido, en el código D.25 se observa como se ha gestionado este longitud con el tamaño de los registros *rs1* y *rd* de 32 bits. El registro fuente *rs2* no se utiliza y el resultado del cálculo de la función sigmoide se guarda en el registro de destino *rd*. Para especificar la función se ha empleado el campo *funct3*, asociando el coprocesador sigmoide a `funct3=000`.

```
493                0 fs - tb_sig                -    INFO - Simulation start (tb_sig.vhd:88)
494         50000000 fs - tb_sig                -    INFO - >>> This test writes eight values to CRIsig <<< (tb_sig.vhd:94)
495         50000000 fs - tb_sig                -    INFO - > First input (tb_sig.vhd:95)
496         50000000 fs - tb_sig                -    INFO - CRIsig input data is: -1.0e1 (tb_sig.vhd:71)
497        105000000 fs - tb_sig                -    INFO - CRIsig output data is: 0000000000000000 (tb_sig.vhd:79)
498        105000000 fs - tb_sig                -    INFO - CRIsig output (real) data is: 0.0 (tb_sig.vhd:80)
499        105000000 fs - tb_sig                -    INFO - > Second input (tb_sig.vhd:96)
500        105000000 fs - tb_sig                -    INFO - CRIsig input data is: -7.5 (tb_sig.vhd:71)
501        165000000 fs - tb_sig                -    INFO - CRIsig output data is: 0000000000000010 (tb_sig.vhd:79)
502        165000000 fs - tb_sig                -    INFO - CRIsig output (real) data is: 7.8125e-3 (tb_sig.vhd:80)
503        165000000 fs - tb_sig                -    INFO - > Third input (tb_sig.vhd:97)
504        165000000 fs - tb_sig                -    INFO - CRIsig input data is: -5.0 (tb_sig.vhd:71)
505        225000000 fs - tb_sig                -    INFO - CRIsig output data is: 0000000000101000 (tb_sig.vhd:79)
506        225000000 fs - tb_sig                -    INFO - CRIsig output (real) data is: 1.5625e-1 (tb_sig.vhd:80)
507        225000000 fs - tb_sig                -    INFO - > Fourth input (tb_sig.vhd:98)
508        225000000 fs - tb_sig                -    INFO - CRIsig input data is: -2.5 (tb_sig.vhd:71)
509        285000000 fs - tb_sig                -    INFO - CRIsig output data is: 0000000001111100 (tb_sig.vhd:79)
510        285000000 fs - tb_sig                -    INFO - CRIsig output (real) data is: 4.84375e-1 (tb_sig.vhd:80)
511        285000000 fs - tb_sig                -    INFO - > Fifth input (tb_sig.vhd:99)
512        285000000 fs - tb_sig                -    INFO - CRIsig input data is: 0.0 (tb_sig.vhd:71)
513        345000000 fs - tb_sig                -    INFO - CRIsig output data is: 0000000100000000 (tb_sig.vhd:79)
514        345000000 fs - tb_sig                -    INFO - CRIsig output (real) data is: 1.0 (tb_sig.vhd:80)
515        345000000 fs - tb_sig                -    INFO - > Sixth input (tb_sig.vhd:100)
516        345000000 fs - tb_sig                -    INFO - CRIsig input data is: 2.5 (tb_sig.vhd:71)
517        405000000 fs - tb_sig                -    INFO - CRIsig output data is: 0000000110000100 (tb_sig.vhd:79)
518        405000000 fs - tb_sig                -    INFO - CRIsig output (real) data is: 1.515625 (tb_sig.vhd:80)
519        405000000 fs - tb_sig                -    INFO - > Seventh input (tb_sig.vhd:101)
520        405000000 fs - tb_sig                -    INFO - CRIsig input data is: 5.0 (tb_sig.vhd:71)
521        465000000 fs - tb_sig                -    INFO - CRIsig output data is: 0000000111011000 (tb_sig.vhd:79)
522        465000000 fs - tb_sig                -    INFO - CRIsig output (real) data is: 1.84375 (tb_sig.vhd:80)
523        465000000 fs - tb_sig                -    INFO - > Eighth input (tb_sig.vhd:102)
524        465000000 fs - tb_sig                -    INFO - CRIsig input data is: 7.5 (tb_sig.vhd:71)
525        525000000 fs - tb_sig                -    INFO - CRIsig output data is: 0000000111111110 (tb_sig.vhd:79)
526        525000000 fs - tb_sig                -    INFO - CRIsig output (real) data is: 1.9921875 (tb_sig.vhd:80)
527        525000000 fs - tb_sig                -    INFO - > Ninth input (tb_sig.vhd:103)
528        525000000 fs - tb_sig                -    INFO - CRIsig input data is: 1.0e1 (tb_sig.vhd:71)
529        585000000 fs - tb_sig                -    INFO - CRIsig output data is: 0000001000000000 (tb_sig.vhd:79)
530        585000000 fs - tb_sig                -    INFO - CRIsig output (real) data is: 2.0 (tb_sig.vhd:80)
531        635000000 fs - tb_sig                -    INFO - Simulation end (tb_sig.vhd:106)
532 simulation stopped @635ns with status 0
533 pass (P=1 S=0 F=0 T=1) lib.tb_sig.Write_nine_values (0.4 s)
534 ==== Summary ==========================================
535 pass lib.tb_sig.Write_nine_values (0.4 s)
536 ==========================================================
537 pass 1 of 1
538 ==========================================================
539 Total time was 0.4 s
540 Elapsed time was 0.4 s
541 ==========================================================
542 All passed!
```

FIGURA 2.13: Verificación en simulación de la correcta operatividad del acelerador sigmoide en un entorno de VUnit.

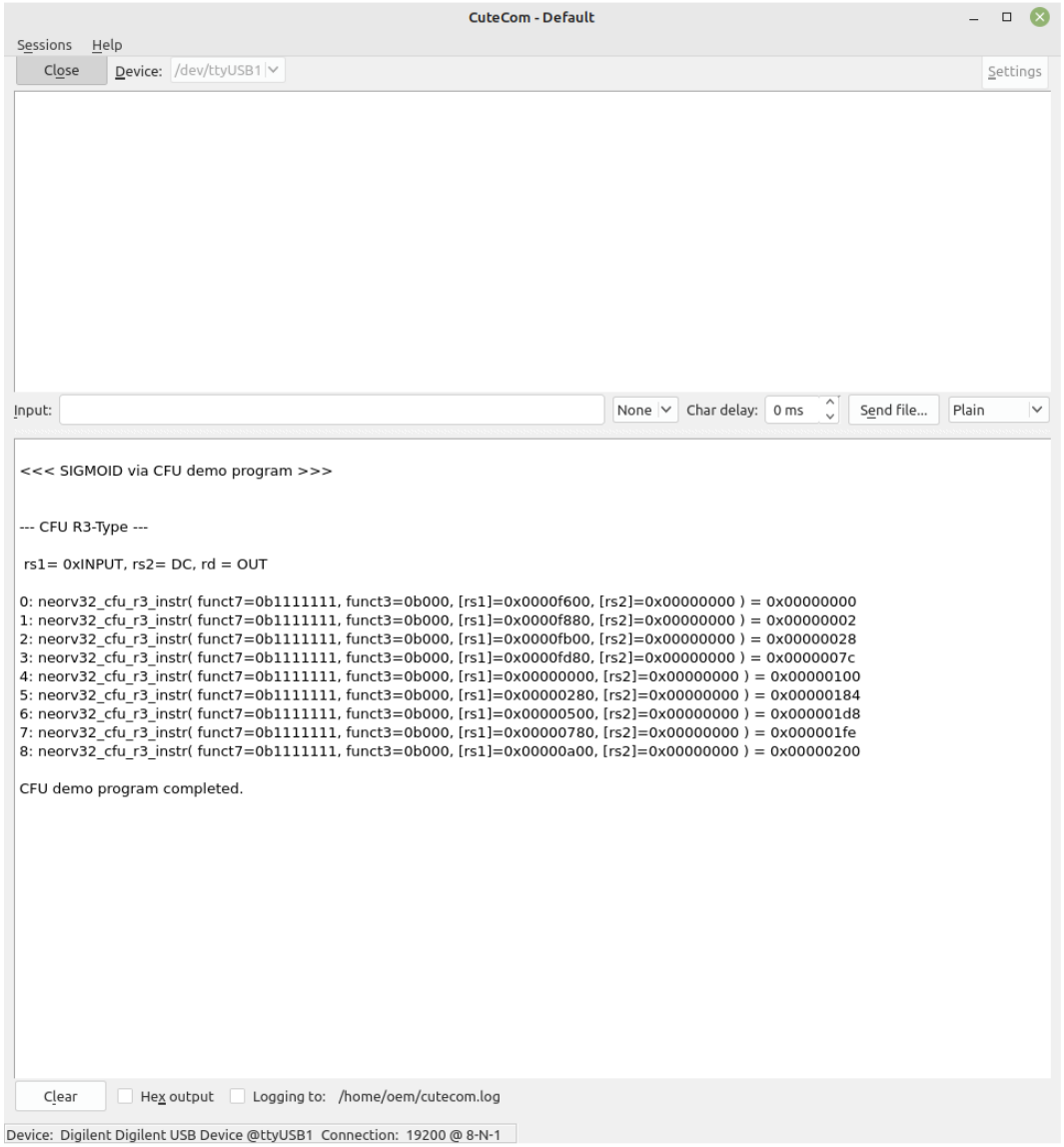


FIGURA 2.14: Verificación en implementación de la correcta operatividad del acelerador sigmoide acoplado al NEORV32 mediante CFU.

Como se puede apreciar en las imágenes los resultados en ambos casos han sido los esperados, cumpliendo de forma óptima la primera etapa de verificación.

2.4.2. Realización del cálculo de la FA mediante la FPU

La tarea de calcular la función de activación sigmoide mediante los recursos por defecto del NEORV32, se ha decidido implementar mediante la Unidad de Coma Flotante (FPU). Es decir, al contrario que el acelerador que se usa coma fija, para el enfoque realizado sin coprocesador se emplean datos en coma flotante. Normalmente, cuando una red se calcula por software, es decir, sin externalizar los cálculos a coprocesadores embebidos, los datos que se suelen usar son de este tipo, por lo que se ha visto adecuado emular este hecho. En concreto, las instrucciones relativas al manejo de la Unidad de Coma Flotante se encuentran en la extensión Zfinx. En este sentido surge un pequeño inconveniente. Debido a que la implementación

hardware actual de la extensión en el NEORV32 es limitada, esta no soporta operaciones de división. A priori, esto puede resultar un problema. Ya que la función sigmoide es una fracción (**), resolverla sin emplear operaciones de división puede resultar engorroso. Es por ello que se ha buscado una alternativa de cálculo. En primer lugar, se ha de tener en cuenta que las operaciones de suma y de multiplicación sí que están soportadas mediante las funciones `riscv_intrinsic_fadds()` y `riscv_intrinsic_fmuls()` respectivamente. Este hecho nos abre una ventana de posibilidad y nos permite implementar una aproximación polinomial a la función sigmoide. Para ello se ha realizado un programa en python descrito en el código D.26. Este programa hace uso de la librería numpy y su funcionamiento es la siguiente: se describe la función a aproximar, en este caso (***) (sigmoide ajustada a saturación = 2 y pendiente = 0,5), se calculan los coeficientes para tres grados distintos de polinomios y se realiza una gráfica comparativa. Cabe destacar que el rango de aproximación definido está condicionado por los valores a testear, es decir, entre -10 y 10. Se decide realizar polinomios de grado 3(****), 5(*****) y 7(******) con objeto de testear varías implementaciones y no comprometer ni el tiempo de ejecución ni la precisión, obteniendo varios ejemplos a comparar con el coprocesador.

$$f(x) = 2 \times \frac{1}{1 + e^{-0{,}5x}} \tag{***}$$

Los polinomios de aproximación a la función sigmoide obtenidos son los siguientes:

$$f(x) = 1 + 0{,}19744040439x - 0{,}00109773200x^3 \tag{****}$$

$$f(x) = 1 + 0{,}22878851178x - 0{,}00253272801x^3 + 0{,}00001266682x^5 \tag{*****}$$

$$f(x) = 1 + 0{,}24190955524x - 0{,}00369209483x^3 + +0{,}00003770457x^5 - 0{,}00000015213x^7 \tag{******}$$

En la figura 2.15, se observa el resultado de las aproximaciones polinómicas propuestas para aproximar la función sigmoide en el rango indicado. En este sentido, en el código D.27 se puede observar como se implementan estas funciones en C. Además, cabe destacar que el NEORV32 no soporta la impresión de flotantes por UART, por lo que se ha tenido que adaptar el resultado a dos enteros que representen la parte entera y la parte decimal. Para este propósito, entre otras cosas, se ha utilizado la función `riscv_intrinsic_flts()` para realizar la comparación «menor que» de un flotante con cero y evaluar su signo. Para emplear las funciones de la extensión Zfinx, esta se debe activar en el TOP del diseño.

```
CPU_EXTENSION_RISCV_Zfinx => true
```

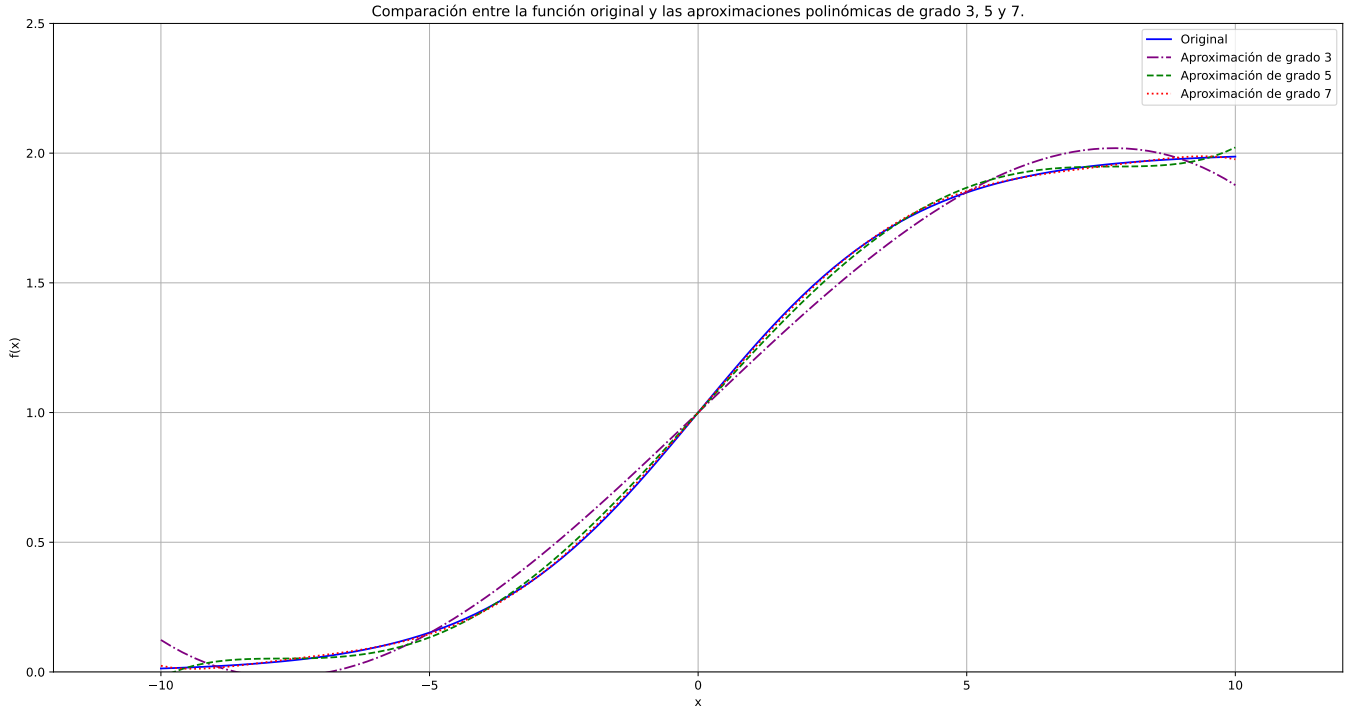


FIGURA 2.15: Comparación entre la función original y las aproximaciones polinómicas de grado 3, 5 y 7.

Cabe destacar un hecho relevante, como se ha mencionado las aproximaciones polinómicas están calculadas en función del rango propuesto. Es decir, su uso no se puede generalizar para todos los valores de entrada de la recta real. Esto se debe a que, para valores fuera del rango indicado, el polinomio se puede inestabilizar y desviar por completo de la aproximación a la función sigmoide. Además, para las aproximaciones de grado 3 y 5, las colas en los extremos no son aceptables. En consecuencia, para una aplicación real, el inicio y el final de estas aproximaciones se deben saturar a 0 y a 2, respectivamente.

2.4.3. Comparación de enfoques

La comparación de ambos enfoques se ha llevado a cabo mediante una simulación de VUnit y una implementación en placa. Tanto la simulación, como la generación del *bitstream* del ensayo en placa, se han automatizo en tareas de integración continua del repositorio de GitLab. Estos ensayos no se encuentran disponibles en el repositorio propio de GitHub.

Simulación

El ensayo de simulación consiste en lanzar un mismo paquete de 9 datos 4 veces, un vez por cada implementación sigmoide a testear y medir los ciclos que tarda cada una de ellas en calcular el resultado para cada dato de entrada. En este sentido, se ha acoplado mediante CFU el coprocesador CRI al NEORV32 con su FPU activada, con la cual se calculan las otras tres implementaciones. El orden seguido es el

siguiente, en primer lugar se lanza el paquete al acelerador CRI, en segundo lugar a la aproximación polinómica de grado 7, en tercer lugar a la aproximación polinómica de grado 5 y por último a la aproximación polinómica de grado 3, 36 datos en total. Para realizar las medidas de latencia, se ha utilizado la metodología expuesta en el apartado 2.3.2. El código empleado en esta simulación se encuentra en la parte de código destinada a simulación del programa D.27. Los resultados obtenidos se muestran en la figura 2.16

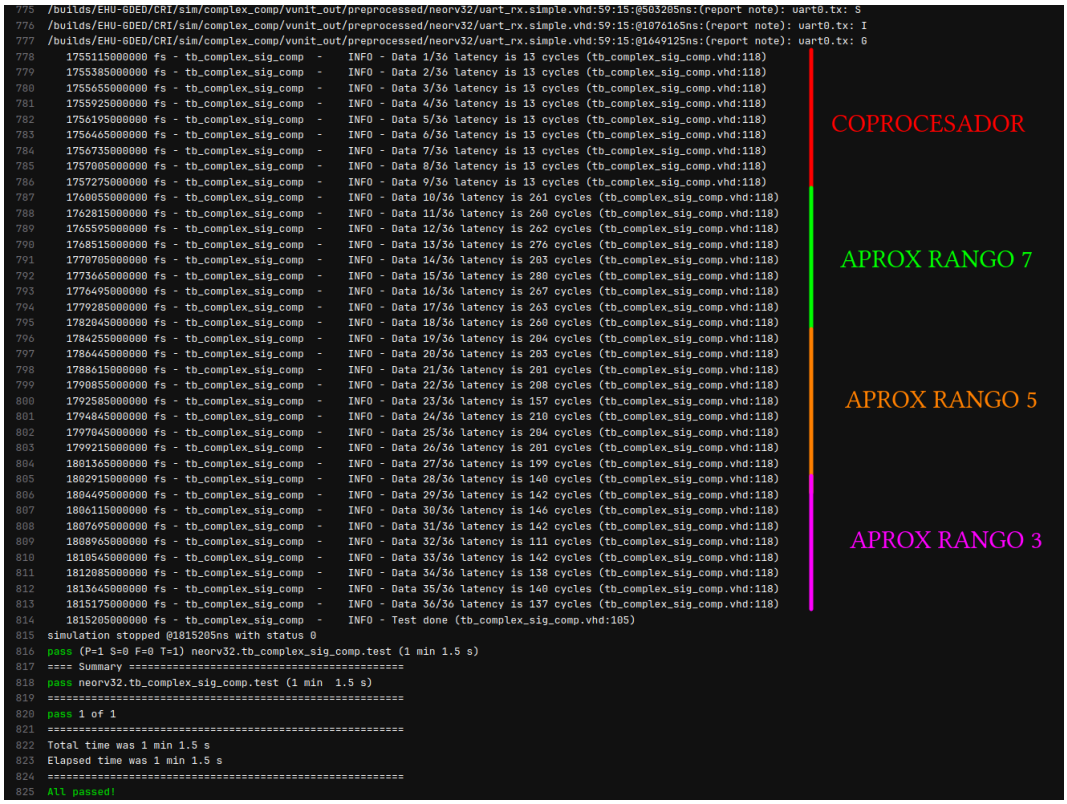


FIGURA 2.16: Resultados de simulación en ciclos de latencia necesarios para calcular la función sigmoide para cada dato de entrada.

Implementación

El ensayo en placa ha consistido en dos partes contenidas en un mismo programa, ver la parte de código destinada a implementación en D.27. Por un lado, se ha computado el resultado de la función sigmoide con los dos enfoques, el distribuido mediante el coprocesador acoplado al NEORV32 y el monolítico a través de las tres aproximaciones mediante la FPU. Para ello, se han utilizado los mismo 9 datos de entrada que en simulación. Por otro lado, se ha repetido cada uno de estos cómputos y mediante el uso del CSR(mcycle) se ha calculado la latencia que tarda cada uno de los enfoques en calcular un paquete completo de datos, dando un punto de vista más global. La figura 2.17 muestra el resultado del ensayo transmitido por UART y visualizado mediante la terminal CuteCom.

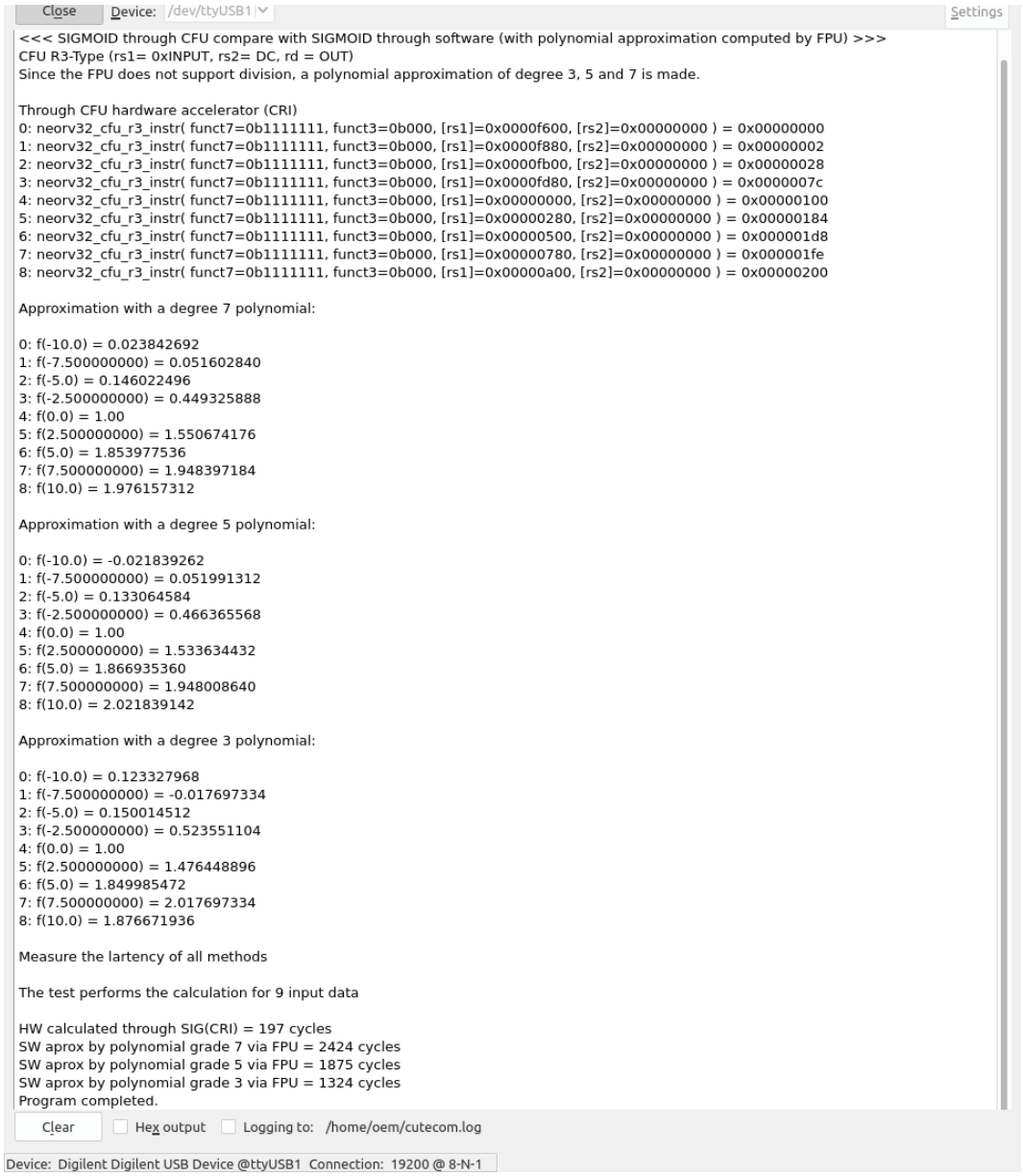


FIGURA 2.17: Resultado del ensayo comparativo en placa.

Resultados

Se distinguen dos tipos de resultados para cada planteamiento, la respuesta de la función sigmoide y la latencia en calcularla. Respecto a los resultados de la sigmoide, se observa que para los valores cercanos a los límites del rango relativos a las aproximaciones de grado bajo es necesario realizar una saturación, como ya se ha destacado en la subsección 2.4.2. No obstante, la aproximación de grado más alto se ajusta con relativa eficacia a los valores esperados. Respecto a la latencia, cabe destacar que la mínima para los casos de las aproximaciones mediante FPU coincide siempre con el 5º dato, el cual es f(0) = 1. De este hecho se deduce que se requiere poca carga computacional para calcular esa salida, es por ello que baja tanto su resultado. Si se observan el resto de casos, figura 2.16, la desviación no es tan notable. Respecto a los resultados del paquete de datos se aprecia que no se ajusta de forma directa al total de ciclos entre 9. Esto es debido a que en el ensayo de simulación

se mide rigurosamente cada ejecución de dato aislada, mientras que la implementación en placa mide, aparte de la ejecución per se, las instrucciones relativas al bucle for que corre la secuencia de introducción del paquete, lo que añade ciclos extra a la computación, ver D.27. Es por ello que se ha mencionado que el ensayo en placa da un punto de vista más global. Debido a que en la práctica no solo se ejecutan los ciclos relativos a las instrucciones propias, es de interés medir la parte del código añadido que compone la aplicación. Respecto al ahorro computacional en ciclos de latencia obtenidos mediante el acelerador CRI, se afirma que en el caso más ajustado se ahorran 111 - 13 = 98 ciclos por dato, lo cual es una aceleración de 8,53:1 y en el caso más holgado se ahorran 280 - 13 = 267 ciclos por dato, lo cual es una aceleración de 21,53:1. Si se realiza la media de las tres medias y se compara se obtiene un ahorro de 198,3 - 13 = 185,3 ciclos por dato, lo cual es una aceleración de 15,25:1. Estos hechos corroboran el beneficio de emplear este tipo de enfoque distribuido y afianzan el argumento de externalizar los cálculos relativos a las redes neuronales a coprocesadores embebidos con la finalidad de llevar a cabo una gestión computacional eficaz y simplificada de IA en el borde. Se procede a agrupar los resultados obtenidos en dos tablas, una referente al resultado del cálculo de la función sigmoide 2.5 y otra a la latencia relativa a cada ensayo 2.6. Además, se visualizan mediante gráficos de barras dichas latencias obtenidas en las figuras 2.18 y 2.19.

TABLA 2.5: Resultados de la función sigmoide obtenidos mediante los cuatro casos ensayados (implementación).

Entrada	**Coprocesador CRI**		**Aproximación mediante FPU**		
			Grado 7	**Grado 5**	**Grado 3**
	Coma fija	**Hexadecimal**	**Coma flotante**		
-10	0	0000	0,023842692	-0,021839262	0,123327968
-7,5	0,0078125	0002	0,051602840	0,051991312	-0,017697334
-5	0,15625	0028	0,146022496	0,133064584	0,150014512
-2,5	0,484375	007C	0,449325888	0,466365568	0,523551104
0	1	0100	1,00	1,00	1,00
2,5	1,515625	0184	1,550674176	1,533634432	1,476448896
5	1,84375	01D8	1,853977536	1,866935360	1,849985472
7,5	1,9921875	01FE	1,948397184	1,948008640	2,017697334
10	2	0200	1,976157312	2,021839142	1,876671936

TABLA 2.6: Resultados de latencia en ciclos de reloj del sistema obtenidos para los cuatro casos ensayados (simulación/implementación).

Un dato (simulación)											
Coprocesador CRI			**Aproximación mediante FPU**								
			Grado 7			**Grado 5**			**Grado 3**		
Min	**M**[8]	**Max**	**Min**	**M**	**Max**	**Min**	**M**	**Max**	**Min**	**M**	**Max**
13	13	13	203	259,1	280	157	198,5	210	111	137,5	146
Un paquete de 9 datos (implementación)											
Coprocesador CRI			**Aproximación mediante FPU**								
			Grado 7			**Grado 5**			**Grado 3**		
197			2424			1875			1324		

[8] «M» refiere a la media.

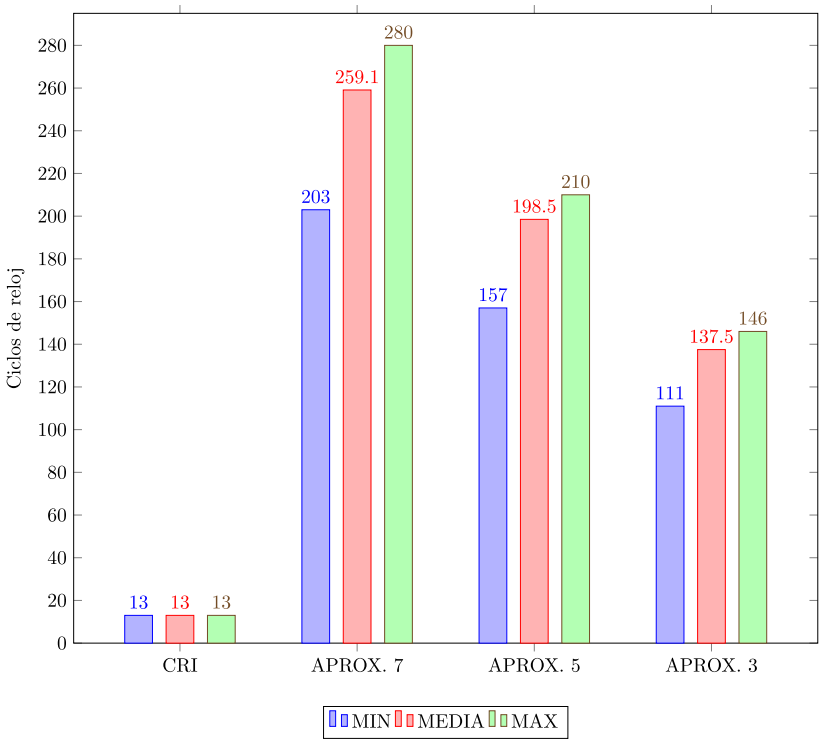


FIGURA 2.18: Resultado de las latencias mínimas, medias y máximas para un dato.

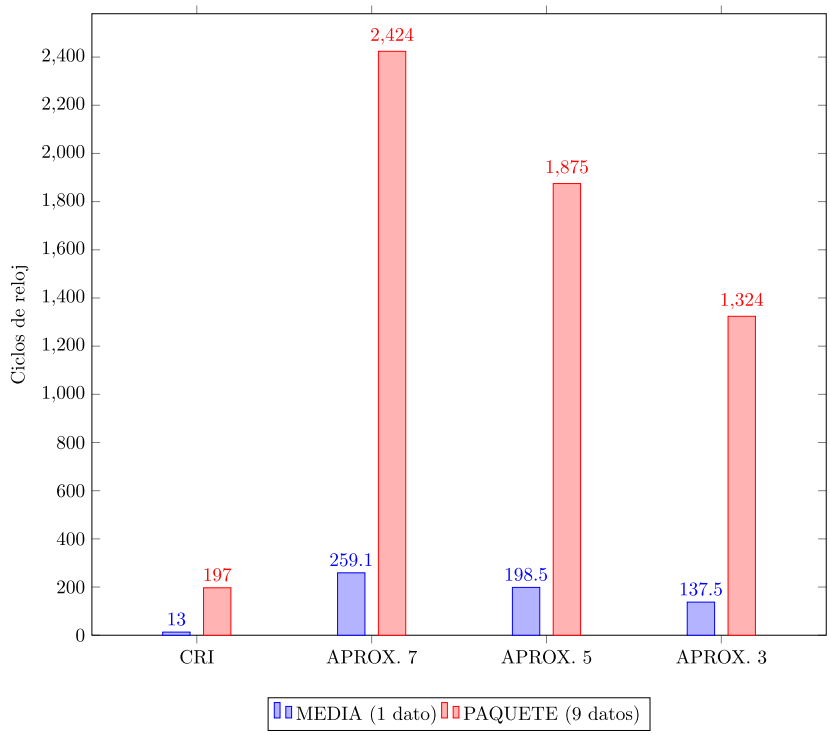


FIGURA 2.19: Resultado de las latencias medias para un dato comparadas con las latencias para un paquete (9 datos).

Capítulo 3

Metodología seguida en el desarrollo del trabajo

3.1. Descripción de tareas, fases y procedimientos

Se procede a describir la metodología llevada a cabo para el desarrollo de este proyecto. Se diferencian 4 fases incluyendo diferentes tareas en cada fase. La primera y la cuarta fase son transversales a todo el proyecto. Sin embargo, la ejecución de la tercera fase depende de la finalización de la segunda. Además, tanto la segunda como la tercera fase dependen de los conocimientos establecidos en la primera.

3.1.1. Fase 1. Recursos de desarrollo hardware y software

Descripción

Esta fase tiene como objetivo afianzar los conocimientos relativos a las plataformas de desarrollo hardware y software que se han utilizado durante la elaboración del proyecto, así como consolidar el uso de herramientas de control de versiones y la metodología de integración continua para la gestión eficaz del código empleado.

Recursos

- **Recursos hardware**
  - Dos PCs (Laboratorio y personal)
  - FPGAs:
    - Una Xilinx Artix-7 35T; Placa Digilent Arty A7 (Laboratorio)
    - Dos Xilinx Artix-7 100T; Placa Digilent Arty A7 (Laboratorio y personal)
    - Una Lattice ICE40 [1]; Placa Alhambra II (Personal)
  - Servido Orion[2] (Laboratorio)
- **Recursos software**
  - FLOS:
    - CuteCom
    - GCC
    - GHDL + GHDL yosys plugin

[1] Su uso ha sido minoritario. No obstante, se ha realizado algún test con ella, ver #726
[2] Se ha empleado mayoritariamente para realizar implementaciones con Vivado.

    - Git + Gitk
    - GTKWave
    - Inkscape
    - KolourPaint
    - LaTeX
    - nextpnr-xilinx
    - openFPGALoader
    - Podman
    - prjx-ray
    - VUnit
    - Wavedrom
    - xed
    - Yosys
- Privativos:
    - Vivado
- Contenedores:
    - docker.io/btdi/latex
    - docker.io/ghdl/vunit
    - docker.io/ghdl/vunit:mcode-master
    - gcr.io/hdl-containers/impl/icestorm
    - gcr.io/hdl-containers/sim/osvb
    - ghcr.io/stnolting/neorv32/sim
    - ghcr.io/unike267/containers/impl-arty
    - ghcr.io/unike267/containers/latex-pygments
- Plataformas
    - GitHub
    - GitLab
- Lenguajes
    - Bash
    - C
    - Markdown
    - Python
    - TCL
    - TeX
    - VHDL
    - YML
- Sistemas operativos:
    - Fedora (Laboratorio)
    - Linux Mint (Personal)

Duración

Transversal, de inicio a fin de proyecto, es decir, del 01/02/2024 al 30/09/2024.

Tareas

Tarea 1

Objetivo: afianzar el uso de las herramientas software relativas al diseño hardware, su verificación y documentación.

Descripción: con objeto de desarrollar el proyecto, se requieren los conocimientos necesarios para manejar con soltura las herramientas software referentes a la elaboración/simulación de VHDL tanto FLOS como privativas, a la síntesis/implementación de VHDL tanto FLOS como privativas y a la compilación cruzada de programas de alto nivel, en concreto C, para ejecutar en RISC-V. Además, de ciertos conocimientos para desarrollar programas en python, así como para desarrollar documentos en markdown y LaTeX. En esta tarea, se adquieren estos conocimientos.

Tarea 2

Objetivo: afianzar el uso de herramientas/metodologías para la gestión eficaz de código.

Descripción: con objeto de gestionar de forma eficaz el código del proyecto, se requieren los conocimientos necesarios para utilizar la herramienta de control de versiones Git, complementada con las plataformas asociadas, así como dominar la metodología de integración continua en los repositorios en línea. Para el caso de Git, se requiere manejar con soltura los comandos asociados para, entre otras cosas, generar/fusionar/eliminar ramas y hacer *rebases* interactivos para reordenar/fusionar/eliminar *commits*, además de realizar de forma correcta *pull requests*. Respecto a lo referente a la integración continua, se requieren los conocimientos relativos al desarrollo de archivos bash, como el realizado en D.23 para realizar la secuencia de comandos necesarios para la generación de *bitstreams* mediante herramientas FLOS, así como los relativos al desarrollo de archivos TCL para realizar la secuencia de comandos necesarios para la generación de *bitstreams* mediante Vivado, con objeto de incluir estos archivos a una lista de ejecución mediante código YML y así llevar a cabo la metodología de integración continua tanto en GitHub como en GitLab. Asimismo, se ha de saber generar contenedores en CI, mediante un *container file*, con objeto de realizar algunos de los contenedores necesarios para nuestras aplicaciones. En esta tarea, se obtienen estos conocimientos.

3.1.2. Fase 2. Caracterización del rendimiento

Descripción

En esta fase se han realizado todas las tareas referentes a la caracterización de los métodos de conexión con los que cuenta el NEORV32. Ha sido la fase más prolongada y de ella ha dependido la fase 3. A su vez, esta fase ha dependido del correcto aprendizaje de las herramientas necesarias adquirido a lo largo de la fase 1.

Recursos

Los mencionados en 3.1.1, excepto:

- docker.io/btdi/latex

- ghcr.io/unike267/containers/latex-pygments
- Inkscape
- LaTeX

Duración

Del 01/02/2024 al 12/06/2024.

Tareas

Tarea 1

Objetivo: realizar el diseño de los multiplicadores a acoplar.

Descripción: con objeto de tener un abanico de aceleradores a acoplar, se decide diseñar 3 multiplicadores con características diferentes. En esta tarea, se realiza el diseño hardware y la simulación de cada uno de estos coprocesadores. Además, se realizan los *wrappers* para AXI-Stream y Wishbone y se verifican con *Verification Components*. Las simulaciones se realizan mediante VUnit y se añaden a la integración continua del repositorio.

Tarea 2

Objetivo: realizar la conexión de los multiplicadores definidos mediante SLINK.

Descripción: con objeto de testear el método de conexión Stream Link Interface (SLINK), se realiza la conexión de los tres multiplicadores mediante este método. Para ello, se implementa en hardware el conjunto del diseño NEORV32 + Mult acoplado mediante SLINK y se verifica por UART. La generación de los *bitstreams* se realiza mediante Vivado, así como mediante herramientas FLOS y se añade a la integración continua del repositorio.

Tarea 3

Objetivo: realizar la conexión de los multiplicadores definidos mediante XBUS.

Descripción: con objeto de testear el método de conexión Processor-External Bus Interface (XBUS), se realiza la conexión de los tres multiplicadores mediante este método. Para ello, se implementa en hardware el conjunto del diseño NEORV32 + Mult acoplado mediante XBUS y se verifica por UART. La generación de los *bitstreams* se realiza mediante Vivado, así como mediante herramientas FLOS y se añade a la integración continua del repositorio.

Tarea 4

Objetivo: realizar la conexión de los multiplicadores definidos mediante CFS.

Descripción: con objeto de testear el método de conexión Custom Functions Subsystem (CFS), se realiza la conexión de los tres multiplicadores mediante este método. Para ello, se implementa en hardware el conjunto del diseño NEORV32 + Mult acoplado mediante CFS adaptando las fuentes del NEORV32 necesarias y se verifica por UART. La generación de los *bitstreams* se realiza mediante Vivado, así como mediante herramientas FLOS y se añade a la integración continua del repositorio.

Tarea 5

Objetivo: realizar la conexión de los multiplicadores definidos mediante CFU.

Descripción: con objeto de testear el método de conexión Custom Functions Unit (CFU), se realiza la conexión de los tres multiplicadores mediante este método. Para

ello, se implementa en hardware el conjunto del diseño NEORV32 + Mult acoplado mediante CFU adaptando las fuentes del NEORV32 necesarias y se verifica por UART. La generación de los *bitstreams* se realiza mediante Vivado, así como mediante herramientas FLOS y se añade a la integración continua del repositorio.

Tarea 6

Objetivo: investigar sobre un método para realizar mediciones de latencia en el entorno NEORV32.

Descripción: con objeto de hacer los ensayos de caracterización del rendimiento, se debe contar con un método generalizado para realizar las mediciones en simulación. Se concluye que la medida mediante el CSR(mcycle) y la posterior extracción de su valor mediante *external names* para plasmar su resultado a través de la función de VUnit *info()*, es la opción más interesante. En este sentido, también se planteó evaluar la señal *valid* y *ack* para los métodos AXI y Wishbone respectivamente, extrayendo en un CSV los *time stamps* de los momentos en los que se daban estas señales y procesar estos valores mediante un programa en python. No obstante, esta metodología era poco generalizable y se desechó la idea. Sin embargo, para los ensayos del multiplicador aislado con *Verification Components*, se ha utilizado este concepto.

Tarea 7

Objetivo: realizar los ensayos de simulación para caracterizar el rendimiento de los métodos de conexión.

Descripción: con objeto de caracterizar el rendimiento de los métodos de conexión, se plantean dos ensayos: de latencia y si es posible de *throughput*. De esta manera, se realizan 29 ensayos mediante simulaciones de VUnit aplicando la metodología concluida en la tarea 6. Se comparan y se extraen conclusiones. Todos los ensayos se añaden como tareas de integración continua en el repositorio.

3.1.3. Fase 3. Integración del coprocesador de IA

Descripción

En esta fase se han realizado todas las tareas referentes a resolver una aplicación de IA mediante un enfoque distribuido (acelerador + micro) y verificar su beneficio frente a un enfoque monolítico (solo micro). Esta fase depende de la fase 2, así como del correcto manejo de las herramientas necesarias adquirido en la fase 1.

Recursos

Los mencionados en 3.1.1, excepto:

- docker.io/btdi/latex
- gcr.io/hdl-containers/impl/icestorm
- gcr.io/hdl-containers/sim/osvb
- ghcr.io/unike267/containers/impl-arty
- ghcr.io/unike267/containers/latex-pygments
- Inkscape
- LaTeX
- nextpnr-xilinx
- prjx-ray

- Yosys

Duración

Del 11/07/2024 al 22/09/2024.

Tareas

Tarea 1

Objetivo: realizar la verificación del coprocesador de IA.

Descripción: con objeto de verificar la operatividad del coprocesador de IA, se procede a testearlo mediante una simulación de VUnit en solitario y una implementación en placa acoplado al NEORV32 mediante CFU. La simulación y la generación del *bitstream* se añaden a la integración continua del repositorio.

Tarea 2

Objetivo: investigar como realizar una sigmoide mediante los recursos por defecto del NEORV32.

Descripción: con objeto de verificar el beneficio del enfoque distribuido, se ha de comparar contra un enfoque monolítico (empleando solo recursos del micro). Para ello, se decide realizar el cálculo de la función sigmoide empleando datos flotantes procesados mediante la FPU del NEORV32. Ya que normalmente los modelos de redes neuronales computados por software en procesadores de ámbito general utilizan este tipo de datos, se ve relevante buscar una forma de emular este hecho. Debido a que la FPU del NEORV32 tiene ciertas limitaciones, de momento no soporta la instrucción de división, se decide computar la sigmoide empleando aproximaciones polinómicas.

Tarea 3

Objetivo: realizar los ensayos tanto de simulación como de implementación para comparar los enfoques.

Descripción: con objeto de visualizar la ventaja de aplicar un enfoque distribuido, se realizan ensayos que materialicen una comparación en términos de resultados y latencia. Estos ensayos se llevan a cabo mediante una simulación de VUnit y un ensayo en placa. En ambos caso se testea el NEORV32 con FPU acoplado al coprocesador mediante CFU. La simulación y la generación del *bitstream* se añaden a la integración continua del repositorio.

3.1.4. Fase 4. Documentación

Descripción

Esta fase tiene como objetivo documentar el transcurso del proyecto llevado a cabo a lo largo de las fases 1, 2 y 3. El público objetivo de las diferentes tareas varía. En el caso de la tarea 1, está orientado al ámbito del grupo de investigación. En el caso de la tarea 2, está orientado al ámbito académico internacional. En el caso de la tarea 3, está orientado al ámbito académico local.

Recursos

De los mencionados en 3.1.1, los referentes a redacción y generación de gráficos.

Duración

Transversal, de inicio a fin de proyecto, es decir, del 01/02/2024 al 30/09/2024.

Tareas

Tarea 1

Objetivo: documentar la mayor parte de problemas enfrentados a lo largo del desarrollo del proyecto en forma de *issues*.

Descripción: con objeto de complementar el repositorio, cada vez que se ha chocado contra un problema considerado relevante, se ha resumido/descrito en una *issue*. En el caso de resolverse, se ha documentado el proceso para ello.

Tarea 2

Objetivo: desarrollar el artículo.

Descripción: con objeto de aportar a la comunidad un criterio basado en hechos que facilite la elección de un método de conexión a la hora de acoplar coprocesadores al NEORV32, se resume la parte del trabajo referente a la caracterización de los métodos de conexión. Además, se acompaña de una introducción y un contexto, así como se describe el flujo de trabajo llevado a cabo remarcando el uso de herramientas FLOS.

Tarea 3

Objetivo: desarrollar el TFM.

Descripción: resumir todo el trabajo realizado en un texto que logre describir de manera clara y concisa, qué se ha realizado, cómo se a realizado y qué resultados se han obtenidos para justificar porqué se ha realizado, así como efectuar las conclusiones y líneas futuras correspondientes. Además, se ha elaborado una memoria que resume el trabajo y lo introduce planteando tanto el contexto como el estado del arte, así como los beneficios y las alternativas del mismo.

3.2. Diagrama de Gantt

En diagrama de Gantt desarrollado en la figura 3.1, se ha realizado en base a la información que almacena Git en los *commits* relativos a cada tarea. El hecho de gestionar todo el código y la documentación bajo un control de versiones, permite conocer con exactitud la fecha en la que se ha desarrollado el contenido de cada fase.

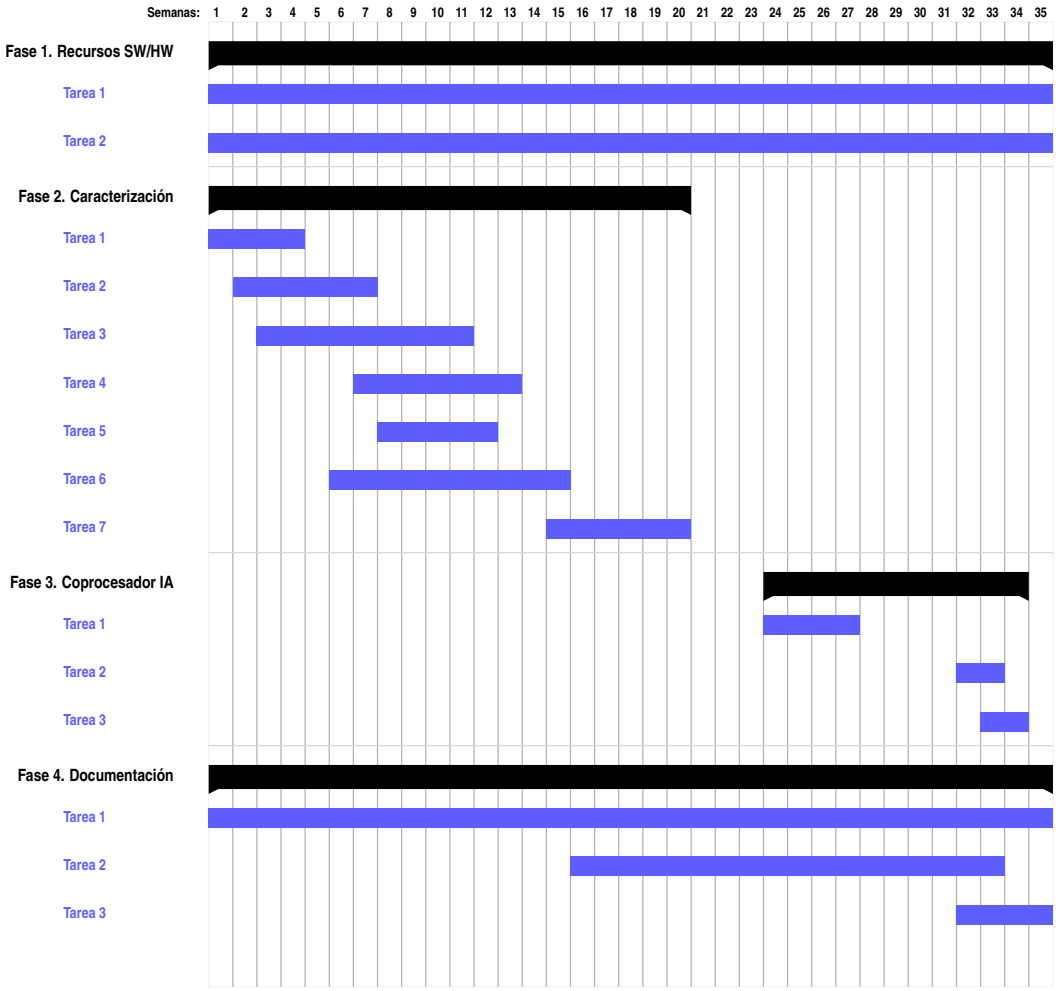


FIGURA 3.1: Diagrama de Gantt del desarrollo del trabajo.

Capítulo 4

Conclusiones

4.1. Conclusiones alcanzadas

El presente trabajo comprende tres vertientes principales: el uso de herramientas FLOS, la caracterización de los métodos de conexión con los que cuenta el NEORV32 y la aceleración de la FA sigmoide mediante un coprocesador CRI para aplicaciones de IA. Se procede a extraer de cada una de ellas las debidas conclusiones.

En primer lugar, el empleo de herramientas FLOS ha mostrado diferentes resultados en los tres entornos en los que se han utilizado: compilación, síntesis e implementación y simulación. Para el caso de la compilación cruzada, el resultado mediante GCC ha sido más que satisfactorio, obteniendo sin ningún tipo de inconveniente los programas para la arquitectura RISC-V desde un entorno x86. Teniendo en cuenta este hecho, es destacable señalar que el compilador de la herramienta privativa Xilinx SDK (actualmente Vitis) está basado en GCC. Esta adaptación tiene una configuración personalizada que optimiza GCC para su uso específico en las arquitecturas ARM (Zynq) y MicroBlaze. El hecho de que compañías privadas utilicen este compilador avalan su robustez. Para el caso de síntesis e implementación, los resultados obtenidos han sido más limitados. A pesar de que la herramienta de síntesis yosys está enormemente optimizada y depurada para el ecosistema de las FPGAs Lattice de bajo coste, para el caso de Xilinx no está bien optimizada. Sin embargo, se han podido generar correctamente *bitstreams* funcionales del SoC. No obstante, estos están limitados a implementarse sin la posibilidad de emplear DSPs ni RAM distribuida. Además, el tamaño de la IMEM que se puede sintetizar mediante estas herramientas está limitada, no por los recursos de la FPGA, sino por el propio sintetizador y/o por la herramienta de P&R. En consecuencia, los programas empleados estaban acotados a un tamaño reducido. A pesar de este hecho, todos los programas utilizados en la caracterización de los métodos de conexión han ocupado menos de 1024 x 6 bits, por lo que han sido implementados mediante herramientas FLOS y verificados correctamente en placa. Esta coyuntura es un poco desalentadora. Debido a que para la generación automática de *bitstreams* mediante CI en repositorios públicos son necesarias herramientas FLOS, es deseable que la síntesis y el P&R mediante ellas estén depurados y sean totalmente funcionales. Para el caso de simulación, los resultados que ofrecen las herramientas FLOS son extraordinarios y nada tienen que envidiar a las herramientas privativas. Al contrario, se han dado casos en los que Vivado se ha «tragado» errores de *loops* combinacionales que han podido ser corregidos del diseño porque GHDL sí que los ha detectado. Lo que hace pensar que objetivamente esta herramienta FLOS es más rigurosa con el estándar VHDL del IEEE. Además, la composición de *test benches* mediante VUnit permite numerosas ventajas. Entre ellas, extraer valores de simulación a archivos CSV, permitir el usos de componentes de verificación (*Verifications components*), simplificar la

sintaxis del código mediante funciones propias y ejecutar consecutivamente varios test en un mismo ensayo, ayudando a la visualización global de resultados. Además, esta herramienta permite que el uso de varios simuladores, tales como Active-HDL, Riviera-PRO, GHDL, NVC y ModelSim/Questa. En el caso de este trabajo, se ha empleado para todas las simulaciones GHDL.

En segundo lugar, los resultados de la caracterización de los métodos de conexión han permitido obtener un criterio objetivo en lo relativo a elegir el modo de acoplar coprocesadores al NEORV32. A este respecto, se ha ratificado la eficacia de integrar el coprocesador mediante la CFU, adaptando una instrucción personalizada para cada acelerador. También se ha observado que los resultados de *throughput* obtenidos con XBUS son los más elevados. Un hecho relevante a recalcar es que en la Tabla 1 *Comparison of On-Chip Extension Options* de la guía de usuario del NEORV32 [55], donde se hace una comparativa cualitativa (sin resultados experimentales) de los métodos de conexión CFU, CFS y XBUS, se afirma que el acceso de latencia para la interfaz CFS es menor que para la interfaz XBUS. Esta afirmación choca con los resultados experimentales obtenidos y puede deberse al siguiente motivo: si se examinan el método aislado se puede obtener conclusiones diferentes. En el caso de los ensayos de simulación realizados, los métodos están influenciados por la lógica de los aceleradores. En algunos casos los aceleradores cuentan con *buffers*, por lo que se han de gestionar las señales de control correspondientes. Como se ha mencionado, para el caso de XBUS estas señales de control las gestiona el *wrapper* con lógica combinacional en función de algunas de las señales propias del estándar, lo que agiliza la transmisión. Para el caso de CFS, es necesario añadir un registro asociado al subsistema para gestionar estas señales de control, lo que añade latencia al proceso. Para el caso del acelerador sin *buffer* se obtiene un resultado más desconcertante, XBUS realiza la transmisión 2 ciclos más rápido que CFS y en ese caso no hay que manejar las señales de control de forma externa. Se desconoce por qué Stephan ha catalogado a XBUS como un método de transmisión más lento que CFS. El código empleado para la caracterización llevada a cabo en este proyecto se puede revisar en [64].

En tercer lugar, los resultados obtenidos para el coprocesador de IA han sido bastante interesantes. La aceleración obtenida mediante el enfoque distribuido genera una ventaja evidente respecto al cálculo monolítico (empleando solo los recursos del microcontrolador) logrando un ratio de aceleración medio de 15,25 a 1. Además, el acelerador CRI es relativamente fácil de configurar en términos de recursiones de interpolación, pendiente y saturación, lo que le da una gran flexibilidad. Con respecto a las aproximaciones polinómicas, se consideran mejorables mediante otros métodos. No obstante, es la solución que se optó ante la carencia de soporte para divisiones en coma flotante en un contexto de tiempo ajustado. Sin embargo, los resultados del cálculo de la sigmoide obtenidos mediante estas aproximaciones, a través de la FPU, no son significativamente malos dentro del rango propuesto teniendo en cuenta que se han de saturar las colas en los extremos. Cabe remarcar que esta comparación se ha realizado en el marco del NEORV32, en consecuencia se ha tenido que lidiar sobre sus limitaciones. Es decir, en el contexto de otros microcontroladores, se podría obtener un ratio de aceleración menor.

Por último, se ha de concluir que el uso de herramientas de control de versiones, así como la aplicación de la metodología de integración continua, han sido esenciales para la gestión del código empleado y la administración de los resultados obtenidos. En mi opinión estas herramientas y sus metodologías asociadas se deberían fomentar más en el ámbito académico universitario.

4.2. Líneas futuras

Una vez concluido el desarrollo del proyecto, se abren varias líneas de investigación que se podrían abordar en el futuro. Principalmente, se van a exponer cuatro de ellas: avanzar en la aplicación de IA propuesta, mejorar la aplicabilidad del proyecto, investigar sobre la gestión de memoria, investigar sobre el uso de RTOS.

En primer lugar, se observa razonable que una vez integrada la lógica relativa al cálculo de la función sigmoide, se debería dar soporte al resto de funciones de activación. Para ello, se acoplaría el CRI completo mediante CFU y se diseñaría una función personalizada de la extensión Zxcfu para cada FA, seleccionando entre ellas a través del campo *funct3*. Además, se puede seguir destinando el registro fuente *rs1* para introducir el valor de entrada, pero se podría utilizar el registro fuente *rs2* para ajustar los parámetros de recursión, saturación y pendiente del CRI. En este sentido, se podría calcular para un mismo dato de entrada la salida mediante varias FAs empleando varias recursiones, saturaciones y pendientes, lo que daría una gran flexibilidad a la hora de decidir la activación de la neurona. Además, con objeto de realizar una comparación más rigurosa, se podría inferir una red sencilla en software y compararla con la misma red pero con los cálculos de las FAs acelerados mediante CRI, lo que daría un punto de vista más global a la comparativa. Eventualmente, se podría escalar la aplicación de IA al coprocesamiento de operaciones de convolución y funciones de tipo *spike*. A este respecto, la característica de gestión segmentada es típica al gestionar operaciones de convolución. Además, una neurona tipo *spike* generalmente utiliza *buffers* en la entrada y la salida para gestionar el flujo de información. Dado que los picos o eventos son de carácter discreto, se emplean *buffers* para acumularlos y permitir que la neurona procese la suma de sus entradas. Asimismo, se emplean *buffers* a la salida para asegurar la sincronía de la salida con elementos de procesamiento posteriores. Por esta razón, se han emulado estas características en los aceleradores empleados para la caracterización de los métodos de conexión, ver sección 2.3.

En segundo lugar, se puede mejorar la aplicabilidad del proyecto de varias maneras. Por un lado, se podría añadir la compilación de software al CI. Este hecho se podría hacer, por ejemplo, generando un contenedor que reúna las herramientas FLOS necesarias para compilar, simular e implementar. Después, se añadirían nuevas sentencias a la lista de ejecución YML. En este sentido, se haría un *git clone* del repositorio del NEORV32, después se movería el programa en C junto a un *makefile* a una nueva carpeta en la ruta sw/examples, se compilaría, se movería el resultado al core y se procedería a la implementación y/o simulación. Por otro lado, se podría hacer una interfaz en linea de comandos (CLI - *Command Line Interface*) mediante python, con el objetivo de gestionar la simulación/implementación de todos los ensayos a través de una sentencia de comandos en la terminal.

En tercer lugar, con objeto de administrar el gran volumen de datos que supone procesar una red neuronal completa, se propone investigar respecto a las posibilidades de gestión de memoria que existen. En el trabajo realizado los datos estaban *hardcoded* en los programas en C, este modo de gestión es sencillo y sirve para hacer ensayos orientativos de latencia/*throughput*. No obstante, para manejar una gran cantidad de datos es necesario utilizar un soporte de memoria externo. Para ello, existen varias posibilidades. Por un lado, se puede utilizar la memoria RAM dinámica DDR con la que cuenta la Arty A7. Sin embargo, se necesita un controlador de memoria. En este sentido, se podría utilizar el que ofrece LiteX, aunque se tendría

que analizar cómo se incrusta el código y cómo se envían/reciben los datos de entrada/salida desde el punto de vista del NEORV32. Por otro lado, se podrían transferir los datos desde el ordenador al NEORV32 a través de JTAG mediante el *debugger*. A este respecto, tanto GDB como OpenOCD se mencionan en la guía del usuario. Además, atendiendo a la discusión 28 se podría asimilar cómo realizar esta operación. A modo de curiosidad, cabe mencionar que en el transcurso del proyecto se investigó el uso de la memoria SPI de la Arty para este propósito, consiguiendo enviar y recibir datos del NEORV32 a esta. A este respecto, se puede cargar mediante el *bootloader* un programa en C con todos los datos *hardcoded* y *flashear* la SPI. Después, cargar el programa de la aplicación específica y leer los datos de entrada desde esta memoria. Sin embargo, este método se desechó, ya que, debido a las características de la SPI, no es eficiente gestionar grandes volúmenes de datos mediante esta memoria. Además, a diferencia de la Lattice ICE40, la Arty A7 cuenta con memorias mucho más efectivas a la hora de administrar datos, por lo que es más lógico destinar el tiempo a estas opciones.

En cuarto lugar, se podría investigar sobre la integración de un sistema operativo en tiempo real en el NEORV32. En lo referente a este trabajo, se ha planteado la gestión de software por parte del microcontrolador desde un punto de vista *Bare metal*. Es decir, ejecutándose únicamente un software en la CPU. En este sentido, si se desean gestionar varios programas a la vez en una misma aplicación, existe un sistema operativo *Open Source* en tiempo real llamado Zephyr OS [65]. Se destaca este RTOS porque está documentado en la discusión 172 un soporte inicial para correrlo en el NEORV32.

Por último, todas estas líneas futuras están pensadas para implementar la arquitectura heterogénea propuesta en un SoC softcore prototipado sobre FPGA. No obstante, atendiendo a la iniciativa actual que promueve económicamente la soberanía europea en el ámbito de la microelectrónica, es razonable pensar en la posibilidad de fabricación de un SoC derivado de este estudio a modo de ASIC.

Apéndice A

Artículo de congreso

El presente trabajo de investigación ha dado pie a presentar un artículo en el marco del congreso *XXXIX Conference on Design of Circuits and Integrated Systems (DCIS)* que se celebrará del 13 al 15 de Noviembre de 2024 en Catania (Italia). Cabe destacar que los artículos aceptados para esta conferencia, con al menos un autor registrado y copyright firmado, se publicarán en IEEEXplore. A este respecto, el siguiente artículo ha sido aceptado y está en espera de ser publicado en dicha base de datos.

Hardware coprocessor integration with NEORV32: characterization for efficient implementation of RISC-V-based AI SoCs

Unai Sainz-Estebanez*, Unai Martinez-Corral†, and Koldo Basterretxea‡
Grupo de Diseño en Electrónica Digital (GDED) †‡Dept. Electronics Technology
University of the Basque Country (UPV/EHU)
Bilbao, Basque Country, Spain
*usainz003@ehu.eus †unai.martinezcorral@ehu.es ‡koldo.basterretxea@ehu.es

***Abstract*—Performing AI inference ubiquitously requires energy-efficient, small footprint and highly reliable processing devices. Heterogeneous processing architectures combining customized CPUs with domain specific coprocessors can provide a good trade-off between computational efficiency and application flexibility for edge AI deployments while shortening development times compared to full custom application-specific processor designs. Following the impulse for the European sovereignty in the microelectronics field, in this work we propose the use of a RISC-V based open-source hardware platform and Free/Libre and/or Open Source (FLOS) Electronic Design Automation (EDA) tools to evaluate the performance of different coprocessor integration options in a System-on-Chip (SoC) prototyped on FPGA. We tested four integration options (XBUS, Stream, CFS and CFU) to obtain precise data that will allow making the correct design decisions for the future development of integrated devices for high-performance AI at the edge.**

***Index Terms*—RISC-V, FPGA, NEORV32, FLOS tools, AI at the edge**

I. Introduction

Fully performing the growing number of AI inferences in the cloud is not only undesirable, it is unsustainable [1]. The future of AI is hybrid and distributed, and this future demands the development of highly efficient devices capable of offloading AI inferences to the edge. Integrating custom designed AI coprocessors with CPU cores in heterogeneous processing SoCs specifically tailored to target applications is a strong driver in the semiconductor market for AI today [2].

Last century, development of custom CPU microarchitectures and customisation of instruction sets was hindered by most available Instruction Set Architectures (ISAs) being paywalled. Hence, achieving a royalty-free design involved not only designing the microarchitecture but also building the software tooling for compilation to some custom ISA and for verification of the whole ecosystem. As a result of such effort, in the early 2000s very few production-ready open source CPUs existed, some of the most notable being the OpenSPARC [3] LEON [4] designed by the European Space Agency (ESA) and Gaisler Research, and the OpenRISC [5] ecosystem by OpenCores [6].

Since mid 2000s to 2010s multiple ISAs were made available royalty-free, such as, Power [7], SuperH [8], or RISC-V [9]. In some cases, original patents expired and/or copyright holders made existing specifications open. Conversely, RISC-V was conceived to be an open standard ISA since the project began in 2010, designed from scratch based on established reduced instruction set computer (RISC) principles.

Availability of open and royalty-free ISAs eased the development of microarchitectures to fit the requirements of application-specific tasks, allowing software tooling development and maintenance to be done collaboratively. Still, integrating custom hardware coprocessors with a CPU is beyond having a working microarchitecture and the ISA related software support. In the RISC-V ecosystem, communications are not limited to memory-mapped and stream interfaces. In 2019, Google [10] proposed the CFU concept [11] and integrated it into VexRiscv [12] CPU. Thereinafter, some others CPU microarchitectures have added this feature, such as NEORV32 [13]. Based on this work, there is a draft to specify Custom Function/Extension Units (CFUs/CXUs) [14] for tight software-hardware integration.

In this paper we present the results obtained from a set of experiments to accurately evaluate the processing performance of a RISC-V based SoC prototyped on FPGA by integrating simple coprocessing cores attached through different data communication modes, including memory-mapped and stream interfaces and custom ISA extensions. The whole design and verification tooling setup, from design entry to implementation, from functional simulation to physical data processing was performed using FLOS EDA tools. Obtained results will be used to propose efficient SoC designs for edge AI execution using custom coprocessing cores.

In section II the open source microarchitecture ecosystem is presented and the selection of the target CPU design is explained. In section III tools used for compilation, simulation,

This work was partially supported by Union Europea-NextGenerationEU through the Cátedras Chip program SOC4SENSING TSI-069100-2023-0004, by the Basque Government under grant KK-2023/00090, and by the Spanish Ministry of Science and Innovation under grant PID2020-115375RB-I00.
979-8-3503-6439-2/24/$31.00 ©2024 IEEE

synthesis and communication are explained. In section IV each of the four communication mechanisms is throroughly characterized. In section V results and future work are summarized.

II. RISC-V ECOSYSTEM AND SOC DESIGN

The use of an open standard ISA offers the possibility of describing soft-core CPUs and microcontroller-like SoC based on this architecture and share these designs freely and openly with the community, if the designer so wishes. In this way, users can contribute to the project, for example finding and fixing bugs and according to the features offered by a version control platform, such as GitHub [15] or GitLab [16], keeping the project active and subject to continuous improvement. In addition to this, one of the great advantages of using an open standard ISA is the possibility of implementing on an ASIC without the need to pay any type of royalties, contrary to what would happen with other closed architectures. There are multiple examples of open standard ISA based CPU projects, such as Microwatt [17] and Chiselwatt [18] which are based on Open POWER ISA. However, in this section we will focus on RISC-V based projects. At the moment, there are several RISC-V based CPU projects on GitHub, from microcontroller-sized to linux capable. These projects are described in scala, verilog, system verilog, VHDL and others HDLs. (e.g. Scala: Rocket Chip [19] and VexRiscv [12]; Verilog: PicoRV32 [20], Hummingbirdv2 E203 [21], DarkRISCV [22], Jasonlin316-RISC-V-CPU [23], RISCV-Atom [24] and RISC-V Steel [25]; VHDL: NEORV32 [13], ORCA RISC-V [26], Potato [27], RPU [28] and ReonV [29]).

Principally, we have two requirements. On the one hand, we need a description in VHDL, since our accelerators are in this language. On the other hand, we need the microcontroller to be provided with CFU, CFS and with a top of memory-mapped and stream. Of all these RISC-V projects, the one that meets our requirements is *NEORV32*. Additionally, it is equipped with an official Open Source RISC-V ID, 19 [30]. For these reasons, we have selected this project for the current work. The NEORV32 Processor [13] [31] [32] is an open source customizable microcontroller-like system on chip (SoC) built around the NEORV32 RISC-V CPU, described in platform-independent VHDL.

Figure 1 illustrates a Custom SoC Design composed of NEORV32 and different accelerators connected through various modes:

- SLINK: Stream Link (AXI4-Stream)
- XBUS: External Bus Interface (Wishbone/AXI4-Lite)
- Custom Functions Subsystem (CFS)
- Custom Functions Unit (CFU/CXU)

Stream Link is an interface to perform a stream transmission compatible with a subset of AXI4-Stream [33]. It provides independent unidirectional RX and TX channels for sending and receiving stream data. Each channel features a configurable internal FIFO to buffer stream data. XBUS is a general bus interface for attaching memory-mapped accelerators compatible with Wishbone [34] and AXI4-Lite [35]. An optional cache module *X-CACHE* can be enabled to improve memory

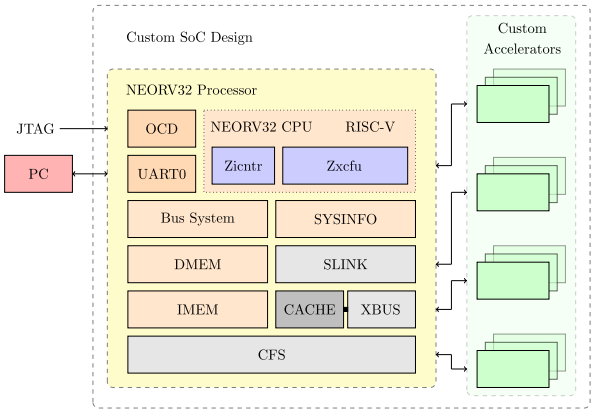


Fig. 1: Scheme of the Custom SoC Design.

access latency. Custom Functions Subsystem is an *empty* template for a memory-mapped, processor-internal module. It provides sixty-four 32-bit memory-mapped read/write registers. It should be noted that CFS does not have direct access to memory, all data (and control instruction) have to be send by the CPU. Custom Functions Unit is a functional unit that is integrated right into the CPU's pipeline. This was added by the NEORV32 developer based on CFU/CXU specification draft. It allows to implement custom RISC-V instructions. The instruction formats supported by NEORV32 are *R3-Type*, *R4-Type* and *R5-Type*. The first two types are a RISC-V standard and the last one is exclusive to NEORV32. Specifically, the first and the second type allows addressing two and three 32-bit input registers respectively. Besides, the function is selected through *funct7* and/or *funct3* in the first case and through *funct3* in the second case. The third type allows addressing four 32-bit input registers. Since it does not have the field *funct3* and/or *funct7* only two custom functions can be performed through this type, A format and B format. Figure 2 shows the 32-bit custom instruction types supported through Zxcfu NEORV32-specific ISA extension.

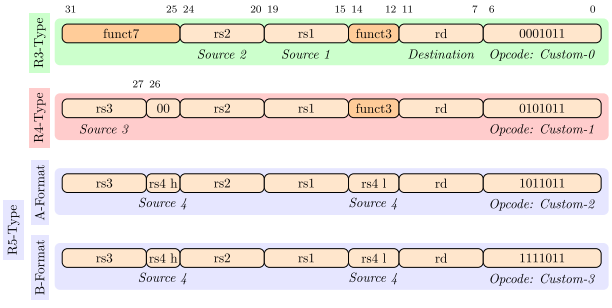


Fig. 2: CFU Custom Instruction Types.

TABLE I: Latency (L) and throughput (T) measurements performed by simulating: *VC*, the accelerator standalone along with Verification Components; and *Complex*, the whole SoC including NEORV32, the accelerator and software execution.

Accelerator		SLINK		XBUS		CFU	CFS
		VC	Complex	VC	Complex	Complex	Complex
Buffered	Not pipelined	Both	Both	Both	Both	L	Both
	Pipelined	Both	Both	Both	Both	L	Both
Unbuffered		L	Both	L	L	L	L

III. Workflow

FLOS and proprietary/commercial tools are not isolated ecosystems, on the contrary, in recent years, we have seen collaborative projects of open source tools integrated in vendor tools, such as RapidWright [36]. In addition to this, we have also seen a contest [37] sponsored by AMD [38] with the goal of promoting and demonstrating the FPGA Interchange Format [39] as an efficient and robust intermediate representation for working on backend FPGA problems, even at industrial scales. In the same line, Siemens [40] has observed a healthy growth between the Open Source VHDL Verification Methodology (OSVVM) [41] and the Universal VHDL Verification Methodology (UVVM) [42] since 2018, which in his own words "is encouraging" [43]. Therefore, in view of these events, we can affirm that traditional vendors are starting to facilitate the use of parts or all of FLOS allowing a hybrid future in the FPGA tools ecosystem.

Figure 3 illustrates the workflows for build, simulation, synthesis, place and root and generate bitstream through FLOS tools and proprietary tools. In this way, the implementation has been successfully tested on Arty A7 35t/100t FPGAs for all accelerators. For this purpose, the bitstream is generated, one the one hand, using *Vivado* [44] and on the other hand, using the following container [45]. This container is built and pushed in continuous integration and contains *GHDL* [46], *yosys* [47], *nextpnr-xilinx* [48] and *prjxray* [49]. It should be noted that the simulation framework allows the use of *ModelSim/QuestaSim* [50].

IV. Performance characterization

The performance characterization has been carried out using *VUnit* verification framework [51]. Since custom accelerators can differ in their internal characteristics, we have tested three designs with different latency and throughput ratios. The defining characteristics of each accelerator are as follows:

- Buffered not pipelined
- Buffered pipelined
- Unbuffered

The tests that have been performed are summarized in Table I and can be divided into two stages. In the first stage each accelerator has been independently tested in simulation with *VUnit Verification Components (VC)* for the Slink and XBUS connection modes. In the second stage the integrating design test bench including the whole SoC (NEORV32, accelerator and software execution) has been tested in simulation for the four selected modes of connection.

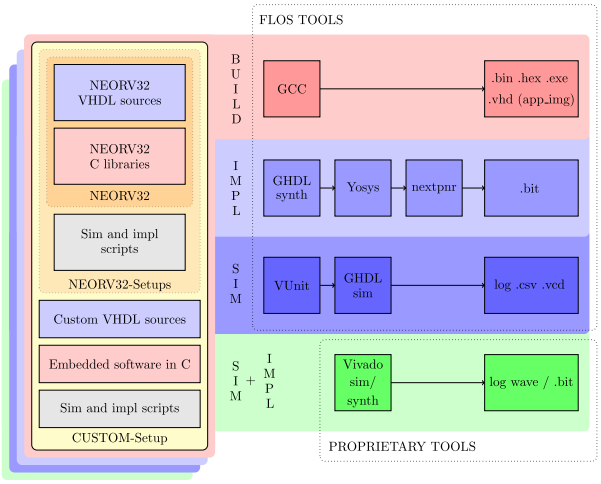


Fig. 3: Workflow of the Custom-Setup.

A. Measurement methodology

To perform a transmission with NEORV32, each connection mode is associated with one or more CPU instructions. Is the running time involved in applying each of these instructions that is measured. For this purpose, four transmission are performed in each test.

In the case of latency measurement, the operations of sending from NEORV32 to the accelerator and receiving from the accelerator to NEORV32 are executed consecutively four times and the running time of each send/receive operation is measured in system clock cycles. In the case of throughput measurement, data are sent from NEORV32 to the accelerator. Then, the receive operation is executed four times and the running time of how many data are received is measured in data per system clock cycles. For this reason, to perform the throughput measurement the accelerator or the mode of connection must have a buffer to store the inputs data from NEORV32. Therefore, for the case of the unbuffered accelerator only the throughput measurement can be performed for the SLINK mode, since this mode has associated transmitter/receiver FIFOs. For the case of CFU mode only latency measurement can be performed because according to the internal characteristics of the custom instruction the sent/received operation is performed in a single step.

The methodology followed to realize the measurements with the whole SoC has been generalized for all tests and the process is graphically summarized in Figure 4. When

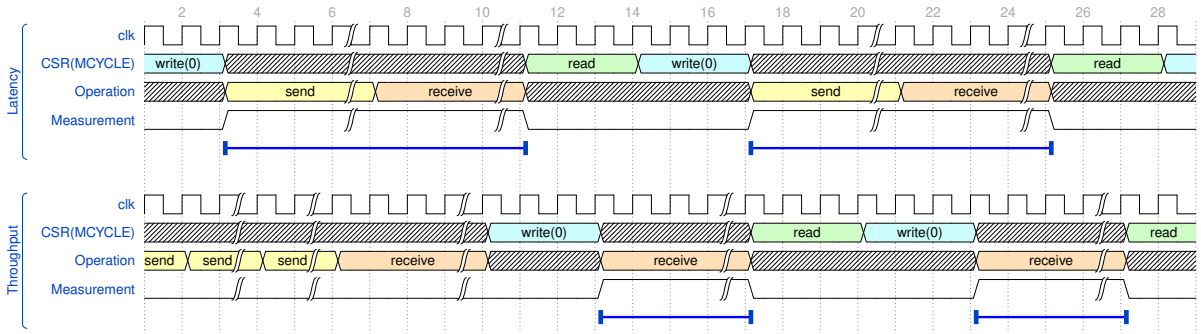


Fig. 4: Latency and throughput measurement process.

```
for x in 0 to test_items-1 loop
  wait until rising_edge(clk) and csr_we = '0' and
  csr_valid = '1' and csr_addr = x"B00" and
  csr_rdata_o /= x"00000000"; -- MCYCLE ADDR 0xB00
  info(logger, "Data " & to_string(x+1) & "/" &
  to_string(test_items) & " latency is " &
  to_string(to_integer(unsigned(csr_rdata_o))-1) &
  " cycles");
  wait until rising_edge(clk);
end loop;
```

Listing 1: VHDL code to extract *CSR(mcycle)* value.

the accelerators are attached to NEORV32 through the four selected modes several test benches are obtained. Each of them is associated with a C program that is compiled and loaded into the NEORV32 instruction memory. This program executes the instructions associated with each connection mode. To measure the execution time of these instructions the *control status register (CSR) mcycle* is used. In this way, whatever we want to measure is placed between `neorv32_cpu_csr_write(CSR_MCYCLE, 0)` instruction, which sets the register to 0 indicating the start of the measurement and `neorv32_cpu_csr_read(CSR_MCYCLE)` instruction, which reads the result indicating the end of the measurement.

The value of the *CSR(mcycle)* is extracted in simulation through the *VUnit info()* function adding Listing 1 to the test bench. Thus, we have automated the latency/throughput measurement every time the simulation of the whole SoC is launched visualizing the results of the measurements at the end of the simulation. It should be noted that the measurement result can be retried e.g. in CSV format using the *VUnit* logger for further processing.

As is shown in Listing 1, when *CSR(mcycle)* value is extracted, 1 is subtracted. This is due to the fact that internally `neorv32_cpu_csr_read(CSR_MCYCLE)` instruction adds one extra cycle to the measurement.

B. Measurement results

The latency and throughput measurement results are summarized in Table II. The entire workflow can be reproduced locally, since all the code and simulation/implementation scripts are available in GitHub. Henceforth, we will discuss the obtained results for each connection mode.

First, for the SLINK Complex test denotes the high latency compared to the SLINK (AXI-Stream) Verification Components test. To perform a send/receive operation with NEORV32 the functions `neorv32_slink_put` and `neorv32_slink_get` are used, respectively. These functions involve moving data through the associated TX/RX FIFOs, which slows down the transmission.

Second, for the XBUS (Wishbone) Verification Component test it should be clarified that it is measured from send acknowledge to receive acknowledge. Normally, XBUS communication takes two cycles between setting the strobe and receiving the acknowledge. In this context, the accelerator starts to operate when the strobe is received, masking one cycle in the measurement compared to SLINK (AXI-Stream) Verification Component test. To perform a send/receive operation with NEORV32 the functions `neorv32_cpu_store_unsigned_word` and `neorv32_cpu_load_unsigned_word` are used, respectively. In the latency case, the first transmission takes two cycles more than the rest because the compiler moves the XBUS address to an immediate register and since the address does not change, this move operation is skipped for all other transmissions. The most restrictive measurement is taken into account in the presentation of the results. Figure 5 illustrates graphically the transmission through XBUS for the buffered pipelined accelerator throughput test.

Third, for the CFU test three custom instructions *R3-type* are defined, one for each accelerator. `funct3=000` for buffered not pipelined, `funct3=001` for buffered pipelined and `funct3=010` for unbuffered. To perform a send/receive operation the `neorv32_cfu_r3_instr(funct7, funct3, rs1, rs2)` function is used. In this function the two 16-bit input operators are contained in *rs1* and the 32-bit result is stored in *rd*. When the *R3-type* custom instruction is executed, the operation of sending the content of *rs1* to

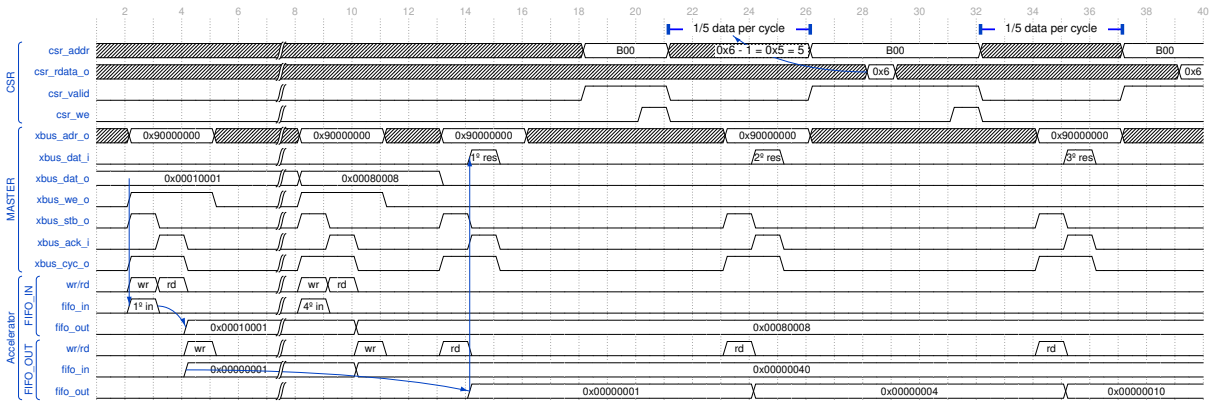


Fig. 5: Waveform of XBUS throughput for buffered pipelined accelerator.

the accelerator and receiving from the accelerator storing the result in *rd* is done consecutively in a single step. Figure 6 illustrates graphically the transmission through CFU for the buffered not pipelined accelerator latency test.

Finally, for the CFS test two situations can be distinguished. In the case of buffered accelerators, two of the sixty-four memory-mapped registers associated with CFS are used. One register to write/read the input/output data and another one to control the accelerator write/read signals. Therefore, to perform a send operation four write operations are needed: write the accelerator input data, write the accelerator write signal, write the accelerator read signal and clean the control register writing a zero in it. To perform a receive operation just one read operation is needed. In the case of unbuffered accelerator, only one memory-mapped register is used to write/read the input/output data. Therefore, to perform a send/receive operation just one write/read operation is needed. This is why the difference between buffered accelerators and unbuffered accelerator is so significant. Also, as in the case of the XBUS mode, the compiler adds extra instructions in the first transmission in order to move data to immediate registers. Again, the most restrictive measurement is taken into account in the presentation of the results.

V. Conclusion

In view of these results we can conclude that CFU is the connection mode that offers the lowest latency, between eight and thirteen system clock cycles depending on the accelerator type. In addition to this, the lowest throughput is obtained with the XBUS mode, one-fifth data per system clock cycle for both accelerators. It is relevant to note that depending on the connection mode, the internal architecture of the accelerator does or does not affect the latency/throughput measurement. This fact can be decisive when the connection mode is selected because according to the internal characteristics of the custom accelerator the transmission performance may be affected. Therefore, for coprocessors with low internal latency, the efficiency is improved through the CFU connection mode, but once the internal latency of the coprocessor is increased, the performance of this connection mode is closer to other modes. Furthermore, if the coprocessor is buffered and we are interested in receiving the highest possible data throughput, the most interesting connection mode would be through XBUS.

The results obtained in this work have allowed to establish which connection mode is the most efficient to attach coprocessors to RISC-V based processors, specifically for NEORV32 case. This information is relevant to us because we are working on optimal integration of different accelerators for efficient execution of AI models, for example, accelerating activation function calculations.

Acknowledgement

Thanks to NEORV32 author Stephan Nolting for his work in Open Source hardware development and documentation, as well as his dedication to keeping the project in constant evolution and up-to-date. His explanations and clarifications have been very useful for the elaboration of this work.

TABLE II: Measurement results: latency (*L*, system clock cycles) and throughput (*T*, data per system clock cycle).

	Accelerator		SLINK		XBUS		CFU	CFS
			VC	Complex	VC	Complex	Complex	Complex
L	Buffered	Not pipelined	6	45	5	16	13	37
		Pipelined	4	45	3	16	11	37
	Unbuffered		1	45	2	16	8	18
T	Buffered	Not pipelined	1/4	1/20	1/2	1/5	X	1/15
		Pipelined	1	1/20	1/2	1/5	X	1/15
	Unbuffered		X	1/20	X	X	X	X

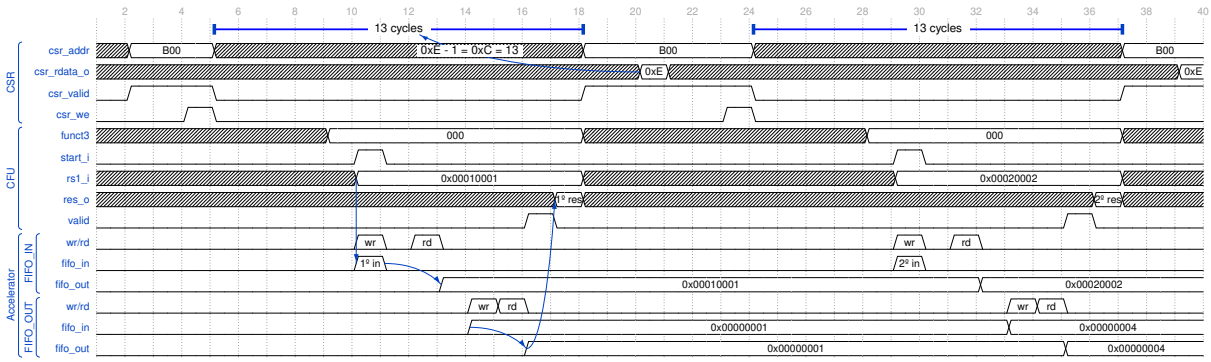


Fig. 6: Waveform of CFU latency for buffered not pipelined accelerator.

REFERENCES

[1] M. M. H. Shuvo, S. K. Islam, J. Cheng, and B. I. Morshed, "Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review," *Proceedings of the IEEE*, vol. 111, no. 1, pp. 42–91, 2023.

[2] S. Duan, D. Wang, J. Ren, F. Lyu, Y. Zhang, H. Wu, and X. Shen, "Distributed Artificial Intelligence Empowered by End-Edge-Cloud Computing: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 591–624, 2023.

[3] Sun Microsystems, Inc., "OpenSPARC." oracle.com/servers/technologies/opensparc-overview, 2005.

[4] European Space Research and Technology Centre, "LEON." esa.int/LEON_the_space_chip_that_Europe_built, 1997.

[5] D. Lampret and the OpenRISC Community, "OpenRISC." gh:openrisc, 2000.

[6] OpenCores Community, "OpenCores." opencores.org, 1999.

[7] OpenPOWER Foundation, "Power ISA." openpower.foundation, 2006.

[8] Hitachi/Renesas, "SuperH." renesas.com/us/en/products/microcontrollers-microprocessors/other-mcus-mpus/superh-risc-engine-family-mcus, 1992.

[9] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. Patterson, and K. Asanovic, "The RISC-V instruction set," in *2013 IEEE Hot Chips 25 Symposium (HCS)*, pp. 1–1, 2013.

[10] Google, "Google." about.google, 1998.

[11] S. Prakash, T. Callahan, J. Bushagour, C. Banbury, A. V. Green, P. Warden, T. Ansell, and V. J. Reddi, "CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (TinyML) Acceleration on FPGAs," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, Apr. 2023.

[12] C. Papon and Contributors, "VexRiscv - A FPGA friendly 32 bit RISC-V CPU." gh:SpinalHDL/VexRiscv, 2016.

[13] S. Nolting and Contributors, "NEORV32 - A tiny, customizable and extensible MCU-class 32-bit RISC-V soft-core CPU and microcontroller-like SoC." gh:stnolting/neorv32, 2024.

[14] J. Gray, "Draft Proposed RISC-V Composable Custom Extensions Specification." raw.githubusercontent.com/grayresearch/CFU/main/spec/spec.pdf, 2019.

[15] Microsoft Corporation, "GitHub." github.com, 2008.

[16] GitLab Inc., "GitLab." gitlab.com, 2011.

[17] A. Blanchard and Contributors, "Microwatt - A tiny Open POWER ISA softcore." gh:antonblanchard/microwatt, 2019.

[18] A. Blanchard and Contributors, "Chiselwatt - A tiny POWER Open ISA soft processor." gh:antonblanchard/chiselwatt, 2019.

[19] A. Waterman and Contributors, "Rocket Chip." gh:chipsalliance/rocket-chip, 2014.

[20] C. Xenia Wolf and Contributors, "PicoRV32 - A Size-Optimized RISC-V CPU." gh:YosysHQ/picorv32, 2021.

[21] Nuclei System Technology, "Hummingbirdv2 E203 Core and SoC." gh:riscv-mcu/e203_hbirdv2, 2020.

[22] M. Samsoniuk, "DarkRISCV." gh:darklife/darkriscv, 2018.

[23] Y.-C. Lin, "RISC-V CPU (Tape-Out with U18 Technology)." gh:jasonlin316/RISC-V-CPU, 2019.

[24] S. Singh, "RISCV-Atom soft-core processor." gh:saursin/riscv-atom, 2021.

[25] R. Calcada and contributors, "RISC-V Steel." gh:riscv-steel/riscv-steel, 2024.

[26] J. Vandergriendt, "ORCA RISC-V RV32IM core." gh:kammoh/ORCA-risc-v, 2015.

[27] K. Klomsten Skordal, "The Potato Processor." gh:skordal/potato, 2015.

[28] C. Riley, "RPU - Basic RISC-V CPU." gh:Domipheus/RPU, 2020.

[29] L. Castro, "ReonV RISC-V." gh:lcbcFoo/ReonV, 2018.

[30] RISC-V Community, "Open-Source RISC-V Architecture IDs." gh:riscv/riscv-isa-manual/blob/main/marchid.md, 2024.

[31] S. Nolting and Contributors, "The NEORV32 RISC-V Processor Datasheet." stnolting.github.io/neorv32/, 2020.

[32] S. Nolting and Contributors, "The NEORV32 RISC-V Processor User Guide." stnolting.github.io/neorv32/ug/, 2020.

[33] ARM, "AXI4-Stream." developer.arm.com/documentation/ihi0051/latest/, 2010.

[34] OpenCores, "Wishbone Bus." cdn.opencores.org/downloads/wbspec_b4.pdf, 2010.

[35] ARM, "AXI4-Lite." developer.arm.com/documentation/ihi0022/e/, 2010.

[36] AMD Research and Advanced Development, "RapidWright." gh:Xilinx/RapidWright, 2018.

[37] AMD, "Runtime-First FPGA Interchange Routing Contes." xilinx.github.io/fpga24_routing_contest/index, 2024.

[38] AMD, "Advanced Micro Devices, Inc.." amd.com, 1969.

[39] CHIPS Alliance, "FPGA Interchange Format." rapidwright.io/docs/FPGA_Interchange_Format, 2020.

[40] Siemens, "Siemens." siemens.com/global, 1847.

[41] J. Lewis and Contributors, "Open Source VHDL Verification Methodology." osvvm.org, 2013.

[42] E. Tallaksen and Contributors, "Universal VHDL Verification Methodology ." uvvm.org, 2013.

[43] Siemens, "The 2022 Wilson Research Group Functional Verification Study." blogs.sw.siemens.com/verificationhorizons/2022/11/21/part-6-the-2022-wilson-research-group-functional-verification-study/, 2022.

[44] Xilinx, "Vivado Design Suite." xilinx.com/products/design-tools/vivado, 2012.

[45] U. Sainz-Estebanez, "Ghdl + yosys + ghdl yosys plugin + nextpnr-xilinx + prjxray container." ghcr.io/unike267/containers/impl-arty:latest, 2024.

[46] T. Gingold and Contributors, "GHDL - VHDL 2008/93/87 simulator." gh:ghdl/ghdl, 2024.

[47] C. Xenia Wolf and Contributors, "YOSYS - Yosys Open SYnthesis Suite." gh:YosysHQ/yosys, 2020.

[48] D. Shah and Contributors, "Nextpnr-Xilinx - Experimental flows using nextpnr for Xilinx devices." gh:gatecat/nextpnr-xilinx, 2020.

[49] Project X-Ray Contributors, "Documenting the Xilinx 7-series bitstream format." gh:f4pga/prjxray, 2020.

[50] Siemens, "Questa advanced simulator." eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/, 2011.

[51] L. Asplund, O. Kraigher, and Contributors, "VUnit - Testing framework for VHDL/SystemVerilog." gh:VUnit/vunit, 2024.

Apéndice B

Formas de onda

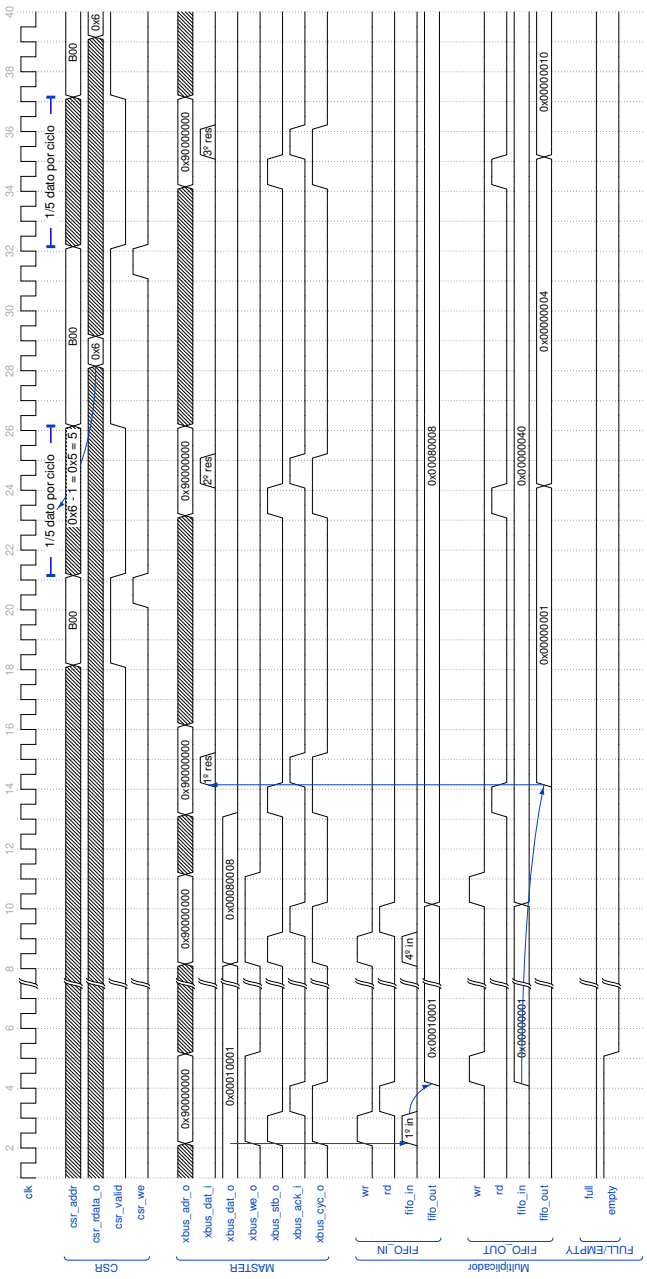


FIGURA B.1: Forma de onda resultante del ensayo de *throughput* para NEORV32 + Mult-BP acoplado mediante XBUS.

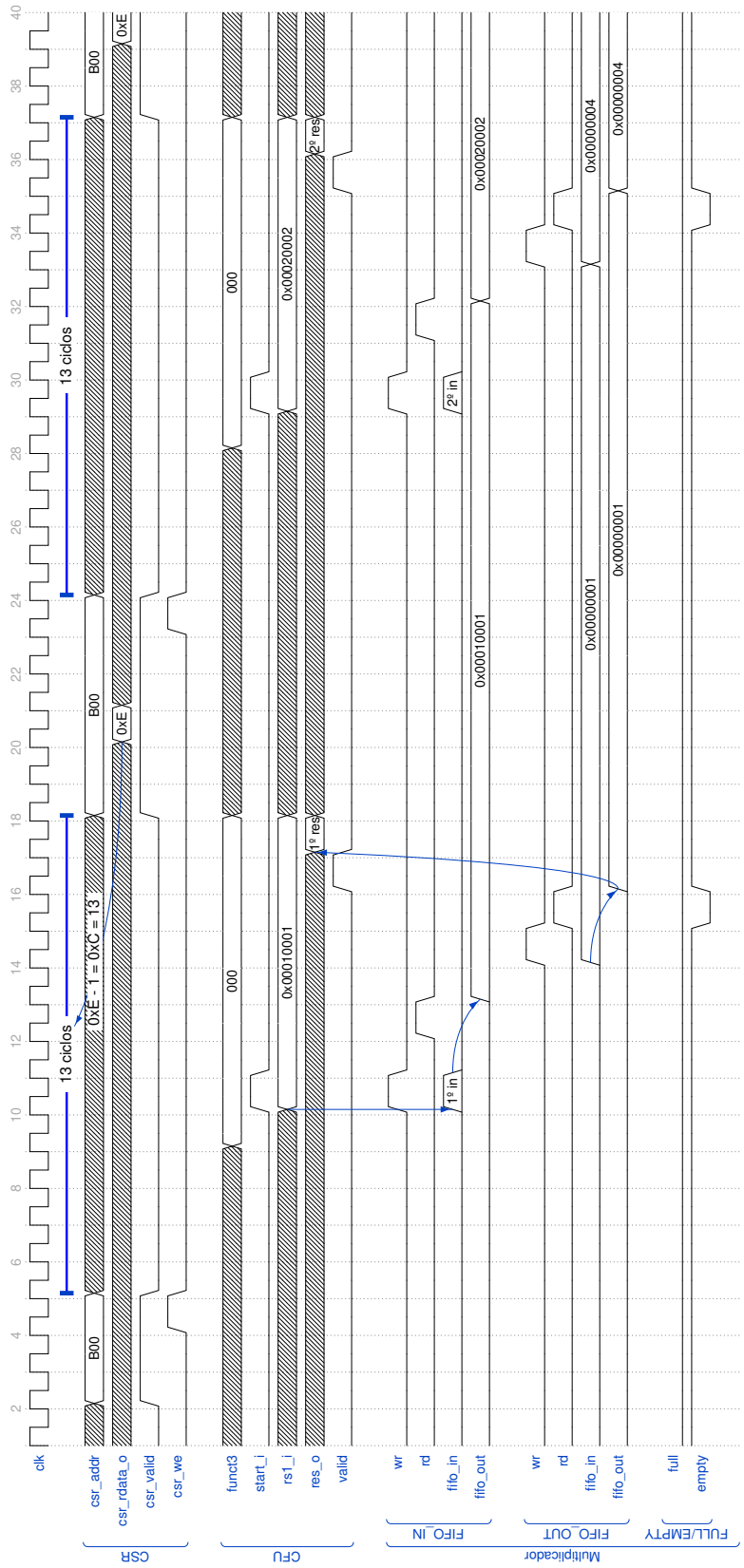


FIGURA B.2: Forma de onda resultante del ensayo de latencia para NEORV32 + Mult-B acoplado mediante CFU.

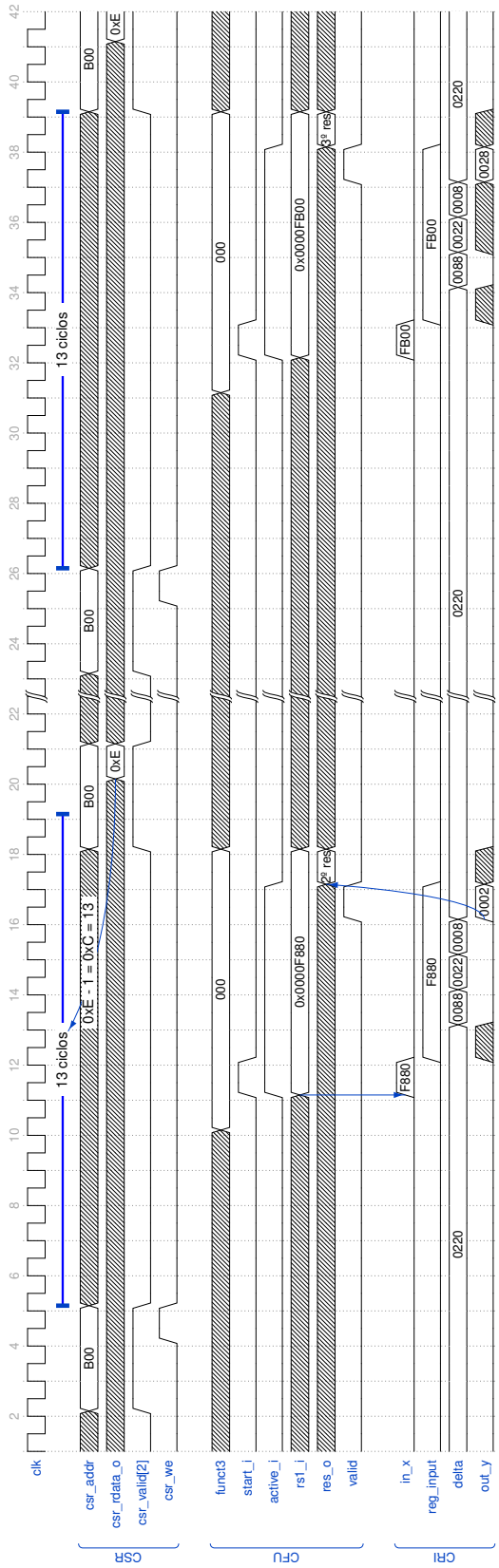


FIGURA B.3: Forma de onda resultante del cálculo de dos sigmoides mediante CRI acoplado vía CFU con el NEORV32.

Apéndice C

Resultados de la caracterización de los métodos de conexíon en simulación

```
All passed!
$ DESIGN=mult ./latency.py -v
---- For tb_mult_wfifos_axis_latency.vhd file ----
The latency is: 60000000 fs - 6 cycles
Data 1 of 4 sent/received latency is: 60000000 fs - 6 cycles
Data 2 of 4 sent/received latency is: 60000000 fs - 6 cycles
Data 3 of 4 sent/received latency is: 60000000 fs - 6 cycles
Data 4 of 4 sent/received latency is: 60000000 fs - 6 cycles
$ DESIGN=multp-wfifos ./latency.py -v
---- For tb_multp_wfifos_axis_latency.vhd file ----
The latency is: 40000000 fs - 4 cycles
Data 1 of 4 sent/received latency is: 40000000 fs - 4 cycles
Data 2 of 4 sent/received latency is: 40000000 fs - 4 cycles
Data 3 of 4 sent/received latency is: 40000000 fs - 4 cycles
Data 4 of 4 sent/received latency is: 40000000 fs - 4 cycles
$ DESIGN=multp ./latency.py -v
---- For tb_multp_axis_latency.vhd file ----
The latency is: 10000000 fs - 1 cycles
Data 1 of 4 sent/received latency is: 10000000 fs - 1 cycles
Data 2 of 4 sent/received latency is: 10000000 fs - 1 cycles
Data 3 of 4 sent/received latency is: 10000000 fs - 1 cycles
Data 4 of 4 sent/received latency is: 10000000 fs - 1 cycles
```

FIGURA C.1: Resultados del ensayo de latencia para Mult-B, Mult-BP y Mult-UBP acoplados mediante *AXI-Stream Verification Componets*.

```
    6336225000000 fs - tb_complex_mult_wfifos_slink -    INFO - Data 1/4 latency is 45 cycles (tb_complex_mult_wfifos_slink.vhd:118)
    6336735000000 fs - tb_complex_mult_wfifos_slink -    INFO - Data 2/4 latency is 45 cycles (tb_complex_mult_wfifos_slink.vhd:118)
    6337245000000 fs - tb_complex_mult_wfifos_slink -    INFO - Data 3/4 latency is 45 cycles (tb_complex_mult_wfifos_slink.vhd:118)
    6337755000000 fs - tb_complex_mult_wfifos_slink -    INFO - Data 4/4 latency is 45 cycles (tb_complex_mult_wfifos_slink.vhd:118)
    6337785000000 fs - tb_complex_mult_wfifos_slink -    INFO - Test done (tb_complex_mult_wfifos_slink.vhd:105)
simulation stopped @6337785ns with status 0
pass (P=2 S=0 F=0 T=3) neorv32.tb_complex_mult_wfifos_slink.test (3 min 19.9 s)
```

FIGURA C.2: Resultados del ensayo de latencia para NEORV32 + Mult-B, acoplado mediante SLINK.

```
791    6336225000000 fs - tb_complex_multp_wfifos_slink -    INFO - Data 1/4 latency is 45 cycles (tb_complex_multp_wfifos_slink.vhd:118)
792    6336735000000 fs - tb_complex_multp_wfifos_slink -    INFO - Data 2/4 latency is 45 cycles (tb_complex_multp_wfifos_slink.vhd:118)
793    6337245000000 fs - tb_complex_multp_wfifos_slink -    INFO - Data 3/4 latency is 45 cycles (tb_complex_multp_wfifos_slink.vhd:118)
794    6337755000000 fs - tb_complex_multp_wfifos_slink -    INFO - Data 4/4 latency is 45 cycles (tb_complex_multp_wfifos_slink.vhd:118)
795    6337785000000 fs - tb_complex_multp_wfifos_slink -    INFO - Test done (tb_complex_multp_wfifos_slink.vhd:105)
796 simulation stopped @6337785ns with status 0
797 pass (P=1 S=0 F=0 T=3) neorv32.tb_complex_multp_wfifos_slink.test (3 min 19.5 s)
```

FIGURA C.3: Resultados del ensayo de latencia para NEORV32 + Mult-BP, acoplado mediante SLINK.

```
943    6336225000000 fs - tb_complex_multp_slink -    INFO - Data 1/4 latency is 45 cycles (tb_complex_multp_slink.vhd:118)
944    6336735000000 fs - tb_complex_multp_slink -    INFO - Data 2/4 latency is 45 cycles (tb_complex_multp_slink.vhd:118)
945    6337245000000 fs - tb_complex_multp_slink -    INFO - Data 3/4 latency is 45 cycles (tb_complex_multp_slink.vhd:118)
946    6337755000000 fs - tb_complex_multp_slink -    INFO - Data 4/4 latency is 45 cycles (tb_complex_multp_slink.vhd:118)
947    6337785000000 fs - tb_complex_multp_slink -    INFO - Test done (tb_complex_multp_slink.vhd:105)
948 simulation stopped @6337785ns with status 0
949 pass (P=3 S=0 F=0 T=3) neorv32.tb_complex_multp_slink.test (3 min 12.2 s)
```

FIGURA C.4: Resultados del ensayo de latencia para NEORV32 + Mult-UBP, acoplado mediante SLINK.

```
1190 $ DESIGN=multp-wfifos ./throughput.py -v
1191 ---- For tb_multp_wfifos_axis_throughput.vhd file ----
1192 The throughput is: 1/1 data per cycle
1193 Between data 1 and 2 is: 1/1 data per cycle
1194 Between data 2 and 3 is: 1/1 data per cycle
1195 Between data 3 and 4 is: 1/1 data per cycle
1196 $ DESIGN=multp ./throughput.py -v
1197 ---- For tb_multp_axis_throughput.vhd file ----
1198 The throughput is: 1/1 data per cycle
1199 Between data 1 and 2 is: 1/1 data per cycle
1200 Between data 2 and 3 is: 1/1 data per cycle
1201 Between data 3 and 4 is: 1/1 data per cycle
1202 Cleaning up project directory and file based variables
1203 Job succeeded
```

FIGURA C.5: Resultados del ensayo de *throughput* para Mult-B, Mult-BP acoplados mediante *AXI-Stream Verification Componets*.

```
865    6337185000000 fs - tb_complex_mult_wfifos_slink -    INFO - Throughput between data 1 and data 2 is: 1/20 data per cycle (tb_complex_mult_wfifos_slink.vhd:118)
866    6337445000000 fs - tb_complex_mult_wfifos_slink -    INFO - Throughput between data 2 and data 3 is: 1/20 data per cycle (tb_complex_mult_wfifos_slink.vhd:118)
867    6337705000000 fs - tb_complex_mult_wfifos_slink -    INFO - Throughput between data 3 and data 4 is: 1/20 data per cycle (tb_complex_mult_wfifos_slink.vhd:118)
868    6337735000000 fs - tb_complex_mult_wfifos_slink -    INFO - Test done (tb_complex_mult_wfifos_slink.vhd:105)
869 simulation stopped @6337735ns with status 0
870 pass (P=2 S=0 F=0 T=3) neorv32.tb_complex_mult_wfifos_slink.test (3 min 16.9 s)
```

FIGURA C.6: Resultados del ensayo de *throughput* para NEORV32 + Mult-B, acoplado mediante SLINK.

```
791    6337185000000 fs - tb_complex_multp_wfifos_slink -    INFO - Throughput between data 1 and data 2 is: 1/20 data per cycle (tb_complex_multp_wfifos_slink.vhd:118)
792    6337445000000 fs - tb_complex_multp_wfifos_slink -    INFO - Throughput between data 2 and data 3 is: 1/20 data per cycle (tb_complex_multp_wfifos_slink.vhd:118)
793    6337705000000 fs - tb_complex_multp_wfifos_slink -    INFO - Throughput between data 3 and data 4 is: 1/20 data per cycle (tb_complex_multp_wfifos_slink.vhd:118)
794    6337735000000 fs - tb_complex_multp_wfifos_slink -    INFO - Test done (tb_complex_multp_wfifos_slink.vhd:105)
795 simulation stopped @6337735ns with status 0
796 pass (P=1 S=0 F=0 T=3) neorv32.tb_complex_multp_wfifos_slink.test (3 min 11.6 s)
```

FIGURA C.7: Resultados del ensayo de *throughput* para NEORV32 + Mult-BP, acoplado mediante SLINK.

```
941    6337185000000 fs - tb_complex_multp_slink -    INFO - Throughput between data 1 and data 2 is: 1/20 data per cycle (tb_complex_multp_slink.vhd:118)
942    6337445000000 fs - tb_complex_multp_slink -    INFO - Throughput between data 2 and data 3 is: 1/20 data per cycle (tb_complex_multp_slink.vhd:118)
943    6337705000000 fs - tb_complex_multp_slink -    INFO - Throughput between data 3 and data 4 is: 1/20 data per cycle (tb_complex_multp_slink.vhd:118)
944    6337735000000 fs - tb_complex_multp_slink -    INFO - Test done (tb_complex_multp_slink.vhd:105)
945 simulation stopped @6337735ns with status 0
946 pass (P=3 S=0 F=0 T=3) neorv32.tb_complex_multp_slink.test (3 min 8.1 s)
```

FIGURA C.8: Resultados del ensayo de *throughput* para NEORV32 + Mult-UBP, acoplado mediante SLINK.

```
1156 All passed!
1157 $ DESIGN=mult ./latency.py -v
1158 ---- For tb_mult_wfifos_wishbone_latency.vhd file ----
1159 The latency is: 50000000 fs - 5 cycles
1160 Data 1 of 4 sent/received latency is: 50000000 fs - 5 cycles
1161 Data 2 of 4 sent/received latency is: 50000000 fs - 5 cycles
1162 Data 3 of 4 sent/received latency is: 50000000 fs - 5 cycles
1163 Data 4 of 4 sent/received latency is: 50000000 fs - 5 cycles
1164 $ DESIGN=multp-wfifos ./latency.py -v
1165 ---- For tb_multp_wfifos_wishbone_latency.vhd file ----
1166 The latency is: 30000000 fs - 3 cycles
1167 Data 1 of 4 sent/received latency is: 30000000 fs - 3 cycles
1168 Data 2 of 4 sent/received latency is: 30000000 fs - 3 cycles
1169 Data 3 of 4 sent/received latency is: 30000000 fs - 3 cycles
1170 Data 4 of 4 sent/received latency is: 30000000 fs - 3 cycles
1171 $ DESIGN=multp ./latency.py -v
1172 ---- For tb_multp_wishbone_latency.vhd file ----
1173 The latency is: 20000000 fs - 2 cycles
1174 Data 1 of 4 sent/received latency is: 20000000 fs - 2 cycles
1175 Data 2 of 4 sent/received latency is: 20000000 fs - 2 cycles
1176 Data 3 of 4 sent/received latency is: 20000000 fs - 2 cycles
1177 Data 4 of 4 sent/received latency is: 20000000 fs - 2 cycles
```

FIGURA C.9: Resultados del ensayo de latencia para Mult-B, Mult-BP y Mult-UBP acoplados mediante *Wishbone Verification Componets*.

```
791    6334865000000 fs - tb_complex_mult_wfifos_wishbone -    INFO - Data 1/4 latency is 16 cycles (tb_complex_mult_wfifos_wishbone.vhd:118)
792    6335065000000 fs - tb_complex_mult_wfifos_wishbone -    INFO - Data 2/4 latency is 14 cycles (tb_complex_mult_wfifos_wishbone.vhd:118)
793    6335265000000 fs - tb_complex_mult_wfifos_wishbone -    INFO - Data 3/4 latency is 14 cycles (tb_complex_mult_wfifos_wishbone.vhd:118)
794    6335465000000 fs - tb_complex_mult_wfifos_wishbone -    INFO - Data 4/4 latency is 14 cycles (tb_complex_mult_wfifos_wishbone.vhd:118)
795    6335495000000 fs - tb_complex_mult_wfifos_wishbone -    INFO - Test done (tb_complex_mult_wfifos_wishbone.vhd:105)
796 simulation stopped @6335495ns with status 0
797 pass (P=1 S=0 F=0 T=3) neorv32.tb_complex_mult_wfifos_wishbone.test (3 min 29.3 s)
```

FIGURA C.10: Resultados del ensayo de latencia para NEORV32 + Mult-B, acoplado mediante XBUS.

```
866    6334865000000 fs - tb_complex_multp_wfifos_wishbone -    INFO - Data 1/4 latency is 16 cycles (tb_complex_multp_wfifos_wishbone.vhd:119)
867    6335065000000 fs - tb_complex_multp_wfifos_wishbone -    INFO - Data 2/4 latency is 14 cycles (tb_complex_multp_wfifos_wishbone.vhd:119)
868    6335265000000 fs - tb_complex_multp_wfifos_wishbone -    INFO - Data 3/4 latency is 14 cycles (tb_complex_multp_wfifos_wishbone.vhd:119)
869    6335465000000 fs - tb_complex_multp_wfifos_wishbone -    INFO - Data 4/4 latency is 14 cycles (tb_complex_multp_wfifos_wishbone.vhd:119)
870    6335495000000 fs - tb_complex_multp_wfifos_wishbone -    INFO - Test done (tb_complex_multp_wfifos_wishbone.vhd:106)
871 simulation stopped @6335495ns with status 0
872 pass (P=2 S=0 F=0 T=3) neorv32.tb_complex_multp_wfifos_wishbone.test (3 min 21.7 s)
```

FIGURA C.11: Resultados del ensayo de latencia para NEORV32 + Mult-BP, acoplado mediante XBUS.

```
941     6334865000000 fs - tb_complex_multp_wishbone -    INFO - Data 1/4 latency is 16 cycles (tb_complex_multp_wishbone.vhd:118)
942     6335065000000 fs - tb_complex_multp_wishbone -    INFO - Data 2/4 latency is 14 cycles (tb_complex_multp_wishbone.vhd:118)
943     6335265000000 fs - tb_complex_multp_wishbone -    INFO - Data 3/4 latency is 14 cycles (tb_complex_multp_wishbone.vhd:118)
944     6335465000000 fs - tb_complex_multp_wishbone -    INFO - Data 4/4 latency is 14 cycles (tb_complex_multp_wishbone.vhd:118)
945     6335495000000 fs - tb_complex_multp_wishbone -    INFO - Test done (tb_complex_multp_wishbone.vhd:105)
946  simulation stopped @6335495ns with status 0
947  pass (P=3 S=0 F=0 T=3) neorv32.tb_complex_multp_wishbone.test (3 min 16.4 s)
```

FIGURA C.12: Resultados del ensayo de latencia para NEORV32 + Mult-UBP, acoplado mediante XBUS.

```
1178  $ DESIGN=mult ./throughput.py -v
1179  ---- For tb_mult_wfifos_wishbone_throughput.vhd file ----
1180  The throughput is: 1/2 data per cycle
1181  Between data 1 and 2 is: 1/2 data per cycle
1182  Between data 2 and 3 is: 1/2 data per cycle
1183  Between data 3 and 4 is: 1/3 data per cycle
1184  $ DESIGN=multp-wfifos ./throughput.py -v
1185  ---- For tb_multp_wfifos_wishbone_throughput.vhd file ----
1186  The throughput is: 1/2 data per cycle
1187  Between data 1 and 2 is: 1/2 data per cycle
1188  Between data 2 and 3 is: 1/2 data per cycle
1189  Between data 3 and 4 is: 1/2 data per cycle
1190  Cleaning up project directory and file based variables
1191  Job succeeded
```

FIGURA C.13: Resultados del ensayo de *throughput* para Mult-B, Mult-BP acoplados mediante *Wishbone Verification Componets*.

```
789     6335195000000 fs - tb_complex_mult_wfifos_wishbone -    INFO - Throughput between data 1 and data 2 is: 1/5 data per cycle (tb_complex_mult_wfifos_wishbon
    e.vhd:118)
790     6335305000000 fs - tb_complex_mult_wfifos_wishbone -    INFO - Throughput between data 2 and data 3 is: 1/5 data per cycle (tb_complex_mult_wfifos_wishbon
    e.vhd:118)
791     6335415000000 fs - tb_complex_mult_wfifos_wishbone -    INFO - Throughput between data 3 and data 4 is: 1/5 data per cycle (tb_complex_mult_wfifos_wishbon
    e.vhd:118)
792     6335445000000 fs - tb_complex_mult_wfifos_wishbone -    INFO - Test done (tb_complex_mult_wfifos_wishbone.vhd:105)
793  simulation stopped @6335445ns with status 0
794  pass (P=1 S=0 F=0 T=2) neorv32.tb_complex_mult_wfifos_wishbone.test (3 min 26.9 s)
```

FIGURA C.14: Resultados del ensayo de *throughput* para NEORV32 + Mult-B, acoplado mediante XBUS.

```
863     6335195000000 fs - tb_complex_multp_wfifos_wishbone -    INFO - Throughput between data 1 and data 2 is: 1/5 data per cycle (tb_complex_multp_wfifos_wishb
    one.vhd:118)
864     6335305000000 fs - tb_complex_multp_wfifos_wishbone -    INFO - Throughput between data 2 and data 3 is: 1/5 data per cycle (tb_complex_multp_wfifos_wishb
    one.vhd:118)
865     6335415000000 fs - tb_complex_multp_wfifos_wishbone -    INFO - Throughput between data 3 and data 4 is: 1/5 data per cycle (tb_complex_multp_wfifos_wishb
    one.vhd:118)
866     6335445000000 fs - tb_complex_multp_wfifos_wishbone -    INFO - Test done (tb_complex_multp_wfifos_wishbone.vhd:105)
867  simulation stopped @6335445ns with status 0
868  pass (P=2 S=0 F=0 T=2) neorv32.tb_complex_multp_wfifos_wishbone.test (3 min 19.7 s)
```

FIGURA C.15: Resultados del ensayo de *throughput* para NEORV32 + Mult-BP, acoplado mediante XBUS.

```
787     6908085000000 fs - tb_complex_mults_cfu -    INFO - Data 1/4 latency is 13 cycles (tb_complex_mults_cfu.vhd:146)
788     6908275000000 fs - tb_complex_mults_cfu -    INFO - Data 2/4 latency is 13 cycles (tb_complex_mults_cfu.vhd:146)
789     6908465000000 fs - tb_complex_mults_cfu -    INFO - Data 3/4 latency is 13 cycles (tb_complex_mults_cfu.vhd:146)
790     6908655000000 fs - tb_complex_mults_cfu -    INFO - Data 4/4 latency is 13 cycles (tb_complex_mults_cfu.vhd:146)
791     6908685000000 fs - tb_complex_mults_cfu -    INFO - Test done (tb_complex_mults_cfu.vhd:133)
792  simulation stopped @6908685ns with status 0
793  pass (P=1 S=0 F=0 T=1) neorv32.tb_complex_mults_cfu.test (3 min 37.8 s)
```

FIGURA C.16: Resultados del ensayo de latencia para NEORV32 + Mult-B, acoplado mediante CFU.

```
788     7481015000000 fs - tb_complex_mults_cfu -    INFO - Data 1/4 latency is 11 cycles (tb_complex_mults_cfu.vhd:146)
789     7481185000000 fs - tb_complex_mults_cfu -    INFO - Data 2/4 latency is 11 cycles (tb_complex_mults_cfu.vhd:146)
790     7481355000000 fs - tb_complex_mults_cfu -    INFO - Data 3/4 latency is 11 cycles (tb_complex_mults_cfu.vhd:146)
791     7481525000000 fs - tb_complex_mults_cfu -    INFO - Data 4/4 latency is 11 cycles (tb_complex_mults_cfu.vhd:146)
792     7481555000000 fs - tb_complex_mults_cfu -    INFO - Test done (tb_complex_mults_cfu.vhd:133)
793  simulation stopped @7481555ns with status 0
794  pass (P=1 S=0 F=0 T=1) neorv32.tb_complex_mults_cfu.test (3 min 58.6 s)
```

FIGURA C.17: Resultados del ensayo de latencia para NEORV32 + Mult-BP, acoplado mediante CFU.

```
787     6908035000000 fs - tb_complex_mults_cfu -    INFO - Data 1/4 latency is 8 cycles (tb_complex_mults_cfu.vhd:146)
788     6908175000000 fs - tb_complex_mults_cfu -    INFO - Data 2/4 latency is 8 cycles (tb_complex_mults_cfu.vhd:146)
789     6908315000000 fs - tb_complex_mults_cfu -    INFO - Data 3/4 latency is 8 cycles (tb_complex_mults_cfu.vhd:146)
790     6908455000000 fs - tb_complex_mults_cfu -    INFO - Data 4/4 latency is 8 cycles (tb_complex_mults_cfu.vhd:146)
791     6908485000000 fs - tb_complex_mults_cfu -    INFO - Test done (tb_complex_mults_cfu.vhd:133)
792  simulation stopped @6908485ns with status 0
793  pass (P=1 S=0 F=0 T=1) neorv32.tb_complex_mults_cfu.test (3 min 40.5 s)
```

FIGURA C.18: Resultados del ensayo de latencia para NEORV32 + Mult-UBP, acoplado mediante CFU.

```
870     7481495000000 fs - tb_complex_mult_wfifos_cfs -    INFO - Data 1/4 latency is 37 cycles (tb_complex_mult_wfifos_cfs.vhd:128)
871     7481845000000 fs - tb_complex_mult_wfifos_cfs -    INFO - Data 2/4 latency is 29 cycles (tb_complex_mult_wfifos_cfs.vhd:128)
872     7482195000000 fs - tb_complex_mult_wfifos_cfs -    INFO - Data 3/4 latency is 29 cycles (tb_complex_mult_wfifos_cfs.vhd:128)
873     7482545000000 fs - tb_complex_mult_wfifos_cfs -    INFO - Data 4/4 latency is 29 cycles (tb_complex_mult_wfifos_cfs.vhd:128)
874     7482575000000 fs - tb_complex_mult_wfifos_cfs -    INFO - Test done (tb_complex_mult_wfifos_cfs.vhd:115)
875  simulation stopped @7482575ns with status 0
876  pass (P=2 S=0 F=0 T=2) neorv32.tb_complex_mult_wfifos_cfs.test (3 min 54.3 s)
```

FIGURA C.19: Resultados del ensayo de latencia para NEORV32 + Mult-B, acoplado mediante CFS.

```
793     7481495000000 fs - tb_complex_multp_wfifos_cfs -    INFO - Data 1/4 latency is 37 cycles (tb_complex_multp_wfifos_cfs.vhd:128)
794     7481845000000 fs - tb_complex_multp_wfifos_cfs -    INFO - Data 2/4 latency is 29 cycles (tb_complex_multp_wfifos_cfs.vhd:128)
795     7482195000000 fs - tb_complex_multp_wfifos_cfs -    INFO - Data 3/4 latency is 29 cycles (tb_complex_multp_wfifos_cfs.vhd:128)
796     7482545000000 fs - tb_complex_multp_wfifos_cfs -    INFO - Data 4/4 latency is 29 cycles (tb_complex_multp_wfifos_cfs.vhd:128)
797     7482575000000 fs - tb_complex_multp_wfifos_cfs -    INFO - Test done (tb_complex_multp_wfifos_cfs.vhd:115)
798  simulation stopped @7482575ns with status 0
799  pass (P=1 S=0 F=0 T=2) neorv32.tb_complex_multp_wfifos_cfs.test (3 min 41.4 s)
```

FIGURA C.20: Resultados del ensayo de latencia para NEORV32 + Mult-BP, acoplado mediante CFS.

```
792     7481305000000 fs - tb_complex_multp_cfs -    INFO - Data 1/4 latency is 18 cycles (tb_complex_multp_cfs.vhd:128)
793     7481505000000 fs - tb_complex_multp_cfs -    INFO - Data 2/4 latency is 14 cycles (tb_complex_multp_cfs.vhd:128)
794     7481705000000 fs - tb_complex_multp_cfs -    INFO - Data 3/4 latency is 14 cycles (tb_complex_multp_cfs.vhd:128)
795     7481905000000 fs - tb_complex_multp_cfs -    INFO - Data 4/4 latency is 14 cycles (tb_complex_multp_cfs.vhd:128)
796     7481935000000 fs - tb_complex_multp_cfs -    INFO - Test done (tb_complex_multp_cfs.vhd:115)
797  simulation stopped @7481935ns with status 0
798  pass (P=1 S=0 F=0 T=1) neorv32.tb_complex_multp_cfs.test (3 min 39.4 s)
```

FIGURA C.21: Resultados del ensayo de latencia para NEORV32 + Mult-UBP, acoplado mediante CFS.

```
867     7482275000000 fs - tb_complex_mult_wfifos_cfs -    INFO - Throughput between data 1 and data 2 is: 1/15 data per cycle (tb_complex_mult_wfifos_cfs.vhd:12
    8)
868     7482485000000 fs - tb_complex_mult_wfifos_cfs -    INFO - Throughput between data 2 and data 3 is: 1/15 data per cycle (tb_complex_mult_wfifos_cfs.vhd:12
    8)
869     7482695000000 fs - tb_complex_mult_wfifos_cfs -    INFO - Throughput between data 3 and data 4 is: 1/15 data per cycle (tb_complex_mult_wfifos_cfs.vhd:12
    8)
870     7482725000000 fs - tb_complex_mult_wfifos_cfs -    INFO - Test done (tb_complex_mult_wfifos_cfs.vhd:115)
871  simulation stopped @7482725ns with status 0
872  pass (P=2 S=0 F=0 T=2) neorv32.tb_complex_mult_wfifos_cfs.test (3 min 55.4 s)
```

FIGURA C.22: Resultados del ensayo de *throughput* para NEORV32 + Mult-B, acoplado mediante CFS.

```
791    7482275000000 fs - tb_complex_multp_wfifos_cfs -    INFO - Throughput between data 1 and data 2 is: 1/15 data per cycle (tb_complex_multp_wfifos_cfs.vhd:1
   28)
792    7482485000000 fs - tb_complex_multp_wfifos_cfs -    INFO - Throughput between data 2 and data 3 is: 1/15 data per cycle (tb_complex_multp_wfifos_cfs.vhd:1
   28)
793    7482695000000 fs - tb_complex_multp_wfifos_cfs -    INFO - Throughput between data 3 and data 4 is: 1/15 data per cycle (tb_complex_multp_wfifos_cfs.vhd:1
   28)
794    7482725000000 fs - tb_complex_multp_wfifos_cfs -    INFO - Test done (tb_complex_multp_wfifos_cfs.vhd:115)
795  simulation stopped @7482725ns with status 0
796  pass (P=1 S=0 F=0 T=2) neorv32.tb_complex_multp_wfifos_cfs.test (3 min 47.8 s)
```

FIGURA C.23: Resultados del ensayo de *throughput* para NEORV32 + Mult-BP, acoplado mediante CFS.

Apéndice D

Código

El código empleado a lo largo de este trabajo es bastante extenso. Comprende desde la descripción hardware en VHDL de los diseños y los *test benches*, pasando por la codificación de software en C, archivos en Python para lanzar las simulaciones de VUnit, así como para gestionar sus archivos de salida en formato CSV, además de archivos YML para gestionar la integración continua, archivos Bash para gestionar la generación de *bitstream* mediante herramientas FLOS y archivos TCL para gestionar la generación de *bitstream* mediante Vivado. El total del código empleado está en el repositorio de GitLab del grupo de investigación y gran parte de él, sobre todo el referente a la sección 2.3, está en el siguiente repositorio público de GitHub [64]. Este apéndice comprende parte del código desarrollado como ejemplificación del trabajo realizado a lo largo de este proyecto de investigación.

```
1  -- RTL of MULT
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5  use ieee.numeric_std.all;
6
7  entity mult is
8  generic (
9          -- Number of bits that the input/output data has.
10         N_bits : in natural
11         );
12 port (
13         -- Clock signal
14         clk : in std_logic;
15         -- Mult in/out signals
16         mult_in : in std_logic_vector (N_bits-1 downto 0);
17         mult_out : out std_logic_vector(N_bits-1 downto 0)
18     );
19 end mult;
20
21 architecture rtl of mult is
22
23 -- Declaration of signals
24
25 signal in_1 : unsigned ((N_bits/2)-1 downto 0) := (others => '0');
26 signal in_2 : unsigned ((N_bits/2)-1 downto 0) := (others => '0');
27
28 begin
```

```
29
30      -- Assign inputs to the multipler
31
32      in_1 <= unsigned(mult_in(N_bits-1 downto N_bits/2));
33      in_2 <= unsigned(mult_in((N_bits/2)-1 downto 0));
34
35      -- Make multiplication and assign output
36
37      mult_make : process ( clk )
38          begin
39              if( rising_edge (clk) ) then
40                  mult_out <= std_logic_vector(in_1 * in_2);
41              end if;
42      end process mult_make;
43
44  end rtl;
```

CÓDIGO D.1: Mult.vhd

```
1   -- RTL of FIFO
2
3   library ieee;
4   use ieee.std_logic_1164.all;
5   use ieee.numeric_std.all;
6
7   entity fifo is
8   generic (
9           -- Number of bits per element
10          N_bits : in natural;
11          -- Log2 of number of elements that the FIFO has; Number of FIFO elements has
            ↪  to be a power of two.
12          Log2_elements : in natural
13          );
14  port (
15          -- Fifo clocks/reset signals
16          clk_wr : in std_logic;
17          clk_rd : in std_logic;
18          rst : in std_logic;
19          -- Fifo in/out signals
20          fifo_in : in std_logic_vector (N_bits-1 downto 0);
21          fifo_out : out std_logic_vector(N_bits-1 downto 0);
22          -- Fifo write/read signals
23          wr : in std_logic;
24          rd : in std_logic;
25          -- Fifo status signals
26          full_o : out std_logic;
27          empty_o : out std_logic
28      );
29  end fifo;
30
31  architecture rtl of fifo is
32
```

```
33  -- Declariation of fifo array
34
35  type array_type is array ((2**Log2_elements)-1 downto 0) of std_logic_vector(N_bits-1
    ↪  downto 0);
36  signal fifo_array : array_type := (others => (others => '0'));
37
38  -- Declaration of signals
39
40  signal wr_pnt : std_logic_vector(Log2_elements downto 0) := (others => '0');
41  signal rd_pnt : std_logic_vector(Log2_elements downto 0) := (others => '0');
42  signal full : std_logic := '0';
43  signal empty : std_logic := '0';
44
45
46  begin
47
48      -- Signals empty and full logic
49
50      full_logic : process ( wr_pnt,rd_pnt )
51      begin
52          if( (wr_pnt(Log2_elements-1 downto 0) = rd_pnt(Log2_elements-1 downto 0)) and
            ↪  ( (wr_pnt(wr_pnt'left) xor rd_pnt(rd_pnt'left)) = '1') ) then
53              full <= '1';
54          else
55              full <= '0';
56          end if;
57      end process full_logic;
58
59      empty_logic : process ( wr_pnt,rd_pnt )
60      begin
61          if ( wr_pnt = rd_pnt ) then
62              empty <= '1';
63          else
64              empty <= '0';
65          end if;
66      end process empty_logic;
67
68      -- Assign control signals output
69
70      full_o <= full;
71      empty_o <= empty;
72
73      -- Write and read process
74
75      write_process: process (clk_wr)
76      begin
77          if( rising_edge( clk_wr ) ) then
78              if( rst = '1' ) then
79                  wr_pnt <= (others => '0');
80              elsif( wr = '1' and full = '0' ) then
81                  fifo_array(to_integer(unsigned(wr_pnt(Log2_elements-1 downto 0)))) <=
                    ↪  fifo_in;
```

```
82                  wr_pnt <= std_logic_vector(unsigned(wr_pnt) + 1);
83              end if;
84          end if;
85      end process write_process;
86
87      read_process: process (clk_rd)
88      begin
89          if( rising_edge( clk_rd ) ) then
90              if( rst = '1' ) then
91                  rd_pnt <= (others => '0');
92                  fifo_out <= (others => '0');
93              elsif( rd = '1' and empty = '0' ) then
94                  fifo_out <= fifo_array(to_integer(unsigned(rd_pnt(Log2_elements-1
                    ↪  downto 0))));
95                  rd_pnt <= std_logic_vector(unsigned(rd_pnt) + 1);
96              end if;
97          end if;
98      end process read_process;
99
100 end rtl;
```

CÓDIGO D.2: fifo.vhd

```
1   -- RTL of Mult_wfifos
2
3   library ieee;
4   use ieee.std_logic_1164.all;
5   use ieee.numeric_std.all;
6
7   entity mult_wfifos is
8   generic (
9           -- Number of bits that the input/output data has
10          N_bits : in natural;
11          -- Log2 of number of elements that the FIFOs have; Both the same size; Number
            ↪  of FIFO elements has to be a power of two
12          Log2_elements : in natural
13          );
14  port (
15          -- Mult_wfifos clocks and reset signals
16          clk_wr : in std_logic;
17          clk_mult : in std_logic;
18          clk_rd : in std_logic;
19          rst : in std_logic;
20          -- Mult_wfifos input/output data
21          din : in std_logic_vector (N_bits-1 downto 0);
22          dout : out std_logic_vector (N_bits-1 downto 0);
23          -- Mult_wfifos write and read signals;
24          wr : in std_logic;
25          rd : in std_logic;
26          -- Mult_wfifos status signals
27          full : out std_logic;
28          empty : out std_logic
```

```
29  );
30  end mult_wfifos;
31
32  architecture rtl of mult_wfifos is
33
34  -- Declaration of signals
35  signal in_pre_mult : std_logic_vector(N_bits-1 downto 0) := (others => '0');
36  signal in_post_mult : std_logic_vector(N_bits-1 downto 0) := (others => '0');
37  signal rd_inter : std_logic := '0';
38  signal wr_inter : std_logic := '0';
39  signal empty_inter : std_logic := '0';
40  signal full_inter : std_logic := '0';
41
42  type t_states is (CHECK,READ,MULTI,WRITE);
43  signal state : t_states := CHECK;
44  signal next_state : t_states;
45
46
47  begin
48
49  -- Fifo IN instantiation
50
51  fifo_IN : entity work.fifo
52                  generic map (N_bits => N_bits,
53                               Log2_elements => Log2_elements)
54                  port map (clk_wr => clk_wr,
55                            clk_rd => clk_mult,
56                            rst => rst,
57                            fifo_in => din,
58                            fifo_out => in_pre_mult,
59                            wr => wr,
60                            rd => rd_inter,
61                            full_o => full,
62                            empty_o => empty_inter);
63
64  -- mult instantiation
65
66  mult_0 : entity  work.mult
67                  generic map (N_bits => N_bits)
68                  port map (clk => clk_mult,
69                            mult_in => in_pre_mult,
70                            mult_out => in_post_mult);
71
72  -- Fifo OUT instantiation
73
74  fifo_OUT : entity work.fifo
75                  generic map (N_bits => N_bits,
76                               Log2_elements => Log2_elements)
77                  port map (clk_wr => clk_mult,
78                            clk_rd => clk_rd,
79                            rst => rst,
80                            fifo_in => in_post_mult,
```

```
81                          fifo_out => dout,
82                          wr => wr_inter,
83                          rd => rd,
84                          full_o => full_inter,
85                          empty_o => empty);
86
87  -- State machine
88      -- Combinational
89
90      Combinational_of_state_machine : process (state, empty_inter, full_inter)
91      begin
92          next_state <= state;
93          case state is
94              when CHECK =>
95                  if (empty_inter = '0' and full_inter = '0') then
96                      next_state <= READ;
97                  else
98                      next_state <= CHECK;
99                  end if;
100             when READ =>
101                 next_state <= MULTI;
102             when MULTI =>
103                 next_state <= WRITE;
104             when WRITE =>
105                 next_state <= CHECK;
106             when others =>
107                 next_state <= CHECK;
108         end case;
109     end process Combinational_of_state_machine;
110
111     -- Outputs
112
113     with state select
114         wr_inter <= '1' when WRITE,
115                     '0' when others;
116     with state select
117         rd_inter <= '1' when READ,
118                     '0' when others;
119
120     -- Sequential
121
122     state_machine_state_reg : process ( clk_mult )
123     begin
124         if( rising_edge(clk_mult) ) then
125             if( rst = '1' ) then
126                 state <= CHECK;
127             else
128                 state <= next_state;
129             end if;
130         end if;
131     end process state_machine_state_reg;
132
```

```
133 end rtl;
```

CÓDIGO D.3: Mult_wfifos.vhd

```
1  -- RTL of mult_axis
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5  use ieee.numeric_std.all;
6
7  entity mult_wfifos_axis is
8  generic (
9          -- Number of bits that the input/output data has
10         N_bits : in natural;
11         -- Log2 of number of elements that the FIFOs have; Both the same size; Number
           ↪  of FIFO elements has to be a power of two
12         Log2_elements : in natural
13         );
14 port (
15     -- Clk mult
16     clk_mult : in std_logic;
17
18     -- Slave signals
19     s_axis_clk   : in  std_logic;
20     s_axis_rstn  : in  std_logic;
21     s_axis_rdy   : out std_logic;
22     s_axis_data  : in  std_logic_vector(N_bits-1 downto 0);
23     s_axis_valid : in  std_logic;
24
25     -- Master signals
26     m_axis_clk   : in  std_logic;
27     m_axis_rstn  : in  std_logic;
28     m_axis_valid : out std_logic;
29     m_axis_data  : out std_logic_vector(N_bits-1 downto 0);
30     m_axis_rdy   : in  std_logic
31     );
32 end mult_wfifos_axis;
33
34 architecture rtl of mult_wfifos_axis is
35
36 signal reset, write, read, valid, empty, full : std_logic;
37
38 begin
39
40 -- Mult_wfifos instantation
41
42 mult_wfifos_0 : entity work.mult_wfifos
43                 generic map (N_bits => N_bits,
44                              Log2_elements => Log2_elements)
45                 port map (clk_wr => s_axis_clk,
46                           clk_mult => clk_mult,
47                           clk_rd => m_axis_clk,
```

```
48                          rst => reset,
49                          din => s_axis_data,
50                          dout => m_axis_data,
51                          wr => write,
52                          rd => read,
53                          full => full,
54                          empty => empty);
55
56  -- Reset (NEORV32 rst is low-active)
57
58  reset <= (s_axis_rstn nand m_axis_rstn);
59
60  -- Write and read signals
61
62  write <= s_axis_valid and not(full);
63
64  read <= not(empty) and (valid nand not(m_axis_rdy));
65
66  -- Make valid signal
67
68  make_valid : process(m_axis_clk) begin
69      if rising_edge(m_axis_clk) then
70          if (((not m_axis_rstn) or ((valid and empty) and m_axis_rdy)) = '1') then
71              valid <= '0';
72          elsif (read = '1') then
73              valid <= '1';
74          end if;
75      end if;
76      end process make_valid;
77
78  -- Assing axi signals
79
80  s_axis_rdy <= not(full);
81  m_axis_valid <= valid;
82
83  end rtl;
```

CÓDIGO D.4: mult_wfifos_axis.vhd

```
1   -- RTL of mult_wfifos_wishbone
2
3   library ieee;
4   use ieee.std_logic_1164.all;
5   use ieee.numeric_std.all;
6
7   entity mult_wfifos_wishbone is
8   generic (
9           -- Number of bits that the input/output data has
10          N_bits : in natural;
11          -- Log2 of number of elements that the FIFOs have; Both the same size; Number
            ↪  of FIFO elements has to be a power of two
12          Log2_elements : in natural
```

```
13          );
14  port (
15      rst_i : in std_logic;
16      clk_i : in std_logic;
17      adr_i : in std_logic_vector(31 downto 0);
18      dat_i : in std_logic_vector(31 downto 0);
19      dat_o : out std_logic_vector(31 downto 0);
20      we_i : in std_logic;
21      sel_i : in std_logic_vector(3 downto 0);
22      stb_i : in std_logic;
23      ack_o : out std_logic;
24      cyc_i : in std_logic;
25      err_o : out std_logic;
26      stall_o : out std_logic
27      );
28  end mult_wfifos_wishbone;
29
30  architecture rtl of mult_wfifos_wishbone is
31
32  signal reset, write, read, empty, full : std_logic;
33  signal ack : std_logic;
34  signal stall : std_logic;
35  signal input : std_logic_vector(31 downto 0);
36  signal output : std_logic_vector(31 downto 0);
37  signal transfer_in : std_logic;
38  signal transfer_out : std_logic;
39  signal output_window : std_logic := '0';
40
41  begin
42
43  -- Mult_wfifos instantation
44
45  mult_wfifos_0 : entity work.mult_wfifos
46                  generic map (N_bits => N_bits,
47                               Log2_elements => Log2_elements)
48                  port map (clk_wr => clk_i,
49                            clk_mult => clk_i,
50                            clk_rd => clk_i,
51                            rst => reset,
52                            din => input,
53                            dout => output,
54                            wr => write,
55                            rd => read,
56                            full => full,
57                            empty => empty);
58
59  -- Reset (NEORV32 rst is low-active)
60
61  reset <= not rst_i;
62
63  -- Make error signal
64
```

```
65  err_o <= '0'; --tie to zero if not explicitly used
66
67  -- Make stall signal
68
69  with we_i select
70       stall <= full  when '1',
71                 empty when others;
72
73  stall_o <= stall;
74
75  -- Make transfer in/out signals
76
77  transfer_in  <= (stb_i and cyc_i and we_i and not(stall)) when adr_i = x"90000000"
    ↪  else  -- The address is 0x90000000; See main.c in sw/EMEM
78                  '0';
79
80  transfer_out <= (stb_i and cyc_i and not(we_i) and not(stall)) when adr_i =
    ↪  x"90000000" else
81                  '0';
82
83  -- Manage input/output and write/read signals
84
85  with transfer_in select
86       input <= dat_i when '1',
87                 (others => '0') when others;
88
89  with transfer_in select
90       write <= '1' when '1',
91                 '0' when others;
92
93  with transfer_out select
94       read  <= '1' when '1',
95                 '0' when others;
96
97  with output_window select
98       dat_o <= output when '1',
99                 (others => '0') when others;
100
101 -- Manage output_window
102
103 process (clk_i) begin
104     if rising_edge(clk_i) then
105       if reset = '1' then
106         output_window <= '0';
107       elsif transfer_out = '1' then
108         output_window <= '1';
109       else
110         output_window <= '0';
111       end if;
112     end if;
113 end process;
114
```

```
115 -- Manage ack signal
116
117 process (clk_i) begin
118     if rising_edge(clk_i) then
119       if reset = '1' then
120         ack <= '0';
121       else
122         if transfer_in or transfer_out then
123           ack <= '1';
124         else
125           ack <= '0';
126         end if;
127       end if;
128     end if;
129   end process;
130
131 ack_o <= ack;
132
133 end rtl;
```

CÓDIGO D.5: mult_wfifos_wishbone.vhd

```
1  -- Authors:
2  --    Unai Martinez-Corral & Unai Sainz-Estebanez
3  --      <unai.martinezcorral@ehu.eus>
4  --      <usainz003@ikasle.ehu.eus>
5  --
6  -- Licensed under the Apache License, Version 2.0 (the "License");
7  -- you may not use this file except in compliance with the License.
8  -- You may obtain a copy of the License at
9  --
10 --      http://www.apache.org/licenses/LICENSE-2.0
11 --
12 -- Unless required by applicable law or agreed to in writing, software
13 -- distributed under the License is distributed on an "AS IS" BASIS,
14 -- WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15 -- See the License for the specific language governing permissions and
16 -- limitations under the License.
17 --
18 -- SPDX-License-Identifier: Apache-2.0
19
20 library ieee;
21 context ieee.ieee_std_context;
22
23 entity multp_op is
24   generic (
25     g_data_width : natural
26   );
27   port (
28     DIN : in std_logic_vector (g_data_width-1 downto 0);
29     DOUT : out std_logic_vector(g_data_width-1 downto 0)
30   );
```

```
31 end multp_op;
32
33 architecture arch of multp_op is
34 begin
35
36   DOUT <= std_logic_vector(
37     signed(DIN(g_data_width-1 downto g_data_width/2))
38     *
39     signed(DIN((g_data_width/2)-1 downto 0))
40   );
41
42 end arch;
43
44
45 library ieee;
46 context ieee.ieee_std_context;
47
48 entity multp is
49   generic (
50     g_data_width : natural
51   );
52   port (
53     CLK : in std_logic;
54     RST : in std_logic;
55     IN_VALID : in std_logic;
56     IN_READY : out std_logic;
57     DIN : in std_logic_vector (g_data_width-1 downto 0);
58     OUT_VALID : out std_logic;
59     OUT_READY : in std_logic;
60     DOUT : out std_logic_vector(g_data_width-1 downto 0)
61   );
62 end multp;
63
64 architecture registered of multp is
65
66   signal ready: std_logic;
67   signal valid : std_logic;
68   signal transfer_in : std_logic;
69   signal transfer_out : std_logic;
70   signal result : std_logic_vector(g_data_width-1 downto 0);
71
72 begin
73
74   transfer_in <= IN_VALID and ready;
75   transfer_out <= valid and OUT_READY;
76   ready <= not rst and ((not valid) or transfer_out);
77
78   IN_READY <= ready;
79   OUT_VALID <= valid;
80
81   i_multp_op : entity work.multp_op
82     generic map (
```

```
83        g_data_width => g_data_width
84      )
85      port map (
86        DIN  => DIN,
87        DOUT => result
88      );
89
90    process (CLK) begin
91      if rising_edge(CLK) then
92        if RST then
93          DOUT <= (others=>'0');
94        elsif transfer_in then
95          DOUT <= result;
96        end if;
97      end if;
98    end process;
99
100   process (CLK) begin
101     if rising_edge(CLK) then
102       if RST then
103         valid <= '0';
104       else
105         if transfer_in then
106           valid <= '1';
107         elsif transfer_out then
108           valid <= '0';
109         end if;
110       end if;
111     end if;
112   end process;
113
114 end registered;
115
116 architecture combinatorial of multp is
117
118 begin
119
120   IN_READY <= OUT_READY;
121   OUT_VALID <= IN_VALID;
122
123   i_multp_op : entity work.multp_op
124     generic map (
125       g_data_width => g_data_width
126     )
127     port map (
128       DIN  => DIN,
129       DOUT => DOUT
130     );
131
132 end combinatorial;
```

CÓDIGO D.6: multp.vhd

```
-- Authors:
--    Unai Martinez-Corral & Unai Sainz-Estebanez
--      <unai.martinezcorral@ehu.eus>
--      <usainz003@ikasle.ehu.eus>
--
-- Licensed under the Apache License, Version 2.0 (the "License");
-- you may not use this file except in compliance with the License.
-- You may obtain a copy of the License at
--
--      http://www.apache.org/licenses/LICENSE-2.0
--
-- Unless required by applicable law or agreed to in writing, software
-- distributed under the License is distributed on an "AS IS" BASIS,
-- WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
-- See the License for the specific language governing permissions and
-- limitations under the License.
--
-- SPDX-License-Identifier: Apache-2.0

library ieee;
context ieee.ieee_std_context;

entity multp_wfifos is
  generic (
    g_data_width : natural := 32;
    g_fifo_depth : natural := 0 -- ceiling of the log base 2 of the desired FIFO
    ↪  length
  );
  port (
     CLK_IN   : in std_logic;
     CLK_MULT : in std_logic;
     CLK_OUT  : in std_logic;
     RST      : in std_logic;
     DIN      : in std_logic_vector (g_data_width-1 downto 0);
     DOUT     : out std_logic_vector (g_data_width-1 downto 0);
     WRITE    : in std_logic;
     READ     : in std_logic;
     FULL     : out std_logic;
     EMPTY    : out std_logic
);
end multp_wfifos;

architecture rtl of multp_wfifos is

  signal data_in    : std_logic_vector(g_data_width-1 downto 0);
  signal data_out   : std_logic_vector(g_data_width-1 downto 0);
  signal i_read     : std_logic;
  signal i_write    : std_logic;
  signal i_empty    : std_logic;
  signal i_full     : std_logic;
  signal in_valid   : std_logic;
```

```
51    signal in_ready   : std_logic;
52    signal out_valid  : std_logic;
53    signal out_ready  : std_logic;
54
55  begin
56
57    fifo_in : entity work.fifo
58      generic map (
59        N_bits => g_data_width,
60        Log2_elements => g_fifo_depth)
61      port map (
62        clk_wr => CLK_IN,
63        clk_rd => CLK_MULT,
64        rst => RST,
65        fifo_in => DIN,
66        fifo_out => data_in,
67        wr => WRITE,
68        rd => i_read,
69        full_o => FULL,
70        empty_o => i_empty
71      );
72
73    i_read <= in_ready and not i_empty;
74
75    process (CLK_MULT) begin
76      if rising_edge(CLK_MULT) then
77        if RST then
78          in_valid <= '0';
79        else
80          in_valid <= i_read;
81        end if;
82      end if;
83    end process;
84
85    multp : entity work.multp(combinatorial)
86      generic map (
87        g_data_width => g_data_width
88      )
89      port map (
90        CLK => CLK_MULT,
91        RST => RST,
92        IN_VALID => in_valid,
93        IN_READY => in_ready,
94        DIN => data_in,
95        OUT_VALID => out_valid,
96        OUT_READY => out_ready,
97        DOUT => data_out
98      );
99
100   i_write <= out_valid and out_ready;
101   out_ready <= not i_full;
102
```

```
103   fifo_out : entity work.fifo
104     generic map (
105       N_bits => g_data_width,
106       Log2_elements => g_fifo_depth
107     )
108     port map (
109       clk_wr => CLK_MULT,
110       clk_rd => CLK_OUT,
111       rst => RST,
112       fifo_in => data_out,
113       fifo_out => DOUT,
114       wr => i_write,
115       rd => READ,
116       full_o => i_full,
117       empty_o => EMPTY
118     );
119
120 end rtl;
```

CÓDIGO D.7: multp_wfifos.vhd

```
1  -- Authors:
2  --   Unai Martinez-Corral & Unai Sainz-Estebanez
3  --     <unai.martinezcorral@ehu.eus>
4  --     <usainz003@ikasle.ehu.eus>
5  --
6  -- Licensed under the Apache License, Version 2.0 (the "License");
7  -- you may not use this file except in compliance with the License.
8  -- You may obtain a copy of the License at
9  --
10 --     http://www.apache.org/licenses/LICENSE-2.0
11 --
12 -- Unless required by applicable law or agreed to in writing, software
13 -- distributed under the License is distributed on an "AS IS" BASIS,
14 -- WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15 -- See the License for the specific language governing permissions and
16 -- limitations under the License.
17 --
18 -- SPDX-License-Identifier: Apache-2.0
19
20 library ieee;
21 context ieee.ieee_std_context;
22
23 entity multp_wfifos_axis is
24   generic (
25     g_data_width : natural := 32;
26     g_fifo_depth : natural := 0 -- ceiling of the log base 2 of the desired FIFO
       ↪ length
27   );
28   port (
29     CLK_MULT     : in std_logic;
30     s_axis_clk   : in  std_logic;
```

```
31      s_axis_rstn  : in  std_logic;
32      s_axis_rdy   : out std_logic;
33      s_axis_data  : in  std_logic_vector(g_data_width-1 downto 0);
34      s_axis_valid : in  std_logic;
35      m_axis_clk   : in  std_logic;
36      m_axis_rstn  : in  std_logic;
37      m_axis_valid : out std_logic;
38      m_axis_data  : out std_logic_vector(g_data_width-1 downto 0);
39      m_axis_rdy   : in  std_logic
40    );
41  end multp_wfifos_axis;
42
43  architecture rtl of multp_wfifos_axis is
44
45  signal read, empty, full, valid : std_logic;
46
47  begin
48
49    s_axis_rdy <= not full;
50
51    i_multp_wfifos : entity work.multp_wfifos
52      generic map (
53        g_data_width => g_data_width,
54        g_fifo_depth => g_fifo_depth
55      )
56      port map (
57        CLK_IN   => s_axis_clk,
58        CLK_MULT => clk_mult,
59        CLK_OUT  => m_axis_clk,
60        RST      => s_axis_rstn nand m_axis_rstn,
61        DIN      => s_axis_data,
62        DOUT     => m_axis_data,
63        WRITE    => s_axis_valid and not full,
64        READ     => read,
65        FULL     => full,
66        EMPTY    => empty
67      );
68
69    read <= (valid nand not m_axis_rdy) and not empty;
70
71    process (m_axis_clk) begin
72      if rising_edge(m_axis_clk) then
73        if (not m_axis_rstn) or ((valid and empty) and m_axis_rdy) then
74          valid <= '0';
75        elsif read then
76          valid <= '1';
77        end if;
78      end if;
79    end process;
80
81    m_axis_valid <= valid;
82
```

```
83 end rtl;
```

CÓDIGO D.8: multp_wfifos_axis.vhd

```
1  -- RTL of multp_wfifos_wishbone
2
3  library ieee;
4  context ieee.ieee_std_context;
5
6  entity multp_wfifos_wishbone is
7  generic (
8          -- Number of bits that the input/output data has
9          N_bits : in natural;
10         -- Log2 of number of elements that the FIFOs have; Both the same size; Number
           ↪  of FIFO elements has to be a power of two
11         Log2_elements : in natural
12         );
13 port (
14     rst_i : in std_logic;
15     clk_i : in std_logic;
16     adr_i : in std_logic_vector(31 downto 0);
17     dat_i : in std_logic_vector(31 downto 0);
18     dat_o : out std_logic_vector(31 downto 0);
19     we_i : in std_logic;
20     sel_i : in std_logic_vector(3 downto 0);
21     stb_i : in std_logic;
22     ack_o : out std_logic;
23     cyc_i : in std_logic;
24     err_o : out std_logic;
25     stall_o : out std_logic
26     );
27 end multp_wfifos_wishbone;
28
29 architecture rtl of multp_wfifos_wishbone is
30
31 signal reset, write, read, empty, full : std_logic;
32 signal ack : std_logic;
33 signal stall : std_logic;
34 signal input : std_logic_vector(31 downto 0);
35 signal output : std_logic_vector(31 downto 0);
36 signal transfer_in : std_logic;
37 signal transfer_out : std_logic;
38 signal output_window : std_logic := '0';
39
40 begin
41
42 -- Multp_wfifos instantation
43
44 multp_wfifos_0 : entity work.multp_wfifos
45                  generic map (g_data_width => N_bits,
46                               g_fifo_depth => Log2_elements)
47                  port map (clk_in => clk_i,
```

```
48                          clk_mult => clk_i,
49                          clk_out => clk_i,
50                          rst => reset,
51                          din => input,
52                          dout => output,
53                          write => write,
54                          read => read,
55                          full => full,
56                          empty => empty);
57
58  -- Reset (NEORV32 rst is low-active)
59
60  reset <= not rst_i;
61
62  -- Make error signal
63
64  err_o <= '0'; --tie to zero if not explicitly used
65
66  -- Make stall signal
67
68  with we_i select
69       stall <= full  when '1',
70                empty when others;
71
72  stall_o <= stall;
73
74  -- Make transfer in/out signals
75
76  transfer_in  <= (stb_i and cyc_i and we_i and not(stall)) when adr_i = x"90000000"
    ↪  else  -- The address is 0x90000000; See main.c in sw/EMEM
77                  '0';
78
79  transfer_out <= (stb_i and cyc_i and not(we_i) and not(stall)) when adr_i =
    ↪  x"90000000" else
80                  '0';
81
82  -- Manage input/output and write/read signals
83
84  with transfer_in select
85       input <= dat_i when '1',
86                (others => '0') when others;
87
88  with transfer_in select
89       write <= '1' when '1',
90                '0' when others;
91
92  with transfer_out select
93       read  <= '1' when '1',
94                '0' when others;
95
96  with output_window select
97       dat_o <= output when '1',
```

```
98              (others => '0') when others;
99
100 -- Manage output_window
101
102 process (clk_i) begin
103     if rising_edge(clk_i) then
104       if reset = '1' then
105         output_window <= '0';
106       elsif transfer_out = '1' then
107         output_window <= '1';
108       else
109         output_window <= '0';
110       end if;
111     end if;
112 end process;
113
114 -- Manage ack signal
115
116 process (clk_i) begin
117     if rising_edge(clk_i) then
118       if reset = '1' then
119         ack <= '0';
120       else
121         if transfer_in or transfer_out then
122           ack <= '1';
123         else
124           ack <= '0';
125         end if;
126       end if;
127     end if;
128   end process;
129
130 ack_o <= ack;
131
132 end rtl;
```

CÓDIGO D.9: multp_wfifos_wishbone.vhd

```
1  -- RTL of multp
2
3  library ieee;
4  context ieee.ieee_std_context;
5
6  entity multp_wishbone is
7  generic (
8          -- Number of bits that the input/output data has
9          N_bits : in natural
10         );
11 port (
12     rst_i : in std_logic;
13     clk_i : in std_logic;
14     adr_i : in std_logic_vector(31 downto 0);
```

```
15      dat_i : in std_logic_vector(31 downto 0);
16      dat_o : out std_logic_vector(31 downto 0);
17      we_i : in std_logic;
18      sel_i : in std_logic_vector(3 downto 0);
19      stb_i : in std_logic;
20      ack_o : out std_logic;
21      cyc_i : in std_logic;
22      err_o : out std_logic;
23      stall_o : out std_logic
24      );
25  end multp_wishbone;
26
27  architecture rtl of multp_wishbone is
28
29  signal reset : std_logic;
30  signal ack : std_logic;
31  signal input : std_logic_vector(31 downto 0);
32  signal output : std_logic_vector(31 downto 0);
33  signal transfer_in : std_logic;
34  signal transfer_out : std_logic;
35
36  begin
37
38  -- Multp instantation
39
40  multp_0 : entity work.multp_op
41                  generic map (g_data_width => N_bits)
42                  port map (din => input,
43                            dout => output);
44
45  -- Reset (NEORV32 rst is low-active)
46
47  reset <= not rst_i;
48
49  -- Make error signal
50
51  err_o <= '0'; --tie to zero if not explicitly used
52
53  -- Make stall signal
54
55  stall_o <= '0';
56
57  -- Make transfer in/out signals
58
59  transfer_in  <= (stb_i and cyc_i and we_i) when adr_i = x"90000000" else -- The
    ↪  address is 0x90000000; See main.c in sw/EMEM
60                  '0';
61
62  transfer_out <= (stb_i and cyc_i and not(we_i)) when adr_i = x"90000000" else
63                  '0';
64
65  -- Manage inputs/outputs
```

```
66
67  process (clk_i) begin
68      if rising_edge(clk_i) then
69        if reset = '1' then
70          input <= (others=>'0');
71          dat_o <= (others=>'0');
72        elsif transfer_in then
73          input <= dat_i;
74          dat_o <= (others=>'0');
75        elsif transfer_out then
76          dat_o <= output;
77        else
78          dat_o <= (others=>'0');
79        end if;
80      end if;
81  end process;
82
83  -- Make ack signal
84
85  process (clk_i) begin
86      if rising_edge(clk_i) then
87        if reset = '1' then
88          ack <= '0';
89        else
90          if transfer_in or transfer_out then
91            ack <= '1';
92          else
93            ack <= '0';
94          end if;
95        end if;
96      end if;
97    end process;
98
99  ack_o <= ack;
100
101 end rtl;
```

CÓDIGO D.10: multp_wishbone.vhd

```
1   #include <neorv32.h>
2   #include <string.h>
3
4   #define BAUD_RATE 19200
5
6   // This defines is used to bypass the intermediate print functions between axi
    ↪  functions (for latency and throughput measurements)
7   // Comment these defines to perform a normal execution
8   // Uncomment latency to perform latency measurements
9   // Uncomment throughput to perform throughput measurements
10  //#define latency
11  //#define throughput
12
```

```
13 int main() {
14
15   // Capture all exceptions and give debug info via UART0
16   neorv32_rte_setup();
17
18   // Setup UART at default baud rate, no interrupts
19   neorv32_uart0_setup(BAUD_RATE, 0);
20
21   // Check if UART0 unit is implemented at all
22   if (neorv32_uart0_available() == 0) {
23     return -1; // abort if not implemented
24   }
25
26   // check if the CPU base counters are implemented
27   if ((neorv32_cpu_csr_read(CSR_MXISA) & (1 << CSR_MXISA_ZICNTR)) == 0) {
28     neorv32_uart0_printf("ERROR! Base counters ('Zicntr' ISA extensions) not
       ↪  implemented!\n");
29     return -1;
30   }
31
32   // check if SLINK is implemented at all
33   if (neorv32_slink_available() == 0) {
34     neorv32_uart0_printf("ERROR! SLINK module not implemented.");
35     return -1;
36   }
37
38   // setup SLINK module
39   neorv32_slink_setup(0, 0);
40
41   // Declaration of variables
42   //0000000000000001 x 0000000000000001
43   static uint32_t fir = 0x00010001;
44   //0000000000000010 x 0000000000000010
45   static uint32_t sec = 0x00020002;
46   //0000000000000100 x 0000000000000100
47   static uint32_t thi = 0x00040004;
48   //0000000000001000 x 0000000000001000
49   static uint32_t fou = 0x00080008;
50
51 #ifdef latency
52   // Intro
53   neorv32_uart0_printf("\n A-lat \n\n");
54   // Write 4 inputs to mult and read the outputs from mult one by one
55     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
56     neorv32_slink_put(fir);
57     neorv32_slink_get();
58     neorv32_cpu_csr_read(CSR_MCYCLE);
59
60     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
61     neorv32_slink_put(sec);
62     neorv32_slink_get();
63     neorv32_cpu_csr_read(CSR_MCYCLE);
```

```
64
65      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
66      neorv32_slink_put(thi);
67      neorv32_slink_get();
68      neorv32_cpu_csr_read(CSR_MCYCLE);
69
70      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
71      neorv32_slink_put(fou);
72      neorv32_slink_get();
73      neorv32_cpu_csr_read(CSR_MCYCLE);
74
75    // End
76    neorv32_uart0_printf("\nEND-lat\n");
77    #elif defined throughput
78    // Intro
79    neorv32_uart0_printf("\n A-thr \n\n");
80    // Write 4 inputs to mult
81
82      neorv32_slink_put(fir);
83
84      neorv32_slink_put(sec);
85
86      neorv32_slink_put(thi);
87
88      neorv32_slink_put(fou);
89
90    // Read outputs from mult
91
92      neorv32_slink_get();
93      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
94
95      neorv32_slink_get();
96      neorv32_cpu_csr_read(CSR_MCYCLE);
97
98      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
99      neorv32_slink_get();
100     neorv32_cpu_csr_read(CSR_MCYCLE);
101
102     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
103     neorv32_slink_get();
104     neorv32_cpu_csr_read(CSR_MCYCLE);
105
106   // End
107   neorv32_uart0_printf("\nEND-thr\n");
108   #else
109   int i, slink_rc;;
110   // show SLINK FIFO configuration
111   int rx_depth = neorv32_slink_get_rx_fifo_depth();
112   int tx_depth = neorv32_slink_get_tx_fifo_depth();
113   // Intro
114   neorv32_uart0_printf("\n<<< MULT(P) via slink (AXI-Stream) demo program >>>\n\n");
115   neorv32_uart0_printf("RX FIFO depth: %u\n"
```

```
116                     "TX FIFO depth: %u\n\n",
117                     rx_depth, tx_depth);
118   neorv32_uart0_printf("-------- Write data to MULT(P) --------\n");
119   // Write 4 inputs to mult
120     for (i=0; i<4 ; i++) {
121     if(i==0){
122             neorv32_uart0_printf("[%i] Sending 0x%x... ", i, fir);
123             slink_rc = neorv32_slink_tx_status();
124                 if (slink_rc == SLINK_FIFO_FULL) {
125                         neorv32_uart0_printf("FAILED! TX FIFO full!\n");
126                         break;
127                 }
128                 else {
129                         neorv32_slink_put(fir);
130                         neorv32_uart0_printf("ok\n");
131                     }
132             }
133     if(i==1){
134             neorv32_uart0_printf("[%i] Sending 0x%x... ", i, sec);
135             slink_rc = neorv32_slink_tx_status();
136                 if (slink_rc == SLINK_FIFO_FULL) {
137                         neorv32_uart0_printf("FAILED! TX FIFO full!\n");
138                         break;
139                 }
140                 else {
141                         neorv32_slink_put(sec);
142                         neorv32_uart0_printf("ok\n");
143                     }
144             }
145     if(i==2){
146             neorv32_uart0_printf("[%i] Sending 0x%x... ", i, thi);
147             slink_rc = neorv32_slink_tx_status();
148                 if (slink_rc == SLINK_FIFO_FULL) {
149                         neorv32_uart0_printf("FAILED! TX FIFO full!\n");
150                         break;
151                 }
152                 else {
153                         neorv32_slink_put(thi);
154                         neorv32_uart0_printf("ok\n");
155                     }
156             }
157     if(i==3){
158             neorv32_uart0_printf("[%i] Sending 0x%x... ", i, fou);
159             slink_rc = neorv32_slink_tx_status();
160                 if (slink_rc == SLINK_FIFO_FULL) {
161                         neorv32_uart0_printf("FAILED! TX FIFO full!\n");
162                         break;
163                 }
164                 else {
165                         neorv32_slink_put(fou);
166                         neorv32_uart0_printf("ok\n");
167                     }
```

```
168             }
169     }
170   neorv32_uart0_printf("\n-------- Read data from MULT(P) --------\n");
171   // Read outputs from mult
172     for (i=0; i<4; i++) {
173             neorv32_uart0_printf("[%i] Reading RX data... ", i);
174             slink_rc = neorv32_slink_rx_status();
175                 if (slink_rc == SLINK_FIFO_EMPTY) {
176                     neorv32_uart0_printf("FAILED! RX FIFO empty!\n");
177                     break;
178                 }
179                 else {
180                     neorv32_uart0_printf("0x%x\n", neorv32_slink_get());
181                 }
182     }
183   // End
184   neorv32_uart0_printf("\nProgram execution completed.\n");
185   #endif
186
187   return 0;
188 }
```

CÓDIGO D.11: SLINK main.c

```
1  #include <neorv32.h>
2  #include <string.h>
3
4  #define BAUD_RATE 19200
5
6  // This defines is used to bypass the intermediate print functions between wishbone
   ↪  functions (for latency and throughput measurements)
7  // Comment these defines to perform a normal execution
8  // Uncomment latency to perform latency measurements
9  // Uncomment throughput to perform throughput measurements
10 //#define latency
11 //#define throughput
12
13 int main() {
14
15   // Capture all exceptions and give debug info via UART0
16   neorv32_rte_setup();
17
18   // Setup UART at default baud rate, no interrupts
19   neorv32_uart0_setup(BAUD_RATE, 0);
20
21   // Check if UART0 unit is implemented at all
22   if (neorv32_uart0_available() == 0) {
23     return -1; // abort if not implemented
24   }
25
26   // check if the CPU base counters are implemented
27   if ((neorv32_cpu_csr_read(CSR_MXISA) & (1 << CSR_MXISA_ZICNTR)) == 0) {
```

```
28      neorv32_uart0_printf("ERROR! Base counters ('Zicntr' ISA extensions) not
        ↪  implemented!\n");
29      return -1;
30    }
31
32    // Declaration of variables
33    // address 0x90000000
34    static uint32_t add = 0x90000000;
35    //0000000000000001 x 0000000000000001
36    static uint32_t fir = 0x00010001;
37    //0000000000000010 x 0000000000000010
38    static uint32_t sec = 0x00020002;
39    //0000000000000100 x 0000000000000100
40    static uint32_t thi = 0x00040004;
41    //0000000000001000 x 0000000000001000
42    static uint32_t fou = 0x00080008;
43
44    #ifdef latency
45    // Intro
46    neorv32_uart0_printf("\n W-lat \n\n");
47    // Write 4 inputs to mult and read the outputs from mult one by one
48      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
49      neorv32_cpu_store_unsigned_word(add, fir);
50      neorv32_cpu_load_unsigned_word(add);
51      neorv32_cpu_csr_read(CSR_MCYCLE);
52
53      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
54      neorv32_cpu_store_unsigned_word(add, sec);
55      neorv32_cpu_load_unsigned_word(add);
56      neorv32_cpu_csr_read(CSR_MCYCLE);
57
58      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
59      neorv32_cpu_store_unsigned_word(add, thi);
60      neorv32_cpu_load_unsigned_word(add);
61      neorv32_cpu_csr_read(CSR_MCYCLE);
62
63      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
64      neorv32_cpu_store_unsigned_word(add, fou);
65      neorv32_cpu_load_unsigned_word(add);
66      neorv32_cpu_csr_read(CSR_MCYCLE);
67
68    // End
69    neorv32_uart0_printf("\nEND-lat\n");
70    #elif defined throughput
71    // Intro
72    neorv32_uart0_printf("\n W-thr \n\n");
73    // Write 4 inputs to mult
74
75      neorv32_cpu_store_unsigned_word(add, fir);
76
77      neorv32_cpu_store_unsigned_word(add, sec);
78
```

```
79      neorv32_cpu_store_unsigned_word(add, thi);
80
81      neorv32_cpu_store_unsigned_word(add, fou);
82
83    // Read outputs from mult
84
85      neorv32_cpu_load_unsigned_word(add);
86      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
87
88      neorv32_cpu_load_unsigned_word(add);
89      neorv32_cpu_csr_read(CSR_MCYCLE);
90
91      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
92      neorv32_cpu_load_unsigned_word(add);
93      neorv32_cpu_csr_read(CSR_MCYCLE);
94
95      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
96      neorv32_cpu_load_unsigned_word(add);
97      neorv32_cpu_csr_read(CSR_MCYCLE);
98
99    // End
100   neorv32_uart0_printf("\nEND-thr\n");
101   #else
102   int i;
103   // Intro
104   neorv32_uart0_printf("\n<<< MULT(P) via external memory interface (EMEM) demo
      ↪  program >>>\n\n");
105   neorv32_uart0_printf("-------- Write data to MULT(P) --------\n");
106   // Write 4 inputs to mult
107     for (i=0; i<4 ; i++) {
108     if(i==0){
109             neorv32_cpu_store_unsigned_word(add, fir);
110             neorv32_uart0_printf("\n[%i] Sending 0x%x to address 0x%x \n",i,fir,add);
111            }
112     if(i==1){
113             neorv32_cpu_store_unsigned_word(add, sec);
114             neorv32_uart0_printf("\n[%i] Sending 0x%x to address 0x%x \n",i,sec,add);
115            }
116     if(i==2){
117             neorv32_cpu_store_unsigned_word(add, thi);
118             neorv32_uart0_printf("\n[%i] Sending 0x%x to address 0x%x \n",i,thi,add);
119            }
120     if(i==3){
121             neorv32_cpu_store_unsigned_word(add, fou);
122             neorv32_uart0_printf("\n[%i] Sending 0x%x to address 0x%x \n",i,fou,add);
123            }
124   }
125   neorv32_uart0_printf("\n-------- Read data from MULT(P) --------\n");
126   // Read outputs from mult
127     for (i=0; i<4; i++) {
128         neorv32_uart0_printf("\n[%i] The read data is 0x%x
          ↪  \n",i,neorv32_cpu_load_unsigned_word(add));
```

```
129       }
130   // End
131   neorv32_uart0_printf("\nProgram execution completed.\n");
132   #endif
133
134   return 0;
135 }
```

CÓDIGO D.12: XBUS main.c

```
1  #include <neorv32.h>
2  #include <string.h>
3
4  #define BAUD_RATE 19200
5
6  // This defines is used to bypass the intermediate print functions between cfu
   ↪  functions (for latency measurements)
7  // Comment these defines to perform a normal execution
8  // Uncomment lat_mult to perform latency measurements with mult_wfifos
9  // Uncomment lat_multpw to perform latency measurements with multp_wfifos
10 // Uncomment lat_multp to perform latency measurements with multp
11 //#define lat_mult
12 //#define lat_multpw
13 //#define lat_multp
14
15 int main() {
16
17   // Capture all exceptions and give debug info via UART0
18   neorv32_rte_setup();
19
20   // Setup UART at default baud rate, no interrupts
21   neorv32_uart0_setup(BAUD_RATE, 0);
22
23   // Check if UART0 unit is implemented at all
24   if (neorv32_uart0_available() == 0) {
25     return -1; // abort if not implemented
26   }
27
28   // check if the CFU is implemented at all (the CFU is wrapped in the core's "Zxcfu"
     ↪  ISA extension)
29   if (neorv32_cpu_cfu_available() == 0) {
30     neorv32_uart0_printf("ERROR! CFU ('Zxcfu' ISA extensions) not implemented!\n");
31     return 1;
32   }
33
34   // check if the CPU base counters are implemented
35   if ((neorv32_cpu_csr_read(CSR_MXISA) & (1 << CSR_MXISA_ZICNTR)) == 0) {
36     neorv32_uart0_printf("ERROR! Base counters ('Zicntr' ISA extensions) not
       ↪  implemented!\n");
37     return -1;
38   }
39
```

```
40    // Declaration of variables
41    //0000000000000001 x 0000000000000001
42    static uint32_t fir = 0x00010001;
43    //0000000000000010 x 0000000000000010
44    static uint32_t sec = 0x00020002;
45    //0000000000000100 x 0000000000000100
46    static uint32_t thi = 0x00040004;
47    //0000000000001000 x 0000000000001000
48    static uint32_t fou = 0x00080008;
49
50    #ifdef lat_mult
51    // Intro
52    neorv32_uart0_printf("\n CFU-mw \n\n");
53   // Perform 4 multiplication through custom instruction (funct3=000)
54      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
55      neorv32_cfu_r3_instr(0b1111111, 0b000, fir, 0);
56      neorv32_cpu_csr_read(CSR_MCYCLE);
57
58      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
59      neorv32_cfu_r3_instr(0b1111111, 0b000, sec, 0);
60      neorv32_cpu_csr_read(CSR_MCYCLE);
61
62      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
63      neorv32_cfu_r3_instr(0b1111111, 0b000, thi, 0);
64      neorv32_cpu_csr_read(CSR_MCYCLE);
65
66      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
67      neorv32_cfu_r3_instr(0b1111111, 0b000, fou, 0);
68      neorv32_cpu_csr_read(CSR_MCYCLE);
69    // End
70    neorv32_uart0_printf("\nEND-mw\n");
71    #elif defined lat_multpw
72    // Intro
73    neorv32_uart0_printf("\n CFU-mpw \n\n");
74   // Perform 4 multiplication through custom instruction (funct3=001)
75      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
76      neorv32_cfu_r3_instr(0b1111111, 0b001, fir, 0);
77      neorv32_cpu_csr_read(CSR_MCYCLE);
78
79      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
80      neorv32_cfu_r3_instr(0b1111111, 0b001, sec, 0);
81      neorv32_cpu_csr_read(CSR_MCYCLE);
82
83      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
84      neorv32_cfu_r3_instr(0b1111111, 0b001, thi, 0);
85      neorv32_cpu_csr_read(CSR_MCYCLE);
86
87      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
88      neorv32_cfu_r3_instr(0b1111111, 0b001, fou, 0);
89      neorv32_cpu_csr_read(CSR_MCYCLE);
90    // End
91    neorv32_uart0_printf("\nEND-mpw\n");
```

```
92    #elif defined lat_multp
93    // Intro
94    neorv32_uart0_printf("\n CFU-mp \n\n");
95   // Perform 4 multiplication through custom instruction (funct3=010)
96      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
97      neorv32_cfu_r3_instr(0b1111111, 0b010, fir, 0);
98      neorv32_cpu_csr_read(CSR_MCYCLE);
99
100     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
101     neorv32_cfu_r3_instr(0b1111111, 0b010, sec, 0);
102     neorv32_cpu_csr_read(CSR_MCYCLE);
103
104     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
105     neorv32_cfu_r3_instr(0b1111111, 0b010, thi, 0);
106     neorv32_cpu_csr_read(CSR_MCYCLE);
107
108     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
109     neorv32_cfu_r3_instr(0b1111111, 0b010, fou, 0);
110     neorv32_cpu_csr_read(CSR_MCYCLE);
111   // End
112   neorv32_uart0_printf("\nEND-mp\n");
113   #else
114   int i;
115   // Intro
116   neorv32_uart0_printf("\n<<< MULT(P) via CFU demo program >>>\n\n");
117   neorv32_uart0_printf("\n--- CFU R3-Type: Multiplier Instruction ---\n");
118   neorv32_uart0_printf("\n rs1= 0xIN1-IN2, rs2= DC, rd = IN1 x IN2 \n\n");
119   // Write 4 inputs to mult and read the outputs from mult one by one in mult_wfifos
120     neorv32_uart0_printf("\n Mult_wfifos: \n\n");
121     for (i=0; i<4 ; i++) {
122     if(i==0){
123             neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                ↪  funct3=0b000, [rs1]=0x%x, [rs2]=0x%x ) = ", i, fir, 0);
124             neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b000, fir,
                ↪  0));
125            }
126     if(i==1){
127             neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                ↪  funct3=0b000, [rs1]=0x%x, [rs2]=0x%x ) = ", i, sec, 0);
128             neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b000, sec,
                ↪  0));
129            }
130     if(i==2){
131             neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                ↪  funct3=0b000, [rs1]=0x%x, [rs2]=0x%x ) = ", i, thi, 0);
132             neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b000, thi,
                ↪  0));
133            }
134     if(i==3){
135             neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                ↪  funct3=0b000, [rs1]=0x%x, [rs2]=0x%x ) = ", i, fou, 0);
```

```
136             neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b000, fou,
                ↪  0));
137             }
138     }
139   // Write 4 inputs to mult and read the outputs from mult one by one in multp_wfifos
140     neorv32_uart0_printf("\n Multp_wfifos: \n\n");
141     for (i=0; i<4 ; i++) {
142     if(i==0){
143             neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                ↪  funct3=0b001, [rs1]=0x%x, [rs2]=0x%x ) = ", i, fir, 0);
144             neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b001, fir,
                ↪  0));
145             }
146     if(i==1){
147             neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                ↪  funct3=0b001, [rs1]=0x%x, [rs2]=0x%x ) = ", i, sec, 0);
148             neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b001, sec,
                ↪  0));
149             }
150     if(i==2){
151             neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                ↪  funct3=0b001, [rs1]=0x%x, [rs2]=0x%x ) = ", i, thi, 0);
152             neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b001, thi,
                ↪  0));
153             }
154     if(i==3){
155             neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                ↪  funct3=0b001, [rs1]=0x%x, [rs2]=0x%x ) = ", i, fou, 0);
156             neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b001, fou,
                ↪  0));
157             }
158     }
159   // Write 4 inputs to mult and read the outputs from mult one by one in multp
160     neorv32_uart0_printf("\n Multp: \n\n");
161     for (i=0; i<4 ; i++) {
162     if(i==0){
163             neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                ↪  funct3=0b010, [rs1]=0x%x, [rs2]=0x%x ) = ", i, fir, 0);
164             neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b010, fir,
                ↪  0));
165             }
166     if(i==1){
167             neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                ↪  funct3=0b010, [rs1]=0x%x, [rs2]=0x%x ) = ", i, sec, 0);
168             neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b010, sec,
                ↪  0));
169             }
170     if(i==2){
171             neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                ↪  funct3=0b010, [rs1]=0x%x, [rs2]=0x%x ) = ", i, thi, 0);
172             neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b010, thi,
                ↪  0));
```

```
173             }
174     if(i==3){
175              neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111,
                 ↪  funct3=0b010, [rs1]=0x%x, [rs2]=0x%x ) = ", i, fou, 0);
176              neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b010, fou,
                 ↪  0));
177             }
178     }
179   // End
180   neorv32_uart0_printf("\nCFU demo program completed.\n");
181   #endif
182
183   return 0;
184 }
```

CÓDIGO D.13: CFU main.c

```
1  #include <neorv32.h>
2
3  #define BAUD_RATE 19200
4
5  // This defines is used to bypass the intermediate print functions between cfs
   ↪  functions (for latency and throughput measurements)
6  // Comment these defines to perform a normal execution
7  // Uncomment latency to perform latency measurements
8  // Uncomment throughput to perform throughput measurements
9  //#define latency
10 //#define throughput
11
12 int main() {
13
14   // Capture all exceptions and give debug info via UART0
15   neorv32_rte_setup();
16
17   // Setup UART at default baud rate, no interrupts
18   neorv32_uart0_setup(BAUD_RATE, 0);
19
20   // Check if UART0 unit is implemented at all
21   if (neorv32_uart0_available() == 0) {
22     return -1; // abort if not implemented
23   }
24
25   // check if CFS is implemented at all
26   if (neorv32_cfs_available() == 0) {
27     neorv32_uart0_printf("Error! No CFS synthesized!\n");
28     return 1;
29   }
30
31   // check if the CPU base counters are implemented
32   if ((neorv32_cpu_csr_read(CSR_MXISA) & (1 << CSR_MXISA_ZICNTR)) == 0) {
33     neorv32_uart0_printf("ERROR! Base counters ('Zicntr' ISA extensions) not
       ↪  implemented!\n");
```

```
34      return -1;
35    }
36
37    // Declaration of variables
38    //0000000000000001 x 0000000000000001
39    static uint32_t fir = 0x00010001;
40    //0000000000000010 x 0000000000000010
41    static uint32_t sec = 0x00020002;
42    //0000000000000100 x 0000000000000100
43    static uint32_t thi = 0x00040004;
44    //0000000000001000 x 0000000000001000
45    static uint32_t fou = 0x00080008;
46
47
48    #ifdef latency
49    // Intro
50    neorv32_uart0_printf("\n CFS-lat \n\n");
51    // Write 4 inputs to mult and read the outputs from mult one by one
52      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
53      NEORV32_CFS->REG[0] = fir; // Write fir to CFS memory-mapped register 0
54      NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1; Write fir
        ↪ to mult
55      NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1; Read fir
        ↪ from mult
56      NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪ control signals
57      NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
58      neorv32_cpu_csr_read(CSR_MCYCLE);
59
60      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
61      NEORV32_CFS->REG[0] = sec; // Write sec to CFS memory-mapped register 0
62      NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1; Write sec
        ↪ to mult
63      NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1; Read sec
        ↪ from mult
64      NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪ control signals
65      NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
66      neorv32_cpu_csr_read(CSR_MCYCLE);
67
68      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
69      NEORV32_CFS->REG[0] = thi; // Write thi to CFS memory-mapped register 0
70      NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1; Write thi
        ↪ to mult
71      NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1; Read thi
        ↪ from mult
72      NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪ control signals
73      NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
74      neorv32_cpu_csr_read(CSR_MCYCLE);
75
76      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
```

```
77      NEORV32_CFS->REG[0] = fou; // Write fou to CFS memory-mapped register 0
78      NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1; Write fou
        ↪  to mult
79      NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1; Read fou
        ↪  from mult
80      NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪  control signals
81      NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
82      neorv32_cpu_csr_read(CSR_MCYCLE);
83
84    // End
85    neorv32_uart0_printf("\nEND-lat\n");
86    #elif defined throughput
87    // Intro
88    neorv32_uart0_printf("\n CFS-thr \n\n");
89    // Write 4 inputs to mult
90      NEORV32_CFS->REG[0] = fir; // Write fir to CFS memory-mapped register 0
91      NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1; Write fir
        ↪  to mult
92      NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪  control signals
93
94      NEORV32_CFS->REG[0] = sec; // Write sec to CFS memory-mapped register 0
95      NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1; Write sec
        ↪  to mult
96      NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪  control signals
97
98      NEORV32_CFS->REG[0] = thi; // Write thi to CFS memory-mapped register 0
99      NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1; Write thi
        ↪  to mult
100     NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪  control signals
101
102     NEORV32_CFS->REG[0] = fou; // Write fou to CFS memory-mapped register 0
103     NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1; Write fou
        ↪  to mult
104     NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪  control signals
105
106   // Read outputs from mult
107     NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1; Read data
        ↪  from mult
108     NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪  control signals
109     NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
110     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
111
112     NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1; Read data
        ↪  from mult
113     NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪  control signals
```

```
114     NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
115     neorv32_cpu_csr_read(CSR_MCYCLE);
116
117     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
118     NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1; Read data
        ↪ from mult
119     NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪ control signals
120     NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
121     neorv32_cpu_csr_read(CSR_MCYCLE);
122
123     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
124     NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1; Read data
        ↪ from mult
125     NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1; Clean the
        ↪ control signals
126     NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
127     neorv32_cpu_csr_read(CSR_MCYCLE);
128
129   // End
130   neorv32_uart0_printf("\nEND-thr\n");
131   #else
132   // Intro
133   int i;
134   neorv32_uart0_printf("\n<<< MULT(P) via CFS demo program >>>\n\n");
135   neorv32_uart0_printf("CFS memory-mapped registers:\n"
136                        " * NEORV32_CFS->REG[0] (r/w): input/output data register.\n"
137                        " * NEORV32_CFS->REG[1] (w): control signals register. 01
                         ↪ Write to MULT(P) - 10 Read from MULT(P) - 00 Clean control
                         ↪ signals\n\n");
138   neorv32_uart0_printf("-------- Write data to MULT(P) --------\n");
139   // Write 4 inputs to mult and read the outputs from mult one by one
140     for (i=0; i<4 ; i++) {
141     if(i==0){
142             NEORV32_CFS->REG[0] = fir; // Write fir to CFS memory-mapped register 0
143             NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1;
                ↪ Write fir to mult
144             NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1;
                ↪ Read fir from mult
145             NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1;
                ↪ Clean the control signals
146             neorv32_uart0_printf("%i: IN = 0x%x, OUT = 0x%x\n", i, fir,
                ↪ NEORV32_CFS->REG[0]);  // Read mult result from CFS memory-mapped
                ↪ register 0
147           }
148     if(i==1){
149             NEORV32_CFS->REG[0] = sec; // Write sec to CFS memory-mapped register 0
150             NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1;
                ↪ Write sec to mult
151             NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1;
                ↪ Read sec from mult
```

```
152             NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1;
                ↪ Clean the control signals
153             neorv32_uart0_printf("%i: IN = 0x%x, OUT = 0x%x\n", i, sec,
                ↪ NEORV32_CFS->REG[0]);  // Read mult result from CFS memory-mapped
                ↪ register 0
154           }
155     if(i==2){
156             NEORV32_CFS->REG[0] = thi; // Write thi to CFS memory-mapped register 0
157             NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1;
                ↪ Write thi to mult
158             NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1;
                ↪ Read thi from mult
159             NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1;
                ↪ Clean the control signals
160             neorv32_uart0_printf("%i: IN = 0x%x, OUT = 0x%x\n", i, thi,
                ↪ NEORV32_CFS->REG[0]);  // Read mult result from CFS memory-mapped
                ↪ register 0
161           }
162     if(i==3){
163             NEORV32_CFS->REG[0] = fou; // Write fou to CFS memory-mapped register 0
164             NEORV32_CFS->REG[1] = 1; // Write 01 to CFS memory-mapped register 1;
                ↪ Write fou to mult
165             NEORV32_CFS->REG[1] = 2; // write 10 to CFS memory-mapped register 1;
                ↪ Read fou from mult
166             NEORV32_CFS->REG[1] = 0; // Write 00 to CFS memory-mapped register 1;
                ↪ Clean the control signals
167             neorv32_uart0_printf("%i: IN = 0x%x, OUT = 0x%x\n", i, fou,
                ↪ NEORV32_CFS->REG[0]);  // Read mult result from CFS memory-mapped
                ↪ register 0
168           }
169     }
170   // End
171   neorv32_uart0_printf("\nCFS demo program completed.\n");
172   #endif
173
174   return 0;
175 }
```

CÓDIGO D.14: CFS *buffered* main.c

```
1  #include <neorv32.h>
2
3  #define BAUD_RATE 19200
4
5  // This defines is used to bypass the intermediate print functions between CFS
   ↪ functions (for latency measurements)
6  // Comment these defines to perform a normal execution
7  // Uncomment latency to perform latency measurements
8  //#define latency
9
10 int main() {
11
```

```
12    // Capture all exceptions and give debug info via UART0
13    neorv32_rte_setup();
14
15    // Setup UART at default baud rate, no interrupts
16    neorv32_uart0_setup(BAUD_RATE, 0);
17
18    // Check if UART0 unit is implemented at all
19    if (neorv32_uart0_available() == 0) {
20      return -1; // abort if not implemented
21    }
22
23    // check if CFS is implemented at all
24    if (neorv32_cfs_available() == 0) {
25      neorv32_uart0_printf("Error! No CFS synthesized!\n");
26      return 1;
27    }
28
29    // check if the CPU base counters are implemented
30    if ((neorv32_cpu_csr_read(CSR_MXISA) & (1 << CSR_MXISA_ZICNTR)) == 0) {
31      neorv32_uart0_printf("ERROR! Base counters ('Zicntr' ISA extensions) not
        ↪  implemented!\n");
32      return -1;
33    }
34
35    // Declaration of variables
36    //0000000000000001 x 0000000000000001
37    static uint32_t fir = 0x00010001;
38    //0000000000000010 x 0000000000000010
39    static uint32_t sec = 0x00020002;
40    //0000000000000100 x 0000000000000100
41    static uint32_t thi = 0x00040004;
42    //0000000000001000 x 0000000000001000
43    static uint32_t fou = 0x00080008;
44
45
46    #ifdef latency
47    // Intro
48    neorv32_uart0_printf("\n CFS-lat \n\n");
49    // Write 4 inputs to mult and read the outputs from mult one by one
50      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
51      NEORV32_CFS->REG[0] = fir; // Write fir to CFS memory-mapped register 0
52      NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
53      neorv32_cpu_csr_read(CSR_MCYCLE);
54
55      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
56      NEORV32_CFS->REG[0] = sec; // Write sec to CFS memory-mapped register 0
57      NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
58      neorv32_cpu_csr_read(CSR_MCYCLE);
59
60      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
61      NEORV32_CFS->REG[0] = thi; // Write thi to CFS memory-mapped register 0
62      NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
```

```
63      neorv32_cpu_csr_read(CSR_MCYCLE);
64
65      neorv32_cpu_csr_write(CSR_MCYCLE, 0);
66      NEORV32_CFS->REG[0] = fou; // Write fou to CFS memory-mapped register 0
67      NEORV32_CFS->REG[0]; // Read mult result from CFS memory-mapped register 0
68      neorv32_cpu_csr_read(CSR_MCYCLE);
69
70    // End
71    neorv32_uart0_printf("\nEND-lat\n");
72    #else
73    // Intro
74    int i;
75    neorv32_uart0_printf("\n<<< MULT(P) via CFS demo program >>>\n\n");
76    neorv32_uart0_printf("CFS memory-mapped registers:\n"
77                         " * NEORV32_CFS->REG[0] (r/w): input/output data
                           ↪  register.\n\n");
78    neorv32_uart0_printf("-------- Write data to MULT(P) --------\n");
79    // Write 4 inputs to mult and read the outputs from mult one by one
80      for (i=0; i<4 ; i++) {
81      if(i==0){
82              NEORV32_CFS->REG[0] = fir; // Write fir to CFS memory-mapped register 0
83              neorv32_uart0_printf("%i: IN = 0x%x, OUT = 0x%x\n", i, fir,
                ↪  NEORV32_CFS->REG[0]);  // Read mult result from CFS memory-mapped
                ↪  register 0
84             }
85      if(i==1){
86              NEORV32_CFS->REG[0] = sec; // Write sec to CFS memory-mapped register 0
87              neorv32_uart0_printf("%i: IN = 0x%x, OUT = 0x%x\n", i, sec,
                ↪  NEORV32_CFS->REG[0]);  // Read mult result from CFS memory-mapped
                ↪  register 0
88             }
89      if(i==2){
90              NEORV32_CFS->REG[0] = thi; // Write thi to CFS memory-mapped register 0
91              neorv32_uart0_printf("%i: IN = 0x%x, OUT = 0x%x\n", i, thi,
                ↪  NEORV32_CFS->REG[0]);  // Read mult result from CFS memory-mapped
                ↪  register 0
92             }
93      if(i==3){
94              NEORV32_CFS->REG[0] = fou; // Write fou to CFS memory-mapped register 0
95              neorv32_uart0_printf("%i: IN = 0x%x, OUT = 0x%x\n", i, fou,
                ↪  NEORV32_CFS->REG[0]);  // Read mult result from CFS memory-mapped
                ↪  register 0
96             }
97      }
98    // End
99    neorv32_uart0_printf("\nCFS demo program completed.\n");
100   #endif
101
102   return 0;
103 }
```

CÓDIGO D.15: CFS *unbuffered* main.c

```
-- ================================================================================ --
-- NEORV32 CPU - Co-Processor: Custom (RISC-V Instructions) Functions Unit (CFU)    --
-- -------------------------------------------------------------------------------- --
-- For custom/user-defined RISC-V instructions (R3-type, R4-type and R5-type        --
-- formats). See the  CPU's documentation for more information. Also take a look at --
-- the "software-counterpart" this default CFU hardware in 'sw/example/demo_cfu'.   --
-- -------------------------------------------------------------------------------- --
-- The NEORV32 RISC-V Processor - https://github.com/stnolting/neorv32              --
-- Copyright (c) NEORV32 contributors.                                              --
-- Copyright (c) 2020 - 2024 Stephan Nolting. All rights reserved.                  --
-- Licensed under the BSD-3-Clause license, see LICENSE for details.                --
-- SPDX-License-Identifier: BSD-3-Clause                                            --
-- ================================================================================ --

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

library neorv32;
use neorv32.neorv32_package.all;

entity neorv32_cpu_cp_cfu is
  port (
    -- global control --
    clk_i       : in  std_ulogic; -- global clock, rising edge
    rstn_i      : in  std_ulogic; -- global reset, low-active, async
    ctrl_i      : in  ctrl_bus_t; -- main control bus
    start_i     : in  std_ulogic; -- trigger operation
    -- CSR interface --
    csr_we_i    : in  std_ulogic; -- write enable
    csr_addr_i  : in  std_ulogic_vector(1 downto 0); -- address
    csr_wdata_i : in  std_ulogic_vector(XLEN-1 downto 0); -- write data
    csr_rdata_o : out std_ulogic_vector(XLEN-1 downto 0) := (others => '0'); -- read
    -- data input --
    rs1_i       : in  std_ulogic_vector(XLEN-1 downto 0); -- rf source 1
    rs2_i       : in  std_ulogic_vector(XLEN-1 downto 0); -- rf source 2
    rs3_i       : in  std_ulogic_vector(XLEN-1 downto 0); -- rf source 3
```

```
38      rs4_i       : in  std_ulogic_vector(XLEN-1 downto 0); -- rf source 4
39      -- result and status --
40      res_o       : out std_ulogic_vector(XLEN-1 downto 0) := (others => '0'); --
        ↪ operation result
41      valid_o     : out std_ulogic := '0' -- data output valid
42    );
43  end neorv32_cpu_cp_cfu;
44
45  architecture neorv32_cpu_cp_cfu_rtl of neorv32_cpu_cp_cfu is
46
47    -- CFU Control ------------------------------------------
48    -- ------------------------------------------------------------
49    type control_t is record
50      busy   : std_ulogic; -- CFU is busy
51      done   : std_ulogic; -- set to '1' when processing is done
52      result : std_ulogic_vector(XLEN-1 downto 0); -- CFU processing result (for
        ↪ write-back to register file)
53      rtype  : std_ulogic_vector(1 downto 0); -- instruction type, see constants below
54      funct3 : std_ulogic_vector(2 downto 0); -- "funct3" bit-field from custom
        ↪ instruction word
55      funct7 : std_ulogic_vector(6 downto 0); -- "funct7" bit-field from custom
        ↪ instruction word
56    end record;
57    signal control : control_t;
58
59    -- instruction format types --
60    constant r3type_c  : std_ulogic_vector(1 downto 0) := "00"; -- R3-type instructions
      ↪ (custom-0 opcode)
61    constant r4type_c  : std_ulogic_vector(1 downto 0) := "01"; -- R4-type instructions
      ↪ (custom-1 opcode)
62    constant r5typeA_c : std_ulogic_vector(1 downto 0) := "10"; -- R5-type instruction
      ↪ A (custom-2 opcode)
63    constant r5typeB_c : std_ulogic_vector(1 downto 0) := "11"; -- R5-type instruction
      ↪ B (custom-3 opcode)
64
65    -- User-Defined Logic -------------------------------------
66    -- ------------------------------------------------------------
67
68    constant N_bits : natural := 32; -- 32 bits (16 bits plus 16 bits)
69    constant Log2_elements : natural := 2; -- Log2 is 2 ergo FIFO has 4 elements
70
71    signal reset : std_logic;
72
73    type mult_wfifos_t is record
74      sreg : std_ulogic_vector(5 downto 0); -- 6 cycles latency = 6 bits in arbitration
        ↪ shift register + 1 cycle for output = 7 cycles in total
75      done : std_logic;
76      --
77      input : std_logic_vector(31 downto 0);
78      output : std_logic_vector(31 downto 0);
79      output_u  : std_ulogic_vector(31 downto 0);
80      wr : std_logic;
```

```
81      rd : std_logic;
82    end record;
83    signal mw : mult_wfifos_t;
84
85    type multp_wfifos_t is record
86      sreg : std_ulogic_vector(3 downto 0); -- 4 cycles latency = 4 bits in arbitration
        ↪  shift register + 1 cycle for output = 5 cycles in total
87      done : std_logic;
88      --
89      input : std_logic_vector(31 downto 0);
90      output : std_logic_vector(31 downto 0);
91      output_u  : std_ulogic_vector(31 downto 0);
92      wr : std_logic;
93      rd : std_logic;
94    end record;
95    signal mpw : multp_wfifos_t;
96
97    type multp_t is record
98      sreg : std_logic; -- 1 cycle latency = 1 bit in arbitration shift register + 1
        ↪  cycle for output = 2 cycles in total
99      done : std_logic;
100     --
101     input : std_logic_vector(31 downto 0);
102     output : std_logic_vector(31 downto 0);
103     output_u  : std_ulogic_vector(31 downto 0);
104   end record;
105   signal mp : multp_t;
106
107 begin
108
109   -- ************************************************************************
110   -- This controller is required to handle the CFU-CPU interface.
111   -- ************************************************************************
112
113   -- CFU Controller
      ↪  ------------------------------------------------------------------------
114   -- ------------------------------------------------------------------------
115   -- The <control> record acts as proxy logic that ensures correct communication with
      ↪  the
116   -- CPU pipeline. However, this control instance adds one additional cycle of
      ↪  latency.
117   -- Advanced users can remove this default control instance to obtain maximum
      ↪  throughput.
118   cfu_control: process(rstn_i, clk_i)
119   begin
120     if (rstn_i = '0') then
121       res_o        <= (others => '0');
122       control.busy <= '0';
123     elsif rising_edge(clk_i) then
124       res_o <= (others => '0'); -- default; all CPU co-processor outputs are
          ↪  logically OR-ed
125       if (control.busy = '0') then -- CFU is idle
```

```
126         control.busy <= start_i; -- trigger new CFU operation
127       else -- CFU operation in progress
128         res_o <= control.result; -- output result only if CFU is processing; has to
            ↪  be all-zero otherwise
129         if (control.done = '1') or (ctrl_i.cpu_trap = '1') then -- operation done or
            ↪  abort if trap (exception)
130           control.busy <= '0';
131         end if;
132       end if;
133     end if;
134   end process cfu_control;
135
136   -- CPU feedback --
137   valid_o <= control.busy and control.done; -- set one cycle before result data
138
139   -- pack user-defined instruction type/function bits --
140   control.rtype  <= ctrl_i.ir_opcode(6 downto 5);
141   control.funct3 <= ctrl_i.ir_funct3;
142   control.funct7 <= ctrl_i.ir_funct12(11 downto 5);
143
144   reset <= not(rstn_i);
145
146   -- Mult_wfifos instantation
147
148   mult_wfifos_0 : entity work.mult_wfifos
149                 generic map (N_bits => N_bits,
150                              Log2_elements => Log2_elements)
151                 port map (clk_wr => clk_i,
152                           clk_mult => clk_i,
153                           clk_rd => clk_i,
154                           rst => reset,
155                           din => mw.input,
156                           dout => mw.output,
157                           wr => mw.wr,
158                           rd => mw.rd,
159                           full => open,
160                           empty => open);
161
162   -- Multp_wfifos instantation
163
164   multp_wfifos_0 : entity work.multp_wfifos
165                  generic map (g_data_width => N_bits,
166                               g_fifo_depth => Log2_elements)
167                  port map (clk_in => clk_i,
168                            clk_mult => clk_i,
169                            clk_out => clk_i,
170                            rst => reset,
171                            din => mpw.input,
172                            dout => mpw.output,
173                            write => mpw.wr,
174                            read => mpw.rd,
175                            full => open,
```

```
176                         empty => open);
177
178   -- Multp instantation
179
180   multp_0 : entity work.multp_op
181                 generic map (g_data_width => N_bits)
182                 port map (din => mp.input,
183                           dout => mp.output);
184
185   -- Inputs
186   mw.input <= To_StdLogicVector(rs1_i)  when control.funct3 = "000" and start_i = '1'
      ↪  else
187               (others => '0');
188   mpw.input <= To_StdLogicVector(rs1_i) when control.funct3 = "001" and start_i = '1'
      ↪  else
189               (others => '0');
190   mp.input <= To_StdLogicVector(rs1_i)  when mp.done = '1' else
191               (others => '0');
192
193   -- Outputs
194   mw.output_u <= To_StdULogicVector(mw.output);
195   mpw.output_u <= To_StdULogicVector(mpw.output);
196   mp.output_u <= To_StdULogicVector(mp.output);
197
198     -- Iteration control
199     iteration_control: process(rstn_i, clk_i)
200     begin
201       if (rstn_i = '0') then
202         mw.sreg <= (others => '0');
203         mpw.sreg <= (others => '0');
204         mp.sreg <= '0';
205       elsif rising_edge(clk_i) then
206         -- operation trigger --
207         if (control.busy = '0') and -- CFU is idle (ready for next operation)
208            (start_i = '1') and -- CFU is actually triggered by a custom instruction
               ↪  word
209            (control.rtype = r3type_c) and -- this is a R3-type instruction
210            (control.funct3 = "000") then -- trigger only for 000 funct3 value
211              mw.sreg(0) <= '1';
212         elsif (control.busy = '0') and -- CFU is idle (ready for next operation)
213               (start_i = '1') and -- CFU is actually triggered by a custom
                  ↪  instruction word
214               (control.rtype = r3type_c) and -- this is a R3-type instruction
215               (control.funct3 = "001") then -- trigger only for 001 funct3 value
216              mpw.sreg(0) <= '1';
217         elsif (control.busy = '0') and -- CFU is idle (ready for next operation)
218               (start_i = '1') and -- CFU is actually triggered by a custom
                  ↪  instruction word
219               (control.rtype = r3type_c) and -- this is a R3-type instruction
220               (control.funct3 = "010") then -- trigger only for 010 funct3 value
221              mp.sreg <= '1';
222         else
```

```
223               mw.sreg(0) <= '0';
224               mpw.sreg(0) <= '0';
225               mp.sreg <= '0';
226           end if;
227           -- simple shift register for tracking operation --
228             mw.sreg(mw.sreg'left downto 1) <= mw.sreg(mw.sreg'left-1 downto 0); --
                ↪ shift left
229             mpw.sreg(mpw.sreg'left downto 1) <= mpw.sreg(mpw.sreg'left-1 downto 0); --
                ↪ shift left
230           end if;
231         end process iteration_control;
232
233         -- Processing has reached last stage (= done) when mult_wfifos sreg's MSB is
            ↪ set --
234         mw.done <= mw.sreg(mw.sreg'left);
235
236         -- Processing has reached last stage (= done) when multp_wfifos sreg's MSB is
            ↪ set --
237         mpw.done <= mpw.sreg(mpw.sreg'left);
238
239         -- Processing has reached last stage (= done) when multp sreg is equal to 1 --
240         mp.done <= mp.sreg;
241
242         -- Write signal for mult_wfifos when the operation starts
243         mw.wr <= start_i when control.funct3 = "000" else
244                  '0';
245         -- Write signal for multp_wfifos when the operation starts
246         mpw.wr <= start_i when control.funct3 = "001" else
247                  '0';
248         -- Read signal for mult_wfifos in the fifth iteration
249         mw.rd <= mw.sreg(4);
250         -- Read signal for multp_wfifos in the third iteration
251         mpw.rd <= mpw.sreg(2);
252
253 -- Output select
    ↪ --------------------------------------------------------------------------
254   -- ------------------------------------------------------------------------
255     out_select: process(control, rs1_i, rs2_i, mw, mpw, mp)
256     begin
257       case control.rtype is
258         when r3type_c => -- R3-type instructions
259           case control.funct3 is
260             when "000" => -- funct3 = "000": mult_wfifos
261               control.result <= mw.output_u;
262               control.done   <= mw.done; -- 6 cycles to perform multiplication
263             when "001" => -- funct3 = "001": multp_wfifos
264               control.result <= mpw.output_u;
265               control.done   <= mpw.done; -- 4 cycles to perform multiplication
266             when "010" => -- funct3 = "010": multp
267               control.result <= mp.output_u;
268               control.done   <= mp.done; -- 1 cycle to perform multiplication
269             when others => -- not implemented
```

```
270               control.result <= (others => '0');
271               control.done   <= '0'; -- this will cause an illegal instruction
                  ↪  exception after timeout
272             end case;
273           when others => -- undefined
274           -- ------------------------------------------------------------------------
275             control.result <= (others => '0');
276             control.done   <= '0';
277         end case;
278       end process out_select;
279
280   end neorv32_cpu_cp_cfu_rtl;
```

CÓDIGO D.16: Archivo neorv32_cpu_cp_cfu.vhd modificado para integrar los multiplicadores.

```
1   -- ================================================================================
    ↪  --
2   -- NEORV32 SoC - Custom Functions Subsystem (CFS)
    ↪  --
3   -- --------------------------------------------------------------------------------
    ↪  --
4   -- Intended for tightly-coupled, application-specific custom co-processors. This
    ↪  --
5   -- module provides up to 64x 32-bit memory-mapped interface registers, one CPU
    ↪  --
6   -- interrupt request signal and custom IO conduits for processor-external or chip-
    ↪  --
7   -- external interface.
    ↪  --
8   -- --------------------------------------------------------------------------------
    ↪  --
9   -- The NEORV32 RISC-V Processor - https://github.com/stnolting/neorv32
    ↪  --
10  -- Copyright (c) NEORV32 contributors.
    ↪  --
11  -- Copyright (c) 2020 - 2024 Stephan Nolting. All rights reserved.
    ↪  --
12  -- Licensed under the BSD-3-Clause license, see LICENSE for details.
    ↪  --
13  -- SPDX-License-Identifier: BSD-3-Clause
    ↪  --
14  -- ================================================================================
    ↪  --
15
16  library ieee;
17  use ieee.std_logic_1164.all;
18  use ieee.numeric_std.all;
19
20  library neorv32;
21  use neorv32.neorv32_package.all;
22
```

```
23 entity neorv32_cfs is
24   generic (
25     CFS_CONFIG   : std_ulogic_vector(31 downto 0); -- custom CFS configuration
       ↪ generic
26     CFS_IN_SIZE  : natural; -- size of CFS input conduit in bits
27     CFS_OUT_SIZE : natural  -- size of CFS output conduit in bits
28   );
29   port (
30     clk_i       : in  std_ulogic; -- global clock line
31     rstn_i      : in  std_ulogic; -- global reset line, low-active, use as async
32     bus_req_i   : in  bus_req_t; -- bus request
33     bus_rsp_o   : out bus_rsp_t := rsp_terminate_c; -- bus response
34     clkgen_en_o : out std_ulogic := '0'; -- enable clock generator
35     clkgen_i    : in  std_ulogic_vector(7 downto 0); -- "clock" inputs
36     irq_o       : out std_ulogic := '0'; -- interrupt request
37     cfs_in_i    : in  std_ulogic_vector(CFS_IN_SIZE-1 downto 0); -- custom inputs
38     cfs_out_o   : out std_ulogic_vector(CFS_OUT_SIZE-1 downto 0) := (others => '0')
       ↪ -- custom outputs
39   );
40 end neorv32_cfs;
41
42 architecture neorv32_cfs_rtl of neorv32_cfs is
43
44   -- MULT(P) interface registers --
45   signal cfs_mult_data : std_ulogic_vector(31 downto 0);
46   signal cfs_mult_control : std_ulogic_vector(31 downto 0);
47   signal cfs_mult_res : std_ulogic_vector(31 downto 0);
48
49 begin
50
51   clkgen_en_o <= '0'; -- not used for this minimal example
52
53   irq_o <= '0'; -- not used for this minimal example
54
55   bus_access: process(rstn_i, clk_i)
56   begin
57     if (rstn_i = '0') then
58       cfs_mult_data <= (others => '0');
59       cfs_mult_control <= (others => '0');
60       --
61       bus_rsp_o.ack  <= '0';
62       bus_rsp_o.err  <= '0';
63       bus_rsp_o.data <= (others => '0');
64     elsif rising_edge(clk_i) then -- synchronous interface for read and write
       ↪ accesses
65       -- transfer/access acknowledge --
66       bus_rsp_o.ack <= bus_req_i.stb;
67
68       -- tie to zero if not explicitly used --
69       bus_rsp_o.err <= '0';
70
71       -- defaults --
```

```
72        bus_rsp_o.data <= (others => '0'); -- the output HAS TO BE ZERO if there is no
          ↪ actual (read) access
73
74        -- bus access --
75        if (bus_req_i.stb = '1') then -- valid access cycle, STB is high for one cycle
76
77          -- write access --
78          if (bus_req_i.rw = '1') then
79            if (bus_req_i.addr(7 downto 2) = "000000") then -- address size is fixed!
80              cfs_mult_data <= bus_req_i.data; -- write to CFS memory-mapped register
                ↪ 0; MULT(P) inputs
81            end if;
82            if (bus_req_i.addr(7 downto 2) = "000001") then
83              cfs_mult_control <= bus_req_i.data; -- write to CFS memory-mapped
                ↪ register 1; MULT(P) control
84            end if;
85
86          -- read access --
87          else
88            case bus_req_i.addr(7 downto 2) is -- address size is fixed!
89              when "000000" => bus_rsp_o.data <= cfs_mult_res; -- read from CFS
                ↪ memory-mapped register 0; MULT(P) outputs
90              when others   => bus_rsp_o.data <= (others => '0');
91            end case;
92          end if;
93
94        end if;
95      end if;
96    end process bus_access;
97
98  -- cfs_mult_control(1) => Read from MULT(P)
99  -- cfs_mult_control(0) => Write to MULT(P)
100
101 -- Concatenate/make output; cfs_out_o => 34 bits (2 bit for control MSB + 32 bit for
    ↪ data)
102  cfs_out_o <= cfs_mult_control(1) & cfs_mult_control(0) & cfs_mult_data;
103
104 -- cfs_in_i => 32 bits; MULT(P) output (16 bits x 16 bits)
105
106  cfs_mult_res <= cfs_in_i;
107
108 end neorv32_cfs_rtl;
```

CÓDIGO D.17: Archivo neorv32_cfs.vhd modificado para gestionar los multiplicadores *buffered*.

```
1  -- ================================================================================
   ↪ --
2  -- NEORV32 SoC - Custom Functions Subsystem (CFS)
   ↪ --
3  -- --------------------------------------------------------------------------------
   ↪ --
```

```
4  -- Intended for tightly-coupled, application-specific custom co-processors. This
   ↪  --
5  -- module provides up to 64x 32-bit memory-mapped interface registers, one CPU
   ↪  --
6  -- interrupt request signal and custom IO conduits for processor-external or chip-
   ↪  --
7  -- external interface.
   ↪  --
8  -- ----------------------------------------------------------------------------
   ↪  --
9  -- The NEORV32 RISC-V Processor - https://github.com/stnolting/neorv32
   ↪  --
10 -- Copyright (c) NEORV32 contributors.
   ↪  --
11 -- Copyright (c) 2020 - 2024 Stephan Nolting. All rights reserved.
   ↪  --
12 -- Licensed under the BSD-3-Clause license, see LICENSE for details.
   ↪  --
13 -- SPDX-License-Identifier: BSD-3-Clause
   ↪  --
14 -- ============================================================================
   ↪  --
15
16 library ieee;
17 use ieee.std_logic_1164.all;
18 use ieee.numeric_std.all;
19
20 library neorv32;
21 use neorv32.neorv32_package.all;
22
23 entity neorv32_cfs is
24   generic (
25     CFS_CONFIG   : std_ulogic_vector(31 downto 0); -- custom CFS configuration
       ↪  generic
26     CFS_IN_SIZE  : natural; -- size of CFS input conduit in bits
27     CFS_OUT_SIZE : natural  -- size of CFS output conduit in bits
28   );
29   port (
30     clk_i       : in  std_ulogic; -- global clock line
31     rstn_i      : in  std_ulogic; -- global reset line, low-active, use as async
32     bus_req_i   : in  bus_req_t; -- bus request
33     bus_rsp_o   : out bus_rsp_t := rsp_terminate_c; -- bus response
34     clkgen_en_o : out std_ulogic := '0'; -- enable clock generator
35     clkgen_i    : in  std_ulogic_vector(7 downto 0); -- "clock" inputs
36     irq_o       : out std_ulogic := '0'; -- interrupt request
37     cfs_in_i    : in  std_ulogic_vector(CFS_IN_SIZE-1 downto 0); -- custom inputs
38     cfs_out_o   : out std_ulogic_vector(CFS_OUT_SIZE-1 downto 0) := (others => '0')
       ↪  -- custom outputs
39   );
40 end neorv32_cfs;
41
42 architecture neorv32_cfs_rtl of neorv32_cfs is
```

```
43
44    -- MULT(P) interface registers --
45    signal cfs_mult_data : std_ulogic_vector(31 downto 0);
46    signal cfs_mult_res : std_ulogic_vector(31 downto 0);
47
48  begin
49
50    clkgen_en_o <= '0'; -- not used for this minimal example
51
52    irq_o <= '0'; -- not used for this minimal example
53
54    bus_access: process(rstn_i, clk_i)
55    begin
56      if (rstn_i = '0') then
57        cfs_mult_data <= (others => '0');
58        --
59        bus_rsp_o.ack  <= '0';
60        bus_rsp_o.err  <= '0';
61        bus_rsp_o.data <= (others => '0');
62      elsif rising_edge(clk_i) then -- synchronous interface for read and write
        ↪  accesses
63        -- transfer/access acknowledge --
64        bus_rsp_o.ack <= bus_req_i.stb;
65
66        -- tie to zero if not explicitly used --
67        bus_rsp_o.err <= '0';
68
69        -- defaults --
70        bus_rsp_o.data <= (others => '0'); -- the output HAS TO BE ZERO if there is no
          ↪  actual (read) access
71
72        -- bus access --
73        if (bus_req_i.stb = '1') then -- valid access cycle, STB is high for one cycle
74
75          -- write access --
76          if (bus_req_i.rw = '1') then
77            if (bus_req_i.addr(7 downto 2) = "000000") then -- address size is fixed!
78              cfs_mult_data <= bus_req_i.data; -- write to CFS memory-mapped register
                ↪  0; MULT(P) inputs
79            end if;
80
81          -- read access --
82          else
83            case bus_req_i.addr(7 downto 2) is -- address size is fixed!
84              when "000000" => bus_rsp_o.data <= cfs_mult_res; -- read from CFS
                ↪  memory-mapped register 0; MULT(P) outputs
85              when others   => bus_rsp_o.data <= (others => '0');
86            end case;
87          end if;
88
89        end if;
90      end if;
```

```
91    end process bus_access;
92
93  -- Make output
94   cfs_out_o <= cfs_mult_data;
95
96  -- cfs_in_i => 32 bits; MULT(P) output (16 bits x 16 bits)
97
98   cfs_mult_res <= cfs_in_i;
99
100 end neorv32_cfs_rtl;
```

CÓDIGO D.18: Archivo neorv32_cfs.vhd modificado para gestionar el multiplicador *unbuffered*.

```
1   library ieee;
2   context ieee.ieee_std_context;
3
4   library neorv32;
5   use neorv32.neorv32_package.all;
6
7   library vunit_lib;
8   context vunit_lib.vunit_context;
9
10  entity tb_complex_mults_cfu is
11    generic (
12      runner_cfg : string
13    );
14  end entity;
15
16  architecture tb of tb_complex_mults_cfu is
17
18  signal clk : std_logic := '0';
19  signal rst : std_logic := '0';
20  signal rstn : std_logic := '0';
21  signal gpio_a : std_ulogic_vector(7 downto 0);
22  signal uart0_txd : std_logic;
23
24  constant baud0_rate_c            : natural := 19200;
25  constant CLOCK_FREQUENCY         : natural := 100000000;
26
27  constant uart0_baud_val_c : real := real(CLOCK_FREQUENCY) / real(baud0_rate_c);
28
29  constant clk_period : time := 10 ns;
30
31  signal ctrl_bus : ctrl_bus_t;
32
33  signal funct3 : std_ulogic_vector(2 downto 0);
34  signal rs1 : std_ulogic_vector(31 downto 0);
35  signal res_CFU : std_ulogic_vector(31 downto 0);
36  signal start_CFU : std_logic;
37
38  signal csr_we : std_logic;
```

```
39  signal csr_valid : std_logic;
40  signal csr_addr : std_ulogic_vector(11 downto 0);
41  signal csr_wdata : std_ulogic_vector(31 downto 0);
42  signal csr_rdata_o : std_ulogic_vector(31 downto 0);
43
44
45  -- Logging
46
47  constant logger : logger_t := get_logger("tb_complex_mults_cfu");
48  constant file_handler : log_handler_t := new_log_handler(
49    output_path(runner_cfg) & "log.csv",
50    format => csv,
51    use_color => false
52  );
53
54  -- Test items (make sure that they are equal to the items defined in the software)
55
56  constant test_items : natural := 4;
57  type test_t is array (0 to test_items-1, 0 to 2) of integer;
58  constant test_data : test_t := (
59    (1, 1, 1),
60    (2, 2, 4),
61    (4, 4, 16),
62    (8, 8, 64)
63  );
64
65  signal start, done: boolean := false;
66
67  begin
68
69  neorv32_mults_cfu_0 : entity work.neorv32_mults_cfu
70                                                  generic map(
71                                                              CLOCK_FREQUENCY =>
                                                                ↪ CLOCK_FREQUENCY,
72                                                              MEM_INT_IMEM_SIZE =>
                                                                ↪ 16384,
73                                                              MEM_INT_DMEM_SIZE => 8192
74                                                             )
75                                                  port map (
76                                                              clk_i => clk,
77                                                              rstn_i => rstn,
78                                                              gpio_o => gpio_a,
79                                                              uart0_txd_o => uart0_txd,
80                                                              uart0_rxd_i => uart0_txd
81                                                            );
82    -- UART Simulation Receiver
      ↪ -------------------------------------------------------------
83    --
      ↪ ------------------------------------------------------------------------------------------
84    uart0_checker: entity work.uart_rx_simple
85    generic map (
86      name => "uart0",
```

```
87      uart_baud_val_c => uart0_baud_val_c
88    )
89    port map (
90      clk => clk,
91      uart_txd => uart0_txd
92    );
93
94    clk <= not clk after clk_period/2;
95    rstn <= not rst;
96
97    -- Capture control bus through external names
98
99    ctrl_bus <= << signal
      ↪  .tb_complex_mults_cfu.neorv32_mults_cfu_0.neorv32_top_inst.core_complex.neorv32_cpu_inst
      ↪  .neorv32_cpu_alu_inst.neorv32_cpu_cp_cfu_inst_true.neorv32_cpu_cp_cfu_inst.ctrl_i
      ↪  : ctrl_bus_t >>;
100
101   -- To display in vcd file
102   funct3 <= ctrl_bus.ir_funct3;
103
104   rs1 <= << signal
      ↪  .tb_complex_mults_cfu.neorv32_mults_cfu_0.neorv32_top_inst.core_complex.neorv32_cpu_inst
      ↪  .neorv32_cpu_alu_inst.neorv32_cpu_cp_cfu_inst_true.neorv32_cpu_cp_cfu_inst.rs1_i
      ↪  : std_ulogic_vector(31 downto 0) >>;
105   res_CFU <= << signal
      ↪  .tb_complex_mults_cfu.neorv32_mults_cfu_0.neorv32_top_inst.core_complex.neorv32_cpu_inst
      ↪  .neorv32_cpu_alu_inst.neorv32_cpu_cp_cfu_inst_true.neorv32_cpu_cp_cfu_inst.res_o
      ↪  : std_ulogic_vector(31 downto 0) >>;
106   start_CFU <= << signal
      ↪  .tb_complex_mults_cfu.neorv32_mults_cfu_0.neorv32_top_inst.core_complex.neorv32_cpu_inst
      ↪  .neorv32_cpu_alu_inst.neorv32_cpu_cp_cfu_inst_true.neorv32_cpu_cp_cfu_inst.start_i
      ↪  : std_logic >>;
107
108   -- Capture CSR signals through external names
109   csr_we <= << signal
      ↪  .tb_complex_mults_cfu.neorv32_mults_cfu_0.neorv32_top_inst.core_complex.neorv32_cpu_inst
      ↪  .neorv32_cpu_control_inst.xcsr_we_o : std_logic >>;
110   csr_addr <= << signal
      ↪  .tb_complex_mults_cfu.neorv32_mults_cfu_0.neorv32_top_inst.core_complex.neorv32_cpu_inst
      ↪  .neorv32_cpu_control_inst.xcsr_addr_o : std_ulogic_vector(11 downto 0) >>;
111   csr_wdata <= << signal
      ↪  .tb_complex_mults_cfu.neorv32_mults_cfu_0.neorv32_top_inst.core_complex.neorv32_cpu_inst
      ↪  .neorv32_cpu_control_inst.xcsr_wdata_o : std_ulogic_vector(31 downto 0) >>;
112   csr_rdata_o <= << signal
      ↪  .tb_complex_mults_cfu.neorv32_mults_cfu_0.neorv32_top_inst.core_complex.neorv32_cpu_inst
      ↪  .neorv32_cpu_control_inst.csr_rdata_o : std_ulogic_vector(31 downto 0) >>;
113   csr_valid <= << signal
      ↪  .tb_complex_mults_cfu.neorv32_mults_cfu_0.neorv32_top_inst.core_complex.neorv32_cpu_inst
      ↪  .neorv32_cpu_control_inst.csr_reg_valid : std_logic >>;
114
115   main: process
116   begin
```

```
117     test_runner_setup(runner, runner_cfg);
118     while test_suite loop
119       if run("test") then
120         set_log_handlers(logger, (display_handler, file_handler));
121         show_all(logger, file_handler);
122         show_all(logger, display_handler);
123 
124         rst <= '1';
125         wait for 15*clk_period;
126         rst <= '0';
127         info(logger, "Init test");
128         wait until rising_edge(clk);
129         start <= true;
130         wait until rising_edge(clk);
131         start <= false;
132         wait until (done and rising_edge(clk));
133         info(logger, "Test done");
134       end if;
135     end loop;
136     test_runner_cleanup(runner);
137     wait;
138   end process;
139 
140   mycycle_capture: process
141   begin
142     done <= false;
143     wait until start and rising_edge(clk);
144     for x in 0 to test_items-1 loop
145       wait until rising_edge(clk) and csr_we = '0' and csr_valid = '1' and csr_addr =
        ↪  x"B00" and csr_rdata_o /= x"00000000"; -- CSR MYCYCLE ADDR IS 0xB00
146       info(logger, "Data " & to_string(x+1) & "/" & to_string(test_items) & " latency
        ↪  is " & to_string(to_integer(unsigned(csr_rdata_o))-1) & " cycles"); --
        ↪  Remove one cycle, see gh:stnolting/neorv32/issues/897
147       wait until rising_edge(clk);
148     end loop;
149 
150     wait until rising_edge(clk);
151     done <= true;
152     wait;
153   end process;
154 
155 end architecture;
```

CÓDIGO D.19: tb_complex_mults_cfu.vhd

```
1  -- Authors:
2  --   Unai Martinez-Corral & Unai Sainz-Estebanez
3  --     <unai.martinezcorral@ehu.eus>
4  --     <usainz003@ikasle.ehu.eus>
5  --
6  -- Licensed under the Apache License, Version 2.0 (the "License");
7  -- you may not use this file except in compliance with the License.
```

```
8  -- You may obtain a copy of the License at
9  --
10 --     http://www.apache.org/licenses/LICENSE-2.0
11 --
12 -- Unless required by applicable law or agreed to in writing, software
13 -- distributed under the License is distributed on an "AS IS" BASIS,
14 -- WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15 -- See the License for the specific language governing permissions and
16 -- limitations under the License.
17 --
18 -- SPDX-License-Identifier: Apache-2.0
19
20 library ieee;
21 context ieee.ieee_std_context;
22
23 library vunit_lib;
24 context vunit_lib.vunit_context;
25 context vunit_lib.vc_context;
26
27 entity mult_wfifos_axis_vcs is
28   generic (
29     m_axis : axi_stream_master_t;
30     s_axis : axi_stream_slave_t;
31     N_bits : natural := 32;
32     Log2_elements : natural := 4;
33     test_items : natural := 4;
34     logger : logger_t
35   );
36   port (
37     clk, rstn: in std_logic
38   );
39 end entity;
40
41 architecture arch of mult_wfifos_axis_vcs is
42
43   signal m_valid, m_ready, s_valid, s_ready : std_logic;
44   signal m_data, s_data : std_logic_vector(data_length(m_axis)-1 downto 0);
45
46 begin
47
48   vunit_axism: entity vunit_lib.axi_stream_master
49   generic map (
50     master => m_axis
51   )
52   port map (
53     aclk   => clk,
54     tvalid => m_valid,
55     tready => m_ready,
56     tdata  => m_data,
57     tlast  => open
58   );
59
```

```
60    vunit_axiss: entity vunit_lib.axi_stream_slave
61    generic map (
62      slave => s_axis
63    )
64    port map (
65      aclk   => clk,
66      tvalid => s_valid,
67      tready => s_ready,
68      tdata  => s_data,
69      tlast  => open
70    );
71
72  --
73
74    uut: entity work.mult_wfifos_axis
75    generic map (
76      N_bits => N_bits,
77      Log2_elements => Log2_elements
78    )
79    port map (
80      clk_mult => clk,
81      s_axis_clk   => clk,
82      s_axis_rstn  => rstn,
83      s_axis_rdy   => m_ready,
84      s_axis_data  => m_data,
85      s_axis_valid => m_valid,
86      m_axis_clk   => clk,
87      m_axis_rstn  => rstn,
88      m_axis_valid => s_valid,
89      m_axis_data  => s_data,
90      m_axis_rdy   => s_ready
91    );
92
93  -- To extract time information through the INFO function, for latency measurements
94
95    send_trigger: process
96    begin
97
98      for x in 0 to test_items-1 loop
99        wait until rising_edge(clk) and m_valid = '1' and m_ready = '1';
100       info(logger, "Data (" & to_string(m_data(31 downto 16)) & "x" &
          ↪  to_string(m_data(15 downto 0))  & ") " & to_string(x+1) & "/" &
          ↪  to_string(test_items) & " sent!");
101     end loop;
102   end process;
103
104   received_trigger: process
105   begin
106
107     for x in 0 to test_items-1 loop
108       wait until rising_edge(clk) and s_valid = '1' and s_ready = '1';
```

```
109       info(logger, "Data (" & to_string(s_data) & ") " & to_string(x+1) & "/" &
          ↪  to_string(test_items) & " received!");
110     end loop;
111   end process;
112
113 end architecture;
```

CÓDIGO D.20: mult_wfifos_axis_vcs.vhd

```
1  -- Authors:
2  --   Unai Martinez-Corral & Unai Sainz-Estebanez
3  --     <unai.martinezcorral@ehu.eus>
4  --     <usainz003@ikasle.ehu.eus>
5  --
6  -- Licensed under the Apache License, Version 2.0 (the "License");
7  -- you may not use this file except in compliance with the License.
8  -- You may obtain a copy of the License at
9  --
10 --     http://www.apache.org/licenses/LICENSE-2.0
11 --
12 -- Unless required by applicable law or agreed to in writing, software
13 -- distributed under the License is distributed on an "AS IS" BASIS,
14 -- WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15 -- See the License for the specific language governing permissions and
16 -- limitations under the License.
17 --
18 -- SPDX-License-Identifier: Apache-2.0
19
20 library ieee;
21 context ieee.ieee_std_context;
22
23 library vunit_lib;
24 context vunit_lib.vunit_context;
25 context vunit_lib.vc_context;
26
27 entity tb_mult_wfifos_axis_latency is
28   generic (
29     runner_cfg : string
30   );
31 end entity;
32
33 architecture tb of tb_mult_wfifos_axis_latency is
34
35   -- Simulation constants
36
37   constant clk_period : time    := 10 ns;
38   constant data_width : natural := 32;
39
40   -- AXI4Stream Verification Components
41
42   constant master_axi_stream : axi_stream_master_t := new_axi_stream_master(
43     data_length => data_width,
```

```
44      stall_config => new_stall_config(0.0, 1, 10)
45    );
46    constant slave_axi_stream  : axi_stream_slave_t  := new_axi_stream_slave(
47      data_length => data_width,
48      stall_config => new_stall_config(0.0, 1, 10)
49    );
50
51    -- Logging
52
53    constant logger : logger_t := get_logger("tb_mult_wfifos_axis_latency");
54    constant file_handler : log_handler_t := new_log_handler(
55      output_path(runner_cfg) & "log.csv",
56      format => csv,
57      use_color => false
58    );
59
60    -- tb signals and variables
61
62    signal clk, rst, rstn : std_logic := '0';
63    signal start, done : boolean := false;
64
65    constant test_items : natural := 4;
66    type test_t is array (0 to test_items-1, 0 to 2) of integer;
67    constant test_data : test_t := (
68      (1, 1, 1),
69      (2, 2, 4),
70      (4, 4, 16),
71      (8, 8, 64)
72    );
73
74  begin
75
76    clk <= not clk after clk_period/2;
77    rstn <= not rst;
78
79    main: process
80    begin
81      test_runner_setup(runner, runner_cfg);
82      while test_suite loop
83        if run("test") then
84          set_log_handlers(logger, (display_handler, file_handler));
85          show_all(logger, file_handler);
86          show_all(logger, display_handler);
87
88          rst <= '1';
89          wait for 15*clk_period;
90          rst <= '0';
91          info(logger, "Init test");
92          wait until rising_edge(clk);
93          start <= true;
94          wait until rising_edge(clk);
95          start <= false;
```

```
96          wait until (done and rising_edge(clk));
97          info(logger, "Test done");
98        end if;
99      end loop;
100     test_runner_cleanup(runner);
101     wait;
102   end process;
103
104   stimuli: process
105     variable word : std_logic_vector(data_width-1 downto 0);
106     variable o : std_logic_vector(31 downto 0);
107     variable last : std_logic:='0';
108   begin
109     done <= false;
110     wait until start and rising_edge(clk);
111
112     for x in 0 to test_items-1 loop
113       word(data_width-1 downto data_width/2) :=
        ↪  std_logic_vector(to_signed(test_data(x, 0), data_width/2));
114       word(data_width/2-1 downto 0) := std_logic_vector(to_signed(test_data(x, 1),
        ↪  data_width/2));
115       push_axi_stream(net, master_axi_stream, word);
116       pop_axi_stream(net, slave_axi_stream, tdata => o, tlast => last);
117       check_equal(signed(o),to_signed(test_data(x,2), data_width),"This is a
        ↪  failure!");
118     end loop;
119
120     wait until rising_edge(clk);
121     done <= true;
122     wait;
123   end process;
124
125   uut_vc: entity work.mult_wfifos_axis_vcs
126   generic map (
127     m_axis => master_axi_stream,
128     s_axis => slave_axi_stream,
129     N_bits => data_width,
130     Log2_elements => 3,
131     test_items => test_items,
132     logger => logger
133   )
134   port map (
135     clk  => clk,
136     rstn => rstn
137   );
138
139 end architecture;
```

CÓDIGO D.21: tb_mult_wfifos_axis_latency.vhd

```
1 #!/usr/bin/env python3
2
```

```
3  from _csv import Error, __version__, writer, reader, register_dialect
4  from pathlib import Path
5  import re
6  import types
7  import csv
8  import os
9
10 # This program reads the simulation output csv and displays the latency of the
   ↪ operations
11 # Run this script after running run.py
12
13 # The path of the folder where the csv files are located is defined
14 ROOT = Path(__file__).parent
15 csv_path = ROOT / "vunit_out" / "outcsv"
16
17 # To choose the design an environment variable is defined. 'mult' value by default.
18 DESIGN = os.environ.get("DESIGN","mult")
19
20 # Specify the csv output file according to envvar
21 if DESIGN == "mult":
22     csv_file = csv_path / "tb_mult_wfifos_axis_latency.csv"
23 elif DESIGN == "multp-wfifos":
24     csv_file = csv_path / "tb_multp_wfifos_axis_latency.csv"
25 elif DESIGN == "multp":
26     csv_file = csv_path / "tb_multp_axis_latency.csv"
27 else:
28     print("The valid envvar values are: mult, multp-wfifos and multp")
29     exit()
30
31 # The lists are defined
32 time=[]
33 time_num=[]
34 line=[]
35 sent=[]
36 received=[]
37
38 # Function to open csv and read from it
39 with open(csv_file, newline='') as csvfile:
40     # Reading function; ',' delimeter is selected
41     csv_reader = csv.reader(csvfile, delimiter=',', quotechar='|')
42     # Take the first value of the fourth column (Name of vhdl file where the csv
       ↪ comes from)
43     name=next(csv_reader)[3]
44     # Reset the csv file handle
45     csvfile.seek(0)
46     # Loop through the csv file
47     for row in csv_reader:
48         # Filling the time/line lists with the second column (time of the operations)
           ↪ and the fifth column (line where the operation comes from)
49         time.append(row[1])
50         line.append(row[4])
51
```

```
52 # Loop for remove " fs"
53 for i in range(0, len(time)):
54     a=time[i]
55     a=a.replace(" fs","")
56     time_num.append(a)
57 
58 # Define the matrix with the time and lines information
59 matrix = [time_num,line]
60 
61 # Filling the sent and the received lists
62 # Note that to have the line information the vu.enable_location_preprocessing()
   ↪ function must be in the program run.py
63 for k in range(0,len(line)):
64     if int(matrix[1][k]) == 100: # If the time information comes from line 99 is sent
      ↪ information
65         sent.append(matrix[0][k]) # Save the sent time information
66     elif int(matrix[1][k]) == 109: # If the time information comes from line 108 is
      ↪ received information
67         received.append(matrix[0][k]) # Save the received time information
68 
69 if len(sent) != len(received):
70     print("Error: the sent length is not equal to the received length")
71     exit()
72 
73 # Print the name of vhdl file where the csv comes from
74 print("---- For",name,"file ----")
75 
76 # Loop for display the latency of the operations
77 for z in range(0,len(sent)):
78     # Casting from string to integer
79     k = int(received[z]) - int(sent[z])
80     if z == 0:
81         print("The latency is:",k,"fs -",int(k/10000000),"cycles")
82     # Send/received data latency
83     print("Data",z+1,"of",len(sent),"sent/received latency is:",k,"fs
      ↪ -",int(k/10000000),"cycles")
```

CÓDIGO D.22: latency.py

```
1  #!/usr/bin/env bash
2  
3  set -ex
4  
5  cd $(dirname "$0")
6  
7  if [[ -z "${Board}" ]]; then
8    Arty='35t'
9  elif [[ $Board == '35t' ]]; then
10   Arty='35t'
11 elif [[ $Board == '100t' ]]; then
12   Arty='100t'
13 else
```

```
14    echo "Error Board must be 35t or 100t"
15    exit
16  fi
17  echo "Selected board is" $Arty
18
19  apt update -qq
20
21  apt install -y git
22
23  cd ../..
24
25  git clone --recursive https://github.com/stnolting/neorv32-setups
26
27  mv rtl/mult/CFU/neorv32_application_image.vhd neorv32-setups/neorv32/rtl/core
28  mv rtl/mult/CFU/neorv32_cpu_cp_cfu.vhd neorv32-setups/neorv32/rtl/core
29
30  mkdir -p build
31
32  echo "Analyze NEORV32 CPU + MULT(P) via CFU"
33
34  ghdl -i --std=08 --workdir=build --work=neorv32
    ↪  ./neorv32-setups/neorv32/rtl/core/*.vhd
35  ghdl -i --std=08 --workdir=build --work=neorv32
    ↪  ./neorv32-setups/neorv32/rtl/core/mem/neorv32_dmem.default.vhd
36  ghdl -i --std=08 --workdir=build --work=neorv32
    ↪  ./neorv32-setups/neorv32/rtl/core/mem/neorv32_imem.default.vhd
37  ghdl -i --std=08 --workdir=build --work=neorv32 ./rtl/mult/*.vhd
38  ghdl -i --std=08 --workdir=build --work=neorv32 ./rtl/multp/*.vhd
39  ghdl -i --std=08 --workdir=build --work=neorv32 ./rtl/mult/CFU/neorv32_mults_cfu.vhd
40  ghdl -m --std=08 --workdir=build --work=neorv32 neorv32_mults_cfu
41
42  echo "Synthesis with yosys and ghdl as module"
43
44  yosys -m ghdl -p 'ghdl --std=08 --workdir=build --work=neorv32 neorv32_mults_cfu;
    ↪  synth_xilinx -nodsp -nolutram -flatten -abc9 -arch xc7 -top neorv32_mults_cfu;
    ↪  write_json neorv32_mults_cfu.json'
45
46  if [[ $Arty == '35t' ]]; then
47    echo "Place and route"
48    nextpnr-xilinx --chipdb /usr/local/share/nextpnr/xilinx-chipdb/xc7a35t.bin --xdc
      ↪  impl/nextpnr/arty.xdc --json neorv32_mults_cfu.json --write
      ↪  neorv32_mults_cfu_routed.json --fasm neorv32_mults_cfu.fasm
49    echo "Generate bitstream"
50    ../../prjxray/utils/fasm2frames.py --part xc7a35tcsg324-1 --db-root
      ↪  /usr/local/share/nextpnr/prjxray-db/artix7 neorv32_mults_cfu.fasm >
      ↪  neorv32_mults_cfu.frames
51    ../../prjxray/build/tools/xc7frames2bit --part_file
      ↪  /usr/local/share/nextpnr/prjxray-db/artix7/xc7a35tcsg324-1/part.yaml
      ↪  --part_name xc7a35tcsg324-1 --frm_file neorv32_mults_cfu.frames --output_file
      ↪  neorv32_mults_cfu_35t.bit
52  elif [[ $Arty == '100t' ]]; then
53    echo "Place and route"
```

```
54    nextpnr-xilinx --chipdb /usr/local/share/nextpnr/xilinx-chipdb/xc7a100t.bin --xdc
      ↪  impl/nextpnr/arty.xdc --json neorv32_mults_cfu.json --write
      ↪  neorv32_mults_cfu_routed.json --fasm neorv32_mults_cfu.fasm
55    echo "Generate bitstream"
56    ../../prjxray/utils/fasm2frames.py --part xc7a100tcsg324-1 --db-root
      ↪  /usr/local/share/nextpnr/prjxray-db/artix7 neorv32_mults_cfu.fasm >
      ↪  neorv32_mults_cfu.frames
57    ../../prjxray/build/tools/xc7frames2bit --part_file
      ↪  /usr/local/share/nextpnr/prjxray-db/artix7/xc7a100tcsg324-1/part.yaml
      ↪  --part_name xc7a100tcsg324-1 --frm_file neorv32_mults_cfu.frames --output_file
      ↪  neorv32_mults_cfu_100t.bit
58  fi
59
60  echo "Implementation completed"
```

CÓDIGO D.23: impl_mults_cfu.sh

```
1   ----------------------------------------------------------------------------------
2   -- Company: UPV/EHU
3   -- Engineer: Koldo Basterretxea & Unai Sainz
4   --
5   -- Create Date: 25.06.2024
6   -- Design Name:
7   -- Module Name: top_CRIsig_vunit - RTL
8   -- Project Name:
9   -- Target Devices:
10  -- Tool Versions:
11  -- Description:
12  --
13  -- Dependencies: Fixed point package --
14  -- Additional Comments: parameters are WL (word-length) and prec (arithmetic
    ↪  precision)
15  -- Programmable slope and saturatio and applied interpolation depth (q) in runtime
16  -- Latency: q+2 clock cycles (this version registers inut and outputs). q+1 if the
    ↪  output or input register is avoided
17  --
18  -- Revision:
19  -- Revision 0.01 - File Created
20  -- Additional Comments:
21  --
22  ----------------------------------------------------------------------------------
23
24  library ieee;
25  use ieee.std_logic_1164.all;
26  use ieee.numeric_std.all;
27  use ieee.fixed_pkg.all;
28  use ieee.fixed_float_types.all;
29
30
31  entity top_CRIsig_vunit is
32   generic (WL: natural := 16; --input&output WL
33            prec : natural := 8); --internal precision (frcational bits)
```

```
34      Port ( clk : in STD_LOGIC;
35             rst : in STD_LOGIC;
36             ce : in STD_LOGIC;
37             q : in STD_LOGIC_VECTOR (2 downto 0);
38             slope : in STD_LOGIC_VECTOR (2 downto 0);
39             satu : in STD_LOGIC_VECTOR (2 downto 0);
40             in_x : in STD_LOGIC_VECTOR (WL-1 downto 0);
41             out_y : out STD_LOGIC_VECTOR (WL-1 downto 0);
42             done : out STD_LOGIC);
43  end top_CRIsig_vunit;
44
45  architecture RTL of top_CRIsig_vunit is
46  --constrained std_logic_vector to avoid indexing errors in type conversion of u_fixed
    ↪  to std_logic_vector
47  subtype result_type is std_logic_vector (WL-1 downto 0);
48  --        Load optimum delta values in a ROM (16 bits (u,2,14))
49  type rom_type is array (1 to 6) of ufixed(1 downto -14);
50  constant rom_sig : rom_type :=
    ↪  (to_ufixed(1.2364,1,-14),to_ufixed(1.1244,1,-14),to_ufixed(1.0636,1,-14)
    ↪  ,to_ufixed(1.0552,1,-14),to_ufixed(1.0512,1,-14),to_ufixed(1.0512,1,-14));
51  -- constant rom_tan : rom_type :=
    ↪  (to_ufixed(2.4728,1,-14),to_ufixed(2.2488,1,-14),to_ufixed(2.1268,1,-14)
    ↪  ,to_ufixed(2.1108,1,-14),to_ufixed(2.1028,1,-14),to_ufixed(2.1020,1,-14));
52
53  --Since we compute not only squashing functions, in this version we consider WL-prec
    ↪  integer bits
54  --for inputs and outputs
55  SIGNAL counter : unsigned(2 downto 0);
56  SIGNAL rl,ssat, sslope : integer range -3 to 7;
57  SIGNAL sel_reg, sel: STD_LOGIC;
58  SIGNAL rom_out : ufixed (4 downto -prec);--min integer bits depend on allowed sat
    ↪  params
59  SIGNAL delta, delta_reg : ufixed (4 downto -prec);--min integer bits depend on
    ↪  allowed sat params
60  SIGNAL input_abs, reg_input, reg_out, output : std_logic_vector(WL-1 downto 0);
61  SIGNAL input_pos, y21, y22: ufixed (WL-prec-1 downto -prec);
62  SIGNAL y2, h, hx, hxp, hxp_reg, hxp_a, hxp_s, g, gx, min_mux: ufixed (6 downto
    ↪  -prec); -- max sat value is 2^7
63  SIGNAL outpre, resta : ufixed (WL-prec-1 downto -prec);
64  SIGNAL outpre2, resta2 : sfixed (WL-prec-1 downto -prec);
65  SIGNAL out_neg, out_CRI : sfixed (WL-prec-1 downto -prec);
66  SIGNAL zeroes: std_logic_vector (WL-prec-3 downto 0):= (others => '0');
67
68  begin
69          rl <= to_integer(unsigned(q));
70          sslope <= to_integer(signed(slope));
71          ssat <= to_integer(signed(satu));
72
73          ------------------------------------------------------------
74          --------------------COUNTER(q) and status-----------------
75          recursion: PROCESS(clk)
76          BEGIN
```

```
77              IF(rising_edge(clk)) THEN
78                  IF(rst='1') OR (ce='1')THEN --start counting when input is registered
79                      counter <= "000";
80                      sel<='0';
81                      done<='0';
82                  ELSIF(counter = rl) THEN
83                          counter <= "000" ;
84                          sel<='0';
85                          done<='1';
86                  ELSE
87                      counter<=counter+1;
88                      sel<='1';
89                      done<='0';
90                  END IF;
91              END IF;
92          END PROCESS;
93          -------------------------------------------------------------
94          ----------------------DELTA--------------------------------
95
96          rom_out <= resize(rom_sig(3),rom_out) sll ssat;  --adjust interpolation depth
            ↪  according to saturation value
97
98          WITH sel SELECT
99               delta <= resize(rom_out,delta) WHEN '0', --First value of delta
100                                 delta_reg WHEN '1',
101                            (others=>'0') when others;
102
103         delta_gen: PROCESS(clk)
104             BEGIN
105                 IF(rising_edge(clk)) THEN
106                     IF (rst='1') THEN
107                         delta_reg <= (others=>'0');
108                     ELSE
109                       delta_reg <= shift_right(delta,2); --divide by four (delta/4)
110                     END IF;
111                 END IF;
112             END PROCESS;
113         -------------------------------------------------------------
114         -- This version registers input (with ce signal)
115         reg_in: PROCESS(clk)
116             BEGIN
117                 IF(rising_edge(clk)) THEN
118                     IF (rst='1') THEN
119                        reg_input <= (others=>'0');
120                     ELSIF (ce='1') THEN
121                        reg_input <= in_x;
122                     END IF;
123                 END IF;
124             END PROCESS;
125         -------------------------------------------------------------
126         -- Convert inputs to positive values u(16,8)
127
```

```
128         input_abs <= std_logic_vector(signed(not(reg_input)) + 1);  --2's complement
129         WITH reg_input(WL-1) SELECT
130              input_pos <= to_ufixed(reg_input,input_pos) WHEN '0',
131                           to_ufixed(input_abs,input_pos) WHEN '1',
132                                        (others=>'0') when others;
133
134         ------------------------------------------------------------------------
135         -------------------------------CRI--------------------------------------
136         -- Initila affine functions
137         -- y2 = (satu/2)*(1+((slope/2)*x));
138         -- y2 <= resize(resize(resize(input_pos sll sslope-1,input_pos)+1,input_pos)
            ↪  sll ssat-1,y2);
139         y21 <= resize(input_pos sll sslope-1,input_pos);
140         y22 <= resize(y21+1,input_pos);
141         y2 <= resize(y22 sll ssat-1,y2);
142         -- compute g(x) at each iteration
143         WITH sel SELECT
144                   g <= y2 WHEN '0',         --g(x)=y2(x) at first iteration
145                   min_mux WHEN '1',  --g(x)=Min[g(x),h(x)] remainding iterations
146             (others=>'0') when others;
147
148         reg_gx: PROCESS(clk) --register g(x)
149             BEGIN
150                 IF(rising_edge(clk)) THEN
151                     IF (rst='1') THEN
152                         gx <= (others=>'0');
153                     ELSE
154                         gx <= g;
155                     END IF;
156                 END IF;
157             END PROCESS;
158
159         -- control signal 'sel' must be delayed one cycle
160         h_select: PROCESS(clk)
161             BEGIN
162                 IF(rising_edge(clk)) THEN
163                     IF (rst='1') THEN
164                         sel_reg <= '1';
165                     ELSE
166                         sel_reg <= sel;
167                     END IF;
168                 END IF;
169             END PROCESS;
170
171         -- Compute h(x) at each iteration
172
173         h <= resize(to_ufixed(1,h) sll ssat,h);
174
175         WITH sel_reg SELECT
176                       hx <= h WHEN '0', --h(x)=y3(x)=sat initially
177                       hxp_reg WHEN '1',
178                 (others=>'0') when others;
```

```
179
180         --h(x)=1/2(g(x)+h(x)-delta)
181         hxp_a <= resize(gx+hx, hxp_a);
182         hxp_s <= resize(hxp_a - delta_reg, hxp_s);
183         --hxp <= '0' & hxp_s(1 DOWNTO -prec+1);
184         hxp <= hxp_s sra 1;
185
186         reg_hxp: PROCESS(clk)
187             BEGIN
188                 IF(rising_edge(clk)) THEN
189                     IF (rst='1') THEN
190                         hxp_reg <= (others=>'0');
191                     ELSE
192                         hxp_reg <= hxp;
193                     END IF;
194                 END IF;
195             END PROCESS;
196
197         -- Min operation
198         min_mux <= gx when gx <= hx else
199                    hx;
200         ------------------------------------------------------------
201         ---------------------OUTPUT---------------------------------
202
203         outpre <= resize(min_mux,outpre);
204         resta <= resize(h, resta);
205
206         -- Transform to signed for substraction
207         resta2  <= to_sfixed(resta (WL-prec-2 downto -prec));
208         outpre2 <= to_sfixed(outpre(WL-prec-2 downto -prec));
209
210         out_neg <= resize(resta2-outpre2,out_neg);
211
212         WITH reg_input(WL-1) SELECT -- negative inputs
213                out_CRI <= resize(to_sfixed(outpre),out_CRI) WHEN '0',
214                --out_CRI <= resize(add_sign(outpre),out_CRI) WHEN '0',
215                                                    out_neg WHEN '1',
216                                                (others=>'0') when others;
217
218         -- output casting(WL-prec integer bits and prec frac bits)
219         output <= result_type(out_CRI);
220         ------------------------------------------------------------
221 --        -- This version registers output (with sel signal)
222 --        reg_output: PROCESS(clk)
223 --            BEGIN
224 --                IF(rising_edge(clk)) THEN
225 --                    IF (rst='1') THEN
226 --                       reg_out <= (others=>'0');
227 --                    ELSIF (sel='0') THEN
228 --                      reg_out <= output;
229 --                END IF;
230 --            END IF;
```

```
231 --        END PROCESS;
232
233         out_y <= output;
234 end RTL;
```

CÓDIGO D.24: top_CRIsig_vunit.vhd

```
1  -- ================================================================================
   ↪  --
2  -- NEORV32 CPU - Co-Processor: Custom (RISC-V Instructions) Functions Unit (CFU)
   ↪  --
3  -- --------------------------------------------------------------------------------
   ↪  --
4  -- For custom/user-defined RISC-V instructions See the  CPU's documentation for
   ↪  --
5  -- more information. Also take a look at the "software-counterpart" this default
   ↪  --
6  -- CFU hardware in 'sw/example/demo_cfu'.
   ↪  --
7  -- --------------------------------------------------------------------------------
   ↪  --
8  -- The NEORV32 RISC-V Processor - https://github.com/stnolting/neorv32
   ↪  --
9  -- Copyright (c) NEORV32 contributors.
   ↪  --
10 -- Copyright (c) 2020 - 2024 Stephan Nolting. All rights reserved.
   ↪  --
11 -- Licensed under the BSD-3-Clause license, see LICENSE for details.
   ↪  --
12 -- SPDX-License-Identifier: BSD-3-Clause
   ↪  --
13 -- ================================================================================
   ↪  --
14
15 library ieee;
16 use ieee.std_logic_1164.all;
17 use ieee.numeric_std.all;
18
19 entity neorv32_cpu_cp_cfu is
20   port (
21     -- global control --
22     clk_i       : in  std_ulogic; -- global clock, rising edge
23     rstn_i      : in  std_ulogic; -- global reset, low-active, async
24     -- operation control --
25     start_i     : in std_ulogic; -- operation trigger/strobe
26     active_i    : in std_ulogic; -- operation in progress, CPU is waiting for CFU
27     -- CSR interface --
28     csr_we_i    : in  std_ulogic; -- write enable
29     csr_addr_i  : in  std_ulogic_vector(1 downto 0); -- address
30     csr_wdata_i : in  std_ulogic_vector(31 downto 0); -- write data
31     csr_rdata_o : out std_ulogic_vector(31 downto 0) := (others => '0'); -- read data
32     -- operands --
```

```
33      rtype_i     : in std_ulogic; -- instruction type (R3-type or R4-type)
34      funct3_i    : in std_ulogic_vector(2 downto 0); -- "funct3" bit-field from custom
        ↪ instruction word
35      funct7_i    : in std_ulogic_vector(6 downto 0); -- "funct7" bit-field from custom
        ↪ instruction word
36      rs1_i       : in  std_ulogic_vector(31 downto 0); -- rf source 1
37      rs2_i       : in  std_ulogic_vector(31 downto 0); -- rf source 2
38      rs3_i       : in  std_ulogic_vector(31 downto 0); -- rf source 3
39      -- result and status --
40      result_o    : out std_ulogic_vector(31 downto 0) := (others => '0'); -- operation
        ↪ result
41      valid_o     : out std_ulogic := '0' -- result valid, operation done; set one
        ↪ cycle before result_o is valid
42    );
43  end neorv32_cpu_cp_cfu;
44
45  architecture neorv32_cpu_cp_cfu_rtl of neorv32_cpu_cp_cfu is
46
47    -- CFU instruction type formats --
48    constant r3type_c : std_ulogic := '0'; -- R3-type CFU instructions (custom-0
      ↪ opcode)
49    constant r4type_c : std_ulogic := '1'; -- R4-type CFU instructions (custom-1
      ↪ opcode)
50
51    constant WL   : natural := 16;     -- Input/Output Word Length
52    constant prec : natural := 8;      -- Internal precision (frcational bits)
53
54    signal reset : std_logic;
55
56    -- processing logic --
57    type sig_t is record
58      ce        : std_logic;
59      done      : std_logic;
60      q         : std_logic_vector(2 downto 0);
61      slope     : std_logic_vector(2 downto 0);
62      satu      : std_logic_vector(2 downto 0);
63      input     : std_logic_vector(WL-1 downto 0);
64      output    : std_logic_vector(WL-1 downto 0);
65      output_u  : std_ulogic_vector(31 downto 0);
66      fill      : std_ulogic_vector(32-WL-1 downto 0);
67    end record;
68    -- Define the signal and initialize it
69    signal sig : sig_t := (ce       => '0',
70                           done     => '0',
71                           q        => "011", -- q=3
72                           slope    => "111", -- slope=0.5 (shift -1)
73                           satu     => "001", -- sat=2 (shift 1)
74                           input    => (others => '0'),
75                           output   => (others => '0'),
76                           output_u => (others => '0'),
77                           fill     => (others => '0'));
78
```

```
79  begin
80
81    reset <= not(rstn_i) or not(active_i);
82
83    -- top_CRIsig_vunit instantation
84
85    top_CRIsig_vunit_0 : entity work.top_CRIsig_vunit
86                         generic map (WL   => WL,
87                                      prec => prec)
88                         port map (clk     => clk_i,
89                                   rst     => reset,
90                                   ce      => sig.ce,
91                                   q       => sig.q,
92                                   slope   => sig.slope,
93                                   satu    => sig.satu,
94                                   in_x    => sig.input,
95                                   out_y   => sig.output,
96                                   done    => sig.done);
97
98    -- Input
99    sig.input <= To_StdLogicVector(rs1_i(WL-1 downto 0)) when funct3_i = "000" and
      ↪  rtype_i = r3type_c and start_i = '1' else
100             (others => '0');
101
102   -- Output
103   out_reg: process(rstn_i,clk_i)
104   begin
105     if(rstn_i='0') then
106       sig.output_u <= (others => '0');
107     elsif rising_edge(clk_i) then
108         if(sig.done = '1') then
109           sig.output_u <= sig.fill & To_StdULogicVector(sig.output);
110         else
111           sig.output_u <= (others => '0');
112         end if;
113     end if;
114   end process out_reg;
115
116   -- CE signal
117   sig.ce <= '1' when funct3_i = "000" and rtype_i = r3type_c and (start_i = '1' or
      ↪  sig.done = '1') else
118             '0';
119
120   -- Function Result Select
      ↪  --------------------------------------------------------------------
121   -- --------------------------------------------------------------------------
122   result_select: process(rtype_i, funct3_i, sig)
123   begin
124     case rtype_i is -- check instruction type
125
126       when r3type_c => -- R3-type instructions; function select via "funct3" and
          ↪  ""funct7
```

```
127       -- ------------------------------------------------------------------------
128         case funct3_i is -- Just check "funct3" here; "funct7" bit-field is ignored
129           when "000" =>
130             result_o <= sig.output_u;
131             valid_o  <= sig.done;
132           when others => -- all unspecified operations
133             result_o <= (others => '0'); -- no logic implemented
134             valid_o  <= '0'; -- this will cause an illegal instruction exception
                ↪  after timeout
135         end case;
136       when others => -- undefined
137       -- ------------------------------------------------------------------------
138         result_o <= (others => '0'); -- no logic implemented
139         valid_o  <= '0'; -- this will cause an illegal instruction exception after
            ↪  timeout
140
141     end case;
142   end process result_select;
143
144 end neorv32_cpu_cp_cfu_rtl;
```

CÓDIGO D.25: Archivo neorv32_cpu_cp_cfu.vhd modificado para integrar el coprocesador sigmoide CRI.

```
1  #!/usr/bin/env python3
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  # Definir la función original
7  def original_function(x):
8      return 2 * (1 / (1 + np.exp(-0.5 * x)))
9
10 # Definir el rangos de x
11 x_range = np.linspace(-10, 10, 100)
12
13 # Calcular los valores de la función original en eso rango
14 y_range = original_function(x_range)
15
16 # Ajustar polinomios de grado 3, 5 y 7
17 poly_3 = np.polyfit(x_range, y_range, 3)
18 poly_5 = np.polyfit(x_range, y_range, 5)
19 poly_7 = np.polyfit(x_range, y_range, 7)
20
21 # Evaluar los polinomios ajustados en el rango respectivo
22 y_poly_3 = np.polyval(poly_3, x_range)
23 y_poly_7 = np.polyval(poly_7, x_range)
24 y_poly_5 = np.polyval(poly_5, x_range)
25
26 # Imprimir los coeficientes de los polinomios de aproximación
27 print("Polinomio de aproximación de grado 3 para el rango [-10, 10]:")
```

```
print(f"{poly_3[3]:.11f} + {poly_3[2]:.11f}x + {poly_3[1]:.11f}x^2 + {poly_3[0]:.11f}x^3")
print("Polinomio de aproximación de grado 5 para el rango [-10, 10]:")
print(f"{poly_5[5]:.11f} + {poly_5[4]:.11f}x + {poly_5[3]:.11f}x^2 + {poly_5[2]:.11f}x^3 + {poly_5[1]:.11f}x^4 + {poly_5[0]:.11f}x^5")
print("Polinomio de aproximación de grado 7 para el rango [-10, 10]:")
print(f"{poly_7[7]:.11f} + {poly_7[6]:.11f}x + {poly_7[5]:.11f}x^2 + {poly_7[4]:.11f}x^3 + {poly_7[3]:.11f}x^4 + {poly_7[2]:.11f}x^5 + {poly_7[1]:.11f}x^6 + {poly_7[0]:.11f}x^7")

# Visualizar los resultados para los valores testeados

data=[-10,-7.5,-5,-2.5,0,2.5,5,7.5,10]

#for i in range(0,9):
#    print("Aproximación Grado 7;   f("+ str(data[i]) +"):", poly_7[7] + poly_7[6]*data[i] + poly_7[5]*data[i]**2 + poly_7[4]*data[i]**3 + poly_7[3]*data[i]**4 + poly_7[2]*data[i]**5 + poly_7[1]*data[i]**6 + poly_7[0]*data[i]**7)
#    print("Aproximación Grado 5;   f("+ str(data[i]) +"):", poly_5[5] + poly_5[4]*data[i] + poly_5[3]*data[i]**2 + poly_5[2]*data[i]**3 + poly_5[1]*data[i]**4 + poly_5[0]*data[i]**5)
#    print("Aproximación Grado 3;   f("+ str(data[i]) +"):", poly_3[3] + poly_3[2]*data[i] + poly_3[1]*data[i]**2 + poly_3[0]*data[i]**3)
#    print("Original coma flotante; f("+ str(data[i]) +"):", 2 * (1 / (1 + np.exp(-0.5 * data[i]))))

for i in range(0,9):
    print("Aproximación Grado 7;   f("+ str(data[i]) +"):", poly_7[7] + poly_7[6]*data[i] + poly_7[4]*data[i]**3 + poly_7[2]*data[i]**5 + poly_7[0]*data[i]**7)
    print("Aproximación Grado 5;   f("+ str(data[i]) +"):", poly_5[5] + poly_5[4]*data[i] + poly_5[2]*data[i]**3 + poly_5[0]*data[i]**5)
    print("Aproximación Grado 3;   f("+ str(data[i]) +"):", poly_3[3] + poly_3[2]*data[i] + poly_3[0]*data[i]**3)
    print("Original coma flotante; f("+ str(data[i]) +"):", 2 * (1 / (1 + np.exp(-0.5 * data[i]))))

# Crear una nueva figura y mejorar la interpretación de la gráfica
plt.figure(figsize=(10, 6))

# Gráficar
plt.plot(x_range, y_range, label="Original", color='blue')
plt.plot(x_range, y_poly_3, '-.', label="Aproximación de grado 3", color='purple')
plt.plot(x_range, y_poly_5, '--', label="Aproximación de grado 5", color='green')
plt.plot(x_range, y_poly_7, ':', label="Aproximación de grado 7", color='red')

# Mejorar la visualización
plt.legend(loc="best")
plt.xlabel('x')
plt.ylabel('f(x)')
```

```
plt.title('Comparación entre la función original y las aproximaciones polinómicas de grado 3, 5 y 7.')

# Ajustar el rango de los ejes para que sea más claro
plt.xlim([-12, 12])  # Ajustar los límites del eje x
plt.ylim([0, 2.5])   # Ajustar los límites del eje y

# Añadir líneas de cuadrícula para facilitar la lectura
plt.grid(True)

# Mostrar la gráfica
plt.show()
```

CÓDIGO D.26: aprox_poli.py

```
#include <neorv32.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <float.h>
#include "neorv32_zfinx_extension_intrinsics.h"

#define BAUD_RATE 19200

// This defines is used to bypass the intermediate print functions between cfu/FPU functions (for simulation)
// Comment this defines to perform a implementation
// Uncomment sim to perform a simulation
//#define sim

//    Floating point operations supported by the NEORV32 FPU
//    riscv_intrinsic_fadds()
//    riscv_intrinsic_fsubs()
//    riscv_intrinsic_fmuls()
//    The division operation is not supported, so the function is approximated with a polynomial.

float sig_aprox_7 (float x){
    // f(x)=1 + 0.24190955524x - 0.00369209483x^3 + 0.00003770457x^5 - 0.00000015213x^7; grade 7 approximation; range [-10 to 10]
    float a1 = 0.24190955524f;
    float a3 = -0.00369209483f;
    float a5 = 0.00003770457f;
    float a7 = -0.00000015213f;
    float inter1, inter2, inter3, inter4, pow1, pow2, pow3, pow4;
    float res, res1, res2, res3;

    inter1 = riscv_intrinsic_fmuls(a1,x);
    pow1 = riscv_intrinsic_fmuls(x,x);
    pow2 = riscv_intrinsic_fmuls(pow1,x);
    inter2 = riscv_intrinsic_fmuls(a3,pow2);
    pow3 = riscv_intrinsic_fmuls(pow2,pow1);
```

```
35      inter3 = riscv_intrinsic_fmuls(a5,pow3);
36      pow4 = riscv_intrinsic_fmuls(pow3,pow1);
37      inter4 = riscv_intrinsic_fmuls(a7,pow4);
38      res1 = riscv_intrinsic_fadds(1,inter1);
39      res2 = riscv_intrinsic_fadds(res1,inter2);
40      res3 = riscv_intrinsic_fadds(res2,inter3);
41      res = riscv_intrinsic_fadds(res3,inter4);
42
43      return res;
44  }
45
46  float sig_aprox_5 (float x){
47      // f(x)=1 + 0.22878851178x - 0.00253272801x^3 + 0.00001266682x^5; grade 5
        ↪  approximation; range [-10 to 10]
48      float a1 = 0.22878851178f;
49      float a3 = -0.00253272801f;
50      float a5 = 0.00001266682f;
51      float inter1, inter2, inter3, pow1, pow2, pow3;
52      float res, res1, res2;
53
54      inter1 = riscv_intrinsic_fmuls(a1,x);
55      pow1 = riscv_intrinsic_fmuls(x,x);
56      pow2 = riscv_intrinsic_fmuls(pow1,x);
57      inter2 = riscv_intrinsic_fmuls(a3,pow2);
58      pow3 = riscv_intrinsic_fmuls(pow2,pow1);
59      inter3 = riscv_intrinsic_fmuls(a5,pow3);
60      res1 = riscv_intrinsic_fadds(1,inter1);
61      res2 = riscv_intrinsic_fadds(res1,inter2);
62      res = riscv_intrinsic_fadds(res2,inter3);
63
64      return res;
65  }
66
67  float sig_aprox_3 (float x){
68      // f(x)=1 + 0.19744040439x - 0.00109773200x^3; grade 3 approximation; range [-10
        ↪  to 10]
69      float a1 = 0.19744040439f;
70      float a3 = -0.00109773200f;
71      float inter1, inter2, pow1, pow2;
72      float res, res1;
73
74      inter1 = riscv_intrinsic_fmuls(a1,x);
75      pow1 = riscv_intrinsic_fmuls(x,x);
76      pow2 = riscv_intrinsic_fmuls(pow1,x);
77      inter2 = riscv_intrinsic_fmuls(a3,pow2);
78      res1 = riscv_intrinsic_fadds(1,inter1);
79      res = riscv_intrinsic_fadds(res1,inter2);
80
81      return res;
82  }
83
84  int n_sign_decimal(float value) {
```

```
85      int parte_entera = (int)value;
86      float decimal = riscv_intrinsic_fsubs(value,(float)parte_entera);
87      float eval_n = riscv_intrinsic_fmuls(decimal,10);
88      float sign = riscv_intrinsic_flts(value,0);
89
90      if(parte_entera == 0 && sign == 1)
91          if((int)eval_n >= 1)
92              return 1; //-0.x
93          else
94              return 2;//-0.0x
95      else
96          if((int)eval_n >= 1)
97              return 3; // != -0. ; x.y
98          else
99              return 4; // != -0. ; x.0y
100 }
101
102 int main() {
103
104   // Capture all exceptions and give debug info via UART0
105   neorv32_rte_setup();
106
107   // Setup UART at default baud rate, no interrupts
108   neorv32_uart0_setup(BAUD_RATE, 0);
109
110   // Check if UART0 unit is implemented at all
111   if (neorv32_uart0_available() == 0) {
112     return -1; // abort if not implemented
113   }
114
115   // check if Zfinx extension is implemented at all
116   if ((neorv32_cpu_csr_read(CSR_MXISA) & (1<<CSR_MXISA_ZFINX)) == 0) {
117     neorv32_uart0_puts("Error! <Zfinx> extension not synthesized!\n");
118     return 1;
119   }
120
121   // check if the CFU is implemented at all (the CFU is wrapped in the core's "Zxcfu"
      ↪ ISA extension)
122   if (neorv32_cpu_cfu_available() == 0) {
123     neorv32_uart0_printf("ERROR! CFU ('Zxcfu' ISA extensions) not implemented!\n");
124     return 1;
125   }
126
127   // check if the CPU base counters are implemented
128   if ((neorv32_cpu_csr_read(CSR_MXISA) & (1 << CSR_MXISA_ZICNTR)) == 0) {
129     neorv32_uart0_printf("ERROR! Base counters ('Zicntr' ISA extensions) not
        ↪ implemented!\n");
130     return -1;
131   }
132
133   neorv32_cpu_csr_write(CSR_FCSR, 0); // clear exception flags and set "round to
      ↪ nearest"
```

```
134
135   // Declaration of variables
136   //-10
137   const uint32_t fir = 0x0000F600;
138   //-7.5
139   const uint32_t sec = 0x0000F880;
140   //-5
141   const uint32_t thi = 0x0000FB00;
142   //-2.5
143   const uint32_t fou = 0x0000FD80;
144   //0
145   const uint32_t fif = 0x00000000;
146   //2.5
147   const uint32_t six = 0x00000280;
148   //5
149   const uint32_t sev = 0x00000500;
150   //7.5
151   const uint32_t eig = 0x00000780;
152   //10
153   const uint32_t nin = 0x00000A00;
154   //Input vector to hardware accelerator
155   static uint32_t data_hw[9] = {fir,sec,thi,fou,fif,six,sev,eig,nin};
156   //Input vector to software
157   float data_sw[9] = {-10.0,-7.5,-5.0,-2.5,0.0,2.5,5.0,7.5,10.0};
158   //Loop variable
159   int i;
160
161   #ifdef sim
162   // Intro
163   // Sigmoide-Comparation
164   neorv32_uart0_printf("SIG-C");
165
166   // Perform 9 operations through hardware accelerator (funct3=000)
167   for (i=0; i<9; i++){
168     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
169     neorv32_cfu_r3_instr(0b1111111, 0b000, data_hw[i], 0);
170     neorv32_cpu_csr_read(CSR_MCYCLE);
171   }
172   // Perform 9 operations through software with FPU (approximation degree 7)
173   for (i=0; i<9; i++){
174     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
175     sig_aprox_7(data_sw[i]);
176     neorv32_cpu_csr_read(CSR_MCYCLE);
177   }
178   // Perform 9 operations through software with FPU (approximation degree 5)
179   for (i=0; i<9; i++){
180     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
181     sig_aprox_5(data_sw[i]);
182     neorv32_cpu_csr_read(CSR_MCYCLE);
183   }
184   // Perform 9 operations through software with FPU (approximation degree 5)
185   for (i=0; i<9; i++){
```

```
186     neorv32_cpu_csr_write(CSR_MCYCLE, 0);
187     sig_aprox_3(data_sw[i]);
188     neorv32_cpu_csr_read(CSR_MCYCLE);
189   }
190   // End
191   neorv32_uart0_printf("END");
192   #else
193   uint32_t time_hw, time_sw7, time_sw5, time_sw3;
194   int parte_entera_in, parte_decimal_in, parte_entera_out, parte_decimal_out;
195   float fact_multi = 1000000000;
196   float res;
197   // Intro
198   neorv32_uart0_printf("\n<<< SIGMOID through CFU compare with SIGMOID through
      ↪  software (with polynomial approximation computed by FPU) >>>\n");
199   neorv32_uart0_printf("CFU R3-Type (rs1= 0xINPUT, rs2= DC, rd = OUT)\n");
200   neorv32_uart0_printf("Since the FPU does not support division, a polynomial
      ↪  approximation of degree 3, 5 and 7 is made.\n");
201
202   // Write 9 inputs to sigmoid and read the outputs one by one
203   neorv32_uart0_printf("\nThrough CFU hardware accelerator (CRI)\n");
204   for (i=0; i<9 ; i++) {
205     neorv32_uart0_printf("%u: neorv32_cfu_r3_instr( funct7=0b1111111, funct3=0b000,
      ↪  [rs1]=0x%x, [rs2]=0x%x ) = ", i, data_hw[i], 0);
206     neorv32_uart0_printf("0x%x\n",neorv32_cfu_r3_instr(0b1111111, 0b000, data_hw[i],
      ↪  0));
207   }
208   neorv32_uart0_printf("\nApproximation with a degree 7 polynomial:\n\n");
209
210   for (i=0; i<9 ; i++) {
211     //In order to print the float via UART
212     parte_entera_in = (int)data_sw[i];
213     parte_decimal_in =
      ↪  (int)(fabs(riscv_intrinsic_fmuls(riscv_intrinsic_fsubs(data_sw[i],
      ↪  (float)parte_entera_in),fact_multi)));
214     neorv32_uart0_printf("%u: f(%d.%d) = ", i, parte_entera_in, parte_decimal_in);
215     res = sig_aprox_7(data_sw[i]);
216     parte_entera_out = (int)res;
217     parte_decimal_out = (int)(fabs(riscv_intrinsic_fmuls(riscv_intrinsic_fsubs(res,
      ↪  (float)parte_entera_out),fact_multi)));
218     // In particular cases when the integer part is 0, and the number is negative,
      ↪  the sign must be managed
219     if(n_sign_decimal(res) == 1){
220         neorv32_uart0_printf("-%d.%d\n", parte_entera_out, parte_decimal_out);
221     } else if (n_sign_decimal(res) == 2){
222         neorv32_uart0_printf("-%d.0%d\n", parte_entera_out, parte_decimal_out);
223     }else if (n_sign_decimal(res) == 3){
224         neorv32_uart0_printf("%d.%d\n", parte_entera_out, parte_decimal_out);
225     }else{
226         neorv32_uart0_printf("%d.0%d\n", parte_entera_out, parte_decimal_out);
227     }
228   }
229
```

```
230   neorv32_uart0_printf("\nApproximation with a degree 5 polynomial:\n\n");
231
232   for (i=0; i<9 ; i++) {
233     //In order to print the float via UART
234     parte_entera_in = (int)data_sw[i];
235     parte_decimal_in =
      ↪  (int)(fabs(riscv_intrinsic_fmuls(riscv_intrinsic_fsubs(data_sw[i],
      ↪  (float)parte_entera_in),fact_multi)));
236     neorv32_uart0_printf("%u: f(%d.%d) = ", i, parte_entera_in, parte_decimal_in);
237     res = sig_aprox_5(data_sw[i]);
238     parte_entera_out = (int)res;
239     parte_decimal_out = (int)(fabs(riscv_intrinsic_fmuls(riscv_intrinsic_fsubs(res,
      ↪  (float)parte_entera_out),fact_multi)));
240     // In particular cases when the integer part is 0, and the number is negative,
      ↪  the sign must be managed
241     if(n_sign_decimal(res) == 1){
242         neorv32_uart0_printf("-%d.%d\n", parte_entera_out, parte_decimal_out);
243     } else if (n_sign_decimal(res) == 2){
244         neorv32_uart0_printf("-%d.0%d\n", parte_entera_out, parte_decimal_out);
245     }else if (n_sign_decimal(res) == 3){
246         neorv32_uart0_printf("%d.%d\n", parte_entera_out, parte_decimal_out);
247     }else{
248         neorv32_uart0_printf("%d.0%d\n", parte_entera_out, parte_decimal_out);
249     }
250   }
251
252   neorv32_uart0_printf("\nApproximation with a degree 3 polynomial:\n\n");
253
254   for (i=0; i<9 ; i++) {
255     //In order to print the float via UART
256     parte_entera_in = (int)data_sw[i];
257     parte_decimal_in =
      ↪  (int)(fabs(riscv_intrinsic_fmuls(riscv_intrinsic_fsubs(data_sw[i],
      ↪  (float)parte_entera_in),fact_multi)));
258     neorv32_uart0_printf("%u: f(%d.%d) = ", i, parte_entera_in, parte_decimal_in);
259     res = sig_aprox_3(data_sw[i]);
260     parte_entera_out = (int)res;
261     parte_decimal_out = (int)(fabs(riscv_intrinsic_fmuls(riscv_intrinsic_fsubs(res,
      ↪  (float)parte_entera_out),fact_multi)));
262     // In particular cases when the integer part is 0, and the number is negative,
      ↪  the sign must be managed
263     if(n_sign_decimal(res) == 1){
264         neorv32_uart0_printf("-%d.%d\n", parte_entera_out, parte_decimal_out);
265     } else if (n_sign_decimal(res) == 2){
266         neorv32_uart0_printf("-%d.0%d\n", parte_entera_out, parte_decimal_out);
267     }else if (n_sign_decimal(res) == 3){
268         neorv32_uart0_printf("%d.%d\n", parte_entera_out, parte_decimal_out);
269     }else{
270         neorv32_uart0_printf("%d.0%d\n", parte_entera_out, parte_decimal_out);
271     }
272   }
273
```

```
274   neorv32_uart0_printf("\nMeasure the lartency of all methods\n");
275   neorv32_uart0_printf("\nThe test performs the calculation for 9 input data\n\n");
276
277   neorv32_cpu_csr_write(CSR_MCYCLE, 0); // start timing
278   for (i=0; i<9 ; i++) {
279     neorv32_cfu_r3_instr(0b1111111, 0b000, data_hw[i], 0);
280   }
281   time_hw = neorv32_cpu_csr_read(CSR_MCYCLE); // stop timing
282
283   neorv32_cpu_csr_write(CSR_MCYCLE, 0); // start timing
284   for (i=0; i<9 ; i++) {
285     sig_aprox_7(data_sw[i]);
286   }
287   time_sw7 = neorv32_cpu_csr_read(CSR_MCYCLE); // stop timing
288
289   neorv32_cpu_csr_write(CSR_MCYCLE, 0); // start timing
290   for (i=0; i<9 ; i++) {
291     sig_aprox_5(data_sw[i]);
292   }
293   time_sw5 = neorv32_cpu_csr_read(CSR_MCYCLE); // stop timing
294
295   neorv32_cpu_csr_write(CSR_MCYCLE, 0); // start timing
296   for (i=0; i<9 ; i++) {
297     sig_aprox_3(data_sw[i]);
298   }
299   time_sw3 = neorv32_cpu_csr_read(CSR_MCYCLE); // stop timing
300
301   neorv32_uart0_printf("HW calculated through SIG(CRI) = %u cycles\n", time_hw);
302   neorv32_uart0_printf("SW aprox by polynomial grade 7 via FPU = %u cycles\n",
      ↪  time_sw7);
303   neorv32_uart0_printf("SW aprox by polynomial grade 5 via FPU = %u cycles\n",
      ↪  time_sw5);
304   neorv32_uart0_printf("SW aprox by polynomial grade 3 via FPU = %u cycles\n",
      ↪  time_sw3);
305   // End
306   neorv32_uart0_printf("\nProgram completed.\n");
307   #endif
308
309   return 0;
310 }
```

CÓDIGO D.27: Compración cálculo de la sigmoide: CRI vs FPU; main.c

Bibliografía

[1] A. Munir, E. Blasch, J. Kwon, J. Kong y A. Aved, «Artificial Intelligence and Data Fusion at the Edge,» *IEEE Aerospace and Electronic Systems Magazine*, vol. 36, n.º 7, págs. 62-78, 2021. DOI: 10.1109/MAES.2020.3043072.

[2] M. M. H. Shuvo, S. K. Islam, J. Cheng y B. I. Morshed, «Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review,» *Proceedings of the IEEE*, vol. 111, n.º 1, págs. 42-91, 2023. DOI: 10.1109/JPROC.2022.3226481.

[3] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. Patterson y K. Asanovic, «The RISC-V instruction set,» en *2013 IEEE Hot Chips 25 Symposium (HCS)*, 2013, págs. 1-1. DOI: 10.1109/HOTCHIPS.2013.7478332.

[4] D. Lampret y the OpenRISC Community, *OpenRISC*, gh:openrisc, 2000.

[5] OpenCores Community, *OpenCores*, opencores.org, 1999.

[6] Sun Microsystems, Inc., *OpenSPARC*, oracle.com/servers/technologies/opensparc-overview, 2005.

[7] European Space Research y Technology Centre, *LEON*, esa.int/Enabling_Support/Space_Engineering_Technology/LEON_the_space_chip_that_Europe_built, 1997.

[8] Hitachi/Renesas, *SuperH*, renesas.com/us/en/products/microcontrollers-microprocessors/other-mcus-mpus/superh-risc-engine-family-mcus, 1992.

[9] Hitachi/Mitsubishi, *Hitachi and Mitsubishi Electric to Establish Renesas Technology Corp., a New Company for Semiconductor Operations*, hitachi.us/press/archive/10032002, 2002.

[10] Jeff Dionne, *J-Core Open Processor*, j-core.org, 2015.

[11] OpenPOWER Foundation, *Power ISA*, openpowerfoundation.org/specifications/isa/, 2024.

[12] OpenPOWER Foundation, *OpenPOWER*, openpower.foundation, 2006.

[13] A. Blanchard y Contribuidores, *Microwatt - A tiny Open POWER ISA softcore*, gh:antonblanchard/microwatt, 2019.

[14] A. Blanchard y Contribuidores, *Chiselwatt - A tiny POWER Open ISA soft processor*, gh:antonblanchard/chiselwatt, 2019.

[15] L. Leighton y Contribuidores, *Libre-SOC*, libre-soc.org/, 2019.

[16] A. Waterman, Y. Lee, D. Patterson y K. Asanovic, *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*, eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.pdf, 2011.

[17] K. Asanovic y D. Patterson, *Instruction Sets Should Be Free: The Case For RISC-V*, eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf, 2014.

[18] J. Bachrach, H. Vo, B. Richards et al., «Chisel: Constructing hardware in a Scala embedded language,» en *DAC Design Automation Conference 2012*, 2012, págs. 1212-1221. DOI: 10.1145/2228360.2228584.

[19] RISC-V Community, *Open-Source RISC-V Architecture IDs*, gh:riscv/riscv-isa-manual/blob/main/marchid.md, 2024.

[20] A. Waterman y Contribuidores, *Rocket Chip*, gh:chipsalliance/rocket-chip, 2014.

[21] C. Papon y Contribuidores, *VexRiscv - A FPGA friendly 32 bit RISC-V CPU*, gh:SpinalHDL/VexRiscv, 2016.

[22] C. Xenia Wolf y Contribuidores, *PicoRV32 - A Size-Optimized RISC-V CPU*, gh:YosysHQ/picorv32, 2021.

[23] Nuclei System Technology, *Hummingbirdv2 E203 Core and SoC*, gh:riscv-mcu/e203_hbirdv2, 2020.

[24] R. Calcada y Contribuidores, *RISC-V Steel*, gh:riscv-steel/riscv-steel, 2024.

[25] O. Kindgren y Contribuidores, *SERV - The SErial RISC-V CPUl*, gh:olofk/serv, 2024.

[26] S. Nolting y Contribuidores, *NEORV32 - A tiny, customizable and extensible MCU-class 32-bit RISC-V soft-core CPU and microcontroller-like SoC*, gh:stnolting/neorv32, 2024.

[27] J. Vandergriendt, *ORCA RISC-V RV32IM core*, gh:kammoh/ORCA-risc-v, 2015.

[28] S. Prakash, T. Callahan, J. Bushagour et al., «CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (TinyML) Acceleration on FPGAs,» en *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, abr. de 2023. DOI: 10.1109/ispass57527.2023.00024. dirección: http://dx.doi.org/10.1109/ISPASS57527.2023.00024.

[29] J. Gray, *Draft Proposed RISC-V Composable Custom Extensions Specification*, raw.github usercontent.com/grayresearch/CFU/main/spec/spec.pdf, 2019.

[30] ARM, *AXI4-Stream*, developer.arm.com/documentation/ihi0051/latest/, 2010.

[31] OpenCores, *Wishbone Bus*, cdn.opencores.org/downloads/wbspec_b4.pdf, 2010.

[32] ARM, *AXI4-Lite*, developer.arm.com/documentation/ihi0022/e/, 2010.

[33] K. Basterretxea, E. Alonso, J. M. Tarela e I. del Campo, «PWL approximation of non-linear functions for the implementation of neuro-fuzzy systems,» *Proceedings of the IMACS/IEEE CSCC*, vol. 99, 1999.

[34] J. Tarela, K Basterretxea, I Del Campo, M. Martínez y E Alonso, «Optimised PWL recursive approximation and its application to neuro-fuzzy systems,» *Mathematical and computer modelling*, vol. 35, n.º 7-8, págs. 867-883, 2002.

[35] U. Sainz-Estebanez, *GHDL + yosys + GHDL yosys plugin + nextpnr-xilinx + prjxray Container*, ghcr.io/unike267/containers/impl-arty:latest, 2024.

[36] N.-D. Nguyen, D.-H. Bui y X.-T. Tran, «Tiny Neuron Network System based on RISC-V Processor: A Decentralized Approach for IoT Applications,» en *2022 International Conference on Advanced Technologies for Communications (ATC)*, 2022, págs. 98-103. DOI: 10.1109/ATC55345.2022.9942990.

[37] X. Yu, Z. Yang, L. Peng, B. Lin, W. Yang y L. Wang, «CNN Specific ISA Extensions Based on RISC-V Processors,» en *2022 5th International Conference on Circuits, Systems and Simulation (ICCSS)*, 2022, págs. 116-120. DOI: 10.1109/ICCSS55260.2022.9802445.

[38] P. Davide Schiavone, F. Conti, D. Rossi et al., «Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications,» en *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, págs. 1-8. DOI: 10.1109/PATMOS.2017.8106976.

[39] M. Gautschi, M. Wegmann y D. Schiavone, *zero-riscy CPU Core)*, tom01h/zero-riscy, 2017.

[40] Q. Wei, E. Cui, Y. Gao y T. Li, «A Review of Edge Intelligence Applications Based on RISC-V,» en *2023 2nd International Conference on Computing, Communication, Perception and Quantum Technology (CCPQT)*, 2023, págs. 115-119. DOI: 10.1109/CCPQT60491.2023.00025.

[41] AMD Research and Advanced Development, *RapidWright*, gh:Xilinx/RapidWright, 2018.

[42] AMD, *Runtime-First FPGA Interchange Routing Contes*, xilinx.github.io/fpga24_routing_contest/index, 2024.

[43] CHIPS Alliance, *FPGA Interchange Format*, rapidwright.io/docs/FPGA_Interchange_Format, 2020.

[44] J. Lewis y Contribuidores, *Open Source VHDL Verification Methodology*, osvvm.org, 2013.

[45] E. Tallaksen y Contribuidores, *Universal VHDL Verification Methodology*, uvvm.org, 2013.

[46] Siemens, *The 2022 Wilson Research Group Functional Verification Study*, blogs.sw.siemens.com/verificationhorizons/2022/11/21/part-6-the-2022-wilson-research-group-functional-verification-study/, 2022.

[47] AMD, *AMD MicroBlaze™ Processor*, amd.com/en/products/software/adaptive-socs-and-fpgas/microblaze.html, 2024.

[48] AMD, *AMD MicroBlaze™ V Processor*, amd.com/en/products/software/adaptive-socs-and-fpgas/microblaze-v.html, 2024.

[49] Enjoy-Digital, *LiteX*, gh:enjoy-digital/litex, 2024.

[50] M-Labs, *Migen (Milkymist generator)*, gh:m-labs/migen, 2020.

[51] V. N. Chander y K. Varghese, «A Soft RISC-V Vector Processor for Edge-AI,» en *2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID)*, 2022, págs. 263-268. DOI: 10.1109/VLSID2022.2022.00058.

[52] S. Nolting y Contribuidores, *Prebuilt RISC-V GCC toolchains for x64 Linux*. gh:stnolting/riscv-gcc-prebuilt, 2023.

[53] S. Nolting y Contribuidores, *Simulation container*, ghcr.io/stnolting/neorv32/sim, 2024.

[54] S. Nolting y Contribuidores, *The NEORV32 RISC-V Processor Datasheet*, stnolting.github.io/neorv32/, 2020.

[55] S. Nolting y Contribuidores, *The NEORV32 RISC-V Processor User Guide*, stnolting.github.io/neorv32/ug/, 2020.

[56] T. Gingold y Contribuidores, *GHDL - VHDL 2008/93/87 simulator*, gh:ghdl/ghdl, 2024.

[57] C. Xenia Wolf y Contribuidores, *YOSYS - Yosys Open SYnthesis Suite*, gh:YosysHQ/yosys, 2020.

[58] T. Gingold y Contribuidores, *ghdl-yosys-plugin: VHDL synthesis (based on GHDL and Yosys)*, gh:ghdl/ghdl-yosys-plugin, 2024.

[59] D. Shah y Contribuidores, *Nextpnr-Xilinx - Experimental flows using nextpnr for Xilinx devices*, gh:gatecat/nextpnr-xilinx, 2020.

[60] Project X-Ray Contribuidores, *Documenting the Xilinx 7-series bit-stream format*, gh:f4pga/prjxray, 2020.

[61] G. Goavec-Merou y Contribuidores, *openFPGALoader: Universal utility for programming FPGA*, gh:trabucayre/openFPGALoader, 2024.

[62] L. Asplund, O. Kraigher y Contribuidores, *VUnit - Testing framework for VHDL/SystemVerilog*, gh:VUnit/vunit, 2024.

[63] A. Neundorf y Contribuidores, *CuteCom - A graphical serial terminal*, gh:neundorf/CuteCom, 2018.

[64] U. Sainz-Estebanez, *Practices*, gh:Unike267/Practices, 2024.

[65] Linux Foundation, *Zephyr OS*, zephyrproject.org, 2016.