
This is the **published version** of the bachelor thesis:

Oliva Abelló, Ariadna. *Acceleració FPGA de la verificació de Circuits Digitals dissenyats amb Python*. Treball de Final de Grau (Universitat Autònoma de Barcelona), 2026 (Grau en Enginyeria de Sistemes de Telecomunicació)

This version is available at <https://ddd.uab.cat/record/335930>

under the terms of the  license.



TREBALL DE FI DE GRAU

GRAU EN ENGINYERIA DE SISTEMES DE TELECOMUNICACIÓ

ACCELERACIÓ FPGA DE LA VERIFICACIÓ DE CIRCUITS DIGITALS DISSENYATS AMB PYTHON

Ariadna Oliva Abelló

DIRECTOR: David Castells Rufas

DEPARTAMENT DE TELECOMUNICACIÓ I ENGINYERIA DE SISTEMES

UNIVERSITAT AUTÒNOMA DE BARCELONA

Bellaterra, Juny 29, 2026

Abstract

The verification of digital circuits is an essential process, but it is costly in terms of computation time, especially in highly complex designs. Purely software-based simulation tools provide flexibility and control, but they present performance limitations. On the other hand, a complete FPGA implementation provides execution speed, but reduces debugging capability.

This work proposes a hybrid Hardware-in-the-Loop approach that combines Python-based simulation, through the *py4hw* library, with the execution of the Design Under Test (DUT) on an AMD Alveo U55C FPGA connected via PCI Express (PCIe). Verification control remains in *py4hw*, while the DUT computation is delegated to hardware.

An architecture has been designed and implemented that, starting from a DUT described in *py4hw*, automatically extracts its ports, adapts them to AXI4-Stream channels through a wrapper, and generates the executable binary for the FPGA using the Vitis flow and the XRT runtime. The platform has been validated with two circuits (an Add/Sub arithmetic module and an RGB-to-YCbCr colour-space converter) confirming that the outputs obtained on the FPGA match, bit by bit, those of the *py4hw* software simulation.

The project provides a basis for future developments in FPGA-Python co-simulation environments and for the automation of the verification flow in digital design.

Resum

La verificació de circuits digitals és un procés essencial però costós en temps de càlcul, especialment en dissenys de gran complexitat. Les eines de simulació purament software ofereixen flexibilitat i control, però presenten limitacions de rendiment. D'altra banda, la implementació completa en FPGA proporciona velocitat d'execució, però redueix la capacitat de depuració.

Aquest treball proposa una aproximació híbrida de tipus Hardware-in-the-Loop que combina la simulació en Python, mitjançant la llibreria *py4hw*, amb l'execució del circuit sota test (DUT) sobre una FPGA AMD Alveo U55C connectada via PCI Express (PCIe). El control de la verificació es manté en *py4hw*, mentre que el càlcul del DUT es delega al hardware.

S'ha dissenyat i implementat una arquitectura que, a partir d'un DUT descrit en *py4hw*, n'obté automàticament els ports, l'adapta a canals AXI4-Stream mitjançant un *wrapper* i genera el binari executable per a la FPGA mitjançant el flux de Vitis i el runtime XRT. La plataforma s'ha validat amb dos circuits (un mòdul aritmètic Add/Sub i un convertidor de color RGB a YCbCr) comprovant que les sortides obtingudes sobre la FPGA coincideixen bit a bit amb les de la simulació software de *py4hw*.

El projecte constitueix una base per a futurs desenvolupaments en entorns de co-simulació FPGA-Python i per a l'automatització del flux de verificació en el disseny digital.

Resumen

La verificación de circuitos digitales es un proceso esencial, pero costoso en tiempo de cálculo, especialmente en diseños de gran complejidad. Las herramientas de simulación puramente software ofrecen flexibilidad y control, pero presentan limitaciones de rendimiento. Por otro lado, la implementación completa en FPGA proporciona velocidad de ejecución, pero reduce la capacidad de depuración.

Este trabajo propone una aproximación híbrida de tipo Hardware-in-the-Loop que combina la simulación en Python, mediante la librería *py4hw*, con la ejecución del circuito bajo prueba (DUT) sobre una FPGA AMD Alveo U55C conectada mediante PCI Express (PCIe). El control de la verificación se mantiene en *py4hw*, mientras que el cálculo del DUT se delega al hardware.

Se ha diseñado e implementado una arquitectura que, a partir de un DUT descrito en *py4hw*, obtiene automáticamente sus puertos, lo adapta a canales AXI4-Stream mediante una envoltura y genera el binario ejecutable para la FPGA mediante el flujo de Vitis y el runtime XRT. La plataforma se ha validado con dos circuitos (un módulo aritmético Add/Sub y un convertidor de color RGB a YCbCr) comprobando que las salidas obtenidas sobre la FPGA coinciden bit a bit con las de la simulación software de *py4hw*.

El proyecto constituye una base para futuros desarrollos en entornos de co-simulación FPGA-Python y para la automatización del flujo de verificación en el diseño digital.

Agraïments

M'agradaria expressar el meu agraïment a totes les persones que m'han acompanyat i donat suport durant el desenvolupament d'aquest Treball de Fi de Grau.

En primer lloc, vull agrair al meu tutor, David Castells Rufas, la seva orientació, ajuda i disponibilitat al llarg del projecte. Els seus consells i indicacions han estat essencials per entendre millor l'entorn de treball, avançar en la implementació i donar forma a la proposta desenvolupada.

També vull agrair especialment a la meva família, amistats i companys el suport i els ànims durant tot aquest procés. El seu acompanyament ha estat molt important, tant en els moments d'avanç com en les etapes més difícils del treball.

A tots ells, moltes gràcies.

Índex

Abstract	I
Resum	III
Resumen	V
Agraïments	VII
Índex de figures	XIII
Índex de taules	XVII
Glossari d'acrònims	XIX
1 Introducció	1
1.1 Motivació i context del projecte	1
1.2 Plantejament del problema i justificació	2
1.3 Objectius generals i específics	3
1.4 Metodologia i abast del treball	4
2 Estat de l'art	7
2.1 Verificació digital i simulació tradicional	7
2.2 Co-simulació i emulació amb FPGA	8
2.3 Disseny i verificació de hardware amb Python	9
2.4 Plataformes FPGA per a acceleració: Vitis, XRT i Alveo	9

2.5	Limitacions actuals i posicionament del projecte	10
3	Fonaments teòrics i conceptes clau	13
3.1	Arquitectura bàsica d'una FPGA	13
3.1.1	Lògica programable i recursos interns	13
3.1.2	FPGA com a plataforma d'acceleració hardware	15
3.2	Verificació de circuits digitals	15
3.2.1	Simulació software i les seves limitacions	16
3.2.2	Simulació híbrida i Hardware-in-the-Loop	17
3.3	Descripció hardware amb Python	18
3.3.1	<i>py4hwcom</i> a entorn de disseny i simulació	18
3.3.2	DUT: circuit sota test	20
3.3.3	Generació de Verilog a partir del model Python	21
3.4	Execució hardware i comunicació host-FPGA	21
3.4.1	Flux Vivado/Vitis	21
3.4.2	Kernels hardware	23
3.4.3	Interfícies AXI i transferència de dades	24
3.5	Verificació funcional del sistema	25
3.5.1	<i>Testbench</i> i comparació de resultats	25
3.5.2	Automatització del flux de generació	25
4	Disseny de l'Arquitectura del Sistema	27
4.1	Descripció general del sistema proposat	27
4.2	Arquitectura software	28
4.2.1	Interfície entre <i>py4hw</i> i el controlador de maquinari	29
4.2.2	Disseny de l'API de comunicació	29
4.3	Arquitectura hardware	30
4.3.1	Jerarquia de mòduls dins la FPGA	30

4.3.2	Interfície PCIe i control de flux	31
4.3.3	DUT <i>wrapper</i> : integració RTL com a kernel	32
4.4	Protocol de comunicació i format de dades	33
4.4.1	Mecanisme de crida i sincronització <i>testbench</i> - FPGA	33
4.4.2	Format de dades	34
5	Implementació pràctica i desenvolupament	35
5.1	Entorn de desenvolupament	35
5.1.1	Eines software: Vivado, Vitis, XRT, Python	35
5.1.2	Configuració del servidor i de la targeta FPGA	39
5.2	Validació inicial del flux Vitis/XRT	40
5.2.1	Prova preliminar amb inversió de píxels en una imatge PPM	43
5.2.2	Prova amb un kernel RTL de conversió RGB a YCbCr	44
5.2.3	Valoracions de les proves preliminars	45
5.3	Desenvolupament del programari	45
5.3.1	Extensió de <i>py4hw</i> per a la co-simulació FPGA	46
5.3.2	API de control i programa host	47
5.3.3	Redirector: integració de la FPGA dins del bucle de <i>py4hw</i>	47
5.4	Desenvolupament del maquinari	49
5.4.1	Circuits sota test: Add/Sub i RGB2YCbCr	51
5.4.2	Kernels reader i writer	53
5.4.3	DUT <i>wrapper</i> i generació del fitxer .xo	54
5.4.4	Integració AXI4-Stream i control de flux	56
5.5	Integració sistema complet	56
5.5.1	Generació dels fitxers XO i XCLBIN	57
5.5.2	Flux de dades host-FPGA-host	60
5.6	Metodologia experimental	60
5.6.1	Circuits de prova (Add/Sub i RGB2YCbCr)	61

5.6.2	Mètriques i procediment de comparació (software vs híbrid)	61
6	Resultats, conclusions i treballs futurs	63
6.1	Verificació funcional	63
6.1.1	Add/Sub: verificació dirigida	63
6.1.2	RGB2YCbCr: verificació per mostreig	64
6.1.3	Verificació automatitzada amb <i>pytest</i>	66
6.2	Anàlisi de rendiment	67
6.2.1	Metodologia de mesura (SW vs HW)	67
6.2.2	Resultats i discussió	67
6.3	Conclusions	69
6.3.1	Valoració dels objectius assolits	69
6.4	Línies futures de desenvolupament	70
6.4.1	Integració en un flux de CI/CD per a la verificació hardware	70
6.4.2	Implementació completa del bloc <i>axi2clk</i>	71
6.4.3	Optimització de la comunicació i millora del rendiment	71
	Bibliografia	I

Índex de figures

2.1	Esquema conceptual d'una arquitectura híbrida de verificació.	8
3.1	Estructura interna general d'una FPGA.	14
3.2	Ubicació del <i>Hardware-in-the-Loop</i> dins del flux de desenvolupament i verificació.	18
3.3	Arquitectura general d'un <i>testbench</i> : el vector d'estímul aplica les entrades al DUT (<i>Unit/Design Under Test</i>) i se'n recullen les sortides.	20
3.4	Flux de compilació amb Vitis per generar el fitxer <code>.xclbin</code> a partir dels kernels del sistema	22
3.5	Handshake de l'AXI4-Stream: la dada (TDATA) només avança quan TVALID i TREADY estan actius alhora; quan TREADY baixa, l'element es manté fins que el receptor el pot acceptar. TLAST marca l'últim element del bloc.	24
3.6	Esquema simplificat d'un flux de verificació basat en simulació.	26
4.1	Esquema de la plataforma de verificació HIL.	28
4.2	Esquema general de la comunicació entre el servidor host i una targeta acceleradora Alveo a través de PCIe. La plataforma proporciona el runtime, els drivers, el DMA, la memòria global i la infraestructura de comunicació amb els kernels.	32
4.3	Format uniforme de les taules d'intercanvi host-FPGA: cada fila correspon a un port del DUT (índex, amplada en bits i valor). La taula d'entrades la prepara el host amb els estímuls; la de sortides es reserva buida i l'omple el hardware amb els resultats. Font: elaboració pròpia.	34
5.1	Visió general de l'entorn de desenvolupament: flux des de la descripció i simulació en py4hw fins a l'execució sobre la FPGA Alveo U55C, amb les eines AMD/Xilinx (Vitis, Vivado i XRT) emprades en cada etapa.	36

5.2	Pantalla inicial de Vivado ML Edition 2023.2 utilitzat durant el desenvolupament del projecte.	37
5.3	Flux de compilació i verificació en Vitis HLS	39
5.4	Vistes externes de la targeta acceleradora AMD/Xilinx Alveo U55C utilitzada en el projecte.	41
5.5	Validació preliminar del flux host–FPGA–host mitjançant una aplicació senzilla de processament d’imatge.	43
5.6	Esquema simplificat del flux host – FPGA – host validat amb la prova preliminar d’inversió de píxels. El host prepara les dades d’entrada, la FPGA executa el <i>pipeline</i> reader – kernel RTL – writer i els resultats es retornen al host per generar la imatge de sortida.	44
5.7	Esquema conceptual del redirector HIL: el DUTProxyXRT substitueix el DUT dins la simulació <i>py4hw</i> i, en lloc de calcular, reenvia les entrades a la FPGA i en recull les sortides; el <i>Waveform</i> compara els senjals simulats amb els reals. S’indica una línia de codi representativa de cada bloc.	48
5.8	Comparació entre el DUT Add/Sub simulat en <i>py4hw</i> i el bloc DUTProxyXRT que redirigeix l’execució cap a la FPGA.	49
5.9	Comparació entre el DUT RGB2YCbCr simulat en <i>py4hw</i> i el bloc DUT-ProxyXRT executat sobre la FPGA.	50
5.10	Esquema del DUT Add/Sub descrit en <i>py4hw</i>	51
5.11	Separació d’una imatge RGB en les components Y, Cr i Cb. La component Y conté principalment la informació de luminància, mentre que Cr i Cb representen la informació de crominància associada al vermell i al blau, respectivament. . . .	52
5.12	Esquema del DUT RGB2YCbCr descrit en <i>py4hw</i>	53
5.13	Estructura interna del <i>DUT wrapper</i> plantejat com a kernel RTL. El <i>wrapper</i> adapta les entrades i sortides del DUT al protocol AXI4-Stream mitjançant els blocs <i>axi2reg</i> , <i>axi2clk</i> i <i>reg2axi</i> , mentre que la màquina d’estats <i>KernelFSM</i> en coordina el control.	55
5.14	Fitxers generats automàticament al directori del DUT després d’executar el flux de construcció per a Vitis.	59
6.1	Verificació del DUT Add/Sub a la finestra Workbench, comparant les sortides del model software amb les obtingudes sobre la FPGA.	63

6.2	Verificació del DUT RGB2YCbCr per a un conjunt de valors d'entrada, comparant els resultats software i hardware.	65
6.3	Verificació del DUT RGB2YCbCr per al píxel d'entrada 0x87CEEB, comparant els resultats software i hardware.	65
6.4	Sortida de l'execució del test de rendiment per al DUT FPAdder_SP, on es mostra la generació del binari, la superació dels tests i la mesura de mostres per segon en simulació software i en execució sobre FPGA.	68

Índex de taules

2.1	Comparació de les principals aproximacions relacionades amb el projecte.	11
5.1	Informació principal de la plataforma Vitis seleccionada per a la targeta AMD/- Xilinx Alveo U55C.	42
6.1	Verificació dirigida del DUT Add/Sub: comparació bit a bit entre la simulació software (<i>py4hw</i>) i l'execució sobre FPGA. Valors en hexadecimal de 32 bits. . .	64
6.2	Verificació per mostreig del DUT RGB2YCbCr: comparació bit a bit entre la simulació software (<i>py4hw</i>) i l'execució sobre FPGA. Píxels d'entrada i sortides YCbCr empaquetades, en hexadecimal de 32 bits (ordre de bytes Y Cb Cr). Els tres primers casos corresponen als components purs vermell, verd i blau; el píxel 0x87CEEB és el blau cel emprat a la prova preliminar del mòdul RTL (Secció 5.2.2). .	66
6.3	Verificació funcional dels DUTs addicionals mitjançant la bateria de tests unitaris (<i>pytest</i>). Cada DUT s'avalua amb mil vectors aleatoris, comparant bit a bit la sortida de la simulació software (<i>py4hw</i>) amb la de l'execució sobre FPGA. . . .	67
6.4	Rendiment: simulació software (<i>py4hw</i>) vs. execució sobre FPGA, per a cent mil avaluacions consecutives.	68

Glossari d'acrònims

Acrònims

Acrònim	Nom complet
API	<i>Application Programming Interface.</i>
AXI	<i>Advanced eXtensible Interface.</i>
AXI4-Stream	<i>Advanced eXtensible Interface 4 Stream Protocol.</i>
BRAM	<i>Block Random Access Memory.</i>
CI/CD	<i>Continuous Integration / Continuous Deployment.</i>
CLB	<i>Configurable Logic Blocks.</i>
CPU	<i>Central Processing Unit.</i>
DMA	<i>Direct Memory Access.</i>
DSP	<i>Digital Signal Processing.</i>
DUT	<i>Design Under Test.</i>
EDA	<i>Electronic Design Automation.</i>
FPGA	<i>Field Programmable Gate Array.</i>
HBM	<i>High Bandwidth Memory.</i>
HDL	<i>Hardware Description Language.</i>
HIL	<i>Hardware-in-the-Loop.</i>
HLS	<i>High-Level Synthesis.</i>
HW	<i>Hardware.</i>
I/O	<i>Input/Output.</i>
IP	<i>Intellectual Property.</i>
LUT	<i>Look-Up Table.</i>
PCIe	<i>PCI Express.</i>
PPM	<i>Portable Pixmap.</i>
RTL	<i>Register Transfer Level.</i>
SW	<i>Software.</i>
TCL	<i>Tool Command Language.</i>
UART	<i>Universal Asynchronous Receiver-Transmitter.</i>

XRT

Xilinx Runtime.

Capítol 1

Introducció

La verificació i validació de circuits digitals complexos és un procés crític que requereix temps i recursos computacionals significatius. Les simulacions software pures, tot i la seva flexibilitat, esdevenen lentes per a dissenys grans, mentre que la implementació completa en una plataforma FPGA (Field-Programmable Gate Array), encara que ràpida, redueix la capacitat de depuració. L'aproximació híbrida, on part del circuit es simula en software i una altra part s'executa en hardware, emergeix com una solució potent que combina el millor dels dos mons: la visibilitat i el control del software amb la velocitat del hardware.

Aquest projecte proposa el desenvolupament d'una plataforma que integri la simulació de circuits amb la llibreria Python *py4hw* amb l'execució accelerada en una placa FPGA connectada via PCI (Peripheral Component Interconnect).

1.1 Motivació i context del projecte

En el disseny de circuits digitals, una de les etapes més importants és la verificació. Abans d'implementar un circuit en hardware real, cal comprovar que el seu funcionament és correcte davant de diferents entrades i situacions. Aquesta comprovació es fa habitualment mitjançant simulació software, ja que permet observar els senyals interns, detectar errors i repetir proves de manera controlada.

Tot i això, quan els circuits creixen en complexitat o quan cal executar moltes proves, la simulació purament software pot esdevenir lenta. Aquesta limitació afecta directament el temps de desenvolupament, ja que cada modificació del disseny pot requerir tornar a executar simulacions llargues per validar-ne el comportament. Per aquest motiu, en molts àmbits del disseny digital s'estudia l'ús de hardware reconfigurable, com les FPGAs, per accelerar part d'aquest procés. L'acceleració, però, no és l'única raó per voler executar els dissenys sobre

hardware real. Una implementació que en l'entorn de simulació ens sembla correcta es pot comportar lleugerament diferent un cop ha passat pel procés de síntesi. Per exemple, en Verilog hi ha clàusules que només són de simulació, com ara els blocs `initial`. En el cas de `py4hw`, ens interessa verificar que la generació automàtica de Verilog a partir de Python no hi ha introduït cap error.

Les FPGAs permeten executar circuits digitals de manera molt més propera al hardware real i aprofitar el paral·lisme propi d'aquest tipus de dispositius. Això pot proporcionar una velocitat d'execució superior a la d'una simulació software tradicional. Tanmateix, implementar completament un disseny en FPGA també presenta inconvenients: el procés d'integració és més complex, la depuració és menys directa i l'observació dels senyals interns no és tan flexible com en una simulació en Python o en un entorn software.

Aquest projecte parteix d'aquesta necessitat de combinar els avantatges dels dos entorns. D'una banda, es vol mantenir la flexibilitat, la facilitat de prova i l'observabilitat que ofereix la simulació en Python mitjançant la llibreria `py4hw`. D'altra banda, es vol aprofitar la capacitat d'execució d'una FPGA per accelerar la verificació d'alguns blocs del circuit. Per aconseguir-ho, es planteja una arquitectura híbrida en què el control de la simulació es manté al host (ordinador o programa principal que controla l'execució i la comunicació amb la FPGA), mentre que una part del disseny es pot executar sobre una targeta FPGA.

En aquest treball, la FPGA s'integra mitjançant dues eines complementàries: *Vitis* i *XRT* (Xilinx Runtime). *Vitis* s'encarrega de la compilació: transforma un disseny en un *kernel* hardware, és a dir, una funció sintetitzada que s'executa físicament sobre la placa. *XRT*, en canvi, actua en temps d'execució i fa de pont entre la FPGA i el *host*, el programa que corre a la CPU (Central Processing Unit) i que controla tot el procés. A través de *XRT*, el host carrega el kernel, hi envia les dades i en recupera els resultats.

L'objectiu del treball no és substituir la simulació software, sinó construir un flux que connecti un DUT (*Design Under Test*), el circuit que es vol verificar, descrit en `py4hw` amb la seva execució real sobre FPGA, d'una manera més automàtica i reutilitzable.

Així, el projecte s'emmarca dins de les tècniques de co-simulació i verificació hardware / software. Aquesta aproximació pot ser especialment útil en entorns de recerca i prototipatge, on és important disposar d'un flux flexible, automatitzable i adaptable a diferents circuits digitals, sense perdre la capacitat de verificació i anàlisi pròpia de la simulació software.

1.2 Plantejament del problema i justificació

La simulació software de dissenys digitals grans pot arribar a ser molt costosa en temps de càlcul. A mesura que augmenta la complexitat del circuit o el nombre de proves a realitzar, el

temps necessari per simular-ne el comportament també creix, fet que pot limitar la productivitat durant el desenvolupament i dificultar l'execució de tests exhaustius.

D'altra banda, implementar tot el disseny directament sobre una FPGA ofereix una execució a velocitat de hardware, però introdueix dificultats que no apareixen en simulació. La principal és la pèrdua d'observabilitat: mentre que en un entorn software es pot inspeccionar qualsevol senyal intern a cada cicle, en hardware només són visibles els senyals que s'han exposat explícitament mitjançant recursos de depuració dedicats integrats a la placa, els quals consumeixen àrea i s'han de preveure abans de la síntesi. A això s'hi afegeix el cost de cada iteració (cada repetició del procés): qualsevol modificació del circuit obliga a repetir el flux de síntesi i implementació, que pot trigar des de minuts fins a hores, de manera que el cicle de depuració és molt més lent que en simulació. Aquestes limitacions són especialment notables quan es treballa amb circuits complexos.

Per aquest motiu, resulta útil disposar d'un sistema híbrid que permeti dividir el disseny entre software i hardware. Les parts més crítiques o costoses computacionalment es poden executar en hardware per aprofitar la velocitat de la FPGA, mentre que les parts que requereixen més control, observabilitat o anàlisi es poden mantenir en software. L'objectiu és que aquesta divisió sigui el més transparent possible per a l'usuari.

El repte principal del projecte és, per tant, crear una infraestructura de comunicació eficient entre l'entorn de simulació software, basat en *py4hw*, i el hardware implementat en FPGA. Aquesta comunicació ha de permetre transferir les dades de manera fiable, mantenir la coherència dels valors intercanviats i garantir una sincronització correcta entre l'execució en Python i l'execució en hardware.

1.3 Objectius generals i específics

L'objectiu principal és dissenyar i implementar una plataforma de test híbrida que permeti avaluar circuits digitals partitionats entre un entorn de simulació Python (*py4hw*) i una placa FPGA. Els objectius específics inclouen:

- Desenvolupar un entorn de simulació Python (amb *py4hw*) capaç de partitionar un disseny digital i delegar la simulació de subcircuits específics a una FPGA.
- Implementar un sistema en la FPGA capaç de rebre, executar i retornar els resultats de subcircuits especificats utilitzant kernels HDL de Xilinx .
- Assegurar la transparència del sistema, de manera que l'usuari pugui interactuar amb la simulació sense preocupar-se de la partition hardware/software.

- Validar la plataforma mitjançant tests d'equivalència per verificar que el comportament dels circuits sintetitzats a la FPGA coincideix exactament amb els seus models simulats en *py4hw*.

1.4 Metodologia i abast del treball

Per començar, s'ha analitzat el problema que motiva el projecte i s'ha realitzat una revisió bibliogràfica sobre co-simulació, acceleració de verificació amb FPGA, sistemes Hardware-in-the-Loop (HIL) i eines de disseny hardware basades en Python. Aquesta fase ha servit per situar el projecte dins del context existent i identificar els principals reptes tècnics a abordar. A continuació es detallen els passos seguits al llarg del projecte.

1. Estudi del problema i revisió de l'estat de l'art

En primer lloc, s'ha analitzat el context que motiva el projecte. La simulació software de circuits digitals ofereix una gran flexibilitat, observabilitat i capacitat de depuració, però executar un disseny directament sobre hardware obre la porta a verificar-ne el comportament en condicions reals d'execució. El repte és connectar tots dos mons de manera fiable, conservant els avantatges de l'entorn software. Aquesta fase també ha inclòs una revisió de conceptes relacionats amb la co-simulació, les plataformes FPGA, els entorns de verificació hardware/software i els fluxos de treball existents.

2. Estudi de les eines i de l'entorn de treball

A continuació, s'han estudiat les eines principals utilitzades durant el projecte. Per una banda, s'ha analitzat *py4hw* com a entorn de descripció i simulació de circuits digitals en Python. Per altra banda, s'han revisat les eines pròpies del flux FPGA, principalment Vivado, Vitis i XRT. Aquesta fase ha permès entendre el paper dels kernels, dels fitxers *.xo*, del fitxer final *.xclbin* i de la comunicació entre el programa host i la targeta FPGA.

3. Familiarització amb la placa FPGA i els tutorials oficials

Abans de desenvolupar el flux propi del projecte, ha estat necessari familiaritzar-se amb l'entorn real de treball, basat en una targeta FPGA AMD Alveo U55C. Aquesta fase ha inclòs l'ús del servidor, la configuració de l'entorn, la càrrega de les eines necessàries i l'execució de tutorials oficials de Vitis, Vivado i XRT. L'objectiu ha estat comprendre el procés bàsic de compilació i execució d'una aplicació accelerada: preparar els kernels, generar el fitxer *.xclbin*, carregar-lo a la FPGA, executar-lo des del host i recuperar els resultats.

4. Validació inicial del flux host-FPGA

Un cop entès el funcionament bàsic de l'entorn, s'han realitzat proves inicials per validar la comunicació amb la FPGA. Primer es va treballar amb una aplicació senzilla de processament d'imatge, en què el host enviava una imatge a la placa i el kernel hardware retornava una imatge amb els píxels invertits. Aquesta prova va servir per comprovar que el flux host-FPGA-host funcionava correctament: transferència de dades, execució del kernel i recuperació dels resultats.

Posteriorment, es va realitzar una segona prova amb un disseny RTL (Register Transfer Level) de conversió de color RGB (Red, Green and Blue) a YCbCr (Luminance, Blue-difference chrominance and Red-difference chrominance). En aquest cas es va generar el fitxer `.xclbin`, es va carregar a la FPGA i es va comprovar que l'execució hardware produïa els resultats esperats. Aquest pas va permetre passar d'un exemple genèric de processament d'imatge a un bloc RTL, més proper al tipus de circuit utilitzat posteriorment en el projecte.

5. Anàlisi del DUT de prova en *py4hw*

Després de validar l'entorn FPGA, s'ha treballat amb un DUT de prova ja proporcionat en *py4hw*. Aquest circuit s'ha utilitzat com a cas representatiu per desenvolupar i comprovar el flux proposat. En aquesta fase no s'ha creat el DUT des de zero, sinó que s'ha analitzat la seva estructura per entendre com *py4hw* representa els mòduls, els ports d'entrada i sortida, els senyals i la jerarquia interna del disseny.

6. Extracció i ús de la informació del DUT

A partir del DUT proporcionat, s'ha desenvolupat una part del flux capaç d'obtenir automàticament la informació necessària del circuit, com ara els ports d'entrada i sortida, el nom de cada senyal i la seva amplada en bits. Aquesta informació s'utilitza per adaptar la generació dels fitxers auxiliars al DUT concret que es vol verificar, evitant haver d'escriure manualment totes les connexions per a cada nou DUT.

7. Disseny de l'arquitectura del sistema

Un cop disponible la informació del DUT, s'ha definit l'arquitectura general del sistema. Aquesta arquitectura es basa en tres blocs principals: un kernel de lectura, el DUT implementat com a bloc hardware i un kernel d'escriptura.

El kernel de lectura envia les dades d'entrada cap al DUT, el DUT processa aquestes dades i el kernel d'escriptura recull les sortides generades. La comunicació entre aquests blocs es planteja mitjançant canals AXI4-Stream, un protocol estàndard de la indústria que funciona com una canonada de dades: la informació flueix de manera ordenada des d'un bloc cap al següent, sense necessitat d'indicar on s'ha de desar cada valor.

8. Implementació de la part software

Després de definir l'arquitectura general, s'ha implementat la part software del sistema. Aquesta part és l'encarregada d'automatitzar el procés a partir de la informació extreta del DUT i de preparar els elements necessaris perquè el circuit pugui integrar-se dins del flux FPGA.

També s'ha desenvolupat el programa host, que serà l'encarregat de controlar l'execució des de l'ordinador, carregar el binari de la FPGA, enviar les dades d'entrada i recuperar els resultats.

9. Implementació de la part hardware

En paral·lel, s'ha treballat en la part hardware del sistema. Aquesta part inclou la preparació dels kernels auxiliars de lectura i escriptura i la integració del DUT dins del flux de dades. El kernel de lectura s'encarrega d'enviar les dades cap al DUT, mentre que el kernel d'escriptura recull les sortides generades. D'aquesta manera, el DUT queda envoltat per una estructura que permet comunicar-lo amb el host mitjançant la infraestructura de la FPGA.

10. Generació dels fitxers del flux Vitis

Un cop definides les parts software i hardware, s'han generat els fitxers necessaris per construir el sistema accelerat amb Vitis. Aquests fitxers inclouen la configuració de connectivitat entre kernels, els scripts TCL (Tool Command Language) i els scripts de compilació. La seva funció és automatitzar el procés d'empaquetat, connexió i compilació dels blocs fins a obtenir el fitxer final que es carregarà a la FPGA.

11. Integració del sistema complet

Un cop desenvolupades les parts anteriors, s'ha integrat el flux complet. El sistema parteix d'un DUT definit en *py4hw*, n'obté la informació necessària, genera els fitxers auxiliars, prepara els kernels, construeix el sistema accelerat i permet executar-lo des del host. Aquesta integració és el punt central del projecte, ja que connecta la descripció del circuit en Python amb l'execució hardware en FPGA.

12. Tests i validació funcional

Finalment, s'ha plantejat una fase de tests i validació funcional. Aquesta fase consisteix a comprovar que els resultats obtinguts mitjançant l'execució en FPGA coincideixen amb els resultats esperats del model software. L'objectiu és verificar que el flux preserva el comportament funcional del DUT i que la comunicació entre el host i la FPGA és coherent. A partir d'aquesta validació, s'identifiquen les limitacions del sistema actual i les possibles línies de millora.

Capítol 2

Estat de l'art

Aquest capítol revisa les principals solucions relacionades amb la verificació de circuits digitals, la co-simulació amb FPGA —l'execució conjunta d'una part del sistema en simulació software i una altra en hardware real— i els entorns de disseny hardware basats en Python.

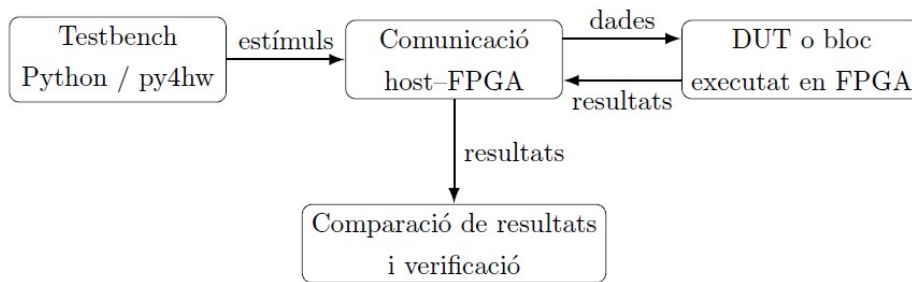
L'objectiu és identificar les limitacions de les solucions actuals i situar el projecte dins d'un enfocament híbrid en què el disseny es descriu i es controla des de *py4hw*, però s'executa físicament sobre la FPGA.

2.1 Verificació digital i simulació tradicional

La verificació funcional és una etapa essencial en el disseny de circuits digitals. El seu objectiu és comprovar que el circuit implementat es comporta d'acord amb les especificacions abans de passar a fases posteriors del desenvolupament. Normalment, el circuit que es vol validar s'anomena DUT (*Design Under Test*) i s'executa dins d'un entorn de proves o *testbench*.

Els fluxos tradicionals de verificació es basen principalment en simuladors HDL (Hardware Description Language), com VHDL (VHSIC + HDL: Very High Speed Integrated Circuit + Hardware Description Language), Verilog o SystemVerilog. Aquests entorns permeten aplicar estímuls al DUT, observar senyals interns i externs, generar formes d'ona i comprovar automàticament el comportament del circuit mitjançant assertions, checkers o regressions[1, 2].

Aquest tipus de verificació ofereix molta flexibilitat i una gran capacitat d'observació, però també presenta una limitació important: quan el disseny creix o cal executar molts tests, la simulació software pot ser lenta. Aquestes pràctiques de verificació sistemàtica i automatitzada -tests unitaris, regressions i, en projectes de software, la integració contínua (CI/CD)- són habituals en el desenvolupament de software, però la seva aplicació a la verificació sobre hardware real encara és poc estesa, ja que requereix integrar el dispositiu físic dins del flux de proves [3, 4].



Font: Elaboració pròpia.

Figura 2.1: Esquema conceptual d'una arquitectura híbrida de verificació.

Aquest problema motiva l'ús de tècniques d'acceleració, com l'execució parcial del circuit en FPGA, per reduir el temps de verificació sense perdre el control que ofereix el *testbench* en software [5, 4].

2.2 Co-simulació i emulació amb FPGA

Una de les alternatives a la simulació purament software és la co-simulació o emulació amb FPGA. Aquest tipus de solucions combinen l'execució d'una part del sistema en software amb l'execució d'una altra part en hardware reconfigurable. D'aquesta manera, el *testbench* o l'entorn de control pot continuar executant-se a l'ordinador host, mentre que el DUT o alguns blocs del disseny s'executen físicament en una FPGA.

La Figura 2.1 mostra de manera simplificada aquest model híbrid, on el *testbench* en Python manté el control de la verificació i una part del disseny s'executa en una FPGA.

Aquest enfocament permet aprofitar la velocitat del hardware sense perdre completament la flexibilitat del software. En un sistema híbrid, el software pot generar els estímuls, controlar l'execució i comprovar els resultats, mentre que la FPGA s'encarrega d'executar el bloc accelerat. Aquesta idea també apareix en entorns de Hardware-in-the-Loop, on un model o component real s'integra dins d'un procés de simulació per validar-ne el comportament en condicions més properes a una execució física [6].

Tot i els seus avantatges, aquestes solucions introdueixen nous reptes. Cal definir interfícies de comunicació entre el host i la FPGA, controlar la sincronització entre ambdós mons i garantir que les dades enviades i rebudes siguin coherents amb la simulació original. A més, si la comunicació és massa freqüent o es fa amb paquets molt petits, la latència pot reduir el benefici obtingut per l'acceleració hardware [5, 4].

Aquest projecte parteix d'aquesta idea d'execució híbrida, però aplicada a un entorn de

verificació basat en Python. L'objectiu és que *py4hw* mantingui el control de la simulació i que determinats blocs del circuit puguin ser executats en una FPGA, resolent la comunicació i la sincronització necessàries entre el *testbench* software i el hardware accelerat.

2.3 Disseny i verificació de hardware amb Python

A més dels llenguatges HDL tradicionals, existeixen frameworks que utilitzen Python per descriure, simular o verificar circuits digitals. Aquestes eines busquen aprofitar la flexibilitat de Python per millorar la productivitat, automatitzar proves i facilitar la integració amb altres eines software [3].

En aquest projecte, l'entorn utilitzat és *py4hw* [3, 7]. Aquest framework permet descriure circuits digitals en Python, simular-los i treballar amb una estructura jeràrquica del disseny. A més, facilita la verificació interactiva i la generació de representacions hardware del circuit. Per aquest motiu, *py4hw* és la base adequada per plantejar una extensió cap a una execució híbrida, on el *testbench* i el control es mantenen en Python mentre que determinats blocs del circuit s'executen en una FPGA [3].

2.4 Plataformes FPGA per a acceleració: Vitis, XRT i Alveo

Les FPGA modernes no només s'utilitzen com a dispositius de prototipat hardware, sinó també com a plataformes d'acceleració connectades a un ordinador host. En aquest tipus d'arquitectura, el programa principal s'executa al processador de l'ordinador, mentre que les parts més costoses del càlcul es deleguen a la FPGA. Les targetes AMD/Xilinx Alveo formen part d'aquest tipus de plataformes i estan pensades per integrar-se en sistemes de servidor mitjançant PCIe [8].

L'ecosistema Vitis/XRT proporciona les eines necessàries per desenvolupar aquest tipus d'aplicacions. Vitis permet crear i empaquetar kernels hardware, tant a partir de HLS com de RTL, i generar el fitxer final que es carrega a la FPGA. XRT (*Xilinx Runtime*) actua com a capa de runtime entre l'aplicació host i el dispositiu FPGA. A través d'aquesta capa, el programa host pot carregar el fitxer d'acceleració, reservar buffers de memòria, transferir dades i llançar l'execució dels kernels [9, 10].

En una arquitectura típica, la comunicació entre l'host i la FPGA es realitza a través de PCIe i mecanismes DMA, que permeten moure blocs de dades entre la memòria del sistema i la memòria del dispositiu amb poca intervenció de la CPU. Dins de la FPGA, els diferents kernels es poden connectar mitjançant interfícies AXI, especialment AXI4-Stream quan es vol construir un flux de dades entre blocs hardware [8, 11].

Aquest model encaixa amb l'objectiu del projecte, ja que permet mantenir el control de la verificació en software, dins de l'entorn *Python/py4hw*, i delegar l'execució d'un subcircuit a la FPGA. En aquest cas, Vitis i XRT no són l'objectiu final del treball, sinó la infraestructura que permet construir el pont entre el *testbench* en Python i el DUT executat com a kernel hardware.

2.5 Limitacions actuals i posicionament del projecte

Les solucions revisades mostren que existeixen eines molt madures per a la verificació de circuits digitals. Els simuladors HDL permeten validar el comportament del disseny amb molta flexibilitat i observabilitat, però poden esdevenir lents quan el nombre de proves o la complexitat del circuit augmenta. D'altra banda, les plataformes d'emulació i acceleració amb FPGA permeten executar dissenys a més velocitat, però acostumen a requerir una infraestructura més complexa i una integració menys directa amb entorns de verificació d'alt nivell [5, 4].

També existeixen frameworks basats en Python que faciliten el disseny i la verificació de hardware, com *py4hw*. Aquests entorns milloren la productivitat i permeten descriure, simular i provar circuits des d'un llenguatge flexible. Tot i això, la connexió entre aquests frameworks i una FPGA d'acceleració no és habitualment transparent: cal generar *wrappers* (capes d'adaptació que connecten el disseny amb les interfícies del kernel), kernels, fitxers de connexió, scripts de compilació i mecanismes de comunicació entre el software i el hardware [3, 9, 10].

El present projecte se situa en aquest punt intermedi. L'objectiu no és substituir els simuladors HDL ni les plataformes comercials d'emulació, sinó explorar una solució híbrida en què *py4hw* manté el control de la verificació en Python i la FPGA executa determinats blocs del circuit per accelerar-ne el comportament. Així, el projecte es centra en la generació i integració de la infraestructura necessària perquè un DUT descrit en Python pugui ser connectat amb l'entorn Vitis/XRT i executat com a part d'un flux de verificació hardware–software.

D'aquesta manera, el treball respon a una necessitat concreta: apropar l'acceleració FPGA a un entorn de verificació Python de manera més automatitzada, mantenint la coherència de les dades, la sincronització entre host i FPGA i la possibilitat de comparar els resultats amb la simulació software original.

Taula 2.1: Comparació de les principals aproximacions relacionades amb el projecte.

Solució	Avantatges	Limitacions
Simulació HDL tradicional	Alta observabilitat i flexibilitat en el <i>testbench</i>	Pot ser lenta en dissenys grans o regressions llargues
Emulació o co-simulació amb FPGA	Permet executar blocs hardware amb més velocitat	Requereix interfícies, sincronització i infraestructura específica
Frameworks Python-HDL	Milloren la productivitat i faciliten la verificació en Python	No sempre integren l'execució en FPGA de forma transparent
Vitis/XRT/Alveo	Proporcionen eines per executar kernels hardware des de l'host	Cal generar kernels, <i>wrappers</i> , fitxers de connexió i scripts de compilació

Font: elaboració pròpia

Capítol 3

Fonaments teòrics i conceptes clau

3.1 Arquitectura bàsica d'una FPGA

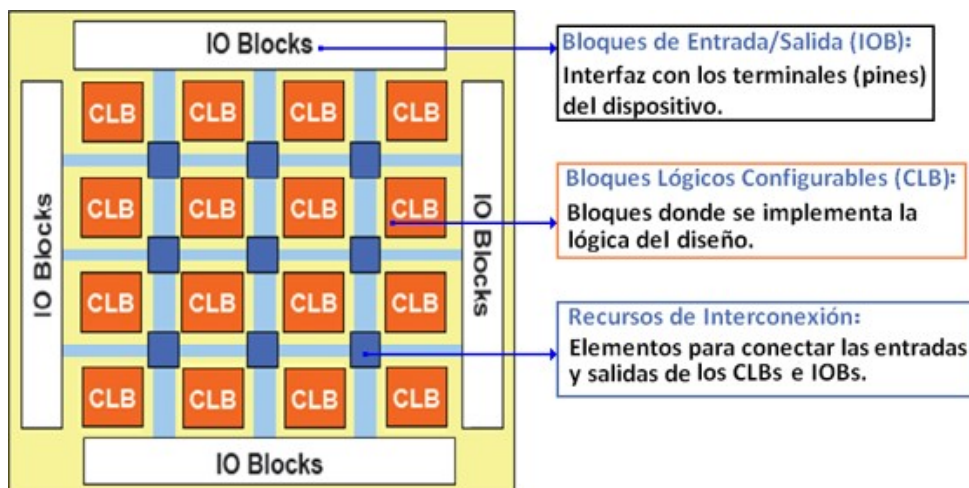
Una FPGA és un dispositiu hardware reconfigurable que permet implementar circuits digitals mitjançant una xarxa de recursos programables. A diferència d'un processador convencional, que executa instruccions de manera seqüencial, una FPGA permet descriure directament una arquitectura hardware específica, aprofitant el paral·lelisme inherent dels circuits digitals.

Aquesta característica fa que les FPGA siguin especialment útils en aplicacions on es requereix una execució ràpida, determinista i amb baixa latència. En lloc d'executar un algoritme com una seqüència d'instruccions, la funcionalitat es materialitza en forma de blocs lògics, registres, memòries i connexions internes. D'aquesta manera, diverses operacions poden executar-se simultàniament, sempre que el disseny hardware ho permeti.

En el context d'aquest projecte, la FPGA s'utilitza com una plataforma d'acceleració hardware. L'objectiu no és substituir completament la simulació software, sinó executar físicament sobre la FPGA el circuit sota test, mantenint el control i la verificació des de l'entorn Python. Aquesta aproximació permet combinar la flexibilitat del software amb la capacitat d'execució paral·lela i eficient pròpia de les FPGA.

3.1.1 Lògica programable i recursos interns

La lògica programable és l'element principal d'una FPGA. Està formada per una matriu de blocs lògics configurables, interconnectats entre si mitjançant una xarxa de connexions programable. Aquesta estructura permet implementar circuits digitals específics a partir d'una descripció hardware, com pot ser Verilog, VHDL o un model generat automàticament des d'un entorn de més alt nivell.



Font: [12].

Figura 3.1: Estructura interna general d'una FPGA.

Els blocs lògics configurables, habitualment anomenats CLB, contenen recursos com taules de consulta o LUT i elements d'emmagatzematge, com biestables o registres. Les LUT permeten implementar funcions combinacionals, mentre que els registres permeten descriure comportament seqüencial i sincronitzat amb un senyal de rellotge. En architectures AMD/-Xilinx UltraScale, els CLB es basen en slices que agrupen LUT, elements d'emmagatzematge, lògica de carry i multiplexors interns, facilitant la implementació eficient de funcions lògiques i aritmètiques bàsiques.

A més de la lògica programable general, les FPGA incorporen recursos especialitzats que permeten millorar el rendiment i reduir l'ús de lògica configurable. Entre aquests recursos destaquen les memòries internes de tipus BRAM, utilitzades per emmagatzemar dades dins del dispositiu, i els blocs DSP, optimitzats per a operacions aritmètiques com multiplicacions, sumes i acumulacions. Aquests blocs són especialment importants quan el circuit implementat requereix processament de dades o operacions repetitives.

Tots aquests elements es connecten mitjançant la xarxa d'interconnexió interna de la FPGA. Aquesta xarxa permet unir els diferents blocs del disseny segons l'arquitectura descrita per l'usuari. Per aquest motiu, la implementació final no consisteix només a sintetitzar la lògica del circuit, sinó també a ubicar els diferents recursos dins del dispositiu i establir les connexions necessàries entre ells.

En aquest projecte, aquests recursos interns són rellevants perquè el circuit sota test es transforma en una implementació hardware que s'executa dins de la FPGA. Això permet aprofitar el paral·lisme i la naturalesa física del hardware per executar físicament una part del procés de verificació, mantenint alhora el control de l'execució des de l'entorn software.

3.1.2 FPGA com a plataforma d'acceleració hardware

Les FPGA són especialment adequades com a plataformes d'acceleració hardware perquè permeten implementar arquitectures específiques per a una aplicació concreta. A diferència d'un processador generalista, que executa instruccions de manera seqüencial, una FPGA permet construir un circuit digital adaptat a la funcionalitat que es vol executar. Això fa possible aprofitar el paral·lelisme intern del hardware i reduir el temps d'execució de determinades operacions.

En un sistema accelerat, el processador principal o *host* manté el control general de l'aplicació, mentre que la FPGA s'encarrega d'executar les parts més costoses o repetitives. Aquesta divisió permet combinar la flexibilitat del software amb l'eficiència del hardware. El software pot gestionar la preparació de les dades, la configuració de l'execució i la comprovació dels resultats, mentre que el hardware pot executar el circuit amb una arquitectura paral·lela i dedicada.

Aquest model és especialment interessant en l'àmbit de la verificació de circuits digitals. La simulació software ofereix una gran capacitat d'observació i control, però pot ser lenta quan el nombre de cicles o la complexitat del disseny augmenta. En canvi, una FPGA pot executar físicament el circuit sota test, aprofitant els recursos interns del dispositiu i reduint el temps necessari per obtenir els resultats.

En aquest projecte, la FPGA s'utilitza per executar físicament una part del procés de verificació. El circuit sota test es transforma en una implementació hardware executable dins del dispositiu, mentre que l'entorn software continua encarregant-se de generar els estímuls, controlar l'execució i comparar els resultats obtinguts. Aquesta aproximació permet mantenir una interfície d'ús propera a la simulació tradicional, però incorporant l'avantatge de l'execució hardware.

Per tant, la FPGA no s'utilitza com un sistema independent, sinó com un recurs d'execució hardware integrat dins d'un flux de verificació híbrid. Aquesta idea és la base del sistema desenvolupat en aquest treball, on el disseny definit en Python es pot portar cap a una implementació hardware i executar-se en una placa FPGA controlada des del host.

3.2 Verificació de circuits digitals

La verificació de circuits digitals és una etapa fonamental dins del procés de disseny hardware. El seu objectiu és comprovar que el circuit implementat es comporta d'acord amb les especificacions funcionals abans de passar a fases posteriors, com la síntesi, la implementació física o la integració en un sistema més gran. Aquesta etapa permet detectar errors de disseny de manera anticipada i reduir el cost associat a corregir-los en fases més avançades del desenvolupament.

En aquest context, el circuit que es vol comprovar s'anomena habitualment DUT, de l'anglès

Design Under Test. El DUT s'executa dins d'un entorn de proves o *testbench*, encarregat de generar els estímuls d'entrada, aplicar-los al circuit i observar les sortides obtingudes. A partir d'aquestes sortides, es pot determinar si el comportament del circuit és correcte o si existeixen diferències respecte al resultat esperat.

Tradicionalment, aquesta verificació es realitza mitjançant simulació software utilitzant llenguatges de descripció hardware, com VHDL, Verilog o SystemVerilog, o entorns de més alt nivell. Aquests simuladors ofereixen un alt grau de control i observabilitat, ja que permeten inspeccionar senyals internes, generar formes d'ona i executar proves de manera repetible. Tanmateix, quan el disseny augmenta de mida o quan cal executar un gran nombre de cicles de simulació, el temps necessari pot créixer de manera significativa.

Per aquest motiu, en alguns casos resulta interessant combinar la flexibilitat de la simulació software amb l'execució hardware sobre una FPGA. Aquesta aproximació permet mantenir el control del procés de verificació des del software, però delegar l'execució d'una part del circuit a un dispositiu hardware reconfigurable. D'aquesta manera, es pot reduir el temps d'execució de determinades proves sense perdre completament la capacitat de control pròpia d'un entorn de simulació.

3.2.1 Simulació software i les seves limitacions

La simulació software és una de les tècniques més utilitzades per verificar circuits digitals abans de passar a la seva implementació física. En aquest tipus de simulació, el comportament del circuit es descriu mitjançant un llenguatge de descripció hardware, com Verilog o VHDL, o bé mitjançant un entorn de més alt nivell. El simulador s'encarrega d'executar aquesta descripció, calcular l'evolució dels senyals al llarg del temps i permetre l'observació del comportament del disseny.

Un dels principals avantatges de la simulació software és el seu alt grau de flexibilitat. Permet modificar fàcilment el disseny, repetir proves, introduir diferents estímuls d'entrada i analitzar amb detall els senyals internes del circuit. Aquesta capacitat d'observació és especialment important durant les primeres fases de desenvolupament, ja que facilita la detecció i correcció d'errors funcionals.

Tot i aquests avantatges, la simulació software presenta una limitació important: el temps d'execució. Encara que el circuit descrit sigui inherentment paral·lel, el simulador s'executa sobre un processador generalista, que ha d'anar calculant l'evolució dels senyals i dels esdeveniments de manera seqüencial o parcialment paral·lela. Això fa que, a mesura que augmenta la mida del disseny o el nombre de cicles simulats, el temps necessari per completar les proves pugui créixer de manera significativa.

Aquesta limitació és especialment rellevant quan es volen executar bancs de proves extensos, regressions automàtiques o simulacions amb un gran volum d'estímuls. En aquests casos, el cost computacional de la simulació pot reduir la productivitat del procés de verificació i dificultar l'exploració exhaustiva de possibles casos de prova.

Per aquest motiu, la simulació software continua sent molt útil per al control, la depuració i l'anàlisi detallada del circuit, però pot no ser suficient quan es busca executar proves llargues o repetitives de manera eficient. Aquesta situació motiva l'ús de plataformes d'acceleració hardware, com les FPGA, que permeten executar físicament una part del circuit i reduir el temps necessari per obtenir els resultats de verificació.

3.2.2 Simulació híbrida i Hardware-in-the-Loop

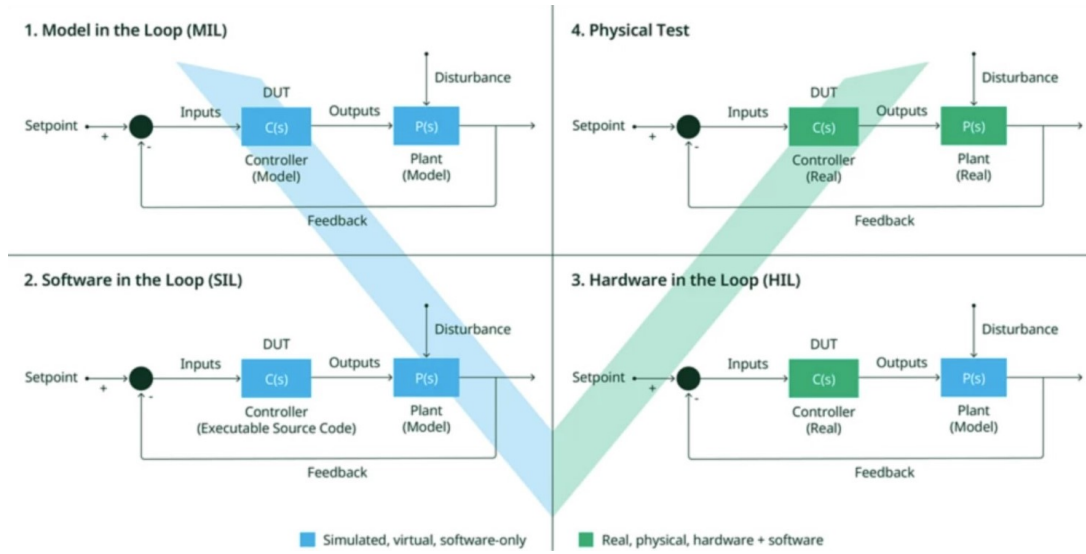
La simulació híbrida consisteix a combinar l'execució software amb l'execució hardware dins d'un mateix procés de verificació. En aquest model, una part del sistema es manté en un entorn de simulació, on és fàcil generar estímuls, controlar l'execució i observar resultats, mentre que una altra part s'executa físicament en un dispositiu hardware, com una FPGA. D'aquesta manera, es busca aprofitar els avantatges de tots dos entorns: la flexibilitat del software i la velocitat d'execució del hardware.

Una de les formes més habituals de simulació híbrida és el concepte de *Hardware-in-the-Loop* [6, 4]. En aquest tipus de sistema, el hardware real forma part del bucle de verificació. El software genera les entrades o estímuls del sistema, els envia al dispositiu hardware, espera l'execució del circuit i rep els resultats obtinguts. Posteriorment, aquests resultats es poden analitzar o comparar amb els valors esperats dins del mateix entorn software. La figura 3.2 mostra la ubicació del *Hardware-in-the-Loop* dins del flux de desenvolupament i verificació.

Aquest enfocament és especialment útil quan es vol accelerar la verificació d'un circuit sense perdre completament la capacitat de control que proporciona la simulació. En una simulació purament software, totes les operacions del circuit són calculades pel processador del host. En canvi, en una simulació híbrida, la part del disseny implementada en hardware s'executa directament sobre la FPGA, aprofitant el paral·lisme i els recursos físics del dispositiu.

En el context d'aquest projecte, el concepte de *Hardware-in-the-Loop* s'aplica a la verificació de circuits digitals descrits amb Python. El circuit sota test es pot transformar en una implementació hardware i executar-se en una FPGA, mentre que l'entorn software manté el control del *testbench*. Això permet generar els estímuls des de Python, enviar-los cap al hardware, recuperar les sortides i comparar-les amb els resultats esperats.

Tot i els avantatges d'aquest enfocament, la simulació híbrida també introdueix nous reptes. Cal definir una interfície clara entre el software i el hardware, establir un format de dades



Font: [13].

Figura 3.2: Ubicació del *Hardware-in-the-Loop* dins del flux de desenvolupament i verificació.

adequat, sincronitzar correctament l'execució i garantir que els resultats retornats per la FPGA siguin coherents amb el model simulat. A més, el temps de comunicació entre el host i la FPGA pot introduir un cost addicional que s'ha de tenir en compte en l'avaluació del sistema.

Així, la simulació híbrida no substitueix completament la simulació software tradicional, sinó que la complementa. La seva utilitat principal és permetre que les parts més costoses o repetitives del procés de verificació s'executin en hardware, mantenint alhora la capacitat de control, configuració i comprovació pròpia de l'entorn software.

3.3 Descripció hardware amb Python

3.3.1 *py4hwcom* a entorn de disseny i simulació

py4hw és un framework de codi obert per descriure, simular, visualitzar i generar circuits digitals des de Python [3]. A diferència de les eines de síntesi d'alt nivell, *py4hw* no parteix d'una descripció algorítmica que cal sintetitzar, sinó que segueix un enfocament *bottom-up*: el circuit es construeix a partir de portes i blocs lògics bàsics, de manera que el dissenyador manté el control sobre l'estructura del hardware resultant. Aquesta filosofia permet, a més, generar Verilog amb un gra molt fi a partir del model Python.

Una característica distintiva de *py4hw* és la separació explícita entre estils de disseny —estructural, combinacional, seqüencial i algorítmic—, que dona claredat al model i en facilita la simulació [3].

El disseny es representa com una jerarquia d'objectes en memòria: cada circuit és un objecte Python que pot contenir altres subcircuits, els seus ports d'entrada i sortida, i els senyals (wires) que els connecten. Aquesta jerarquia és directament accessible i manipulable des de Python, sense necessitat d'una representació intermèdia (IR) separada, fet que és clau per a aquest projecte: permet recórrer el model i extreure'n automàticament els ports, els noms dels senyals i les seves amplades en bits.

Els elements bàsics amb què es treballa són:

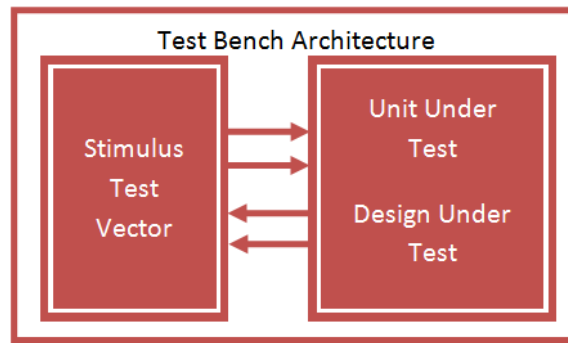
- contenidor de sistema (la instància `HWSystem`)
- els senyals, que es declaren amb una amplada determinada (exemple: `hw.wire('a', 32)`)
- els circuits, descrits com a classes que exposen els seus ports

Sobre aquests elements, *py4hw* ofereix blocs específics per generar estímuls de simulació, com ara **Constant**, que fixa un valor constant, i **Sequence**, que aplica una seqüència repetitiva de valors a un senyal [3].

El simulador integrat és de tipus *cycle-based* [3]. En inicialitzar-se, obté una vista aplanada de tots els elements primitius del sistema i els classifica en combinacionals o seqüencials. Les cadenes combinacionals s'ordenen topològicament —cosa possible perquè no s'admeten bucles asíncrons— de manera que es poden avaluar correctament en una sola passada per cicle, sense necessitat de propagació d'esdeveniments; un cop avaluades, els elements seqüencials s'actualitzen en qualsevol ordre. Aquest model determinista és el que proporciona els resultats de referència amb què, més endavant, es compara l'execució sobre la FPGA.

Finalment, *py4hw* posa un èmfasi especial en la visualització i la verificació interactiva. La classe **Schematic** genera automàticament l'esquemàtic d'un circuit a partir de la seva jerarquia, i l'entorn interactiu (*Workbench*) permet inspeccionar, a cada pas de simulació, la jerarquia del disseny, els valors actuals dels senyals i l'esquemàtic, mentre que la finestra de formes d'ona (*Waveform*) en mostra l'evolució temporal. Aquestes eines són les que, en aquest treball, fan visible la comparació entre la simulació software i l'execució física del DUT (Capítol 6).

Per tot això, *py4hw* és una base especialment adequada per plantejar una extensió cap a l'execució híbrida: la introspecció del model en memòria permet automatitzar la generació de la infraestructura de la FPGA, la generació de Verilog proporciona l'RTL del circuit, i el control de la verificació pot romandre en Python.



Font: [14].

Figura 3.3: Arquitectura general d'un *testbench*: el vector d'estímuls aplica les entrades al DUT (*Unit/Design Under Test*) i se'n recullen les sortides.

3.3.2 DUT: circuit sota test

En el context d'aquest treball, el DUT (*Design Under Test*) és el circuit que es vol verificar, descrit com un objecte de *py4hw*. Com qualsevol circuit del framework, queda definit per un conjunt de ports d'entrada i de sortida, cadascun amb una amplada en bits, i per una estructura interna que pot ser jeràrquica, és a dir, composta al seu torn de subcircuits més senzills. El DUT no s'avalua de manera aïllada, sinó dins d'un entorn de proves (*testbench*) que li aplica els estímuls d'entrada i en recull les sortides per comparar-les amb els valors esperats, tal com mostra la Figura 3.3.

El que importa de cara a la plataforma desenvolupada no és tant la funció concreta que calcula el DUT, sinó la seva interfície: quants ports té, quins són d'entrada i quins de sortida, i quina amplada té cadascun. Aquesta informació, accessible mitjançant la introspecció del model en memòria (Secció 3.3.1), és la que permet adaptar automàticament la resta del flux a qualsevol circuit, sense haver d'escriure manualment les connexions per a cada cas. Des d'aquest punt de vista, el DUT es tracta com una unitat: l'entorn de verificació l'avalua com un bloc més dins del graf de simulació, independentment de la seva complexitat interna.

Per desenvolupar i validar la plataforma s'han emprat dos DUTs de complexitat creixent, tots dos descrits en *py4hw*: un mòdul aritmètic Add/Sub i un convertidor d'espai de color RGB a YCbCr. El primer, amb dues entrades i dues sortides, serveix com a cas mínim per validar el flux; el segon, amb més ports d'entrada i senyals de control, exercita escenaris més exigents de sincronització. Els detalls de cada circuit -ports, operacions internes i descripció- es presenten al Capítol 5 (Secció 5.4.1).

3.3.3 Generació de Verilog a partir del model Python

Un dels components fonamentals de *py4hw* és la generació de RTL: a partir del model descrit en Python, el framework pot emetre el codi Verilog equivalent del circuit [3]. En la pràctica, aquesta generació es fa construint un objecte `py4hw.VerilogGenerator` a partir del circuit i invocant-ne el mètode `getVerilogForHierarchy()`, que recorre tota la jerarquia del disseny i en retorna el Verilog. Atès l'enfocament *bottom-up* del framework, aquesta generació és de grafi i reflecteix directament l'estructura descrita pel dissenyador.

Aquesta capacitat és precisament la que fa possible el pont entre l'entorn de simulació i la FPGA. El DUT, dissenyat i verificat en *py4hw*, s'exporta a Verilog, i aquest RTL és el que posteriorment s'empaqueta com a kernel per integrar-lo en el flux de Vitis (Capítol 5). Cal remarcar que aquesta exportació preserva el comportament del circuit: el Verilog generat descriu el mateix circuit que s'ha simulat en Python, de manera que el que s'executa sobre la FPGA és funcionalment equivalent al model software. Aquesta equivalència és el fonament sobre el qual descansa tota la verificació del projecte, que consisteix a comprovar que les sortides obtingudes sobre el hardware coincideixen, bit a bit, amb les de la simulació de referència.

El detall concret de com s'invoca aquesta generació dins del flux automatitzat, i com el Verilog resultant s'integra amb la infraestructura de la FPGA, es descriu al Capítol 5 (Secció 5.3.1).

3.4 Execució hardware i comunicació host-FPGA

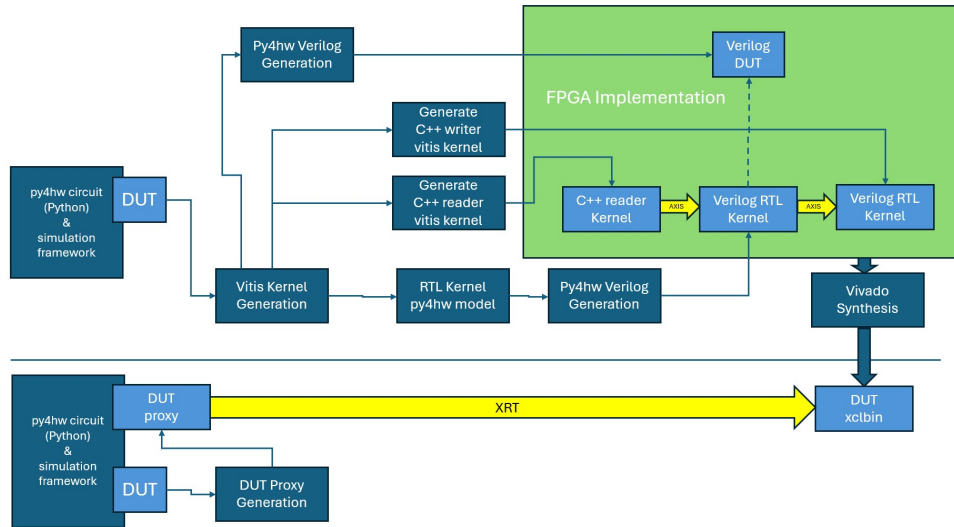
3.4.1 Flux Vivado/Vitis

El pas d'una descripció hardware a un binari executable sobre la FPGA es recolza en dues eines complementàries d'AMD/Xilinx: Vivado i Vitis. Vivado s'encarrega de la síntesi, la implementació física i l'empaquetament de blocs com a propietat intel·lectual (IP), mentre que Vitis gestiona la compilació dels *kernels* i la seva integració en el binari d'acceleració.

Conceptualment, el flux s'organitza en dues etapes:

1. cada bloc funcional es transforma en un kernel encapsulat en un fitxer amb extensió `.xo`
2. aquests fitxers `.xo` es combinen durant l'etapa de *link* per generar el fitxer final `.xclbin`, que es carrega a la FPGA mitjançant XRT

La Figura 3.4 mostra de manera simplificada aquest procés: els kernels es compilen o empaqueten inicialment com a fitxers `.xo` i, posteriorment, l'eina `v++` els enllaça juntament amb la configuració de connectivitat per generar el fitxer final `.xclbin`. Aquest fitxer és el binari que es carrega a la FPGA mitjançant XRT.



Font: Elaboració pròpia.

Figura 3.4: Flux de compilació amb Vitis per generar el fitxer `.xclbin` a partir dels kernels del sistema

Un *kernel* es pot obtenir per dues vies:

- La primera, mitjançant síntesi d'alt nivell (HLS), parteix d'una descripció en C/C++ que Vitis HLS converteix en RTL i empaqueta automàticament.
- La segona parteix d'un disseny RTL ja existent, que s'empaqueta directament com a *kernel* en Vivado.

En aquest projecte, el DUT —descriu en *py4hw* i exportat a Verilog— segueix aquesta segona via, ja que es disposa del seu RTL i interessa preservar-ne exactament el comportament.

Empaquetar un disseny RTL com a kernel és, conceptualment, un procés de dos passos:

- primer Vivado empaqueta el RTL com a IP (generant-ne la descripció de component i les interfícies AXI)
- després aquest IP, juntament amb un fitxer que en descriu la interfície de cara al linker, s'agrupa en el fitxer `.xo`. A l'entorn gràfic de Vivado, tota aquesta tasca es dispara amb una única acció (*Generate RTL Kernel*).

El fet rellevant per a aquest treball és que aquesta acció gràfica no fa res més que executar una seqüència fixa i determinista d'ordres Tcl. Per tant, aquestes mateixes ordres es poden escriure en un script i fer que Vivado les executi en mode no interactiu, sense obrir la interfície

gràfica: tot allò que es fa manualment amb un clic es pot deixar escrit perquè s'executi de forma automàtica.

Aquesta possibilitat és la base de l'automatització proposada en el projecte, en què un programa en Python genera els fitxers i scripts necessaris a partir d'un DUT descrit en *py4hw* i llança Vivado per empaquetar-ne el kernel sense cap intervenció manual. La implementació concreta d'aquesta seqüència es detalla al Capítol 5 (Secció 4.3.3).

3.4.2 Kernels hardware

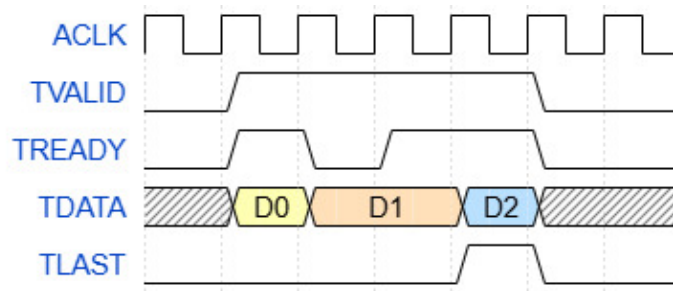
En el model d'acceleració de Vitis, la unitat de còmput que s'executa sobre la FPGA és el *kernel*: una funció sintetitzada que es materialitza com un circuit dins del dispositiu i que el host pot llançar a través del runtime XRT [9]. Mentre que el programa principal s'executa a la CPU, el kernel s'encarrega d'executar físicament la part del càlcul que se li ha delegat, aprofitant el paral·lelisme del hardware.

Un kernel es pot obtenir per dues vies, ja introduïdes a la Secció 3.4.1: a partir de síntesi d'alt nivell (HLS), que parteix d'una descripció en C/C++, o a partir d'un disseny RTL ja existent, que s'empaqueta directament.

Sigui quin sigui l'origen, un cop empaquetat el kernel exposa dos tipus d'interfície ben diferenciats:

- una interfície de *control*, mitjançant la qual el host n'inicia l'execució i en consulta l'estat; el mecanisme estàndard (*ap_ctrl*) es basa en un conjunt de senyals de handshake —típicament **ap_start** per arrencar i **ap_done**, **ap_idle** i **ap_ready** per senyalitzar la finalització i la disponibilitat— que permeten una crida síncrona: el host llança el kernel i espera que aquest indiqui que ha acabat abans de llegir-ne els resultats
- una o més interfícies de *dades*, per les quals entren i surten els valors a processar

Aquesta separació entre control i dades és la que permet construir sistemes de diversos kernels: el host orquestra l'execució a través de les interfícies de control, mentre que les dades poden circular directament entre kernels sense passar per la CPU. En aquest projecte, el DUT no s'executa com un kernel aïllat, sinó dins d'una estructura de tres kernels —un de lectura, el DUT *wrapper* i un d'escriptura— connectats com un flux de dades; aquesta organització es descriu a la Secció 4.3.1 i se'n detalla la implementació al Capítol 5.



Font: Elaboració pròpia.

Figura 3.5: Handshake de l'AXI4-Stream: la dada (TDATA) només avança quan TVALID i TREADY estan actius alhora; quan TREADY baixa, l'element es manté fins que el receptor el pot acceptar. TLAST marca l'últim element del bloc.

3.4.3 Interfícies AXI i transferència de dades

La comunicació entre els blocs d'un sistema accelerat es construeix sobre el conjunt d'interfícies AXI (*Advanced eXtensible Interface*), de la família de protocols AMBA (*Advanced Microcontroller Bus Architecture*) d'Arm, que és l'estàndard de la indústria per interconnectar blocs hardware. Dins d'aquesta família, en aquest projecte intervenen dues variants amb papers ben diferents.

La primera és AXI4-Stream [11], un protocol orientat a flux de dades. A diferència de les variants amb adreces, AXI4-Stream no defineix posicions de memòria: les dades hi circulen com un corrent ordenat d'un bloc cap al següent, de manera semblant a una canonada. La transferència es regula mitjançant un handshake de dos senyals:

- TVALID, que indica que l'emissor té una dada preparada
- TREADY, que indica que el receptor la pot acceptar

de manera que la dada (TDATA) només avança quan tots dos estan actius simultàniament. Un senyal addicional:

- TLAST, marca l'últim element d'un bloc de dades

Això permet als blocs consumidors saber quan han de finalitzar, tal com mostra la Figura 3.5. Aquesta naturalesa de flux fa que AXI4-Stream sigui especialment adequat per connectar kernels que formen un *pipeline* (adena d'etapes on la sortida d'una alimenta l'entrada de la següent), com és el cas de l'estructura reader-DUT-writer d'aquest treball.

Un cop descrita la comunicació interna entre kernels, cal considerar també com les dades arriben inicialment a la FPGA i com els resultats retornen cap al programa de control. Aquesta

segona via de comunicació correspon al moviment de blocs de dades entre la memòria del *host* i la memòria del dispositiu. En el cas de les targetes Alveo, aquesta connexió amb el servidor es realitza a través del bus PCIe [8], i les transferències es duen a terme mitjançant mecanismes DMA (*Direct Memory Access*), que permeten moure grans blocs de dades entre la memòria del sistema i la memòria HBM de la placa amb poca intervenció de la CPU. Sobre aquest canal, el runtime XRT [9] ofereix al *host* les operacions per reservar buffers a la memòria del dispositiu, transferir-hi les dades d'entrada i recuperar-ne els resultats.

D'aquesta manera, els dos nivells es complementen:

- el DMA sobre PCIe s'encarrega del transport de dades de gran volum entre host i dispositiu
- AXI4-Stream governa la circulació de dades *dins* de la FPGA, entre els kernels

La manera concreta com s'apliquen aquestes interfícies a la plataforma —nombre de canals, amplades i control de flux— es detalla al Capítol 4 (Seccions 4.3.1 i 4.3.2) i al Capítol 5.

3.5 Verificació funcional del sistema

3.5.1 *Testbench* i comparació de resultats

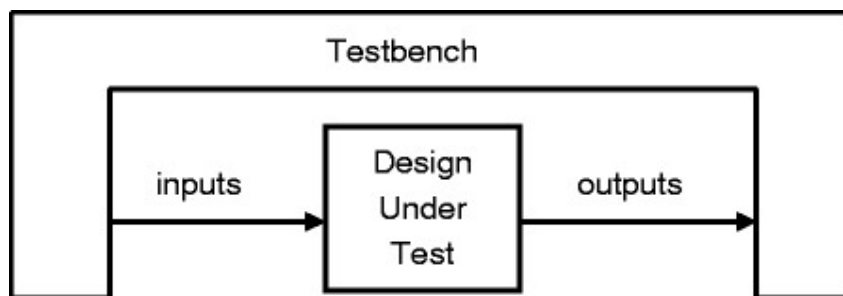
En un procés de verificació funcional, el *testbench* és l'element encarregat de generar els estímuls d'entrada del DUT, aplicar-los al circuit i recollir-ne les sortides. A partir d'aquestes sortides es pot comprovar si el comportament obtingut coincideix amb el resultat esperat.

En aquest treball, el *testbench* es manté dins de l'entorn *py4hw* i actua com a referència funcional del sistema. La verificació es basa a executar el mateix DUT en dues condicions diferents: d'una banda, mitjançant la simulació software de *py4hw*, i de l'altra, mitjançant l'execució del circuit sobre la FPGA. Les sortides obtingudes en tots dos casos es comparen bit a bit.

Aquesta comparació permet validar que el flux complet —generació de Verilog, empaquetament com a kernel, integració amb Vitis/XRT i execució sobre la FPGA— preserva el comportament funcional del DUT descrit originalment en Python.

3.5.2 Automatització del flux de generació

Un dels objectius principals d'un sistema de verificació híbrid és reduir la intervenció manual necessària per executar un DUT sobre la FPGA. En un flux manual, cada nou circuit pot requerir adaptar ports, escriure *wrappers*, generar fitxers de connectivitat i preparar scripts de compilació específics. Aquest procés és costós i pot introduir errors d'integració.



Font: [15].

Figura 3.6: Esquema simplificat d'un flux de verificació basat en simulació.

En aquest projecte, l'automatització es planteja a partir de la informació disponible dins del model *py4hw*. A partir del DUT, es poden obtenir automàticament els ports d'entrada i sortida, els noms dels senyals i les seves amplades en bits. Aquesta informació permet generar els fitxers auxiliars necessaris per adaptar el DUT al flux Vitis/XRT.

Així, l'automatització no només simplifica el procés de construcció del sistema, sinó que també fa que la plataforma sigui reutilitzable amb diferents circuits. Aquesta idea es desenvolupa en detall al Capítol 5.3.1, on es descriu la implementació pràctica del flux generat.

Capítol 4

Disseny de l'Arquitectura del Sistema

4.1 Descripció general del sistema proposat

El sistema proposat és una plataforma de verificació híbrida en què el control de la simulació es manté en software, dins de l'entorn *py4hw*, mentre que el circuit sota test (DUT) s'executa físicament sobre la FPGA. D'aquesta manera, el *testbench* continua generant els estímuls, controlant l'execució i comparant els resultats des de Python, però el càlcul del DUT es delega al hardware.

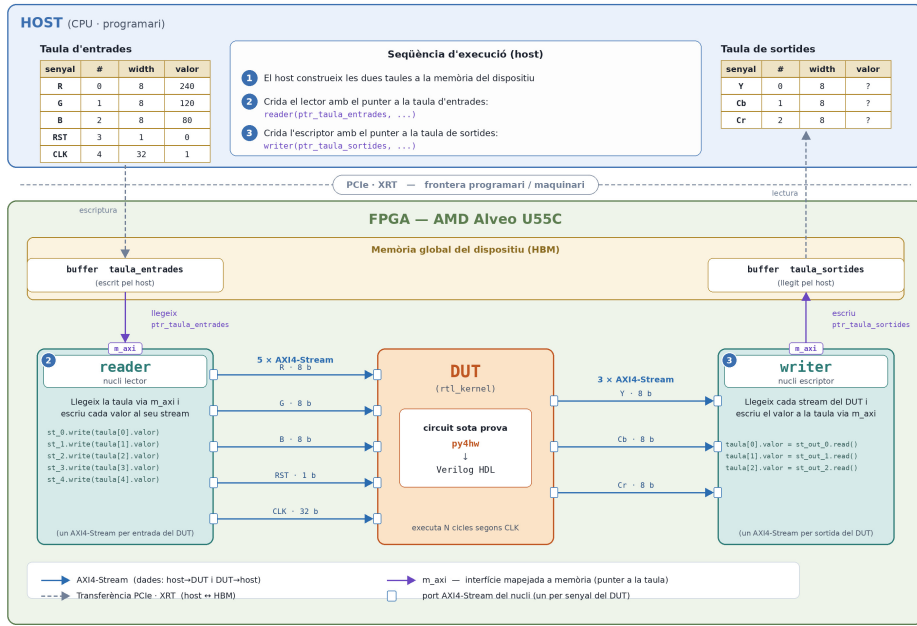
L'objectiu de la plataforma no és substituir la simulació software, sinó proporcionar un flux que connecti, de manera el més automàtica i reutilitzable possible, un DUT descrit en *py4hw* amb la seva execució real sobre FPGA.

A nivell d'arquitectura, el DUT no s'executa aïlladament, sinó envoltat per una estructura de tres blocs connectats per canals AXI4-Stream:

- un kernel de lectura (*reader*), que injecta les dades d'entrada provinents del host
- el DUT *wrapper*, que executa el circuit
- un kernel d'escriptura (*writer*), que recull les sortides i les retorna a la memòria del dispositiu

El host, mitjançant PCIe i el runtime XRT, orquestra tot el procés:

- carrega el binari a la FPGA
- transfereix els estímuls



Font: Elaboració pròpia.

Figura 4.1: Esquema de la plataforma de verificació HIL.

- llança l'execució
- recupera els resultats per verificar-los

Aquesta organització reader–DUT–writer es mostra de manera esquemàtica a la Figura 4.1 i se'n detalla la implementació al Capítol 5.

Convé precisar en quin sentit aquesta plataforma és *híbrida*, ja que el plantejament inicial del projecte parlava de *particionar* el disseny entre software i hardware. En aquest sistema, la divisió no es fa tallant el circuit per dins, sinó tractant el DUT com una unitat indivisible: l'entorn de verificació i el control resten en software (*py4hw*), mentre que el circuit sota test, sencer, s'executa sobre la FPGA. És a dir, el DUT es trasllada complet al hardware, sense separar-lo en una part que s'executa en software i una altra en hardware.

Amb aquesta precisió, el sistema desenvolupat respon a l'objectiu de delegar l'execució de subcircuits específics a la FPGA mantenint el control de la verificació en Python, que és el nucli de la proposta.

4.2 Arquitectura software

La part software de la plataforma és la responsable d'integrar-se amb *py4hw* i de permetre l'execució a la FPGA de manera transparent des de Python. Aquesta secció en descriu l'organització

en capes, el disseny de l'API de comunicació i l'abast de la delegació al hardware, és a dir, quina part del disseny es delega al hardware.

4.2.1 Interfície entre *py4hw* i el controlador de maquinari

La part software de la plataforma fa de pont entre el model de *py4hw* i l'execució sobre la FPGA. El seu disseny es recolza en una decisió clau: no reimplementar l'accés de baix nivell al dispositiu, sinó utilitzar directament el runtime XRT [10], que ja proporciona des de Python les operacions necessàries per comunicar-se amb la targeta (carregar el binari, reservar memòria al dispositiu, transferir dades i executar els kernels).

Tota la gestió del transport físic —PCIe i DMA— queda, doncs, a càrrec de XRT i de la plataforma de la targeta, i no del codi del projecte.

Sobre aquesta base, la part software del projecte aporta dues coses que XRT no ofereix per si sol:

- La primera és la generació automàtica, a partir d'un DUT descrit en *py4hw*, de tots els fitxers auxiliars que el flux de la FPGA necessita; aquesta part es descriu a la Secció 5.3.1.
- La segona és la integració de l'execució sobre la FPGA dins del bucle de simulació de *py4hw*, de manera que el circuit es pugui verificar des de l'entorn Python sense que l'usuari hagi de gestionar manualment cap detall de la comunicació amb el dispositiu.

És aquesta segona part la que fa que el sistema sigui realment transparent per a l'usuari: des de *py4hw*, el DUT s'avalua com qualsevol altre bloc, i tota la interacció amb la FPGA —càrrega del binari, transferència de dades i execució— queda amagada darrere d'aquesta capa d'integració.

4.2.2 Disseny de l'API de comunicació

L'API de comunicació és la concreció de la capa intermèdia descrita a la Secció 4.2.1: el conjunt d'operacions que el host ofereix per controlar l'execució sobre la FPGA. Aquestes operacions es poden agrupar en quatre categories:

- la *configuració i càrrega* (carregar el binari `.xclbin` i seleccionar el dispositiu)
- la *gestió de memòria* (reservar i alliberar els buffers de la targeta)
- la *transferència de dades* (moviment `host → dispositiu` i `dispositiu → host`)
- el *control d'execució* (llançar els kernels i esperar-ne la finalització [10])

El disseny de l'API segueix dos principis:

- El primer és la *transparència*, l'usuari treballa amb una interfície Python homogènia i no ha de gestionar manualment la comunicació de baix nivell amb el dispositiu.
- El segon és la *reutilitzabilitat*, com que els paràmetres concrets (nombre i amplada de canals, mida dels buffers) es deriven automàticament de la introspecció del DUT, la mateixa API serveix per a qualsevol circuit sense haver de reescriure-la.

La implementació concreta d'aquestes operacions sobre XRT, així com el programa host que les utilitza, es detalla a la Secció 5.3.2.

4.3 Arquitectura hardware

4.3.1 Jerarquia de mòduls dins la FPGA

Dins de la FPGA, el circuit sota test no s'executa de manera aïllada, sinó envoltat per una estructura de tres blocs que el connecten amb la resta del sistema. Aquesta estructura segueix un esquema de productor–processador–consumidor:

- un kernel de lectura (*reader*) que injecta les dades d'entrada
- el DUT *wrapper* que executa el circuit
- un kernel d'escriptura (*writer*) que recull les sortides

Tots tres es comuniquen mitjançant canals AXI4-Stream [11], de manera que les dades hi circulen com un flux ordenat d'un bloc al següent.

El kernel de lectura és l'únic bloc del costat d'entrada que accedeix a la memòria global de la targeta. Llegeix d'aquesta memòria les dades d'entrada preparades pel host i les converteix en fluxos AXI4-Stream, generant un canal independent per a cada port d'entrada del DUT. D'aquesta manera, la complexitat de l'accés a memòria queda confinada en aquest bloc, i el DUT en queda completament aïllat: només rep i emet fluxos de dades, sense saber d'on vénen ni on van.

El DUT *wrapper* ocupa la posició central. Rep els fluxos d'entrada provinents del reader, els presenta al circuit sota test, en gestiona l'execució i recull els resultats, que torna a emetre com a fluxos AXI4-Stream cap al bloc següent. La seva estructura interna —com adapta els ports del DUT al protocol i com en controla l'execució— es detalla a la Secció 4.3.3.

El kernel d'escriptura tanca la cadena fent el camí invers al de lectura: consumeix els fluxos de sortida del DUT i escriu els resultats de nou a la memòria global de la targeta, des d'on el host els pot recuperar. Igual que el reader, és l'únic extrem del seu costat que toca la memòria, de manera que el DUT mai no hi accedeix directament.

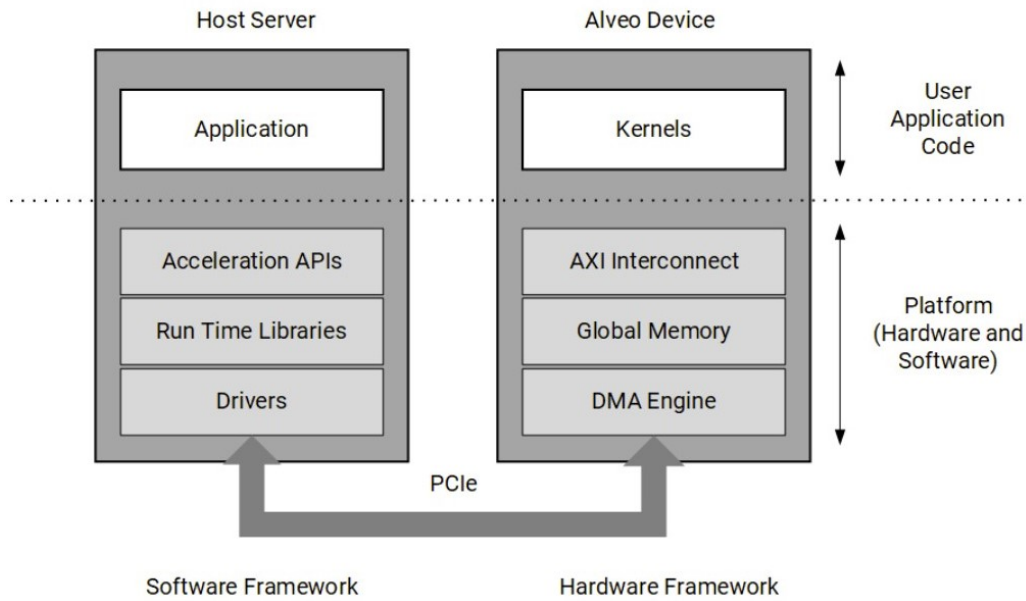
La connexió entre blocs es fa a raó d'un canal AXI4-Stream per cada port del DUT: tants canals d'entrada com ports d'entrada té el circuit, i tants canals de sortida com ports de sortida. El nombre total de canals és, per tant, la suma d'entrades i sortides del DUT, sense cap canal addicional. Aquesta correspondència directa port–canal és el que permet que la mateixa estructura s'adapti automàticament a qualsevol DUT, sigui quina sigui la seva interfície, sense haver de redissenyar la jerarquia per a cada circuit.

4.3.2 Interfície PCIe i control de flux

La targeta Alveo U55C és una acceleradora connectada al servidor amfitrió mitjançant el bus PCI Express (PCIe) [8] que és el canal físic pel qual circulen totes les dades i ordres entre el host i la FPGA. Tota la comunicació host–FPGA descansa, en última instància, sobre aquest enllaç.

La infraestructura d'aquesta comunicació no l'ha de construir el disseny, sinó que la proporciona la plataforma de la targeta. Aquesta plataforma defineix una regió estàtica (*shell*) que implementa el controlador PCIe, el motor de DMA i l'accés a la memòria HBM (High Bandwidth Memory) de la placa; els kernels del projecte s'allotgen en la regió dinàmica que es construeix sobre aquesta shell. El disseny, doncs, només ha d'utilitzar aquesta infraestructura, no reimplementar-la [8].

La Figura 4.2 mostra aquesta separació entre el codi d'aplicació que s'executa al host, la infraestructura software proporcionada pel runtime i els recursos hardware disponibles a la targeta Alveo.



Font: [16].

Figura 4.2: Esquema general de la comunicació entre el servidor host i una targeta acceleradora Alveo a través de PCIe. La plataforma proporciona el runtime, els drivers, el DMA, la memòria global i la infraestructura de comunicació amb els kernels.

El moviment de dades entre la memòria del host i la memòria HBM del dispositiu es fa mitjançant DMA, que transfereix blocs de dades amb poca intervenció de la CPU. Les dades d'entrada que el host prepara s'allotgen en buffers a la HBM; els kernels hi llegeixen i hi escriuen, i els resultats es recuperen pel mateix camí cap a la memòria del host.

Sobre aquest canal físic, el control de l'intercanvi el porta el host a través del runtime XRT (Xilinx Runtime) [10]: carrega el binari a la FPGA, transfereix les dades d'entrada a la memòria del dispositiu, llança l'execució dels kernels i, un cop han finalitzat, en recupera els resultats. El mecanisme concret de crida i sincronització entre el *testbench* i la FPGA es detalla a la Secció 4.4.1, i la implementació del programa host, a la Secció 5.3.2.

4.3.3 DUT *wrapper*: integració RTL com a kernel

En l'arquitectura proposada, els diferents blocs hardware s'organitzen com una cadena de processament, o *pipeline*, formada pels kernels de lectura, el kernel RTL que conté el DUT i el kernel d'escriptura. Les dades avancen d'un bloc al següent mitjançant canals AXI4-Stream, seguint el flux reader–DUT–writer. Tot i això, el DUT descrit en *py4hw* i exportat a Verilog no es pot connectar directament a aquest *pipeline*, ja que els seus ports són ports ordinaris —entrades i sortides registrades, governades per un rellotge— i no coneixen el protocol AXI4-Stream que

regeix la comunicació entre kernels. Cal, per tant, un bloc intermedi que faci de pont: *wrapper*, que adapta els ports del DUT al protocol de flux i en gestiona l'execució.

Pel costat de l'entrada, el *wrapper* rep cada canal AXI4-Stream provinent del kernel de lectura i en captura el valor en un registre, que alimenta el port d'entrada corresponent del DUT. Així es desacobla l'arribada de les dades, regida pel protocol de flux, del funcionament intern del circuit sota test. Des del punt de vista del DUT, les entrades són simplement valors estables disponibles als seus ports, sense que aquest hagi de gestionar cap senyalització AXI.

Com que el DUT és un circuit síncron, necessita un rellotge per avançar el seu càlcul intern. El *wrapper* genera el tren de polsos de rellotge necessari perquè el DUT progressi a través de la seva lògica fins que el resultat és a punt. Aquest és un dels motius pels quals el *wrapper* no és una simple connexió de fils: ha de governar activament l'execució del circuit i coordinar-la amb el flux de dades del *pipeline*.

Pel costat de la sortida, un cop el càlcul ha finalitzat, el *wrapper* recull els valors produïts pel DUT i els emet a través dels canals AXI4-Stream de sortida cap al kernel d'escriptura. Les sortides només es donen per vàlides quan el càlcul s'ha completat realment —és a dir, quan totes les entrades s'han carregat i el DUT ha produït resultats estables—, de manera que el bloc d'escriptura mai no llegeix valors prematurs.

El conjunt —el DUT més la lògica d'adaptació— es presenta al flux de Vitis com un únic kernel RTL. D'aquesta manera, el linker el pot connectar amb els kernels de lectura i d'escriptura com qualsevol altre bloc del *pipeline* hardware. És important remarcar que el *wrapper* només adapta la interfície del DUT: no n'altera el comportament, i el circuit calcula exactament el mateix que en simulació. Els detalls del procés d'empaquetament d'aquest kernel es descriuen a la Secció 5.4.3.

4.4 Protocol de comunicació i format de dades

4.4.1 Mecanisme de crida i sincronització *testbench* - FPGA

La plataforma híbrida funciona deixant que el *testbench* delegui a la FPGA cada avaluació del DUT. Conceptualment, allà on una simulació purament software calcularia internament les sortides del circuit, aquí el *testbench* fa una crida a la FPGA: li lliura els valors d'entrada actuals, la FPGA executa el càlcul i en retorna les sortides.

Aquesta crida és síncrona. El *testbench* no continua fins que la FPGA ha acabat i els resultats són disponibles: el host llança l'execució i espera que els kernels en senyalitzin la finalització abans de llegir les sortides de la memòria del dispositiu. D'aquesta manera es garanteix que els valors amb què continua la simulació són els que ha calculat realment el hardware, i no valors

parcials o previs.

El mecanisme està pensat perquè sigui transparent per al *testbench*: des del punt de vista de *py4hw*, el DUT s'avalua com de costum, i la redirecció cap a la FPGA passa darrere d'un component que ocupa el lloc del DUT dins del graf de simulació. La realització concreta d'aquest component es descriu a la Secció 5.3.3.

4.4.2 Format de dades

El host i la FPGA s'intercanvien les dades a través de dues estructures amb un format uniforme: una taula d'entrades i una taula de sortides. Totes dues tenen la mateixa estructura, fet que les fa independents del DUT concret que s'estigui verificant.

Cada entrada d'una taula correspon a un port del DUT i conté tres camps: un índex que identifica de quin port es tracta, l'amplada del port en bits, i el valor. La taula d'entrades l'omple el host amb els valors dels estímuls, una entrada per cada port d'entrada; la taula de sortides es reserva buida i és la FPGA qui l'omple amb els resultats, una entrada per cada port de sortida.

Taula d'entrades (l'omple el host)

Índex	Amplada	Valor
0	32	a
1	32	b

Taula de sortides (l'omple la FPGA)

Índex	Amplada	Valor
0	32	(buit)
1	32	(buit)

Figura 4.3: Format uniforme de les taules d'intercanvi host-FPGA: cada fila correspon a un port del DUT (índex, amplada en bits i valor). La taula d'entrades la prepara el host amb els estímuls; la de sortides es reserva buida i l'omple el hardware amb els resultats. Font: elaboració pròpia.

Aquesta representació uniforme —índex, amplada i valor— és el que permet que la mateixa infraestructura serveixi per a qualsevol DUT: cada port, sigui quin sigui el seu significat, es converteix en una entrada de la taula. El host no necessita saber què calcula el circuit, només quants ports té i quina amplada tenen, informació que s'obté automàticament de la introspecció del DUT.

Convé remarcar que el camp d'amplada viatja com a metadada al costat del valor, però no altera l'amplada física del transport: pels canals AXI4-Stream, cada valor es transmet com una paraula de mida fixa (vegeu el *handshake* a la Figura 3.5), independentment de l'amplada lògica del port [11]. L'amplada indica al hardware quants d'aquests bits són significatius, però el canal en si sempre té la mateixa mida.

Capítol 5

Implementació pràctica i desenvolupament

5.1 Entorn de desenvolupament

El desenvolupament del projecte combina dos àmbits diferenciats:

- l'entorn de descripció i simulació de circuits en Python
- la cadena d'eines d'AMD/Xilinx necessària per dur un disseny fins a la seva execució sobre la FPGA

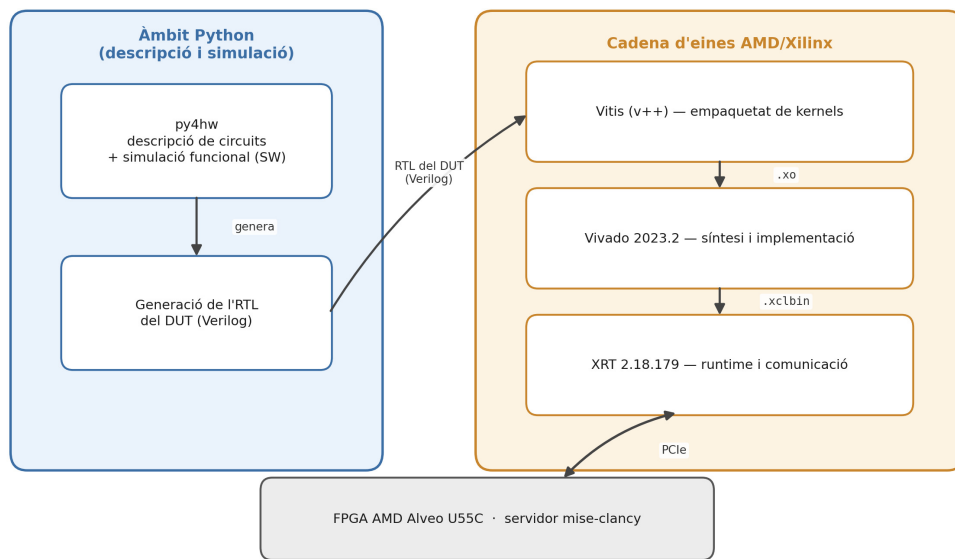
Aquesta secció descriu les eines software emprades i el seu paper dins del flux, així com la configuració del servidor i de la targeta sobre els quals s'ha treballat.

La figura [5.1](#) resumeix aquest flux i la relació entre els dos àmbits, des de la descripció en py4hw fins a l'execució sobre la FPGA.

5.1.1 Eines software: Vivado, Vitis, XRT, Python

En aquest projecte s'han utilitzat diverses eines software per cobrir les diferents fases del flux de desenvolupament.

- Python i py4hw s'han emprat per descriure i simular els circuits digitals, així com per automatitzar la generació dels fitxers necessaris.
- Vivado ML Edition 2023.2 s'ha utilitzat per a la generació i l'empaquetament dels kernels RTL, mentre que Vitis ha permès compilar i enllaçar els kernels fins a obtenir el fitxer final `.xclbin`.



Font: Elaboració pròpia.

Figura 5.1: Visió general de l'entorn de desenvolupament: flux des de la descripció i simulació en py4hw fins a l'execució sobre la FPGA Alveo U55C, amb les eines AMD/Xilinx (Vitis, Vivado i XRT) emprades en cada etapa.

- Finalment, XRT s'ha utilitzat com a runtime per carregar el binari a la FPGA, gestionar buffers de memòria i controlar l'execució des del host.

La Figura 5.2 mostra la versió de Vivado emprada durant el desenvolupament del projecte.

Flux de treball Vitis HLS (Flow Navigator)

Aquest flux també és útil en el cas d'integrar blocs RTL externs o *black boxes*, ja que la cosimulació permet comprovar que aquests blocs s'interconnecten correctament amb el codi generat per HLS.

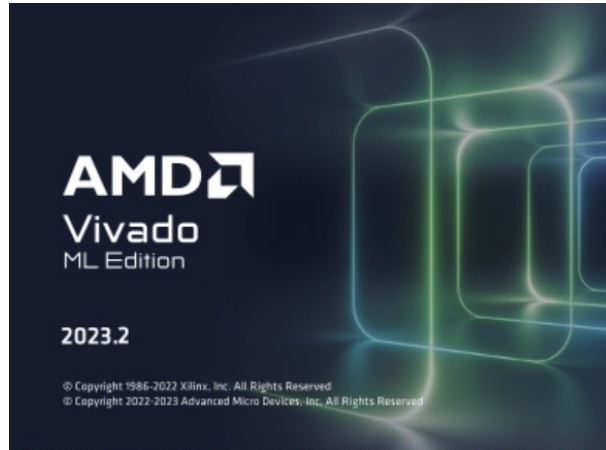
Els ítems que hi ha al Flow Navigator de Vitis HLS és el flux típic per passar de C/C++ a RTL i validar-ho abans d'empaquetar-ho i implementar-ho amb Vivado. Aquest flux s'organitza en 5 etapes.

La Figura 5.3 resumeix visualment aquest flux de treball, des de la simulació inicial del codi C/C++ fins a la implementació final del disseny hardware.

Primera etapa:

C Simulation (CSIM)

- Què fa: compila i executa el codi C/C++ com si fos un programa normal (CPU), amb el



Font: AMD Vivado.

Figura 5.2: Pantalla inicial de Vivado ML Edition 2023.2 utilitzat durant el desenvolupament del projecte.

testbench (o el *main()* de prova).

- Per a què serveix: validar funcionalitat ràpidament (resultats correctes, vectors de test, fitxers d'entrada/sortida, etc.).

Important: no genera RTL, no té cicles/clock real de hardware; és pur software.

Segona etapa:

C Synthesis (CSYNTH)

- Què fa: aplica HLS: converteix el C/C++ del top function en RTL (Verilog/VHDL) + interfícies (AXI, streams...) segons pragmes.
- Per a què serveix: veure si el disseny és sintetitzable i obtenir reports:
 - latència estimada (cicles)
 - II (initiation interval) de *pipelines*
 - ús de recursos (LUT/FF/BRAM/DSP) estimat
 - estructura d'interfícies

Important: aquí és on es “materialitza” de veritat el que fan els `#pragma HLS`.

Tercera etapa:

C/RTL Co-simulation

- Què fa: simula el RTL generat (i/o el RTL blackbox) amb un simulador RTL, fent que el *testbench* “parli” amb el disseny RTL.
- Per a què serveix:
 - comprovar que el RTL fa exactament el mateix que el C (comparació C vs RTL)
 - detectar errors d’interfície, protocols AXI/streaming, resets, latències, etc.
 - és especialment útil quan hi ha RTL blackbox, perquè aquí es valida la integració real

Important: és molt més lent que *CSIM*, però és molt més realista

Quarta etapa:

Package

- Què fa: “empaqueta” el resultat perquè es pugui reutilitzar fora de Vitis HLS. Què genera típicament (segons target):
 - un IP (per Vivado IP Integrator) o
 - un kernel object per Vitis (p. ex. .xo) amb metadades d’interfície
- Per a què serveix: és el pas que deixa el component llest per integració en un disseny més gran / en un projecte Vitis (link per fer l’xclbin).

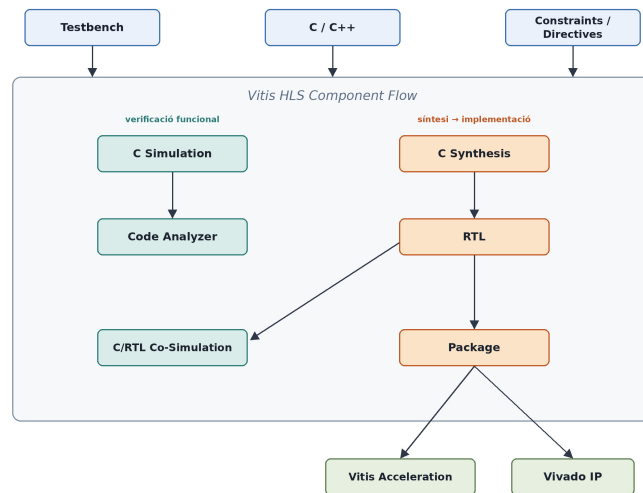
Cinquena etapa:

Implementation

- Què fa: llança una implementació amb Vivado sobre el RTL (synthesis + place&route) amb un dispositiu i clocks reals.
- Per a què serveix:
 - obtenir números més fiables de recursos,
 - comprovar si tanca timing a la freqüència objectiu,
 - tenir reports post-implementació (timing, utilització, etc.).

Important: tot i que aquesta etapa requereix més temps de compilació, és necessària per comprovar que el disseny és implementable sobre la FPGA real i que els seus camins interns permeten treballar a la freqüència de rellotge definida com a objectiu.

El flux de Vitis permet compilar una mateixa aplicació per a tres objectius (*targets*) diferents, segons el grau de fidelitat al hardware real que es vulgui:



Font: Elaboració pròpia.

Figura 5.3: Flux de compilació i verificació en Vitis HLS

- En el mode d'emulació software (**sw_emu**), tant el host com els kernels s'executen sobre la CPU; és el mode més ràpid de compilar i executar, i serveix per validar la funcionalitat i la lògica del programa host sense necessitat de hardware.
- En el mode d'emulació hardware (**hw_emu**), els kernels es compilen a un model RTL que s'executa sobre un simulador, mentre que el host s'executa de manera nativa; és més lent que l'anterior, però permet comprovar de manera realista la integració, els protocols de les interfícies i la comunicació entre el host i els kernels abans de passar a la implementació completa.
- En el mode hardware (**hw**) es duu a terme la síntesi i la implementació completes per generar el binari que s'executa sobre la FPGA física; és el mode més costós en temps de compilació, ja que inclou tot el procés de síntesi i *place&route*, però és l'únic que proporciona execució real sobre la placa.

5.1.2 Configuració del servidor i de la targeta FPGA

Tot el treball relacionat amb la construcció i l'execució dels binaris (Vivado, Vitis i XRT) s'ha dut a terme sobre un servidor Linux compartit, *mise-clancy*, al qual s'accedeix de manera remota mitjançant una sessió de terminal. Aquest servidor disposa de la targeta acceleradora AMD/Xilinx Alveo U55C connectada per PCIe. L'accés és multiusuari, de manera que la cadena d'eines i les variables d'entorn es carreguen a cada sessió abans d'iniciar qualsevol tasca de compilació o execució.

Abans de compilar o executar qualsevol projecte Vitis, cal inicialitzar l'entorn de treball del servidor. En aquest projecte, això es va fer carregant els mòduls corresponents de Vitis i XRT, o bé inicialitzant directament els scripts d'entorn proporcionats per Xilinx:

```
module load xilinx/xrt
module load xilinx/vitis_2023.2

source /opt/xilinx/xrt/setup.sh
source /opt/xilinx/Vitis/2023.2/settings64.sh
```

També es va comprovar la plataforma disponible mitjançant les eines de Vitis/XRT, per tal d'assegurar que el projecte es compilava per a la targeta Alveo U55C utilitzada en el servidor.

La targeta utilitzada és una AMD/Xilinx Alveo U55C, i la plataforma de desplegament corresponent és `xilinx_u55c_gen3x16_xdma_3_202210_1`. Aquesta plataforma defineix la *shell* estàtica de la targeta, que inclou el controlador PCIe, el motor DMA i l'accés a la memòria HBM, sobre la qual s'implementa la regió dinàmica que conté els kernels del projecte.

La informació de plataforma mostrada per Vitis permet comprovar que el projecte es compila per al dispositiu correcte i resumeix alguns dels recursos disponibles a la FPGA, com les freqüències de rellotge i els recursos interns principals. Aquesta informació es recull a la Taula 5.1.

La Figura 5.4 mostra diverses vistes externes de la targeta acceleradora utilitzada en el projecte.

La preparació de l'entorn consisteix a:

- inicialitzar la cadena de Vitis i el runtime XRT (*source* dels respectius scripts d'entorn)
- a seleccionar la plataforma objectiu

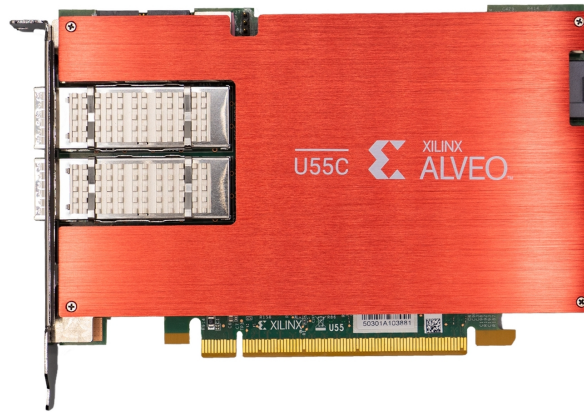
L'estat i la disponibilitat de la targeta es comproven amb les utilitats de XRT `xbutil examine`.

El desenvolupament i la prova interactiva del flux en Python es fan des de la terminal, mentre que els empaquetaments de Vivado s'executen en mode *batch*.

5.2 Validació inicial del flux Vitis/XRT

Abans de construir el flux automatitzat propi d'aquest treball, va caldre familiaritzar-se amb el cicle bàsic d'una aplicació accelerada:

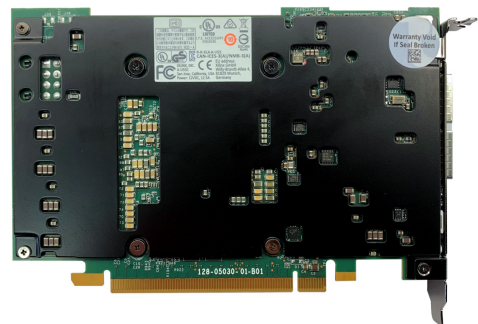
- com es compila un kernel



(a) Vista principal de la targeta AMD/Xilinx Alveo U55C.



(b) Vista del panell frontal i dels ports de comunicació.



(c) Vista posterior de la targeta.

Font: [17].

Figura 5.4: Vistes externes de la targeta acceleradora AMD/Xilinx Alveo U55C utilitzada en el projecte.

Taula 5.1: Informació principal de la plataforma Vitis seleccionada per a la targeta AMD/Xilinx Alveo U55C.

Paràmetre	Valor
Plataforma	xilinx_u55c_gen3x16_xdma_3_202210_1
Targeta	AMD/Xilinx Alveo U55C
Dispositiu	xcu55c-fsvh2892-2L-e
Família	virtexuplusHBM
Sistema host	Linux, x86_64
Relotge PL0	300 MHz
Relotge PL1	500 MHz
Relotge PL2	100 MHz
DSP	9024
FF	2607360
BRAM	2016
LUT	1303680

Font: elaboració pròpia a partir de la informació de plataforma mostrada per Vitis.

- com es genera el fitxer `.xclbin`
- com es carrega a la FPGA des del host
- com s'hi transfereixen dades i se'n recuperen els resultats

Amb aquest objectiu es van seguir els tutorials i exemples oficials de Vitis i XRT [10], que il·lustren el patró host–kernel sobre casos senzills. Un cop entès aquest patró, es van dur a terme dues proves preliminars pròpies, descrites a continuació, que es reproduïen sobre casos progressivament més propers al projecte.

En particular, es va treballar amb el tutorial *RTL Stream Kernel Integration* del repositori oficial de Vitis [18], ja que presenta una estructura propera a la utilitzada en aquest projecte: un programa host que controla l'execució, diversos kernels hardware i la comunicació entre blocs mitjançant canals AXI4-Stream. Aquest tutorial va servir com a punt de partida per comprendre com es construeix una aplicació accelerada amb kernels connectats en forma de *pipeline* i com es genera el fitxer final `.xclbin` a partir dels diferents components del sistema.

5.2.1 Prova preliminar amb inversió de píxels en una imatge PPM

Abans de desenvolupar el flux automatitzat proposat en aquest treball, es va realitzar una prova preliminar basada en una aplicació senzilla de processament d'imatge.

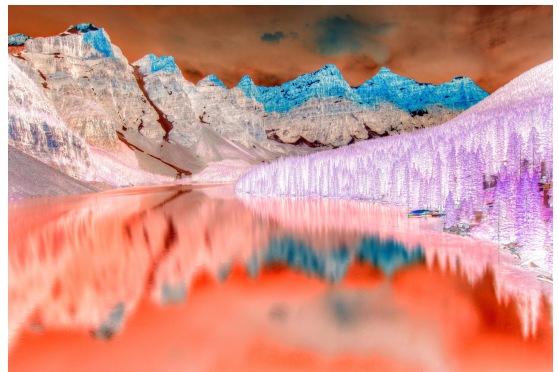
L'objectiu d'aquesta prova era validar el funcionament bàsic de l'entorn Vitis/XRT i comprovar que era possible transferir dades des del host cap a la FPGA, executar un kernel hardware i recuperar els resultats generats.

La prova consistia a llegir una imatge en format PPM, enviar els seus píxels a la FPGA i executar un kernel RTL encarregat d'invertir-ne els valors. Finalment, el resultat es retornava al host i s'escrivia una nova imatge PPM, fet que permetia comprovar visualment que el flux complet funcionava correctament.

La Figura 5.5 mostra el resultat visual d'aquesta prova preliminar: la imatge d'entrada en format PPM i la imatge obtinguda després d'executar el kernel d'inversió de píxels sobre la FPGA.



(a) Imatge d'entrada en format PPM, abans d'aplicar el kernel d'inversió de píxels.



(b) Imatge de sortida després d'aplicar el kernel d'inversió de píxels.

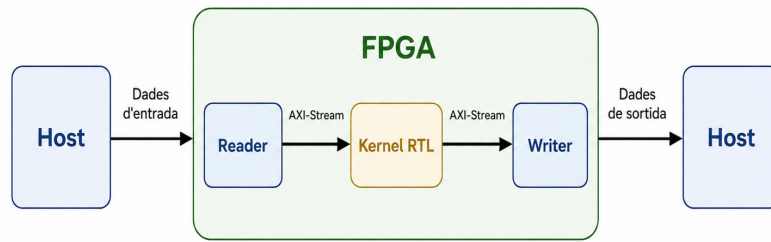
Font: Elaboració pròpia, resultat de la simulació.

Figura 5.5: Validació preliminar del flux host–FPGA–host mitjançant una aplicació senzilla de processament d'imatge.

La Figura 5.6 resumeix el flux validat en aquesta prova: el host prepara les dades de la imatge, el reader les introdueix al *pipeline* hardware, el kernel RTL en fa la inversió i el writer retorna els resultats al host.

Aquesta prova no formava part encara del flux final basat en *py4hw*, però va permetre validar els elements bàsics necessaris per al projecte:

- la generació del fitxer `xclbin`



Font: Elaboració pròpia.

Figura 5.6: Esquema simplificat del flux host – FPGA – host validat amb la prova preliminar d’inversió de píxels. El host prepara les dades d’entrada, la FPGA executa el *pipeline* reader – kernel RTL – writer i els resultats es retornen al host per generar la imatge de sortida.

- la càrrega del binari a la FPGA
- la transferència de dades mitjançant XRT
- la connexió entre kernels mitjançant AXI4-Stream

Cal destacar que aquestes proves preliminars es van validar primer en mode d’emulació [`sw_emu`, `hw_emu`] abans d’executar-les sobre la placa, fet que va permetre comprovar el flux sense ocupar la FPGA ni esperar el temps complet de síntesi.

5.2.2 Prova amb un kernel RTL de conversió RGB a YCbCr

Després de validar el flux amb una operació senzilla d’inversió de píxels, es va realitzar una segona prova amb un disseny RTL més representatiu: un conversor de color RGB a YCbCr. Aquesta prova tenia com a objectiu comprovar que el flux també podia integrar un kernel RTL propi, generar el fitxer *xclbin* corresponent i executar-lo correctament sobre la FPGA.

Després de validar el flux amb una operació senzilla d’inversió de píxels, es va realitzar una segona prova amb un disseny RTL més representatiu: un conversor de color RGB a YCbCr. En aquesta fase, l’objectiu no era encara analitzar en profunditat la conversió de color, sinó comprovar que el flux Vitis/XRT podia integrar un kernel RTL propi, generar-ne el fitxer *.xclbin* i executar-lo correctament sobre la FPGA.

El convertidor RGB a YCbCr s’utilitza posteriorment com un dels DUTs principals del projecte, i per aquest motiu la seva funcionalitat, les seves entrades i sortides i les equacions de conversió es descriuen amb més detall a la Secció 5.4.1.

A diferència de la prova anterior, aquest bloc ja representa un circuit de processament amb una funcionalitat més propera als DUTs utilitzats posteriorment en el projecte. Per aquest

motiu, aquesta prova va servir com a pas intermedi entre els exemples bàsics de Vitis i la integració del DUT dins del flux automatitzat desenvolupat.

El procés va consistir en preparar el RTL del conversor, empaquetar-lo com a kernel, generar el fitxer `.xo`, integrar-lo amb els kernels auxiliars de lectura i escriptura, generar el fitxer `.xclbin` i executar-lo des del host. La comprovació dels resultats va permetre verificar que el kernel RTL s'havia integrat correctament en el flux Vitis/XRT.

5.2.3 Valoracions de les proves preliminars

Les proves preliminars van permetre validar el funcionament bàsic del flux Vitis/XRT abans d'integrar-lo amb *py4hw*. La prova d'inversió de píxels va confirmar que era possible transferir dades des del host cap a la FPGA, executar un kernel hardware i recuperar correctament els resultats generats. Aquesta validació va permetre comprovar el cicle complet host – FPGA – host en un cas senzill i fàcilment observable.

La prova amb el kernel RTL de conversió RGB a YCbCr va representar un pas més proper al flux final del projecte, ja que va permetre integrar un bloc RTL propi dins de l'entorn Vitis/XRT. Aquesta prova va servir per verificar que el sistema podia treballar no només amb kernels HLS senzills, sinó també amb un disseny RTL empaquetat com a kernel.

En conjunt, aquestes proves van servir per familiaritzar-se amb l'entorn de desenvolupament, validar la comunicació amb la targeta FPGA i establir la base tècnica sobre la qual posteriorment es va construir el flux automatitzat a partir de DUTs descrits en *py4hw*.

5.3 Desenvolupament del programari

Aquesta secció descriu la part software desenvolupada en el projecte: l'extensió de *py4hw* que, a partir d'un DUT, genera automàticament els fitxers necessaris per al flux de Vitis, i el programa host que controla l'execució sobre la FPGA des del servidor.

La implementació software s'integra dins del repositori principal de *py4hw*. En concret, els dos fitxers més rellevants per al flux HIL amb Vitis són `vitiswrapping.py`, situat dins de `py4hw/emulation`, i `tb_HILWrapper.Vitis.py`, situat dins de `test/interactive`. El primer concentra la lògica d'automatització del flux: extracció de la informació del DUT, generació del Verilog, creació del *wrapper*, generació dels fitxers auxiliars i preparació dels scripts de construcció per a Vitis. El segon actua com a banc de proves o *testbench* principal, des d'on es defineix el DUT, es generen els estímuls, s'invoca el flux HIL i es comproven els resultats obtinguts.

5.3.1 Extensió de *py4hw* per a la co-simulació FPGA

El component central de la infraestructura és el mòdul `vitiswrapping.py`, que pren un DUT descrit en *py4hw* i en deriva, de manera automàtica, els artefactes necessaris per construir el sistema accelerat. El procés s'organitza en quatre passos.

- En primer lloc, es fa una *introspecció* del DUT: es recorre el model de *py4hw* per obtenir-ne els ports d'entrada i de sortida, el nom de cada senyal i la seva amplada en bits. Aquesta informació és la que permet adaptar la resta del flux a cada circuit concret sense haver d'escriure manualment les connexions.
- En segon lloc, es genera el *Verilog* del DUT a partir del model Python.
- En tercer lloc, es construeix l'*embolcall AXI4-Stream*: per a cada port d'entrada i de sortida del DUT es defineix un canal AXI4-Stream, de manera que el nombre total de canals és

$$\text{total_streams} = \text{num_ins} + \text{num_outs}.$$

- Finalment, es generen els fitxers que requereix el flux de Vitis (entre d'altres, el `kernel.xml`, els scripts TCL d'empaquetament i el `connectivity.cfg`), descrits a les seccions següents.

Durant la integració es van detectar diverses incompatibilitats entre l'API de *py4hw* i el que esperava el flux RTL de Xilinx, que es van haver de resoldre perquè la generació fos correcta. Es documenten aquí perquè formen part del desenvolupament real i poden ser útils per a futures extensions:

- El càlcul inicial del nombre de canals incloïa un +1 espuri associat a un senyal `Axi2Clk`; un cop identificat, es va eliminar, deixant la relació `total_streams = num_ins + num_outs`.
- En la generació del *wrapper*, l'operació d'extensió de bits es va substituir de `BitExtend` a `Buf`, que és la primitiva adequada de *py4hw* per a aquest cas.
- L'accés als senyals del model es va corregir per fer-se a través del diccionari intern `_wires` en lloc del mètode `getWire`.
- En l'script d'empaquetament (`generate_package_tcl`), les seccions corresponents a la interfície de control `s_axi_control` i al mapa de memòria AXI4-Lite es van comentar, ja que no apareixen al `rtl_kernel.v` generat per *py4hw*; igualment, el senyal de rellotge `ap_clk` es va corregir a `clk` per fer-lo coincidir amb la interfície real del DUT.

El codi desenvolupat per implementar aquesta funcionalitat s'ha integrat dins del repositori de *py4hw*, concretament al directori *py4hw/emulation*. En particular, la lògica principal del flux de generació i integració amb Vitis/XRT es troba al fitxer *vitiswrapping.py*.

5.3.2 API de control i programa host

El programa host és el responsable de controlar l'execució des de la CPU. Per fer-ho, utilitza les vinculacions Python de XRT, disponibles a través de *pyxrt*, que proporcionen l'API necessària per interactuar amb la targeta FPGA des del software. Aquesta API permet obrir el dispositiu, carregar el binari *.xclbin*, reservar buffers de memòria, configurar els arguments dels kernels, iniciar-ne l'execució i sincronitzar el final del processament.

El cicle de treball segueix l'esquema habitual d'una aplicació accelerada. En primer lloc, el host carrega el binari *.xclbin* a la FPGA i reserva els buffers de memòria de dispositiu necessaris per a les dades d'entrada i de sortida. A continuació, configura els arguments dels kernels, com ara els punters als buffers i el nombre d'elements a processar, en llança l'execució i espera que aquesta finalitzi.

Un cop el runtime indica que l'execució dels kernels ha acabat, el host llegeix el buffer de sortida des de la memòria del dispositiu i en realitza la verificació funcional, comparant els resultats obtinguts bit a bit amb els que produeix el model software de *py4hw*. D'aquesta manera, el host tanca el cicle host-FPGA-host i actua, alhora, com a controlador de l'execució i com a banc de proves de la plataforma.

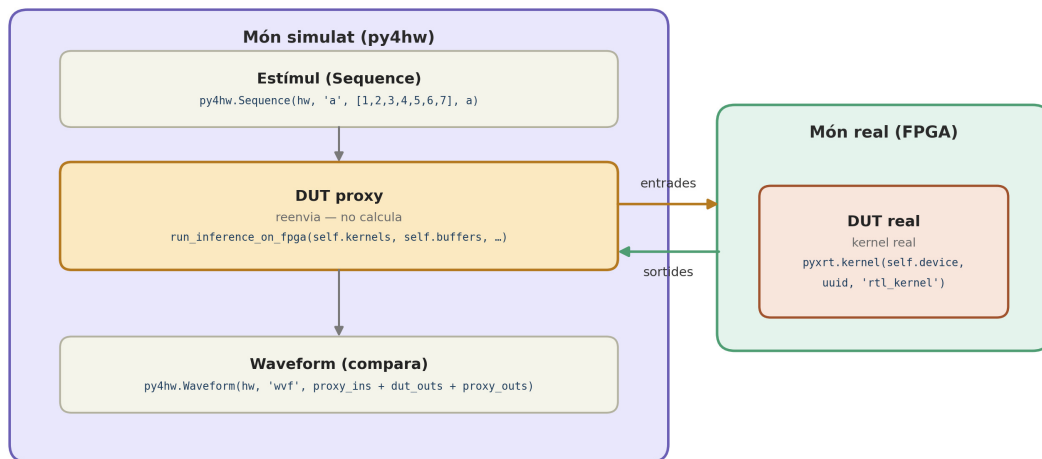
5.3.3 Redirector: integració de la FPGA dins del bucle de *py4hw*

Un cop es va muntar tot el sistema, es va incorporar un component addicional, el *redirector*, amb una finalitat diferent: no corregeix cap error, sinó que tanca l'estructura permetent que la FPGA participi directament dins del bucle de simulació de *py4hw*.

A diferència de la resta de la part hardware, el redirector és un component *software*. S'implementa com la classe *DUTProxyXRT*, que estén *py4hw.Logic* i, per tant, s'integra com un bloc més dins del graf de simulació. La clau del seu disseny és que exposa exactament els mateixos ports -mateix nombre i amplada d'entrades i sortides- que el DUT original, de manera que pot ocupar el lloc del DUT dins del graf de *py4hw* sense alterar-ne la resta.

El seu funcionament, a cada avaluació de la simulació, segueix tres passos.

- Primer, llegeix els valors presents als ports d'entrada del bloc dins de la simulació.
- A continuació, els *redirigeix* cap a la FPGA: construeix la taula d'entrades, les transfereix



Font: Elaboració pròpia.

Figura 5.7: Esquema conceptual del redirector HIL: el `DUTProxyXRT` substitueix el DUT dins la simulació `py4hw` i, en lloc de calcular, reenvia les entrades a la FPGA i en recull les sortides; el `Waveform` compara els senjals simulats amb els reals. S'indica una línia de codi representativa de cada bloc.

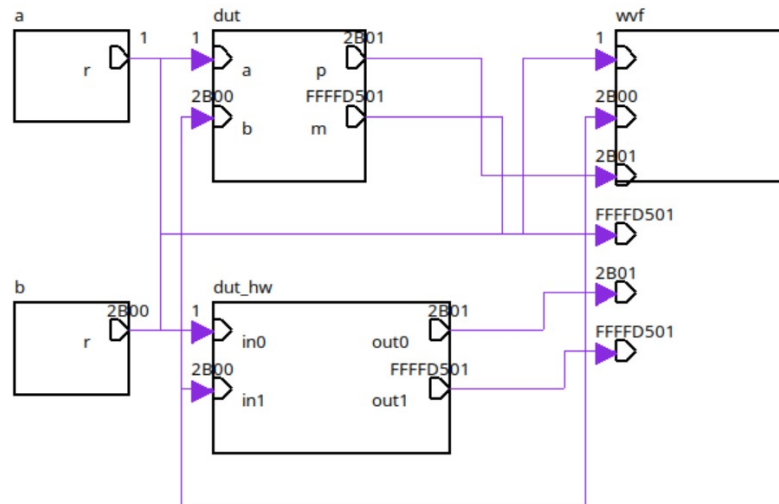
per XRT, executa el *pipeline* de kernels sobre el hardware i en recupera els resultats.

- Finalment, escriu aquests resultats —els que ha produït la FPGA, no un model software— als ports de sortida del bloc, de manera que la simulació continua amb els valors calculats pel hardware real.

La Figura 5.7 resumeix aquest mecanisme: el bloc `DUTProxyXRT` ocupa el lloc del DUT dins del graf de simulació, però redirigeix l'avaluació cap a la FPGA i retorna els resultats al mateix entorn `py4hw`.

Aquesta substitució es pot apreciar a la finestra de simulació de `py4hw`. Les Figures 5.8 i 5.9 en mostren el muntatge per als dos DUTs emprats: en tots dos casos, les mateixes seqüències d'estímul alimenten en paral·lel dos blocs que exposen ports idèntics —el DUT descrit en `py4hw`, que es continua simulant en software, i el `DUTProxyXRT`, que redirigeix l'avaluació cap a la FPGA—, i totes dues sortides conflueixen al bloc `Waveform`, on es comparen. D'aquesta manera, el redirector no altera l'estructura del *testbench*: ocupa el lloc d'un bloc més dins del graf, amb el mateix nombre i amplada de ports que el DUT original, i la diferència és únicament l'origen dels valors de sortida —calculats pel maquinari en el cas del proxy, i pel model software en el cas del DUT de referència.

Per evitar el cost de reinicialitzar XRT a cada cicle, la càrrega del `.xclbin`, la creació dels kernels i la reserva dels buffers es fan una sola vegada en construir l'objecte, i es reutilitzen en



Font: Elaboració pròpia, resultat de la simulació.

Figura 5.8: Comparació entre el DUT Add/Sub simulat en *py4hw* i el bloc DUTProxyXRT que redirigeix l'execució cap a la FPGA.

totes les avaluacions posteriors.

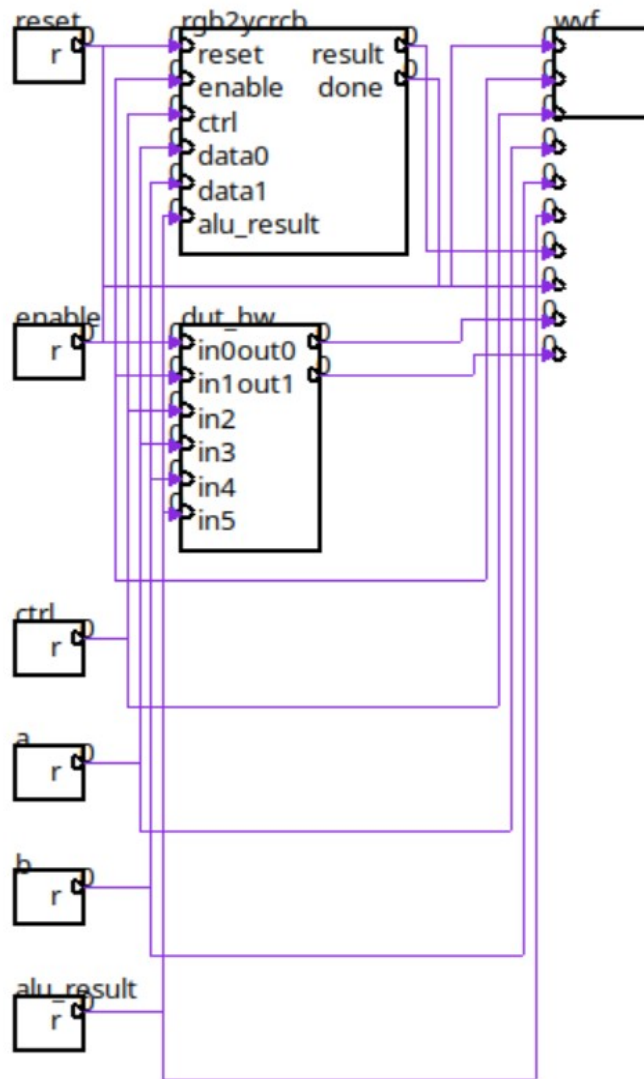
Així, el redirector converteix la plataforma en un veritable entorn *Hardware-in-the-Loop*: *py4hw* manté el control de la verificació mentre el càlcul del DUT s'executa físicament a la FPGA, de forma transparent per a la resta del *testbench*.

5.4 Desenvolupament del maquinari

La part hardware del sistema s'organitza com un *pipeline* de tres kernels connectats per canals AXI4-Stream:

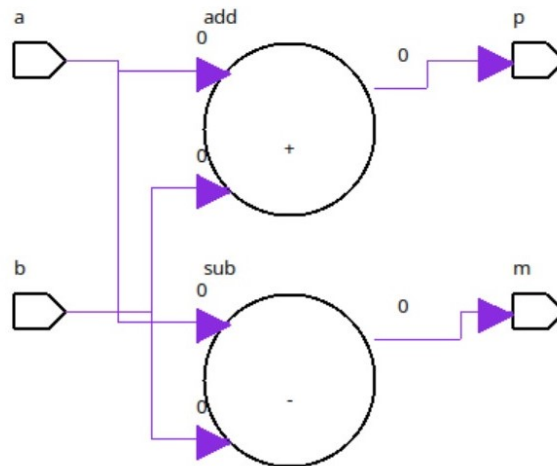
- un kernel de lectura que injecta les dades d'entrada
- el DUT *wrapper* que executa el circuit
- un kernel d'escriptura que recull les sortides

Aquesta secció descriu cadascun d'aquests blocs, la manera com s'empaqueta el DUT com a kernel, la seva integració mitjançant AXI4-Stream i, finalment, el redirector, el component software que permet que la FPGA participi directament dins del bucle de simulació de *py4hw*.



Font: Elaboració pròpia, resultat de la simulació.

Figura 5.9: Comparació entre el DUT RGB2YCbCr simulat en *py4hw* i el bloc DUTProxyXRT executat sobre la FPGA.



Font: Elaboració pròpia, resultat de la simulació.

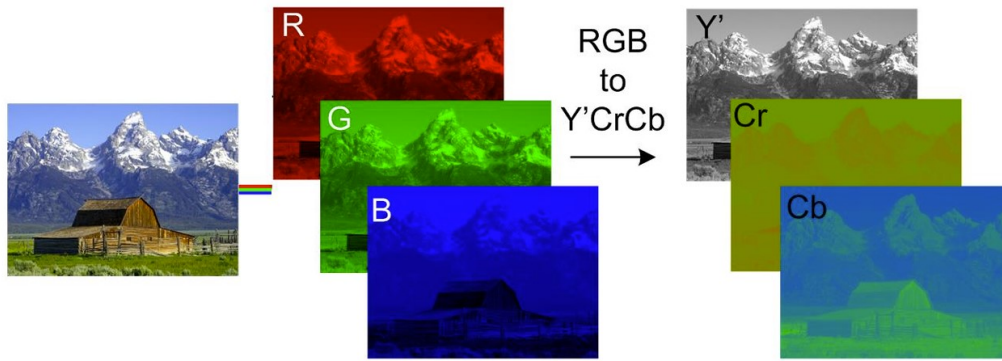
Figura 5.10: Esquema del DUT Add/Sub descrit en *py4hw*

5.4.1 Circuits sota test: Add/Sub i RGB2YCbCr

Per desenvolupar i validar la plataforma s'han emprat dos circuits sota test, descrits en *py4hw*, de complexitat creixent. Tots dos serveixen per comprovar el flux complet i, alhora, per posar a prova casos diferents: un de mínim i un altre amb múltiples sortides.

El primer és un circuit aritmètic senzill (Add/Sub) que rep dos operands i en retorna la suma i la diferència. Té 2 entrades i 2 sortides. Es verifica amb un conjunt de vectors dirigits escollits per cobrir casos representatius. El seu esquema es pot veure a la figura 5.10.

El segon és un convertidor d'espai de color de RGB a YCbCr. Pren els tres components R , G i B d'un píxel i en calcula la luminància i les dues crominàncies. A diferència de la prova preliminar de la Secció 5.2.1, on aquest bloc es va emprar només per validar la integració d'un kernel RTL dins del flux Vitis/XRT, en aquesta secció es descriu el circuit com a cas de prova del sistema complet desenvolupat. El convertidor opera sobre l'espai de color YCbCr, una representació alternativa a l'RGB molt emprada en codificació i compressió d'imatge. En lloc de descriure cada píxel pels seus tres components de color (vermell, verd i blau), el separa en una component de *luminància* (Y), que recull la informació d'intensitat -essencialment, la imatge en escala de grisos—, i dues components de *crominància* (Cb i Cr), que codifiquen la informació de color com a diferències respecte de la luminància: Cb la diferència cap al blau i Cr la diferència cap al vermell. La Figura 5.11 il·lustra aquesta descomposició sobre una imatge de mostra.



Font: [19].

Figura 5.11: Separació d'una imatge RGB en les components Y, Cr i Cb. La component Y conté principalment la informació de luminància, mentre que Cr i Cb representen la informació de crominància associada al vermell i al blau, respectivament.

El convertidor es basa en les equacions estàndard de conversió RGB→YCbCr de la recomanació BT.601 en rang complet (*full-range*), que en aritmètica de nombres reals s'expressen com:

$$Y = 0,299 R + 0,587 G + 0,114 B$$

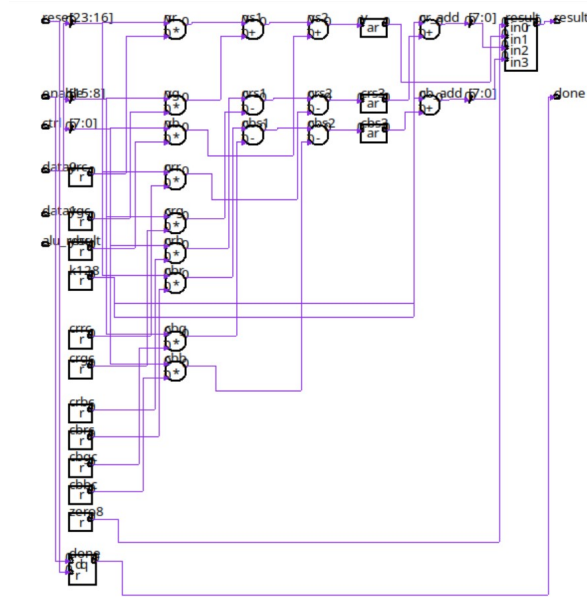
$$Cr = 0,500 R - 0,419 G - 0,081 B + 128$$

$$Cb = -0,169 R - 0,331 G + 0,500 B + 128$$

Aquestes equacions, però, defineixen la conversió ideal en coma flotant. La implementació hardware del DUT —i, per tant, també el model de *py4hw* que se'n simula— no opera amb nombres reals, sinó que n'aproxima els coeficients mitjançant aritmètica de punt fix: cada coeficient es representa com un enter escalat els productes es fan amb multiplicació entera i el resultat es reescala amb un desplaçament, de manera que la sortida és una aproximació en punt fix dels valors ideals.

Convé remarcar que aquesta aproximació no resta vàlida a la verificació bit a bit. El model de referència amb què es compara l'execució sobre la FPGA no són les equacions ideals en coma flotant, sinó la simulació en *py4hw* del mateix circuit en punt fix. Tots dos —simulació i hardware— executen exactament la mateixa aritmètica entera, de manera que la coincidència bit a bit confirma que la FPGA reproduïx fidelment el circuit descrit en *py4hw*, no pas que aquest implementi exactament la conversió real.

Conceptualment, el convertidor pren els tres components RGB i en produeix els tres components YCbCr. Ara bé, la interfície real del DUT descrit en *py4hw* no exposa sis ports de color separats, sinó que treballa amb dades empaquetades: el píxel d'entrada arriba en una única paraula (el port `data0`) i el resultat YCbCr surt empaquetat en una sola paraula (`result`), amb



Font: Elaboració pròpia, resultat de la simulació.

Figura 5.12: Esquema del DUT RGB2YCbCr descrit en *py4hw*

l'ordre de bytes Y|Cb|Cr. El circuit disposa, a més, dels senyals de control **reset**, **enable** i **ctrl**, que cal seqüenciar perquè el càlcul progressi. Aquests senyals de control, absents en el DUT senzill, fan del RGB2YCbCr un cas de sincronització més exigent. L'esquema del DUT RGB2YCbCr es pot observar a la Figura 5.12.

5.4.2 Kernels reader i writer

Els kernels de lectura (**reader_kernel**) i d'escriptura (**writer_kernel**) actuen com a pont entre la memòria global de la targeta i el flux AXI4-Stream que envolta el DUT. El reader llegeix les dades d'entrada des d'un buffer de memòria de dispositiu i les converteix en un flux AXI4-Stream, generant un element per a cada mostra a processar; el writer fa el camí invers, consumint el flux de sortida i escrivint cada resultat de nou a memòria. Tots dos kernels estan implementats en HLS en C/C++ i parametritzats segons el nombre i l'amplada dels canals del DUT, de manera que el mateix parell de kernels serveix per a qualsevol circuit sense reescriure'ls.

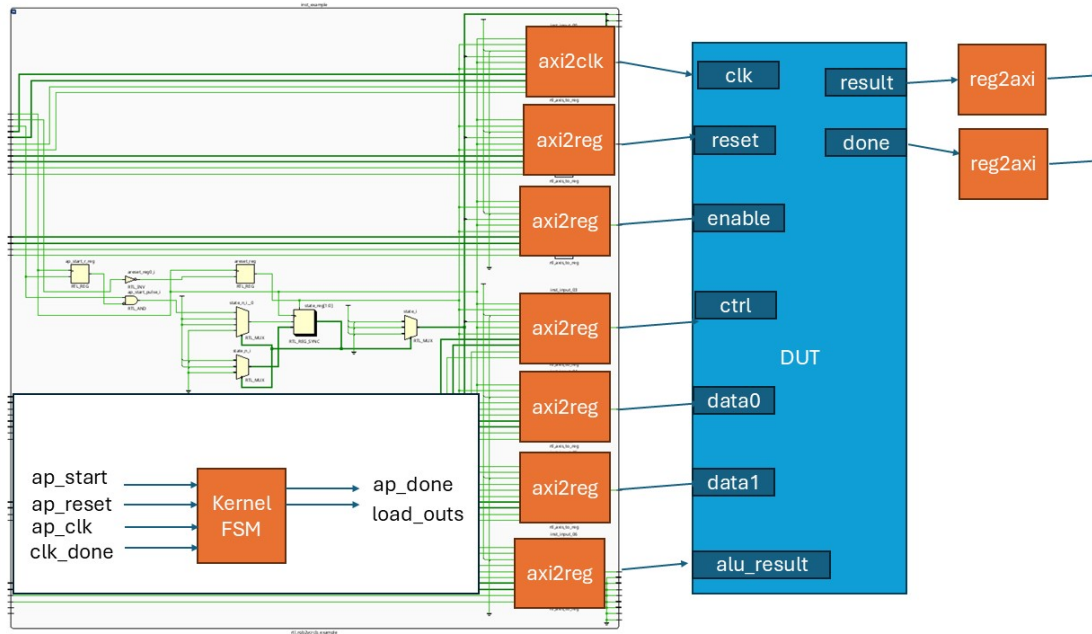
El final de la transferència es marca amb el senyal **TLAST** de l'AXI4-Stream, que indica l'últim element del bloc de dades (per exemple, l'últim píxel d'una imatge). Aquest senyal és el que permet als kernels consumidors saber quan han de finalitzar, i té un paper central en la sincronització del *pipeline* (Secció 5.5.2).

5.4.3 DUT *wrapper* i generació del fitxer .xo

Per empaquetar el DUT com a kernel RTL es va partir de l'assistent *RTL Kernel Wizard* de Vivado i de la documentació oficial d'AMD sobre la creació de kernels RTL que descriuen com configurar la interfície del kernel —ports AXI4-Stream, interfície de control, rellotge i reset— i com generar-ne una estructura de projecte de partida.

A partir d'aquest punt es va reproduir, en mode no interactiu, la seqüència que l'entorn gràfic executa en prémer *Generate RTL Kernel*. El DUT, descrit en py4hw i exportat a Verilog, no es pot connectar directament al *pipeline*: cal embolcallar-lo en un kernel que adapti els seus ports al protocol AXI4-Stream i que en gestioni l'execució. Aquest *wrapper*, que actua com a kernel intermedi, registra els valors que arriben pels canals d'entrada, genera el tren de polsos de rellotge necessari perquè el DUT avanci i, un cop el càlcul ha finalitzat, presenta els resultats als canals de sortida. D'aquesta manera, un circuit definit en py4hw queda envoltat per una estructura que el comunica amb la resta del sistema sense alterar-ne el comportament.

La Figura 5.13 mostra l'estructura interna del *DUT wrapper*. El bloc central és el DUT generat a partir del model py4hw, mentre que al seu voltant s'afegeixen blocs auxiliars encarregats d'adaptar la comunicació amb el *pipeline* AXI4-Stream. Les entrades del DUT es carreguen mitjançant blocs `axi2reg`, el rellotge del circuit es genera amb el bloc `axi2clk`, i les sortides es retornen al *pipeline* a través dels blocs `reg2axi`. A més, una màquina d'estats `KernelFSM` coordina l'execució global del kernel mitjançant els senyals de control `ap_start`, `ap_reset`, `ap_done`, `clk_done` i `load_outs`.



Font: Elaboració pròpia.

Figura 5.13: Estructura interna del *DUT wrapper* plantejat com a kernel RTL. El *wrapper* adapta les entrades i sortides del DUT al protocol AXI4-Stream mitjançant els blocs **axi2reg**, **axi2clk** i **reg2axi**, mentre que la màquina d'estats **KernelFSM** en coordina el control.

El bloc **axi2reg** s'utilitza per adaptar cada canal AXI4-Stream d'entrada a un registre intern connectat al port corresponent del DUT. En aquest bloc, el senyal **TREADY** indica que el *wrapper* està preparat per rebre una dada, mentre que **TVALID** és activat pel bloc emissor quan la dada és disponible. El valor rebut es carrega al registre quan es compleix el handshake del protocol, és a dir, quan **TVALID** i **TREADY** estan actius al mateix cicle. A partir d'aquest moment, el valor queda estable a l'entrada del DUT.

El bloc **axi2clk** es planteja com el bloc encarregat de generar el senyal de rellotge que fa avançar el DUT. En l'arquitectura proposada, aquest bloc hauria de rebre per AXI4-Stream el nombre de cicles que cal executar, generar el tren de polsos corresponent i activar el senyal **clk_done** un cop finalitzada la seqüència. En el prototip actual, aquesta funcionalitat queda definida a nivell d'arquitectura, però la seva implementació completa es deixa com a línia futura de desenvolupament.

Finalment, el bloc **reg2axi** fa el procés invers al de **axi2reg**: captura una sortida del DUT i l'envia cap al kernel d'escriptura mitjançant AXI4-Stream. Aquest bloc no pot enviar la dada immediatament, sinó que espera el senyal **load_outs**. Quan aquest senyal s'activa, la sortida del DUT es carrega en un registre intern i el bloc activa **TVALID**. La transmissió es considera completada quan el receptor també activa **TREADY**. D'aquesta manera, es garanteix que el writer

només rebí dades quan les sortides del DUT ja són vàlides.

Un cop construït l'RTL del *wrapper*, cal empaquetar-lo com a kernel en un fitxer `.xo` perquè el linker de Vitis el pugui connectar amb els kernels de lectura i d'escriptura. Aquest empaquetament reproduïx, en mode no interactiu, la seqüència que l'entorn gràfic de Vivado executa en primer *Generate RTL Kernel*; la seqüència exacta d'ordres i les funcions que l'automatitzen es detallen a la Secció 5.5.1.

5.4.4 Integració AXI4-Stream i control de flux

Amb els tres kernels disponibles, la integració consisteix a connectar-los mitjançant canals AXI4-Stream i a garantir que les dades hi circulin de manera coherent.

Durant aquesta fase van aparèixer dues incidències de control de flux que convé documentar. Totes dues es manifestaven igual —amb resultats incorrectes a la sortida—, però per causes diferents: en un cas perquè es descartaven dades de manera silenciosa, i en l'altre perquè el kernel d'escriptura llegia valors abans que el càlcul hagués acabat.

La primera era una incompatibilitat d'amplada en els canals AXI4-Stream. Si l'amplada no es fixa de manera explícita, l'eina `v++` pot inserir convertidors d'amplada que descarten dades de manera silenciosa.

La solució va ser configurar explícitament l'amplada de cada canal en bytes (`CONFIG.AXIS{nn}_NUM_BYTES {8}`) en la creació del projecte del kernel, evitant així la inserció d'aquests convertidors.

Com s'ha mostrat a la Figura 5.13, el senyal `load_outs` és el que habilita els blocs `reg2axi` perquè capturin les sortides del DUT i les transmetin cap al writer. Per aquest motiu, aquest senyal és crític per a la sincronització del *pipeline*: si s'activa massa aviat, el writer pot llegir valors que encara no corresponen al resultat final del DUT.

La segona afectava la senyalització de validesa de les sortides. El senyal que indica que les sortides del DUT ja estan disponibles (`load_outs`) estava fixat permanentment a actiu (`Constant(1)`), de manera que el kernel d'escriptura podia llegir valors abans que el càlcul hagués acabat.

5.5 Integració sistema complet

Un cop disponibles els kernels individuals i el DUT *wrapper*, l'últim pas és integrar-los en un únic binari executable a la FPGA i definir com circulen les dades entre el host i el dispositiu. Aquesta secció descriu la generació del fitxer `.xclbin` i la configuració de connectivitat entre

kernels, i a continuació el flux de dades complet host-FPGA-host, amb l'ordre de llançament necessari per evitar bloquejos.

En aquest punt cal distingir entre dues fases diferents del flux. La primera és la fase de construcció o compilació, en què es generen els fitxers `.xo`, es defineix la connectivitat entre kernels i s'obté el binari final `.xclbin`. Aquesta fase es fa amb Vivado i Vitis i només cal repetir-la quan canvia el disseny hardware o la configuració del sistema. La segona és la fase d'execució, en què el programa host carrega el fitxer `.xclbin` a la FPGA mitjançant XRT, transfereix les dades d'entrada, llança els kernels i recupera els resultats. Per tant, compilar equival a construir el binari hardware que s'executarà a la placa, mentre que executar equival a fer servir aquest binari ja generat per processar dades reals o vectors de verificació.

5.5.1 Generació dels fitxers XO i XCLBIN

En el flux de Vitis, cada kernel es genera inicialment com un fitxer objecte amb extensió `.xo`. Aquest fitxer encapsula el kernel, ja sigui generat a partir de HLS o empaquetat com a RTL, juntament amb la informació d'interfície necessària per integrar-lo dins del sistema accelerat. Posteriorment, els diferents fitxers `.xo` es combinen durant l'etapa de *link* per generar el fitxer final `.xclbin`. Aquest fitxer és el binari que es carrega a la targeta FPGA mitjançant XRT i conté la configuració hardware i els kernels que s'executaran al dispositiu.

Per reproduir en mode no interactiu, el que l'entorn gràfic de Vivado fa en prémer *Generate RTL Kernel*, calia determinar amb exactitud quines ordres executa internament. Aquesta informació no és directament visible: el registre d'execució (`.jou`) només mostra una crida `source rtl_kernel_ex.tcl -notrace`, i l'opció `-notrace` amaga totes les ordres internes de l'script. Per obtenir la seqüència real es va inspeccionar la sortida de la consola Tcl durant l'empaquetament, juntament amb el contingut de l'script de packaging generat per Vivado (`package_kernel.tcl`). Aquesta anàlisi va revelar que l'acció es redueix a tres ordres:

- carregar `package_kernel.tcl` —que defineix el procediment `package_project`—
- executar `package_project` per empaquetar l'IP
- cridar `package_xo` per generar el `.xo`

aquesta darrera, significativament, fora de l'script d'empaquetament. A partir d'aquesta seqüència es van construir les funcions descrites a continuació, que la repliquen en mode *batch*:

```
generate_package_tcl(dut, dut_files, output_dir)
```

Aquesta funció genera —escriu a disc— l'script TCL `package_rtl_kernel.tcl` dins del

directori del projecte. L'script reproduïx la seqüència d'empaquetament que Vivado executa internament en prémer *Flow* → *Generate RTL Kernel*: obre el projecte d'exemple (`rtl_kernel_ex.xpr`), carrega l'script de packaging del kernel (`package_kernel.tcl`, que defineix el procediment `package_project` propi del kernel), empaqueta l'IP mitjançant les ordres `ipx` —tot generant el `component.xml`—, elimina qualsevol `.xo` previ i, finalment, crida `package_xo` per agrupar l'IP empaquetat i el `kernel.xml` en el fitxer `rtl_kernel.xo`. Cal remarcar que la funció només escriu l'script; no l'executa.

`generate_rtl_kernel(output_path)`

Aquesta funció executa el flux de Vivado en mode batch que produeix el `.xo`, en dues fases independents.

A la primera crida Vivado carregant `create_project.tcl`, que crea el projecte, instancia l'IP *RTL Kernel Wizard* i obre l'*example design* (generant la carpeta `imports/` amb el `kernel.xml` i el `package_kernel.tcl`).

A la segona, torna a cridar Vivado carregant `package_rtl_kernel.tcl`, que obre l'*example project* i executa l'empaquetament (`package_project` i `package_xo`) per generar el `rtl_kernel.xo`. Cada fase és una crida `vivado -mode batch` separada —un procés nou—, motiu pel qual la segona ha de tornar a obrir el projecte.

La construcció del binari es controla mitjançant l'script `build_xclbin.sh`, generat automàticament pel flux. Aquest script encapsula les crides a `v++` necessàries per compilar els kernels HLS i enllaçar-los amb el kernel RTL empaquetat prèviament. De manera simplificada, el procés segueix l'estructura següent:

```
v++ -c -t hw --platform $PLATFORM
-k krnl_reader
-o krnl_reader.xo krnl_reader.cpp

v++ -c -t hw --platform $PLATFORM
-k krnl_writer
-o krnl_writer.xo krnl_writer.cpp

v++ -l -t hw --platform $PLATFORM
--config connectivity.cfg
-o kernel.xclbin
krnl_reader.xo rtl_kernel.xo krnl_writer.xo
```

Les dues primeres ordres corresponen a la fase de compilació dels kernels HLS de lectura i escriptura, mentre que la tercera correspon a la fase d'enllaç. En aquesta última etapa, Vitis combina els fitxers `.xo` amb la configuració de connectivitat definida a `connectivity.cfg` i genera el fitxer final `.xclbin`, que posteriorment es carrega a la FPGA mitjançant XRT.

```

(.venv) aoliva@mise-clancy:~/py4hw/test/interactive$ ls -la /tmp/vitis_hil_aoliva/dut_simp/
total 45936
drwxrwxr-x 8 aoliva aoliva 4096 Jun 25 18:39 .
drwxrwxr-x 3 aoliva aoliva 4096 Jun 25 17:09 ..
-rwxrwxr-x 1 aoliva aoliva 1425 Jun 25 17:10 build_xclbin.sh
-rw-rw-r-- 1 aoliva aoliva 374 Jun 25 17:09 connectivity.cfg
-rw-rw-r-- 1 aoliva aoliva 899 Jun 25 17:09 create_project.tcl
-rw-rw-r-- 1 aoliva aoliva 423 Jun 25 18:39 DUT_7d44bb27a5a0.v
-rw-rw-r-- 1 aoliva aoliva 423 Jun 25 17:09 DUT_7d44bb27a5a0.v
-rw-rw-r-- 1 aoliva aoliva 1599 Jun 25 17:55 hil.ltx
-rw-rw-r-- 1 aoliva aoliva 46607798 Jun 25 17:55 hil.xclbin
-rw-rw-r-- 1 aoliva aoliva 14958 Jun 25 17:55 hil.xclbin.info
-rw-rw-r-- 1 aoliva aoliva 38151 Jun 25 17:55 hil.xclbin.link_summary
-rw-rw-r-- 1 aoliva aoliva 809 Jun 25 17:09 krnl_reader.cpp
-rw-rw-r-- 1 aoliva aoliva 128212 Jun 25 17:11 krnl_reader.xo
-rw-rw-r-- 1 aoliva aoliva 6403 Jun 25 17:11 krnl_reader.xo.compile_summary
-rw-rw-r-- 1 aoliva aoliva 852 Jun 25 17:09 krnl_writer.cpp
-rw-rw-r-- 1 aoliva aoliva 123544 Jun 25 17:11 krnl_writer.xo
-rw-rw-r-- 1 aoliva aoliva 6403 Jun 25 17:11 krnl_writer.xo.compile_summary
-rw-rw-r-- 1 aoliva aoliva 11207 Jun 25 17:09 package_rtl_kernel.tcl
drwxrwxr-x 3 aoliva aoliva 4096 Jun 25 17:09 rtl_kernel.cache
drwxrwxr-x 11 aoliva aoliva 4096 Jun 25 17:10 rtl_kernel_ex
-rw-rw-r-- 1 aoliva aoliva 5214 Jun 25 17:09 rtl_kernel_example.v
drwxrwxr-x 3 aoliva aoliva 4096 Jun 25 17:09 rtl_kernel.gen
drwxrwxr-x 2 aoliva aoliva 4096 Jun 25 17:09 rtl_kernel.hw
drwxrwxr-x 2 aoliva aoliva 4096 Jun 25 17:09 rtl_kernel.ip_user_files
drwxrwxr-x 3 aoliva aoliva 4096 Jun 25 17:09 rtl_kernel.srds
-rw-rw-r-- 1 aoliva aoliva 11101 Jun 25 17:09 rtl_kernel.xpr

```

Font: Elaboració pròpia.

Figura 5.14: Fitxers generats automàticament al directori del DUT després d'executar el flux de construcció per a Vitis.

Pel cas del dut senzill, dins del *py4hw* es fa:

1. `cd test/interactive/`
2. `python tb_HILWrapper_Vitis.py`
3. `bash /tmp/vitis_hil_aoliva/dut_simp/build_xclbin.sh`
4. `python tb_HILWrapper_Vitis.py`

Entre el pas 3 i 4 podem mirar quins arxius s'han creat durant el procés fent `ls -la /tmp/vitis_hil_aoliva/dut_simp/`. Veiem el resultat d'aquesta instrucció a la Figura 5.14, on s'observen diversos fitxers generats automàticament pel flux, com els scripts de construcció, els fitxers de configuració i els directoris associats a la generació del binari per a Vitis. El pas 4 és el que permet veure si els resultats són els esperats o no. Això ho veurem amb més detall a la Secció 6.1.1, pel cas del DUT Add/Sub; i a la Secció 6.1.2

Cal tenir en compte que la generació del fitxer `.xclbin` és una fase costosa en temps, ja que inclou la síntesi, la implementació i l'enllaç dels diferents kernels del sistema. En l'entorn de treball utilitzat, una compilació completa en mode hardware (**hw**) podia tenir una durada aproximada d'uns 45 minuts, depenent del DUT i de l'estat del servidor compartit. Aquesta durada correspon únicament a la fase de construcció del binari, i no al temps d'execució del DUT sobre la FPGA. Un cop generat el fitxer `.xclbin`, aquest es pot reutilitzar en execucions posteriors mentre no es modifiqui el disseny hardware ni la configuració de connectivitat.

Pel cas del DUT `rgb` es segueix el mateix procés esmentat anteriorment, amb un pas a destacar. Dins del codi `tb_HILWrapper_Vitis` s'han de comentar/descomentar les corresponents línies de codi que corresponen a la creació dels cables de l'esquema i les seqüències, ja que aquests són específics per cada DUT. Veiem-les a continuació.

- Pel cas de la creació dels cables de l'esquemàtic:

```
a = hw.wire('a', 32)
```

- Pel cas de l'assignació de valors a les senyals d'entrada:

```
py4hw.Sequence(hw, 'a', [1,2,3,4,5,6,7], a)
```

- Cal destacar la importància de definir la variable *DUT* correctament en cada cas, com es fa en aquesta línia:

```
dut = DUT(hw, 'dut', a, b, p, m)
```

5.5.2 Flux de dades host–FPGA–host

El flux complet s'organitza com un *pipeline* de tres kernels connectats per AXI4-Stream:

- `reader_kernel` llegeix les dades d'entrada de memòria i genera els streams d'entrada
- el kernel RTL `rtl_mid` registra aquests valors, genera el tren de polsos de rellotge per al DUT i en produeix el resultat
- finalment, `writer_kernel` consumeix l'stream de sortida i escriu el resultat a memòria

Un cop el *pipeline* ha finalitzat, el host llegeix la memòria de sortida i realitza la verificació funcional, comparant els resultats amb els del model software de *py4hw*.

5.6 Metodologia experimental

Aquesta secció descriu com s'ha avaluat la plataforma:

- quins circuits s'han utilitzat com a casos de prova
- quins estímuls i paràmetres s'han aplicat
- quines mètriques i procediment de comparació s'han seguit per contrastar l'execució híbrida amb la simulació purament software

Els resultats obtinguts es presenten i es discuteixen al Capítol 6.

5.6.1 Circuits de prova (Add/Sub i RGB2YCbCr)

Els dos circuits descrits a la Secció 5.4.1 s'han emprat com a casos de prova amb papers complementaris.

El circuit aritmètic Add/Sub, per la seva simplicitat, permet validar el flux amb un conjunt de vectors d'entrada dirigits, escollits per cobrir casos representatius (valors nuls, operands petits i casos que exerciten la resta amb resultat negatiu).

El convertidor RGB2YCbCr, en canvi, representa un bloc de processament més realista i, en tenir múltiples sortides, permet validar el comportament del sistema en el cas que motiva el redirector (Secció 5.3.3).

5.6.2 Mètriques i procediment de comparació (software vs híbrid)

L'avaluació té dues vessants: la *correcció funcional* i el *rendiment*.

Verificació funcional. Per a cada cas de prova s'executa primer la simulació purament software del DUT en *py4hw*, que proporciona els resultats de referència, i després l'execució híbrida sobre la FPGA. Tots dos conjunts de resultats es comparen *bit a bit*: per a l'Add/Sub, sobre l'espai d'entrades dirigit; per al RGB2YCbCr, sobre el conjunt de mostres seleccionat. Es considera que el sistema és funcionalment equivalent si la coincidència és total. Aquesta comparació es fa de manera manual/visual.

Rendiment. Per avaluar el rendiment es mesura el temps d'execució de la simulació software i el temps de l'execució híbrida. En el cas software, el temps correspon a l'execució del DUT dins de *py4hw*. En el cas híbrid, el temps inclou la comunicació host-FPGA, la transferència de dades, el llançament dels kernels i la recuperació dels resultats. No s'hi inclou el temps de compilació ni de generació del fitxer *.xclbin*, ja que aquesta fase només cal repetir-la quan es modifica el disseny hardware. A partir d'aquests temps es calcula el factor d'acceleració com: A partir d'aquests temps es calcula el factor d'acceleració. On (T_{software}) és el temps de la simulació purament software i (T_{hibrid}) és el temps de l'execució sobre la FPGA.

Capítol 6

Resultats, conclusions i treballs futurs

6.1 Verificació funcional

6.1.1 Add/Sub: verificació dirigida

Pel cas del dut de suma/resta, es fan servir les següents entrades:

```
py4hw.Sequence(hw, 'a', [1,2,3,4,5,6,7], a)
py4hw.Sequence(hw, 'b', [0x1A00,0x2B00,0x3C00], b)
```

Un cop s'han creat tots els arxius corresponents per poder executar el circuit a la FPGA, tal com s'ha detallat al Capítol 5 anterior, s'executa el binari a la placa fent la línia *python tb_HILWrapper_Vitis.py*. Podem observar que s'obre una finestra "Workbench" on podem veure les senyals i els blocs. Quan es fa el primer cicle s'aprecien les entrades i les sortides esperades. Pel primer cicle tenim la Figura 6.1, on podem observar que, efectivament, la sortida del hardware i la sortida del software coincideixen. El mateix passa pels sis valors restants d'entrada.

La Taula 6.1 recull els resultats: en els vuit vectors, les sortides de la FPGA coincideixen

Name	Direction	Width	Value
a	in	32	0x1
b	in	32	0x1a00
p	in	32	0x1a01
m	in	32	0xffffe601
p_hw	in	32	0x1a01
m_hw	in	32	0xffffe601

Figura 6.1: Verificació del DUT Add/Sub a la finestra Workbench, comparant les sortides del model software amb les obtingudes sobre la FPGA.

Taula 6.1: Verificació dirigida del DUT Add/Sub: comparació bit a bit entre la simulació software (*py4hw*) i l'execució sobre FPGA. Valors en hexadecimal de 32 bits.

#	<i>a</i>	<i>b</i>	Software (<i>py4hw</i>)		FPGA		SW = HW
			<i>p</i>	<i>m</i>	<i>p_{hw}</i>	<i>m_{hw}</i>	
1	0x00000001	0x00001a00	0x00001a01	0xffffe601	0x00001a01	0xffffe601	✓
2	0x00000002	0x00002b00	0x00002b02	0xffffd502	0x00002b02	0xffffd502	✓
3	0x00000003	0x00003c00	0x00003c03	0xffffc403	0x00003c03	0xffffc403	✓
4	0x00000004	0x00001a00	0x00001a04	0xffffe604	0x00001a04	0xffffe604	✓
5	0x00000005	0x00002b00	0x00002b05	0xffffd505	0x00002b05	0xffffd505	✓
6	0x00000006	0x00003c00	0x00003c06	0xffffc406	0x00003c06	0xffffc406	✓
7	0x00000007	0x00001a00	0x00001a07	0xffffe607	0x00001a07	0xffffe607	✓
8	0x00000001	0x00002b00	0x00002b01	0xffffd501	0x00002b01	0xffffd501	✓

bit a bit amb les de la simulació software.

6.1.2 RGB2YCbCr: verificació per mostreig

La verificació del convertidor RGB2YCbCr es fa per mostreig, amb un conjunt de píxels representatius dels quals es comprova que la sortida de la FPGA coincideix bit a bit amb la de la simulació software.

Pel cas del DUT RGB2YCbCr, es fan servir les següents entrades:

```
py4hw.Sequence(hw, 'reset',      [0], reset)
py4hw.Sequence(hw, 'enable',     [0, 1], enable)
py4hw.Sequence(hw, 'ctrl',       [0], ctrl)
py4hw.Sequence(hw, 'a', [0xFF0000, 0x00FF00, 0x0000FF], a)
py4hw.Sequence(hw, 'b',          [0], b)
py4hw.Sequence(hw, 'alu_result', [0], alu_result)
```

Aquests tres píxels corresponen als components primaris purs —vermell (0xFF0000), verd (0x00FF00) i blau (0x0000FF)—, que exerciten per separat cadascun dels canals de color.

Un cop s'han creat tots els arxius corresponents per poder executar el circuit a la FPGA, s'executa el binari a la placa fent la línia *python tb_HILWrapper_Vitis.py*.

Podem observar que s'obre una finestra "Workbench" on podem veure les senyals i els blocs. Quan es fa el primer cicle s'aprecien les entrades i les sortides esperades. Pel primer cicle tenim la Figura 6.2, on podem observar que, efectivament, la sortida del hardware i la sortida del software coincideixen. El píxel d'entrada arriba empaquetat en la paraula **a** i el resultat YCbCr

Name	Direction	Width	Value
reset	in	1	0x0
enable	in	1	0x0
ctrl	in	32	0x0
a	in	32	0xff0000
b	in	32	0x0
alu_result	in	32	0x0
result	in	32	0x4c54ff
done	in	1	0x0
result_hw	in	32	0x4c54ff
done_hw	in	1	0x0

Figura 6.2: Verificació del DUT RGB2YCbCr per a un conjunt de valors d'entrada, comparant els resultats software i hardware.

Name	Direction	Width	Value
reset	in	1	0x0
enable	in	1	0x0
ctrl	in	32	0x0
a	in	32	0x87ceeb
b	in	32	0x0
alu_result	in	32	0x0
result	in	32	0xbc9a5a
done	in	1	0x0
result_hw	in	32	0xbc9a5a
done_hw	in	1	0x0

Figura 6.3: Verificació del DUT RGB2YCbCr per al píxel d'entrada 0x87CEEB, comparant els resultats software i hardware.

surt empaquetat en **result**, amb l'ordre de bytes Y|Cb|Cr. El mateix passa per a tots els valors restants d'entrada.

També s'ha mirat el cas concret de posar a l'entrada un píxel corresponent al color blau cel, seguint el mateix patró que al test preliminar que es va fer per verificar el mòdul RTL del DUT. Per tant, es canvia l'entrada "a" vista anteriorment, per la següent:

```
py4hw.Sequence(hw, 'a', [0x87CEEB], a) #0x87CEEB blue pixel
```

També observem que els resultats entre el hw i sw coincideixen correctament, veure Figura 6.3.

La Taula 6.2 recull tots els casos verificats: en tots, les sortides de la FPGA coincideixen bit a bit amb les de la simulació software, fet que confirma l'equivalència funcional del DUT sobre el maquinari. Convé remarcar que aquest DUT, en tenir més ports d'entrada i senyals de control que cal seqüenciar (**reset**, **enable** i **ctrl**), exercita un escenari de sincronització més exigent que l'Add/Sub.

Taula 6.2: Verificació per mostreig del DUT RGB2YCbCr: comparació bit a bit entre la simulació software (*py4hw*) i l'execució sobre FPGA. Píxels d'entrada i sortides YCbCr empaquetades, en hexadecimal de 32 bits (ordre de bytes Y|Cb|Cr). Els tres primers casos corresponen als components purs vermell, verd i blau; el píxel 0x87CEEB és el blau cel emprat a la prova preliminar del mòdul RTL (Secció 5.2.2).

#	<i>a</i> (píxel RGB)	result (SW)	result_hw (HW)	SW = HW
1	0xFF0000	0x4C54FF	0x4C54FF	✓
2	0x00FF00	0x952B15	0x952B15	✓
3	0x0000FF	0x1DFF6B	0x1DFF6B	✓
4	0x87CEEB	0xBC9A5A	0xBC9A5A	✓

6.1.3 Verificació automatitzada amb *pytest*

A més de la verificació dels dos DUTs principals, la plataforma s'ha sotmès a una bateria de tests unitaris ja inclosa al *framework* (disponible al repositori de *py4hw* [7], dins del directori `test/hil_unit_AlveoU55C/`), basada en *pytest* i en comprovacions, sobre un conjunt de circuits addicionals.

Aquests tests no s'han desenvolupat en aquest treball, sinó que ja estaven muntats; la contribució pròpia ha estat connectar-hi l'execució real sobre la FPGA, de manera que la comparació que abans es feia contra el model de referència software ara es fa també contra el maquinari.

Per a cada DUT, el test genera mil vectors d'entrada aleatoris i, per a cadascun, compara mitjançant una comprovació, la sortida de la simulació software de *py4hw* amb la produïda per l'execució sobre la FPGA. Convé precisar que la bateria funcional i la de rendiment no avaluen exactament el mateix conjunt de circuits, perquè responen a preguntes diferents.

La verificació funcional (Taula 6.3) s'aplica als DUTs amb el test de vectors aleatoris actiu — Abs, FXPadder i SignedAdd—; el FPadder_SP en queda exclòs perquè té aquest test desactivat.

La mesura de rendiment (Secció 6.2), en canvi, contrasta deliberadament un circuit mínim (Abs) amb un de més càrrega de càlcul (FPadder_SP) per fer visible com varia la relació entre el temps de còmput i el de comunicació.

La majoria dels circuits superen la totalitat dels vectors, confirmant l'equivalència funcional bit a bit. En el cas del SignedAdd, però, la verificació automatitzada va detectar una discrepància sistemàtica en els casos amb *carry-in* actiu; aquest comportament, que una comparació manual difícilment hauria revelat, il·lustra precisament la utilitat de l'aproximació automatitzada. La determinació de la causa exacta es deixa com a treball futur. La Taula 6.3 en recull el resum.

Taula 6.3: Verificació funcional dels DUTs addicionals mitjançant la bateria de tests unitaris (*pytest*). Cada DUT s'avalua amb mil vectors aleatoris, comparant bit a bit la sortida de la simulació software (*py4hw*) amb la de l'execució sobre FPGA.

DUT	Vectors	Resultat
Abs	1000 aleatoris	✓ Tots passen
FXPAdder	1000 aleatoris	✓ Tots passen
SignedAdd	1000 aleatoris	Discrepància amb ci=1 (pendent)

6.2 Anàlisi de rendiment

A més de la verificació funcional, s'ha caracteritzat el cost temporal de l'execució HIL, comparant la velocitat de la simulació purament software de *py4hw* amb la de l'execució redirigida a la FPGA. Aquesta mesura permet quantificar el preu que es paga per executar el DUT sobre el maquinari real en l'estat actual de la plataforma.

6.2.1 Metodologia de mesura (SW vs HW)

El test de rendiment (`test_speedup`) executa cent mil cicles consecutius del DUT en dues configuracions.

En primer lloc, sobre la simulació purament software de *py4hw*, mesurant el temps total i derivant-ne el nombre de mostres processades per segon.

A continuació, substituint el DUT pel redirector *DUTProxyXRT*, de manera que cada cicle es delega a la FPGA, i mesurant de nou el mateix indicador. La comparació d'ambdues taxes quantifica el cost de l'execució HIL respecte de la simulació software.

6.2.2 Resultats i discussió

Per al DUT Abs, la sortida del test és la següent:

```
SW = Run 100000 tests in 0.1167 seconds => 857115.65 samples / second
HW = Run 100000 tests in 11.2193 seconds => 8913.23 samples / second
```

I per al DUT FXPAdder.SP:

```
SW = Run 100000 tests in 8.8084 seconds => 11352.75 samples / second
HW = Run 100000 tests in 12.5256 seconds => 7983.62 samples / second
```

```

INFO: [v++ 60-1441] [15:39:49] Run run_link: Step generate_sc_driver: Completed
Time (s): cpu = 00:00:00 ; elapsed = 00:00:00 . Memory (MB): peak = 467.883 ; gain = 0.000 ; free physical = 119028 ; f
e virtual = 134941
Check POST-VPL, containing 1 checks, has run: 0 errors
INFO: [v++ 60-244] Generating system estimate report...
INFO: [v++ 60-1092] Generated system estimate report: /home/aoliva/py4hw/test/unit_hil_AlveoU55C/_x/reports/link/system
estimate_hil.txt
INFO: [v++ 60-586] Created /tmp/vitis_hil_aoliva/test_fpadder_sp/hil.ltx
INFO: [v++ 60-586] Created /tmp/vitis_hil_aoliva/test_fpadder_sp/hil.xclbin
INFO: [v++ 60-1307] Run completed. Additional information can be found in:
Guidance: /home/aoliva/py4hw/test/unit_hil_AlveoU55C/_x/reports/link/v++_link_hil_guidance.html
Timing Report: /home/aoliva/py4hw/test/unit_hil_AlveoU55C/_x/reports/link/impl_1_hw_bb_locked_timing_summar
routed.rpt
Vivado Log: /home/aoliva/py4hw/test/unit_hil_AlveoU55C/_x/logs/link/vivado.log
Steps Log File: /home/aoliva/py4hw/test/unit_hil_AlveoU55C/_x/logs/link/link.steps.log
INFO: [v++ 60-2343] Use the vitis_analyzer tool to visualize and navigate the relevant reports. Run the following comma
nd:
vitis_analyzer /tmp/vitis_hil_aoliva/test_fpadder_sp/hil.xclbin.link_summary
INFO: [v++ 60-791] Total elapsed time: 0h 51m 29s
INFO: [v++ 60-1653] Closing dispatch client.
=====
XCLBIN generated:
/tmp/vitis_hil_aoliva/test_fpadder_sp/hil.xclbin
=====
[OK] script executed
FPadder_SP compilation time: 3203 seconds = 0 : 53 : 23

SW = Run 100000 tests in 8.808441877365112 seconds => 11352.74562655265 samples / second
HW = Run 100000 tests in 12.525645017623901 seconds => 7983.62079232626 samples / second

===== 2 passed in 3229.67s (0:53:49) =====

```

Figura 6.4: Sortida de l'execució del test de rendiment per al DUT FPadder_SP, on es mostra la generació del binari, la superació dels tests i la mesura de mostres per segon en simulació software i en execució sobre FPGA.

La Taula 6.4 resumeix aquestes mesures. En tots dos casos, l'execució sobre la FPGA processa moltes menys mostres per segon que la simulació software. Aquest resultat és coherent amb la naturalesa del sistema en el seu estat actual: cada avaluació del DUT comporta una transferència completa host-FPGA-host a través de PCIe/XRT, de manera que la latència de comunicació per crida domina sobre el temps de càlcul, especialment per a DUTs senzills com els avaluats.

Taula 6.4: Rendiment: simulació software (*py4hw*) vs. execució sobre FPGA, per a cent mil avaluacions consecutives.

DUT	SW (mostres/s)	HW (mostres/s)
Abs	857115,65	8913,23
FPadder_SP	11352,75	7983,62

Convé remarcar la diferència de comportament entre els dos DUTs avaluats. En el cas de l'Abs, que és un circuit molt senzill, el càlcul és pràcticament immediat i el temps total queda dominat per la comunicació host-FPGA-host. Per aquest motiu, l'execució sobre la FPGA resulta molt més lenta que la simulació software.

En canvi, en el cas del Test_FPadder_SP, que presenta una complexitat computacional més elevada, el pes relatiu de la latència de comunicació és menor i la diferència entre l'execució software i hardware es redueix notablement. Aquesta tendència indica que l'aproximació HIL només pot resultar avantatjosa per a DUTs amb una càrrega de càlcul prou elevada, o bé quan la latència de comunicació s'amortitza mitjançant el processament per lots.

6.3 Conclusions

Aquest treball ha presentat el disseny i la implementació d'una plataforma de verificació híbrida de tipus Hardware-in-the-Loop que connecta la descripció i simulació de circuits digitals en *py4hw* amb la seva execució física sobre una FPGA AMD Alveo U55C. El control de la verificació es manté en l'entorn Python, mentre que el càlcul del DUT es delega al hardware a través d'una infraestructura —generada automàticament a partir del mateix model *py4hw*— que adapta els ports del circuit a canals AXI4-Stream, l'empaqueta com a kernel RTL i l'integra en el flux Vitis/XRT.

El resultat principal del projecte és que s'ha aconseguit una equivalència funcional bit a bit entre la simulació software de *py4hw* i l'execució sobre la FPGA per als dos circuits de prova emprats: el mòdul aritmètic Add/Sub i el convertidor d'espai de color RGB2YCbCr. Aquesta coincidència, verificada per a tots els vectors d'estímul aplicats, confirma que el flux proposat preserva el comportament del circuit al llarg de tota la cadena —generació de Verilog, empaquetament, integració i execució— i que la comunicació host-FPGA manté la coherència de les dades.

Més enllà d'aquests dos circuits, la verificació automatitzada amb *pytest* (Secció 6.1.3) va ampliar la cobertura a DUTs addicionals. La majoria reproduïen l'equivalència bit a bit, però en el cas del SignedAdd es va detectar una discrepància sistemàtica per als casos amb carry-in actiu, la causa de la qual resta pendent de determinar. Lluny de ser un punt feble del treball, aquesta detecció il·lustra el valor de l'automatització: una comparació manual sobre uns pocs vectors dirigits difícilment l'hauria revelada, mentre que la generació massiva de vectors aleatoris la va fer evident. La seva resolució queda apuntada com a treball futur.

A més de la verificació funcional, s'ha realitzat una primera caracterització del rendiment de la plataforma. Els resultats mostren que, en l'estat actual, l'execució sobre FPGA no proporciona encara una acceleració respecte de la simulació software, ja que el temps total queda dominat per la latència de comunicació host-FPGA-host. Aquest resultat no invalida la plataforma, sinó que delimita clarament el seu estat actual: el flux és funcionalment correcte, però requereix optimitzacions de comunicació, especialment processament per lots, per poder obtenir guanys de rendiment. Aquesta deficiència, identificada amb dades concretes, defineix la principal línia de treball futur (Secció 6.4).

6.3.1 Valoració dels objectius assolits

Recuperant els objectius específics plantejats a la Secció 1.3, se'n fa a continuació una valoració.

Pel que fa a l'**objectiu de desenvolupar un entorn capaç de delegar la simulació de subcircuits a una FPGA**, s'ha assolit amb una precisió: la divisió entre software i hardware

no es fa partint el circuit per dins, sinó tractant el DUT com una unitat indivisible que s'executa sencera sobre el hardware, mentre el control i la comparació resten en *py4hw*. Aquesta és la interpretació de *partició* que el sistema realitza i respon a la finalitat de l'objectiu —delegar l'execució d'un subcircuit a la FPGA mantenint el control en Python—, encara que no implementi un particionat intern automàtic del disseny.

El segon **objectiu, implementar sobre la FPGA un sistema capaç de rebre, executar i retornar els resultats del DUT mitjançant kernels**, s'ha complert amb l'estructura de tres kernels (reader–DUT–writer) connectats per AXI4-Stream. El DUT, exportat a Verilog des de *py4hw*, s'empaqueta com a kernel RTL i s'integra en aquest *pipeline* sense alterar-ne el comportament.

El tercer **objectiu, la transparència de cara a l'usuari**, s'ha materialitzat en el redirector (DUTProxyXRT): un component software que ocupa el lloc del DUT dins del graf de simulació de *py4hw* i redirigeix cada avaluació cap a la FPGA, de manera que des de l'entorn Python el circuit s'avalua com qualsevol altre bloc, sense que l'usuari hagi de gestionar manualment la comunicació amb el dispositiu.

El quart **objectiu, la validació per equivalència**, s'ha acomplert mitjançant la comparació bit a bit dels resultats software i hardware per als dos DUTs, amb coincidència total en tots els casos provats.

Més enllà del compliment dels objectius, el desenvolupament ha posat de manifest una sèrie de reptes d'integració, la resolució dels quals forma part del valor del treball.

6.4 Línies futures de desenvolupament

6.4.1 Integració en un flux de CI/CD per a la verificació hardware

La conversió de la verificació en tests automatitzats amb *pytest* (Secció 6.1.3) obre la porta a incorporar la plataforma dins d'un flux d'integració i desplegament continu (CI/CD).

En els projectes de software és habitual que, a cada modificació del codi, un sistema automàtic executi la bateria de tests i n'informi els resultats, de manera que qualsevol regressió es detecta immediatament. Aquesta pràctica, però, és poc comuna en disseny digital, perquè la verificació sobre hardware real exigeix tenir el dispositiu dins del bucle.

La plataforma desenvolupada elimina aquesta barrera: en exposar la verificació SW–HW com a tests *pytest* amb resultat *pass/fail*, aquests poden incorporar-se directament a un *pipeline* de CI/CD, de manera que cada canvi en un DUT o en la pròpia infraestructura desencadeni automàticament la seva verificació sobre la FPGA i la comparació bit a bit amb el model de

py4hw. Així, l'equivalència funcional quedaria garantida de forma contínua al llarg de tot el desenvolupament, sense intervenció manual.

6.4.2 Implementació completa del bloc *axi2clk*

Una altra línia futura important és completar la implementació del bloc *axi2clk* dins del *DUT wrapper*. En l'arquitectura plantejada, aquest bloc és l'encarregat de generar el senyal de rellotge intern del DUT a partir d'una dada rebuda per AXI4-Stream. La idea és que el host o el kernel de lectura indiqui el nombre de cicles que el DUT ha d'executar, i que el bloc *axi2clk* generi el tren de polsos corresponent abans d'activar el senyal *clk_done*.

En el prototip desenvolupat en aquest treball, aquesta funcionalitat s'ha definit a nivell d'arquitectura, però no s'ha implementat completament. Això limita el suport actual a DUTs amb una seqüenciació senzilla, ja que el control del nombre de cicles d'execució no queda encara automatitzat dins del wrapper. Per tant, una millora natural del sistema seria implementar aquest bloc de manera completa, incorporant un comptador intern que rebí el nombre de cicles a executar, generi el senyal *DUT_CLK* i indiqui la finalització de l'execució mitjançant *clk_done*.

Aquesta millora permetria donar suport de manera més general a DUTs seqüencials, en què el resultat no es produeix immediatament després de carregar les entrades, sinó després d'un nombre determinat de cicles de rellotge. També faria que el wrapper fos més autònom i reutilitzable, ja que el temps d'execució del DUT podria adaptar-se automàticament a cada circuit sense haver de modificar manualment la lògica de control.

6.4.3 Optimització de la comunicació i millora del rendiment

L'anàlisi de rendiment (Secció 6.2) ha mostrat que, en l'estat actual, la latència de la comunicació domina el temps total d'execució. Això es deu al fet que la plataforma processa una mostra per crida: cada avaluació del DUT comporta una transferència completa host-FPGA-host a través de PCIe/XRT, de manera que el cost fix d'aquesta transferència es paga tantes vegades com mostres s'avaluen.

La principal via de millora és el processament per lots (*batching*): en lloc d'enviar una única mostra per transferència, s'acumularia un conjunt ampli de mostres d'entrada, es transferirien totes juntes a la memòria del dispositiu, la FPGA les processaria de manera encadenada i els resultats es recuperarien en una sola transferència de retorn. D'aquesta manera, el cost fix de la comunicació s'amortitzaria entre totes les mostres del lot, en lloc de pagar-se per cadascuna. Aquesta estratègia encaixa de manera natural amb la infraestructura existent, ja que els kernels de lectura i escriptura es comuniquen amb el DUT mitjançant canals AXI4-Stream, pensats precisament per a fluxos continus de dades.

Amb un volum de mostres prou gran per amortitzar la latència, és previsible que l'execució sobre la FPGA pugui superar la simulació software, especialment per a DUTs amb una càrrega de càlcul elevada. La quantificació d'aquest guany, així com l'exploració de l'ús de transferències DMA de gran bloc i del paral·lelisme entre mostres, constitueixen la continuació natural d'aquest treball.

Bibliografia

- [1] *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*, IEEE, 2024.
- [2] *IEEE Standard for Universal Verification Methodology Language Reference Manual*, IEEE, 2020.
- [3] D. Castells-Rufas, G. Rotger, and D. Novo, “Streamlining fpga circuit design and verification with python and py4hw,” in *XI Southern Programmable Logic Conference (SPL)*, San Luis, Argentina, Mar. 2023.
- [4] D. Castells-Rufas and J. Carrabina, “Jumble: A hardware-in-the-loop simulation system for jhdl,” in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2007, pp. 345–348.
- [5] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović, “Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud,” in *Proceedings of the 45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE/ACM, 2018.
- [6] J. Ledin, “Hardware-in-the-loop simulation,” *Embedded Systems Programming*, vol. 12, no. 2, pp. 42–62, 1999.
- [7] D. Castells-Rufas, “py4hw: A python library for hardware design,” Repositori de programari, GitHub, 2026, consulta: 25 de juny de 2026. [Online]. Available: <https://github.com/davidcastells/py4hw>
- [8] *Alveo Data Center Accelerator Card Platforms User Guide*, AMD, 2023, consulta: 8 de juny de 2026. [Online]. Available: <https://docs.amd.com/r/en-US/ug1120-alveo-platforms>
- [9] *Vitis Unified Software Platform Documentation: Application Acceleration Development*, AMD, 2023, versió 2023.2. Consulta: 8 de juny de 2026. [Online]. Available: <https://docs.amd.com/r/2023.2-English/ug1393-vitis-application-acceleration>
- [10] *Xilinx Runtime (XRT) Documentation*, AMD/Xilinx, 2023, versió 2023.2. Consulta: 8 de juny de 2026. [Online]. Available: <https://xilinx.github.io/XRT/2023.2/html/index.html>
- [11] *AMBA AXI-Stream Protocol Specification*, Arm, consulta: 8 de juny de 2026. [Online]. Available: <https://developer.arm.com/documentation/ih0051/latest>
- [12] J. A. Castaño, V. Moreno, A. Cabrera, A. Gutiérrez, and V. Martínez, “Red de procesadores evolutivos para solucionar el problema de los tres colores. implementación en

- hardware,” *Revista Cubana de Ciencias Informáticas*, vol. 9, no. 4, pp. 155–170, 2015, consulta: 8 de juny de 2026. [Online]. Available: <http://scielo.sld.cu/scielo.php?script=sci.arttext&pid=S2227-18992015000400009>
- [13] *What Is Hardware-in-the-Loop (HIL)?*, NI (National Instruments), 2017, white Paper. Consulta: 8 de juny de 2026. [Online]. Available: <https://www.ni.com/es/solutions/transportation/hardware-in-the-loop/what-is-hardware-in-the-loop-.html>
- [14] All About FPGA, “VHDL testbench tutorial,” <https://allaboutfpga.com/vhdl-testbench-tutorial/>, consulta: 20 de juny de 2026.
- [15] N. Prajapati, “How easier to built basic verification testbench using UVM compared to SystemVerilog,” *International Journal of Engineering Research & Technology (IJERT)*, vol. 2, no. 11, pp. 2514–2518, Nov. 2013. [Online]. Available: <https://www.ijert.org/research/how-easier-to-built-basic-verification-testbench-using-uvm-compared-to-systemverilog-IJERTV2IS110828.pdf>
- [16] Advanced Micro Devices, Inc., “Platform Overview,” *Alveo Versal Vitis Platforms 2022.2 Documentation*, 2023. [Online]. Available: https://xilinx.github.io/Alveo-Versal-Platforms/alveoversalplatforms/build/html/docs/architecture/arch_overview.html
- [17] PC Server & Parts, “Xilinx Alveo U55C 16GB HBM PCIe FPGA Data Center Accelerator Card,” <https://pcserverandparts.com/xilinx-alveo-u55c-16gb-hbm-pcie-fpga-data-center-accelerator-card-2x-100g-qsfp28-passive-cooling-used-a-u55c/>, fotografies de la targeta. Consulta: 20 de juny de 2026.
- [18] Xilinx, “Rtl stream kernel integration,” Tutorial de Vitis, GitHub, 2023, tutorial 03 del repositori Vitis Tutorials. Consulta: 25 de juny de 2026. [Online]. Available: https://github.com/Xilinx/Vitis-Tutorials/tree/2023.1/Hardware_Acceleration/Design_Tutorials/03-rtl_stream_kernel_integration
- [19] T. Csapó, “Infocommunication: Video broadcasting,” Lecture notes, n.d., consulta: 25 de juny de 2026. [Online]. Available: https://smartlab.tmit.bme.hu/downloads/education/infocomm/Infocomm_13_video_broadcasting_v8_notes.pdf