Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Generador de Matrices de Covarianza basado en Vivado HLS

Autor: Rodrigo César Jiménez

Tutor: Ignacio Bravo Muñoz

2016

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Generador de Matrices de covarianza basado en Vivado HLS

Autor: Rodrigo César Jiménez

Tutor/es: Ignacio Bravo Muñoz

TRIBUNAL:

Presidente: Alfredo Gardel Vicente

Vocal 1º: Ignacio Fernández Lorenzo

Vocal 2º: Ignacio Bravo Muñoz

Calificación: .....

Fecha: .....

Índice general

1. Introducción

1.1. Resumen

En el presente proyecto se aborda el diseño e implementación en una FPGA de un generador de matrices de covarianza empleando Vivado HLS. El objetivo fundamental es llevar a cabo un estudio de dicha herramienta de diseño en cuanto a la programación de algoritmos complejos en FPGAs y evaluar qué ventajas presenta con respecto a la metodología tradicional de diseño hardware.

Palabras clave:

- PCA
- Covarianza
- HLS
- FPGA

1.2. Abstract

The Design and Implementation of a covariance matrix generator in a FPGA making us of Vivado HLS program ("*Vivado High Level Synthesis*") is addressed throughout this project. Its main purpose is to carry out an study of this design tool with the aim to evaluate the advantages and facilities that this tool provides in comparison with the traditional methodology of hardware design.

Key words:

- PCA
- HLS
- FPGA
- Covariance

1.3. Resumen extendido

Este proyecto surge de la necesidad de responder a las siguientes preguntas: ¿Qué pasaría si existiese una herramienta que permitiera realizar diseños hardware de un alto grado de complejidad empleando conocimientos básicos de programación C/C++? ¿Cuánto tiempo se ahorraría y cómo de válida sería esa solución?

Dar respuesta a estas preguntas es posible gracias a Vivado HLS[1]. Esta herramienta de Xilinx permite generar diseños hardware a partir de lenguaje de alto nivel (C/C++ y System C), para que sean directamente implementados sobre cualquier plataforma programable de Xilinx, es decir, permite un alto nivel de abstracción para el diseñador.

Para poder sacar conclusiones válidas con respecto al tiempo de diseño y a la validez del mismo, la mejor forma es compararlo con un diseño existente, el cual, en el caso de este proyecto inicialmente iba a ser la implementación en FPGA[2]del algoritmo PCA[3] realizada por el tutor del mismo, Ignacio Bravo Muñoz, pero dado el alto nivel de complejidad finalmente se ha desarrollado solo la parte correspondiente a la fase offline del mismo, es decir, el cálculo de la escena estática, que se compone del cálculo de la media y la obtención de la matriz de covarianza, lo cual se explicará más en profundidad en los siguientes apartados.

En los sucesivos capítulos de este proyecto, se realiza el estudio teórico que servirá de base para el desarrollo del mismo, así como un estudio detallado de Vivado HLS junto con un ejemplo para mostrar el flujo de diseño a seguir con este tipo de herramientas. También se muestran las distintas opciones hardware sobre las que se puede implementar nuestro sistema, una explicación detallada del diseño realizado y se dedica una sección para analizar el código empleado para ello.

[1] High Level Sintesis
[2] Field Programmable Gate Array
[3] Principal Component Analysis

2. Memoria

2.1. Fundamentos Teóricos

Introducción:

En este apartado se pretende dar al lector una visión general de un caso práctica de uso de un generador de matrices de covarianza. Éste se usa en múltiples algoritmos aplicado a procesado de señales o imágenes, por ejemplo. Así, a continuación, se muestra cómo dentro del algoritmo de componentes principales (PCA) esta parte se considera como crítica ya que es prácticamente el inicio del algoritmo.

El algoritmo PCA[4] se emplea fundamentalmente para reducir la dimensionalidad de un conjunto de datos, con el fin de ordenarlos según su importancia. Es utilizado en diferentes campos, como la economía, la electrónica de potencia, la ingeniería del software o la visión artificial. Dentro de éste último es donde se encontraría la aplicación que es objeto de estudio en este proyecto.

En el campo de la visión artificial uno de los grandes problemas o dificultades a la hora de realizar cualquier diseño es que los datos que se tienen que manejar son bastante grandes, lo cual supone un aumento del uso de memoria, y dado que es un recurso bastante limitado en dispositivos hardware programables, puede volverse una restricción crítica a la hora de realizar cualquier diseño. De ahí que el uso de este tipo de algoritmos sea fundamental para el desarrollo de aplicaciones.

En este caso, aplicar la técnica de PCA permitirá:

- Reducir la información redundante de los datos de entrada, para generar un conjunto de componentes principales que describan la información fundamental de una imagen.
- Determinar el grado de similitud entre dos o más imágenes analizando únicamente sus características fundamentales.

Esta última característica es el fundamento del proyecto del Dr. Bravo, ya que lo empleó para la detección de objetos en movimiento. De esta manera, *"se captan un conjunto de imágenes de una escena sin movimiento (escena estática) y de ellas se extrae sus*

[4] Principal Component Analysis

vectores característicos o componentes principales, se obtiene un conjunto de vectores que almacena la información fundamental de una escena. Si ahora se desea determinar si en esa misma escena existe un objeto nuevo, simplemente se debe captar una nueva imagen y comprobar si sus características fundamentales son análogas a las que describirían las componentes principales de la escena estática. En caso de no ser así será indicativo de que la escena ha cambiado (nuevo objeto en la escena).” [5]

Algoritmo PCA:

Partiendo de un conjunto de datos iniciales con media nula y alta correlación entre ellos [x_1, x_2,…, x_n] , se busca generar un nuevo grupo de variables [y_1, y_2,…, y_m] incorreladas entre sí, donde cada elemento y_j es una combinación lineal de los datos originales de forma que:

$$Y_1^T = [y_1, y_2,\ldots, y_m] = [u_{11}, u_{21},\ldots, u_{n1}] \cdot \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1j} \\ x_{21} & x_{22} & \cdots & x_{2j} \\ \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nj} \end{bmatrix} = u_1^T x \qquad \textbf{(2.1)}$$

Es importante tener en cuenta que los vectores u_j son ortogonales entre sí ($u_j^T \cdot u_i = 0$) y ortonormales ($u^T \cdot u_j = 1$).

Si llamamos a y_1, primera componente principal, ésta corresponde con la combinación lineal de las variables iniciales que presenta máxima varianza. Así que el objetivo es calcular los valores de u_1^T que obtienen dicha y_1. Para calcular la varianza de y_1 sujeta a la restricción de ortonormalidad anterior se emplea la siguiente expresión:

$$\text{Var}(y_1) = \frac{1}{n} y_1^T y_1 = \frac{1}{n} u_1^T \cdot x \cdot x^T = u_1^T C u_1 \qquad \textbf{(2.2)}$$

donde C es la matriz de covarianza del vector x, siendo su valor el mostrado a continuación:

$$C = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i \cdot x_i^T \qquad \textbf{(2.3)}$$

Donde la matriz de covarianza es una matriz cuadrada que contiene las varianzas y covarianzas asociadas a cada uno de nuestro conjunto de datos. Los elementos que componen la diagonal principal de la matriz contienen las varianzas entre cada conjunto de datos y los elementos que se encuentran fuera de la diagonal contienen las

[5] Fragmento extraído de la tesis doctoral de Ignacio Bravo Muñoz

covarianzas entre todos los pares posibles de elementos. Esta matriz es una matriz simétrica, dado que la covarianza entre cada par de elementos se muestra dos veces en la matriz, ya que, entre los elementos i-ésimo y j-ésimo se muestra en las posiciones (i, j) y (j, i).

Esta característica resultará muy útil a la hora de realizar nuestro diseño ya que dará pie a importantes optimizaciones del mismo.

Una condición muy importante que debemos aplicar es que la varianza de cada componente debe ser máxima y además, que todas las componente debe estar incorreladas entre sí. Así pues, para calcular las sucesivas componentes se debe aplicar que la suma de varianzas entre ellas debe ser máxima.

De esta forma, podemos decir que partimos de un conjunto de datos que presentan alta correlación entre ellos y que tienen media no nula ($[x_1, x_2, \ldots, x_n]$), para obtener un nuevo conjunto de datos cuyos elementos ($[y_1, y_2, \ldots, y_m]$) son incorrelados con media nula.

Aplicación al campo de la visión artificial:

En ese caso, la matriz de autovectores U contiene las componentes principales de una escena estática. Para obtener esta matriz, se deben de seguir los siguientes pasos:

- Captura de imágenes para construir el modelo de la escena estática: En este caso, el conjunto de datos inicial, son una serie de imágenes que describen una misma escena. Por lo tanto, se parte de un total de M imágenes $I_i \in \Re^{N \times N}$, donde i = 1, …, M. Dentro de PCA todas las imágenes son tratadas como vectores de dimensiones N^2 x 1.
- Cálculo de la media: Acorde a la descripción de PCA del apartado anterior, los datos iniciales deben poseer media nula. Por lo tanto, a continuación, se debe calcular el vector media de las M imágenes ($\boldsymbol{\Psi} \in \Re^{N^2 x 1}$) y restarlo a cada imagen, generando así el vector $\boldsymbol{\Phi}_i \in \Re^{N^2 x 1}$. Llamamos $A \in \Re^{N^2 x M}$ a la matriz que contiene en sus columnas cada uno de los vectores $\boldsymbol{\Phi}_i$ tal y como se muestra a continuación:

$$\boldsymbol{\Psi} = \frac{1}{M} \cdot \sum_{i=1}^{M} I_i = \begin{bmatrix} \boldsymbol{\Psi}_1 \\ \boldsymbol{\Psi}_2 \\ \ldots \\ \boldsymbol{\Psi}_{N^2} \end{bmatrix}_{N^2 x 1} = \frac{1}{M} \cdot \begin{bmatrix} I_{1,1} + I_{2,1} + \ldots + I_{M,1} \\ I_{1,1} + I_{2,1} + \ldots + I_{M,1} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ I_{1,N^2} + I_{2,N^2} + \ldots + I_{M,N^2} \end{bmatrix}_{N^2 x 1} \quad \textbf{(2.4)}$$

$$\mathbf{\Phi}_i = I_i - \mathbf{\Psi} = \begin{bmatrix} I_{1,1} + I_{2,1} + ... + I_{M,1} \\ I_{1,1} + I_{2,1} + ... + I_{M,1} \\ ..... \\ I_{1,N^2} + I_{2,N^2} + ... + I_{M,N^2} \end{bmatrix}_{N^2 x 1} = \begin{bmatrix} \mathbf{\Phi}_{i,1} \\ \mathbf{\Phi}_{i,2} \\ ... \\ \mathbf{\Phi}_{i,N^2} \end{bmatrix}_{N^2 x 1} ; i = 1,...,M \quad \textbf{(2.5)}$$

$$A = [\mathbf{\Phi}_1 ... \mathbf{\Phi}_M] = \begin{bmatrix} \mathbf{\Phi}_{1_{(1,1)}} \mathbf{\Phi}_{2_{(1,1)}} ... \mathbf{\Phi}_{M_{(1,1)}} \\ \mathbf{\Phi}_{1_{(2,1)}} \mathbf{\Phi}_{2_{(2,1)}} ... \mathbf{\Phi}_{M_{(2,1)}} \\ ..... \\ \mathbf{\Phi}_{1_{(N^2,1)}} \mathbf{\Phi}_{1_{(N^2,1)}} ... \mathbf{\Phi}_{M_{(N^2,1)}} \end{bmatrix}_{N^2 x M} \quad \textbf{(2.6)}$$

- Generación de la matriz de covarianza: El siguiente paso es la obtención de la matriz de covarianza ($C \in \Re^{N^2 x M}$). Aplicando la expresión de covarianza (2.3) se obtiene la siguiente expresión:

$$C = \frac{1}{M} \cdot A \cdot A^T \quad \textbf{(2.7)}$$

- Obtención de la matriz de autovectores: Una vez obtenida la matriz de covarianza, el siguiente paso es el cálculo de la matriz de los autovectores de C ($U \in \Re^{N^2 x M}$). Se calcularán de la siguiente manera:

$$C \cdot U = \lambda I \cdot U \quad \textbf{(2.8)}$$

donde $\lambda = \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \dots \\ \lambda_M \end{bmatrix}^T$ es el vector de autovalores.

Analizando la expresión (2.7), es posible introducir un cambio que a nivel matemático no supone un cambio sustancial, pero sus implicaciones a nivel hardware y computacional son enormes, ya que disminuye en gran medida el número de operaciones a realizar.

Debido a que la matriz A es de un tamaño N^2 x M, las dimensiones de la matriz C son N^2 x N^2. Cuando el tamaño de la imagen (N^2) es grande, esto provoca un número muy elevado de operaciones aritméticas. Por lo tanto, para poder reducir dicho número, se opta por la siguiente alternativa: Dado que los autovalores de ($A^T \cdot A$) son iguales a los de ($A \cdot A^T$), en primer lugar se calculan los autovectores de ($A^T \cdot A$):

$$A^T \cdot A \cdot V = \lambda I \cdot V \qquad \textbf{(2.9)}$$

donde $V \in \Re^{MxM}$ es la matriz de autovectores de $A^T \cdot A$.
Si en la expresión (2.9) se multiplica por A, se obtiene:

$$(A \cdot A^T) \cdot A \cdot V = \lambda I \cdot A \cdot V \qquad \textbf{(2.10)}$$

Por lo tanto, los autovectores de C se obtienen como:

$$U = A \cdot V \qquad \textbf{(2.11)}$$

Podemos concluir con que, si calculamos la matriz transpuesta de C, es decir, C_T, la carga operacional de nuestro sistema será bastante inferior dado que el tamaño de la misma es menor:

(2.12)

$$C_{N^2xN^2} = \frac{1}{M} \cdot A_{N^2xM} \cdot A^T{}_{MxN^2}$$

$$C^T{}_{MxM} = \frac{1}{M} \cdot A^T{}_{MxN^2} \cdot A_{N^2xM} \quad \textbf{(2.13)}$$

- Reducción de componentes principales: Una vez determinada la matriz de transformación U, el siguiente paso es determinar cuántos autovectores son necesarios. La matriz U, de tamaño N^2 x M posee M autovectores siendo el tamaño de cada uno de N^2. De los M autovectores se deben coger aquellos que estén asociados a los t autovalores de mayor peso. Para determinar t, en este caso, se emplea el método RMSE[6] o Error Cuadrático Residual Medio, eligiendo el porcentaje, P, de forma experimental.
  Durante el desarrollo de este proyecto, se abordan los aparatados referentes al cálculo de la media y de la matriz de covarianza transpuesta C_T.

2.2. Análisis Hardware

Introducción:

Otro aspecto importante para entender este proyecto es conocer el hardware para el que se va a diseñar. En este punto tenemos varias opciones, las cuales, se van a explicar a continuación:

FPGA:

Este dispositivo hardware ha sido el elegido para realizar la implementación de nuestro diseño, en primer lugar, porque fue el sistema empleado por el Dr. Bravo en su tesis y no tendría sentido realizar este proyecto con el fin de compararlos sin que la plataforma a implementarlo fuese distinta.

[6] Root Mean Square Error

Una FPGA no es más que una serie de áreas de silicio reprogramables, que al utilizar bloques de lógica pre-construidos, permiten implementar una amplia variedad de funcionalidades personalizadas sin tener en cuenta ninguna restricción, salvo las que imponen los límites constructivos de este tipo de dispositivos.

Otra importante característica a tener en cuenta es que este tipo de tecnología permite paralelizar el hardware de forma que se obtiene una gran potencia cómputo, además de conseguir tiempos de respuesta bastante rápidos, aunque menores que los ASICs.

Como inconvenientes, convendría destacar por encima del resto que este tipo de sistemas son fundamentalmente sistemas digitales, en los cuales la integración de un ADC/DAC es mucho más compleja que realizar este tipo de conversiones por software. Además, son sistemas que tienen un coste elevado, un alto consumo de energía.

En este caso se ha seleccionado una FPGA de la familia Virtex-7 de Xilinx. Esta familia de dispositivos está compuesta por tres familias nuevas que abordan una amplia gama de requisitos tales como la disminución del coste, aumento de la conectividad, de la lógica programable y de la capacidad de procesamiento de señales. Las tres familias que la componen son:

- Artix: diseñadas para reducir el consumo de potencia y del tamaño del dispositivo, logrando además una reducción en el coste del mismo.
- Kintex: esta familia incluye, además, mejoras de optimización y del rendimiento con respecto a los dispositivos de la anterior generación.
- Virtex: empleando la tecnología SSI[7], estos dispositivos consiguen un aumento de la capacidad y del rendimiento, convirtiéndose en la gama alta de la Serie 7 de Xilinx.

Maximum Capability	Artix-7 Family	Kintex-7 Family	Virtex-7 Family
Logic Cells	215K	478K	1,955K
Block RAM(1)	13 Mb	34 Mb	68 Mb
DSP Slices	740	1,920	3,600
Peak DSP Performance(2)	929 GMAC/s	2,845 GMAC/s	5,335 GMAC/s
Transceivers	16	32	96
Peak Transceiver Speed	6.6 Gb/s	12.5 Gb/s	28.05 Gb/s
Peak Serial Bandwidth (Full Duplex)	211 Gb/s	800 Gb/s	2,784 Gb/s
PCIe Interface	x4 Gen2	x8 Gen2	x8 Gen3
Memory Interface	1,066 Mb/s	1,866 Mb/s	1,866 Mb/s
I/O Pins	500	500	1,200
I/O Voltage	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V
Package Options	Low-Cost, Wire-Bond, Lidless Flip-Chip	Low-Cost, Lidless Flip-Chip and High-Performance Flip-Chip	Highest Performance Flip-Chip

Notes:
1. Additional memory available in the form of distributed RAM.
2. Peak DSP performance numbers are based on symmetrical filter implementation.

Figura 2.1. Características de las distintas familias de la Serie 7.

[7] Stacked Silicon Interconnect

Cabe destacar que el grado de integración de estos sistemas alcanza los 28 nm, lo cual permite el aumento de su capacidad y una importante reducción del consumo de potencia, lo cual, hoy en día, se está convirtiendo en un requerimiento básico a la hora de realizar cualquier diseño. A continuación, podemos observar la importante mejora que esto ha supuesto:

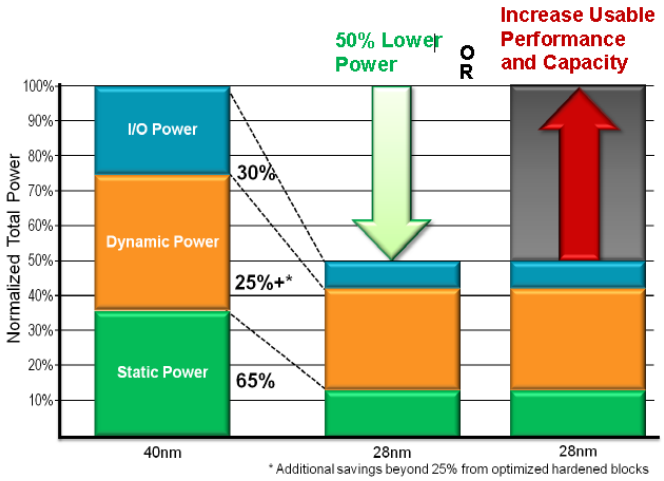


Figura 2.2. Relación de consumos en función del nivel de integración.

Dentro de la Virtex 7, se ha elegido el modelo xc7vx485tffg1927-1, la cual, como comprobaremos a continuación, podríamos decir que es una FPGA de gama media dentro de esta familia:

Virtex-7 FPGA Feature Summary

Device(1)	Logic Cells	Configurable Logic Blocks (CLBs)		DSP Slices(3)	Block RAM Blocks(4)			CMTs(5)	PCIe(6)	GTX	GTH	GTZ	XADC Blocks	Total I/O Banks(7)	Max User I/O(8)	SLRs(9)
		Slices(2)	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)									
XC7V585T	582,720	91,050	6,938	1,260	1,590	795	28,620	18	3	36	0	0	1	17	850	N/A
XC7V2000T	1,954,560	305,400	21,550	2,160	2,584	1,292	46,512	24	4	36	0	0	1	24	1,200	4
XC7VX330T	326,400	51,000	4,388	1,120	1,500	750	27,000	14	2	0	28	0	1	14	700	N/A
XC7VX415T	412,160	64,400	6,525	2,160	1,760	880	31,680	12	2	0	48	0	1	12	600	N/A
XC7VX485T	485,760	75,900	8,175	2,800	2,060	1,030	37,080	14	4	56	0	0	1	14	700	N/A
XC7VX550T	554,240	86,600	8,725	2,880	2,360	1,180	42,480	20	2	0	80	0	1	16	600	N/A
XC7VX690T	693,120	108,300	10,888	3,600	2,940	1,470	52,920	20	3	0	80	0	1	20	1,000	N/A
XC7VX980T	979,200	153,000	13,838	3,600	3,000	1,500	54,000	18	3	0	72	0	1	18	900	N/A
XC7VX1140T	1,139,200	178,000	17,700	3,360	3,760	1,880	67,680	24	4	0	96	0	1	22	1,100	4
XC7VH580T	580,480	90,700	8,850	1,680	1,880	940	33,840	12	2	0	48	8	1	12	600	2
XC7VH870T	876,160	136,900	13,275	2,520	2,820	1,410	50,760	18	3	0	72	16	1	6	300	3

Notes:
1. EasyPath™-7 FPGAs are also available to provide a fast, simple, and risk-free solution for cost reducing Virtex-7 T and Virtex-7 XT FPGA designs
2. Each 7 series FPGA slice contains four LUTs and eight flip-flops; only some slices can use their LUTs as distributed RAM or SRLs.
3. Each DSP slice contains a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator.
4. Block RAMs are fundamentally 36 Kb in size; each block can also be used as two independent 18 Kb blocks.
5. Each CMT contains one MMCM and one PLL.
6. Virtex-7 T FPGA Interface Blocks for PCI Express support up to x8 Gen 2. Virtex-7 XT and Virtex-7 HT Interface Blocks for PCI Express support up to x8 Gen 3, with the exception of the XC7VX485T device, which supports x8 Gen 2.
7. Does not include configuration Bank 0.
8. This number does not include GTX, GTH, or GTZ transceivers.
9. Super logic regions (SLRs) are the constituent parts of FPGAs that use SSI technology. Virtex-7 HT devices use SSI technology to connect SLRs with 28.05 Gb/s transceivers.

Figura 2.3. Especificaciones de los distintos dispositivos de la familia Virtex 7.

System on Chip:

Esta familia de dispositivos está basada en la arquitectura “All programable SoC”. Estos sistemas son los que se están colocando en la vanguardia dentro de este ámbito. Por un lado, llevan integrado un chip de procesamiento basado en el Cortex-A9 de ARM y por otro, lógica programable, con un nivel de integración de 28 nm.

La implicación de ello, es que se ha conseguido juntar en un mismo dispositivo la potencia computacional de un microprocesador, junto con la eficiencia en consumo, la versatilidad y el aumento de velocidad de respuesta que obtenemos gracias a la lógica programable.

Además, no es necesario que la unidad de procesamiento y la lógica programable estén funcionando simultáneamente y de forma continuada, sino que, para aquellas aplicaciones donde la parte software controla la acción sobre la lógica programable, ésta puede activarse o desactivarse a voluntad del usuario, consiguiendo un importante ahorro energético que era inconcebible en el sistema FPGA clásico.

Este tipo de sistemas también contienen puertos de I/O, de comunicación e incluso conversores tipo ADC/DAC.

Para concluir, cabe decir que este tipo de dispositivos representan el futuro, al menos el futuro inmediato, dado que integran la mayoría de las ventajas de los microprocesadores y de las FPGAs.

2.3. Vivado HLS

Introducción

Esta herramienta surge de la necesidad de aumentar la productividad de un diseñador hardware, permitiendo disminuir el tiempo de diseño, especialmente, cuando se trata de sistemas de alta complejidad.

Tradicionalmente se trabajaba a bajo nivel, empleando lenguajes de descripción hardware como el VHDL o el Verilog. Sin embargo, en Vivado HLS, se trabaja empleando lenguajes de alto nivel como C/C++ o System C, lo cual, permite un mayor nivel de abstracción y ofrece la posibilidad de realizar diseños mucho más flexibles. Además, esta herramienta posee una interfaz gráfica que resulta bastante intuitiva para el usuario, de forma que se puedan realizar modificaciones que a nivel hardware suponen cambios fundamentales en la arquitectura del sistema, pero que sin embargo a nivel software,

es tan inmediato como modificar una opción de configuración en una de sus pestañas, como por ejemplo realizar un diseño para una tarjeta específica, pero que con el tiempo por requerimientos técnicos, se necesite que ese diseño sea portable a otras tarjetas. Esto último es posible gracias a la herramienta, ya que basta con cambiar la tarjeta "destino" en el panel de configuración del proyecto y volver a implementarlo.

Éstas últimas son, probablemente, algunas de las características más importantes de esta herramienta, ya que permiten realizar diseños parametrizables de forma que podemos modificar por ejemplo, el tamaño de una variable, el número de decimales a tener en cuenta o el número de puertos que tiene un bloque hardware, simplemente con realizar unos pequeños cambios que a nivel de programación de alto nivel no suponen una excesiva dificultad mientras que si pretendiésemos hacer esto mismo en lenguaje de descripción hardware puede ser una tarea muy complicada y sobretodo, nos llevaría una importante cantidad de tiempo.

Otro aspecto a remarcar es que, entre sus múltiples opciones, esta herramienta permite introducir una serie de directivas de optimización, que se explicarán a continuación, gracias a las cuales, se pueden conseguir distintos diseños a partir de una misma aplicación software, de forma que adecuemos nuestro diseño a las distintas especificaciones o necesidades que fijemos, tales como latencia, paralelismo, recursos consumidos, … Además, estas directivas sirven, entre otras cosas, para fijar la interfaz de comunicación entre los puertos de I/O del periférico que hayamos generado con esta herramienta, y el resto de elementos pertenecientes del sistema.

Una característica importante de esta herramienta, es que facilita inmensamente la portabilidad entre dispositivos, de forma que si inicialmente, el diseño realizado está pensado para un tipo de FPGA, pero queremos migrarlo hacia otra con distintas características, basta con cambiar una opción de configuración de nuestro proyecto de Vivado HLS y volver a exportar el diseño.

Por contra, cabe destacar que Vivado HLS sólo puede implementar diseños para tarjetas que estén soportadas en Vivado, actualmente, cualquier diseño para familias más antiguas que la Serie 7, es decir, permitirá realizar la verificación del diseño, pero a la hora de exportarlo, no te da la opción de añadirlo al catálogo de IPs de Vivado, lo cual, tal y como se explicará en secciones posteriores, nos permite incluir el módulo que hemos generado como un periférico más que forma parte de un sistema más grande, permitiendo así realizar diseños mixtos empleando VHDL y C/C++ alternativamente. Además, el problema de que todo el proceso de optimización sea transparente para el usuario es que no se tiene control sobre él, lo que puede llevar a sistemas que consumen más recursos hardware de los necesarios.

Síntesis de Alto Nivel

- La síntesis que realiza Vivado HLS por defecto se puede resumir de la siguiente forma: Sintetiza los argumentos de la función principal (top) como puertos I/O RTL[8].
- Sintetiza las funciones en C como bloques que componen la jerarquía RTL. Si dentro de nuestra función principal hubiera sub-funciones, se sintetizaría siguiendo dicha jerarquía.
- La síntesis de todos los lazos se realiza de forma que se implementa el hardware necesario para una iteración y el diseño RTL ejecuta esta lógica en cada secuencia del lazo.
- Los arrays se sintetizan como bloques de memoria RAM en el diseño final sobre FPGA.

Etapas de la Síntesis de Alto Nivel

La síntesis de alto nivel se realiza en los siguientes pasos, los cuales son absolutamente transparentes para el usuario:

- Extracción del control y del flujo de datos: durante esta etapa, se genera la máquina de estados interna responsable de la sincronización de todo el sistema. Se obtiene a partir de, por un lado, el comportamiento del sistema, que viene determinado por los bucles y el resto de sentencias condicionales incluidas en el diseño. Cada vez que haya movimiento de datos debido a que se cumple o no, alguna condición en cualquiera de estos tipos de sentencias, se interpreta como entrar o no en cada uno de los estados de nuestra FSM[9] interna. En cuanto al flujo de datos, inicialmente suponemos que para realizar cada operación vamos a necesitar únicamente un flanco de reloj, cosa que en la práctica no siempre es así, lo cual repercutirá sobre la complejidad de la máquina de estados dado que los protocolos de comunicación son los que sufren esos retardos.
- Esquematizado y unión: en el proceso de esquematizado, se genera la implementación más óptima, basándose para ello, en el propio comportamiento del sistema, sus limitaciones, y en las directivas de optimización que el usuario haya introducido. Durante el proceso de unión, se selecciona los recursos hardware necesarios para implementar nuestro sistema.

[8] Register Transfer Level
[9] Máquina de Estados.

Por último, el usuario debe seleccionar el tipo de datos que va a utilizar, así como las limitaciones (constraints) que podrían afectar al valor de la latencia, la frecuencia de reloj, … y las directivas de optimización. Vivado HLS cuenta con librerías para el manejo de datos enteros, flotantes y en coma fija.

Visión general del diseño en Vivado HLS

En la siguiente imagen se puede ver de forma esquemática el modelo de un diseño en HLS:

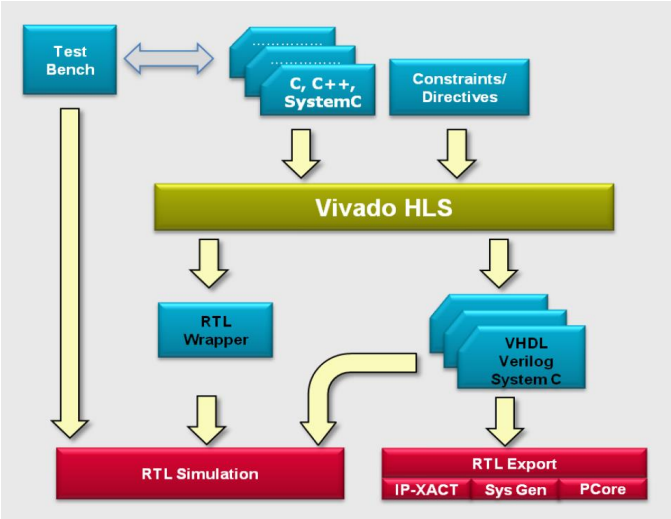


Figura 2.4. Vista general de la arquitectura de un diseño en HLS

Donde los elementos fundamentales son los siguientes:

- Archivos de diseño: son archivos codificados en C/C++. En caso de que añadamos también un archivo de test bench, éste será reutilizado durante la fase RTL de forma que mejoremos la productividad del diseño dado que no existe la necesidad de crear un test bench nuevo a la hora de hacer la verificación RTL. Este test bench es el que hace las veces de función main del sistema, y desde él realizaremos la llamada a las distintas funciones que lo conforman que se encuentran en el archivo .cpp. También es habitual generar un archivo .h con las definiciones de la función principal, así como de todas las constantes que se vayan a emplear en el programa, lo cual nos permite generar un sistema altamente parametrizable tal y como se muestra en secciones posteriores.

- Librerías: vienen integradas dentro de nuestra herramienta, de forma que hacen posible la compilación de nuestro sistema, además de contener los modelos de tiempos y de recursos de todos los dispositivos Xilinx soportados.

- Directivas y restricciones: se especifican a través de la interfaz de usuario (GUI[10]) de HLS o empleando comandos Tcl. Se emplean para que nuestro diseño cumpla con las especificaciones que le marquemos.

- Salida RTL: son los archivos que se generan una vez la etapa de sintetización ha finalizado satisfactoriamente. HLS soporta los siguientes lenguajes de descripción hardware: VHDL, Verilog y System C. Este diseño RTL se puede exportar al Catálogo de IPs (opción por defecto), como un System Generator para DSP[11] o como Pcore para EDK[12]..
- Co-Simulación RTL: una vez se realiza satisfactoriamente la etapa de síntesis, HLS genera una serie de archivos necesarios para verificar el correcto funcionamiento del sistema RTL generado. Vivado HLS permite generar la co-simulación para cualquiera de estos simuladores: Vivado Simulator (XSim), QuestaSim simulator, VCS, ISE Simulator (ISim), Riviera y OSCI System C. Por lo tanto, para llevarla a cabo se emplean los archivos comentados anteriormente y alguno de estos simuladores.

Una vez finalizada la etapa de síntesis, Vivado HLS genera un reporte con información referente al diseño implementado:

- Información general: en este apartado aparecen detalles referentes a la fecha, la versión, el nombre del proyecto, el nombre de la solución implementada y la tarjeta empleada para ello.
- Temporización: en ella se muestran los valores referentes al ciclo de reloj requerido, el estimado después de sintetizarlo y una estimación del error que comete.
- Latencia: nos dice el número de ciclos de reloj que pasan desde que nuestro sistema recibe una entrada, hasta que sitúa el valor correspondiente en la salida. En este apartado también se muestra el número de ciclos de reloj necesario para que nuestro diseño pueda recibir datos de entrada.
- Recursos consumidos: muestra el número de recursos hardware empleados por nuestro diseño, además de ofrecernos un porcentaje del grado de utilización de los mismos. Los recursos que se especifican son los bloques RAM, los multiplicadores (DSP48), los Look-Up Tables (LUT) y los registros.

[10] Graphical User Interface
[11] Digital Signal Processing
[12] Embedded Development Kit

- Interfaz: nos ofrece un resumen de los puertos de I/O sintetizados, especificando el número de bits, el protocolo empleado y si son de entrada o de salida fundamentalmente.

Metodología de trabajo en Vivado HLS

Podemos resumir el proceso para realizar un diseño en Vivado HLS de la siguiente manera:

- Compilar nuestro código en alto nivel y realizar una simulación software del mismo.
- Sintetizar nuestro diseño.
- Analizar el reporte para comprobar que nuestro diseño cumpla las especificaciones deseadas.
- En caso contrario, debemos aplicar directivas de optimización para mejorar el sistema.
- Realizar la Co-Simulación que es básicamente una simulación funcional a nivel hardware. Si el resultado de la misma es satisfactorio podemos concluir con que nuestro sistema funciona correctamente. Esta simulación resulta muy útil de cara a incluir nuestro diseño en un sistema más grande, ya que podemos observar e interpretar la secuencia de las señales de entrada que generar la salida que queremos, ya que no siempre es algo inmediato.
- Por último, se exporta nuestro bloque RTL para emplearlo en alguna de las distintas plataformas antes mencionadas.

Directivas de optimización:

En este apartado se va a explicar más en detalle cada una de las directivas que Vivado HLS nos ofrece, así como los efectos que éstas producen en un diseño sencillo:

Las siguientes sentencias se pueden aplicar tanto a funciones como a variables globales, dependiendo del efecto que el usuario desee conseguir y para aplicarlas, podemos hacerlo mediante la GUI o empleando comandos pragma situados en el mismo código fuente:

Allocation:

Define, y puede llegar a limitar, el número de instancias RTL empleadas para implementar funciones u operaciones específicas. Esta instancia puede ser útil, por ejemplo, en caso de que dentro de nuestro diseño llamemos varias veces a la misma función, de forma que no sería necesario implementar todos esos bloques que compondrían la función a la que estamos llamando, sino que con que se haga una vez y se emplee varias veces es suficiente y mucho más óptimo a nivel de recursos hardware consumidos. La sentencia pragma asociada es la siguiente:

#pragma HLS allocation instances = <Instance Name List> limit = <Integer Value> <operation,function>

Array_map:

Permite mapear un array pequeño en uno más grande. El uso más habitual de esta sentencia es emplearlo de forma sucesiva sobre varios arrays con el fin de condensar todos ellos en un array mayor que luego pueda ser dirigido hacia módulos de memoria del tipo RAM o FIFO. Mediante las opciones que ofrece la propia directiva se puede mapear de forma horizontal, es decir, aumentar la longitud del array, o de forma vertical, es decir, aumentando el ancho de palabra de cada elemento. La sentencia pragma asociada es la siguiente:

#pragma HLS array_map variable = <variable> instance = <instance> <horizontal, vertical> offset <int>.

EL valor de offset se emplea para dejar posiciones libres entre cada array en caso de que el usuario así lo desee.

Array_partition:

Es la directiva complementaria a la anterior, de forma que reduce un array en arrays más pequeños, e incluso, en elementos individuales. Se aplica fundamentalmente para emplear pequeñas unidades de memoria y múltiples registros en lugar de emplear una memoria muy grande, lo cual mejora el throughput del sistema, pero a costa de emplear un mayor número de bloques de memoria y de registros. La sentencia pragma asociada es la siguiente:

#pragma HLS array_partition variable= <variable> <block, cyclic, complete> factor= <int> dim= <int>.

Donde dim expresa la dimensión del array, lo cual es útil si deseamos arrays multidimensionales, factor expresa el número de divisiones que vamos a aplicar al array grande y <block, cyclic y complete> se emplean para determinar la forma en que se va a dividir ese array.

Array_reshape:

Este comando se emplea para combinar varias divisiones de un array, realizando un mapeo vertical de forma que generemos un array nuevo con pocos elementos donde cada uno de ellos tiene un ancho de palabra mayor, lo cual es muy útil para incrementar el ancho de banda de nuestro sistema, además de disminuir ligeramente la latencia y el uso de LUT. La sentencia pragma asociada es la siguiente:

#pragma HLS array_reshape variable= <variable> <block, ciclyc, complete> factor= <int> dim <int>.

Clock:

En diseños basados en C/C++, esta directiva sirve para que a la hora de realizar la implementación, se intente forzar al sistema para llegar al valor de frecuencia de reloj que le hayamos fijado, esto puede ser en detrimento de los recursos consumidos fundamentalmente. En el caso de los diseños basados en SystemC, dado que puede haber más de una fuente de reloj, esta directiva es útil para poder seleccionar la frecuencia de reloj de cada uno de esos relojes adicionales, además de poder intentar modificar el del sistema como en el caso anterior. La sentencia pragma asociada es la siguiente:

#pragma HLS clock domain= <string>.

Dataflow:

Esta directiva especifica la optimización del flujo de datos que se realiza durante la etapa de síntesis (ya explicada anteriormente). Generalmente, esta optimización trata de minimizar la latencia, mejorando la concurrencia de datos. Sin embargo, la existencia de dependencias entre los datos puede limitar dicha optimización, lo cual, es habitual en diseños donde empleemos bucles ya que no podemos continuar el flujo del programa

sin que se haya realizado el bucle completo. Esta directiva permite hacer que la dependencia entre datos no sea tan crítica de forma que las funciones y los bucles puedan operar en "paralelo" siempre y cuando no se necesite que todas las operaciones de un bucle se hayan finalizado. Como consecuencia de ello se reduce la latencia del sistema, además de mejorar el throughput. La sentencia pragma asociada es la siguiente:

#pragma HLS dataflow.

Datapack:

Esta directiva se emplea para unir en un solo número, todos los campos correspondientes a una estructura, aumentando el ancho de palabra, lo cual, optimiza la implementación. La sentencia pragma asociada es la siguiente:

#pragma HLS data_pack variable=<variable> instance= <string> struc_level

Dependence:

Vivado HLS detecta automáticamente el grado de dependencia de datos dentro de bucles o entre las diferentes iteracciones de un bucle, lo cual, como hemos visto, es crítico a la hora de realizar la síntesis. Esta directiva nos es de utilidad desde el punto de vista de encontrar zonas de nuestro código donde haya una fuerte dependencia de datos para intentar optimizarla y mejorar nuestro diseño. La sentencia pragma asociada es la siguiente:

#pragma HLS dependence variable= <variable> <array, pointer> <inter, intra> <RAW, WAR, WAW> distance= <int> <false,true>.

Donde <array, pointer> se emplea para especificar el tipo de variable que es, <inter, intra> para especificar si hay que analizar la dependencia entre los elementos de un bucle o entre las distintas funciones y bucles de nuestro código, <RAW, WAR, WAW> se especifica la dirección de la dependencia: RAW (Read-After-Write: la instrucción escrita emplea el valor usado por un instrucción leída), WAR (Write-After-Read: la instrucción leída obtiene el valor que es sobrescrito por una variable de escritura) y WAW (Write-After-Write: dos instrucciones de escritura escriben en la misma zona de memoria pero en distinto orden), <true, false> se emplea para especificar la acción que se debe hacer sobre esa dependencia, forzarla (true) o eliminarla (false).

Expression_balance:

En lenguaje C/C++ se puede dar el caso en que debamos realizar varias operaciones en una misma línea de código, lo cual puede acabar generando una larga cadena de operaciones en RTL, para evitarlo, esta directiva optimiza esas operaciones aplicando las propiedades asociativa y conmutativa para disminuir la latencia y el sobrecoste de recursos hardware. También se permite realizar lo contrario, de forma que, si no deseamos que a la hora de realizar la síntesis RTL se optimice alguna expresión matemática, podemos expresárselo mediante esta directiva. La sentencia pragma asociada es la siguiente:

#pragma HLS expression_balance <off>.

Donde en <off> podemos especificar si deseamos que se realice la optimización (on) o si por el contrario deseamos que no se realice (off).

Function_Instantiate:

Por defecto, las funciones en RTL se sintetizan como bloques de jerarquía separados. Todas las instancias de una función, del mismo nivel de jerarquía, se implementarán como el mismo bloque RTL. Sin embargo, gracias a esta directiva, podemos generar un bloque RTL por cada instancia, permitiendo una optimización individualizada. La sentencia pragma asociada es:

#pragma HLS function_instantiate variable= <variable>.

Inline:

Esta directiva es nuevamente, la complementaria a la anterior, mientras que la directiva function_instantiate se emplea para separar instancias de un mismo nivel de jerarquía, ésta se emplea para unirlas, de forma que, en ciertos casos, al estar dos instancias en el mismo nivel jerárquico facilite las operaciones entre ellas (en caso de que las haya) y por lo tanto se logre un funcionamiento más óptimo del sistema. En ciertos casos, emplear esta directiva puede suponer un aumento de los recursos hardware consumidos. La sentencia pragma asociada es:

#pragma HLS inline region recursive off.

Donde región se emplea para seleccionar las funciones a las que se les va a aplicar la directiva, recursive, su valor por defecto indica que sólo a un nivel jerárquico se le va a aplicar la directiva, si lo modificamos, se le aplicará la directiva al resto de niveles jerárquicos inferiores, off se emplea para impedir que se aplique a funciones que no queremos.

Interface:

Este comando se emplea para especificar cómo se deben crear los puertos RTL durante la etapa de la síntesis de la interfaz. En caso de que se empleen variables globales, pero las operaciones de lectura y escritura sólo se realicen a nivel local, no será necesario que se genere un puerto I/O en la RTL. Las distintas interfaces que podemos generar para los distintos niveles funcionales son las siguientes:

- ap_ctrl_none: no se generará ninguna interfaz de control para nuestro diseño. Esta alternativa es recomendable para sistemas con un flujo de datos continuo.

- ap_ctrl_hs: éste es la forma de generar una interfaz de handshaking. Es el modo por defecto, y se emplean generalmente en diseños secuenciales donde un control simple es suficiente. Los puertos de la interfaz que se generan en este caso son:
    - ap_start (entrada): esta señal se debe de poner a 1 para que nuestro sistema comience a funcionar.
    - ap_ready (salida): esta señal indica cuándo está listo nuestro diseño para recibir nuevas entradas.
    - ap_done (salida): esta señal se pone a nivel alto cuando nuestro sistema ha realizado todas las operaciones pertinentes para la entrada introducida.
    - ap_idle (salida): esta señal se pondrá a nivel alto cuando nuestro diseño esté activado pero, sin embargo, no tiene ninguna entrada válida, lo que significa que no está “trabajando”. A continuación, se muestra una imagen extraída del manual de usuario de Vivado HLS correspondiente a un sumador con interfaz handshaking.

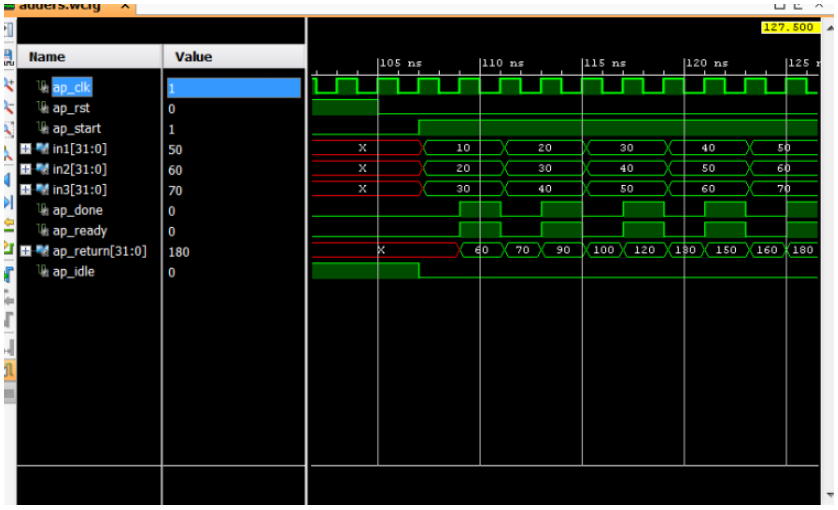


Figura 2.5. Protocolo handshaking

Como se puede observar en la imagen anterior, el sistema comienza a funcionar una vez se pone a nivel alto la señal ap_start, evento que coincide con la puesta nivel bajo de la señal ap_idle, lo cual tiene mucho sentido, ya que el paso a nivel alto de ap_start significa el comienzo del funcionamiento del bloque sumador. Hay que destacar también que la señal ap_ready se pone a nivel alto para indicar que el diseño está preparado para recibir nuevas entradas en el siguiente flanco de reloj. Además, la señal ap_done sigue un comportamiento similar, ya que mientras esté a nivel alto, está indicando que el valor de la salida es un dato válido. En el momento en que estas dos señales pasan a nivel bajo, el valor de la entrada cambia para volver a comenzar una nueva iteración del ciclo.

- ap_ctrl_chain: este protocolo es muy parecido al anterior, con la diferencia de que, en éste, se añade un puerto adicional, ap_continue, el cual debe estar a nivel alto cuando ap_done se pone a 1 para que nuestro diseño pueda seguir cogiendo el valor de las entradas y comenzar así un nuevo ciclo.

Para los argumentos de la función principal y variables globales, para cada tipo de argumento, se implementarán las siguientes interfaces, que se explicarán a continuación:

- Sólo lectura (ap_none)
- Sólo escritura (ap_vld)
- Lectura/Escritura (ap_ovld)
- Arrays (ap_memory)
- Arrays (bram)

Las distintas interfaces son:

- ap_none: no se genera ninguna interfaz. Se corresponde con un simple cable.
- ap_stable: sólo se puede aplicar a puertos de entrada, se emplea para informar a la herramienta de que el valor de entrada de este puerto es constante, por lo tanto se implementará una interfaz muy similar a la de ap_none, pero permitiendo que se realicen optimizaciones internas durante la fase de implementación.
- ap_vld: se genera un puerto de validación adicional (<port_name> _vld) que se emplea para validar el dato de entrada correspondiente al puerto donde se aplique esta directiva.
- ap_ack: se genera un puerto de "acknowledge" adicional (<port_name>_ack) de forma que cuando se ponga a 1, nos dé información adicional sobre los procesos internos que se están llevando a cabo (lectura, escritura).
- ap_hs: esta opción genera una combinación de ambas, de forma que esta directiva genera dos puertos adicionales, uno de validación y otro de "acknowledge".
- ap_ovld: para entradas, esta directiva genera una interfaz ap_none, para salidas, una interfaz ap_vld y para puertos de entrada/salida, implementa ambas.
- ap_memory: este modo implementa los argumentos y accesos para un RAM de datos externa. Se generan los puertos de datos, direcciones y de control de la RAM para leer desde/escribir en la RAM externa. Los tipos y tamaños específicos de dichos puertos dependerán de la memoria RAM a la que queramos acceder.
- bram: esta directiva es igual que la anterior, con la salvedad de que, en vez de ser para una RAM externa, es para una BRAM externa.
- ap_fifo: implementa un array, un puntero y los pasa como puertos de referencia como en el acceso para una FIFO. El puerto de datos de entrada, hará que se ponga a 1 el puerto asociado de salida de lectura asociado (port_name _read) cuando esté listo para leer nuevos valores provenientes de la FIFO externa y detendrá el desarrollo de las funciones hasta que la entrada correspondiente (port_name _empty_n) se ponga a nivel alto, indicando que hay un dato disponible para ser leído. Para el caso de la escritura el proceso es análogo.
- ap_bus: esta interfaz es similar a la anterior, con la diferencia de que ésta genera los puertos pertinentes para la lectura/escritura sobre un bus de datos.
- axis: genera una interfaz de comunicación AXI-Stream, que genera automáticamente los puertos TVALID y TREADYpara realizar el "handshake". Además, se puede especificar si esta interfaz debe ser implementada para un "esclavo" (s_axilite) o para un "maestro" (m_axi).

La sentencia pragma asociada es:

#pragma HLS interface <mode> register port= <string>.

Donde en el mode se debe de seleccionar alguno de los citados anteriormente y en string el nombre del elemento al que le vamos a aplicar la directiva.

Latency:

Permite al usuario seleccionar un margen para las latencias del sistema de cara a la implementación.Se pueden seleccionar tanto para la función principal, como para sub-funciones dentro del programa principal, así como para distintos fragmentos dentro de una función, como en bucles for, while, etc. Durante esta etapa, Vivado HLS intenta reducir la latencia todo lo posible. De los dos márgenes, el más restrictivo es el de la latencia máxima, la cual, en caso de ser superada por nuestro sistema, provocaría que durante la implementación se consuman muchos más recursos hardware para intentar llegar a dicha cifra. La sentencia pragma asociada es:

#pragma HLS latency min=<int> max= <int>.

Loop_flatten:

Esta directiva se emplea para unir varios bucles en uno solo, de forma que sean necesarios menos ciclos de reloj durante la RTL, con el fin de paralelizar dichos bucles en la medida de lo posible. Se diferencian dos tipos de bucles:

- Bucle perfecto: la lógica de cada uno de los bucles es distinta y no se relacionan entre ellas. Los bucles también tienen el mismo número de iteracciones.
- Bucle semi-perfecto: es igual que el anterior, salvo porque el bucle de mayor jerarquía tiene un número de iteracciones variable.

La sentencia pragma asociada es:

#pragma HLS loop_flatten off.

Loop_merge:

Se emplea para optimizar de forma conjunta la lógica hardware de varios bucles consecutivos y así, disminuir la latencia. Para poder aplicar esta directiva se deben de cumplir ciertos requisitos:

- El número de iteracciones de los bucles en cuestión debe ser el mismo, en caso de que dicho número lo marque una variable.
- Si el número de iteracciones lo marca una constante, esa contante debe ser igual para todos los bucles.
- No están permitidas sentencias que hagan que el valor que tome una variable se sobrescriba a sí misma (a = a +1)

La sentencia pragma asociada es:

#pragma HLS loop_merge force.

Loop_occurrence:

La función de esta directiva es conseguir una mejor optimización a la hora de aplicar pipeline sobre algún bucle o función. La empleamos para especificar zonas de código cuya ejecución está sujeta a condicionales dado que este tipo de zonas se ejecutan muchas menos veces que otras que no lo están. En estos casos, el intervalo a aplicar durante el pipeline será mayor (más lento) consiguiendo que la prioridad del pipeline sea acelerar las zonas de código que más veces se repiten para finalmente, obtener un diseño mucho más óptimo. La sentencia pragma asociada es:

#pragma HLS occurrence cycle= <int>.

Loop_tripcount:

Llamamos “tripcount” al número total de interacciones realizadas por un lazo. Vivado HLS da el reporte de la latencia total del sistema, así como, la individual de cada lazo. Pero puede haber situaciones en las que este dato no aparece en el reporte de la síntesis, lo cual, puede ser debido a que dicho bucle tenga un número variable de iteracciones, o dicho número dependa del valor que tomen las propias variables internas del bucle. Esta directiva se emplea para que a pesar de que la latencia no aparezca en el reporte, podamos conocer esa información, de forma, que fijemos los valores máximo, mínimo y medio de iteracciones para que Vivado HLS pueda darnos la información de la latencia específica del bucle. La sentencia pragma asociada es:

#pragma HLS loop_tripcount min=<int> max=<int> avg=<int>.

Pipeline:

Esta directiva se emplea para especificar detalles a la hora de realizar este tipo de optimización. Permite seleccionar la función/bucle específico al que aplicársela. Generalmente, realizar un pipeline sobre una región de código implica “desenrollar” todos los bucles haya, es decir, al separar en el flujo de la ejecución del programa las distintas operaciones, permites optimizar el diseño dado que se paraleliza cada proceso en la medida de lo posible, es decir, disminuimos la latencia a costa de aumentar los recursos hardware consumidos. La sentencia pragma asociada es:

#pragma HLS pipeline ||= <int> enable_flush rewind

Protocol:

Se emplea para indicar zonas de código donde no se aplicarán operaciones de reloj del tipo de ap_wait() (C/C++), o wait() (SystemC). La sentencia pragma asociada es:

#pragma HLS protocol <floating, fixed>.

Resource:

Esta directiva, se emplea para especificar una librería de cores y emplearlos durante la implementación RTL de una variable (array, operaciones aritméticas, operaciones y argumentos de funciones). Generalmente se emplea para seleccionar qué tipo de memoria RAM se va a emplear. La sentencia pragma asociada es:

#pragma HLS resource variable = <variable> core= <core>.

Stream:

Esta directiva afecta sobre los arrays involucrados en la optimización del flujo de datos ya que, generalmente, cuando se pasan arrays por argumento, se implementan como RAMs interconectadas. Si empleamos el comando “config_dataflow” podemos convertir todos esos bloques RAM en FIFOs. Sin embargo, empleando esta directiva podemos decidir individualmente qué arrays se implementarán como RAMs y qué arrays se implementarán como FIFOs. La sentencia pragma asociada es:

#pragma HLS stream variable= <variable> off depth= <int>.

Donde depth se emplea para indicar el tamaño de la FIFO.

Top:

Esta directiva asocia un nombre a una función, de forma que ésta pueda ser seleccionada como función principal del sistema. En Vivado HLS la forma más sencilla de hacerlo es mediante la GUI al crear un proyecto nuevo. La sentencia pragma asociada es:

#pragma HLS top name= <string>

Unroll:

Esta directiva se emplea para desenrollar un bucle con el fin de paralelizarlo, y así disminuir la latencia a costa de un mayor uso de recursos hardware. Se emplea implícitamente en la directiva pipeline. La sentencia pragma asociada es:

#pragma HLS unroll skip_exit_check factor= <int> región.

Donde skip_exit_check se emplea cuando aplicamos unroll de forma parcial dentro de un bucle. En función de lo que se ponga, se tendrán en cuenta sentencias condicionales del bucle (variable bounds) o no (fixed bounds).

Ejemplo directivas de optimización:

Para poder entender cómo aplicar las directivas de optimización y los efectos que producen vamos a realizar un ejemplo sencillo que proporciona Xilinx sobre un multiplicador de matrices. Los archivos fuente se encuentran en la página web de Xilinx, en el siguiente enlace:
https://secure.xilinx.com/webreg/login.do?oamProtectedResource=wh%3Dwww.xilinx.com%20wu%3D%2Fwebreg%2Fclickthrough.do%3Fcid%3D356028%26license%3DRefDesLicense%26filename%3Dug871-vivado-high-level-sythesis-tutorial.zip%20wo%3D1%20rh%3Dhttps%3A%2F%2Fsecure.xilinx.com%20ru%3D%252Fwebreg%252Fclickthrough.do%20rq%3Dcid%253D356028%2526license%253DRefDesLicense%2526filename%253Dug871-vivado-high-level-sythesis-tutorial.zip

Realizaremos todos los pasos necesarios para generar nuestro diseño RTL final:

En primer lugar, iniciamos Vivado HLS, para ello, hacemos doble clic sobre el acceso directo que se debe haber creado durante el proceso de instalación, o bien, hacemos doble clic en: Inicio -> Todos los programas -> Xilinx Design Tools -> Vivado HLS 2014.4. Se abrirá la siguiente ventana:

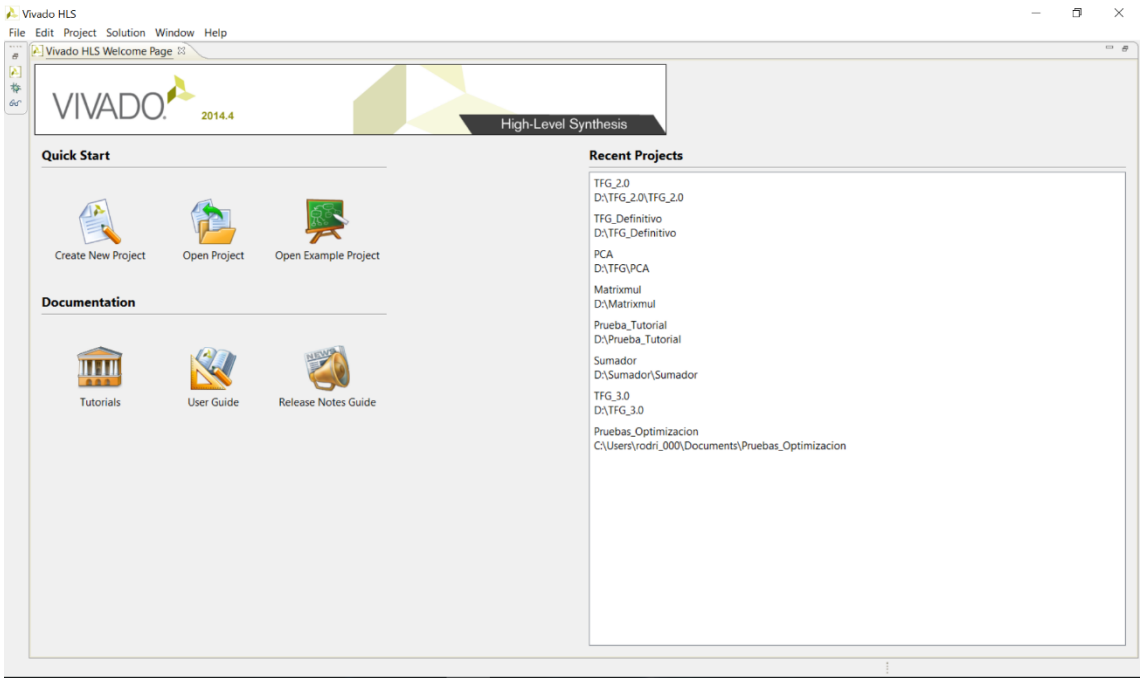


Figura 2.6. Pantalla de inicio de Vivado HLS

A continuación, hay que hacer clic en Create New Project y se nos abrirá la siguiente ventana:

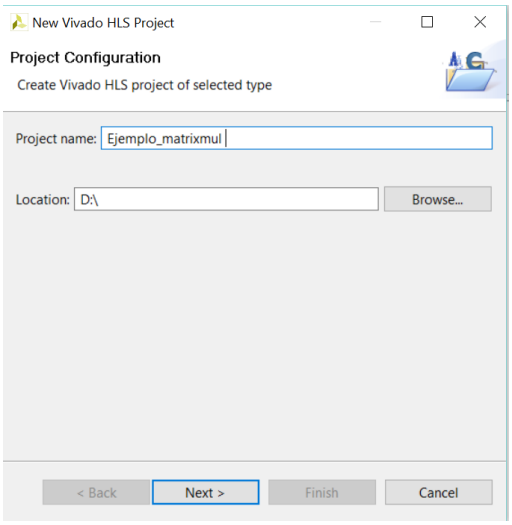


Figura 2.7. Generar un nuevo proyecto en Vivado HLS

Debemos introducir el nombre del proyecto que queramos y seleccionar una ruta donde depositarlo. Hacemos clic en Next. Y nos aparecerá la siguiente ventana:

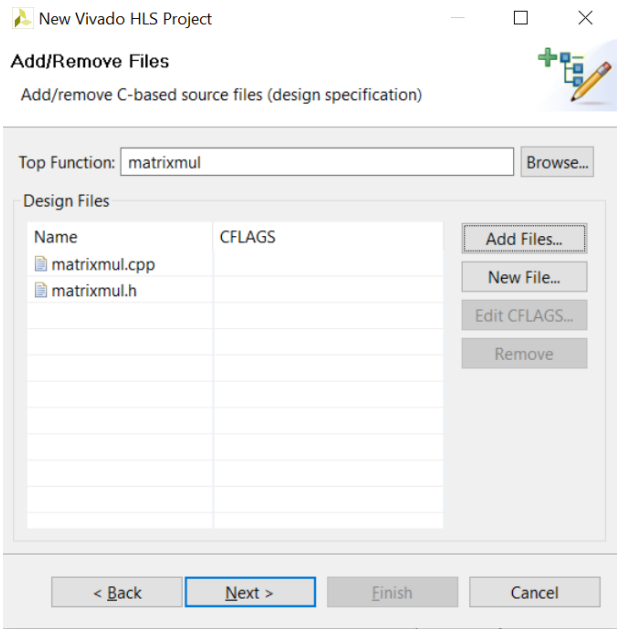
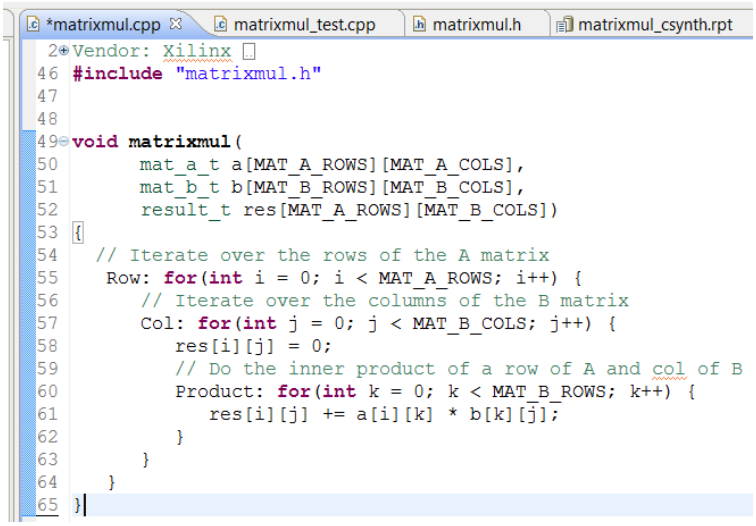


Figura 2.8. Seleccionar el código fuente en Vivado HLS

Debemos seleccionar Add Files e introducir los archivos matrixmul.cpp y martixmul.h (Se encuentran en la siguiente ruta: \Vivado_HLS_Tutorial\Design_Optimization\lab1), además de indicar cuál será la función top de nuestro diseño (como se ha comentado anteriormente, esto puede realizarse desde aquí o mediante la directiva TOP). Hacemos clic en Next.



Como podemos observar en la siguiente figura, el código que vamos a emplear en este ejemplo es el método clásico para multiplicar matrices en C.

Para aplicar las directivas de optimización se han empleado las etiquetas Row, Col y Product.

Figura 2.9. Vista del código fuente en Vivado HLS

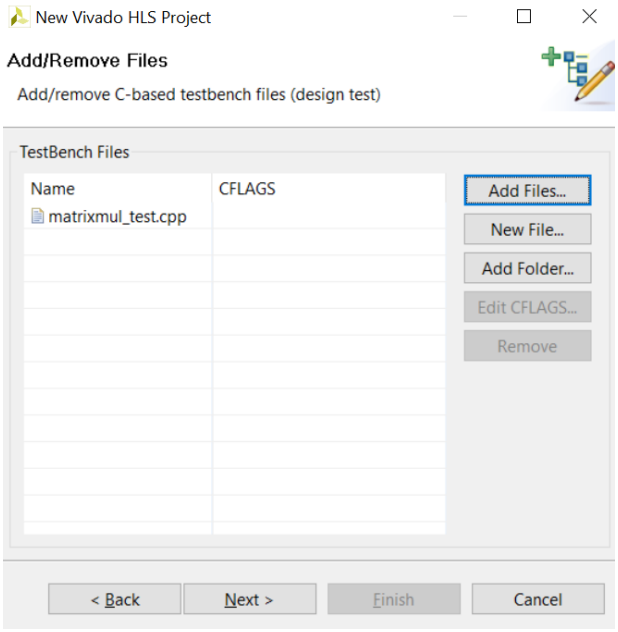


Figura 2.10. Seleccionar el testbench del diseño en Vivado HLS

En esta ventana seleccionamos el test-bench que nos servirá para validar el sistema. Hacemos clic en Next.

```
46 #include <iostream>
47 #include "matrixmul.h"
48 using namespace std;
49 int main(int argc, char **argv)
50 {
51    mat_a_t in_mat_a[3][3] = {
52        {11, 12, 13},
53        {14, 15, 16},
54        {17, 18 , 19}
55    };
56    mat_b_t in_mat_b[3][3] = {
57        {20, 21, 22},
58        {23, 24, 25},
59        {26, 27, 28}
60    };
61    result_t hw_result[3][3], sw_result[3][3];
62    int err_cnt = 0;
63    // Generate the expected result
64    // Iterate over the rows of the A matrix
65    for(int i = 0; i < MAT_A_ROWS; i++) {
66       for(int j = 0; j < MAT_B_COLS; j++) {
67          // Iterate over the columns of the B matrix
68          sw_result[i][j] = 0;
69          // Do the inner product of a row of A and col of B
70          for(int k = 0; k < MAT_B_ROWS; k++) {
71             sw_result[i][j] += in_mat_a[i][k] * in_mat_b[k][j];
72          }
```

Figura 2.11. Imagen del testbench del diseño en Vivado HLS

En este caso, dado que es un sistema muy sencillo, la forma en que se ha realizado el test bench es poniendo directamente el mismo código que el que se encuentra en

matrixmul.cpp. Esto no es lo habitual, de forma general, este testbench sería de la siguiente forma:

```
#include <iostream>
#include "matrixmul.h"
using namespace std;
int main(int argc, char **argv)
{
    mat_a_t in_mat_a[3][3] = {
        {11, 12, 13},
        {14, 15, 16},
        {17, 18 , 19}
    };
    mat_b_t in_mat_b[3][3] = {
        {20, 21, 22},
        {23, 24, 25},
        {26, 27, 28}
    };
    result_t hw_result[3][3], sw_result[3][3];

    matrixmul(in_mat_a, in_mat_b, hw_result);
```

Figura 2.12. Imagen del testbench general de un diseño en Vivado HLS

De forma que el testbench actúe de función main y haga la llamada a la función top de nuestro diseño.

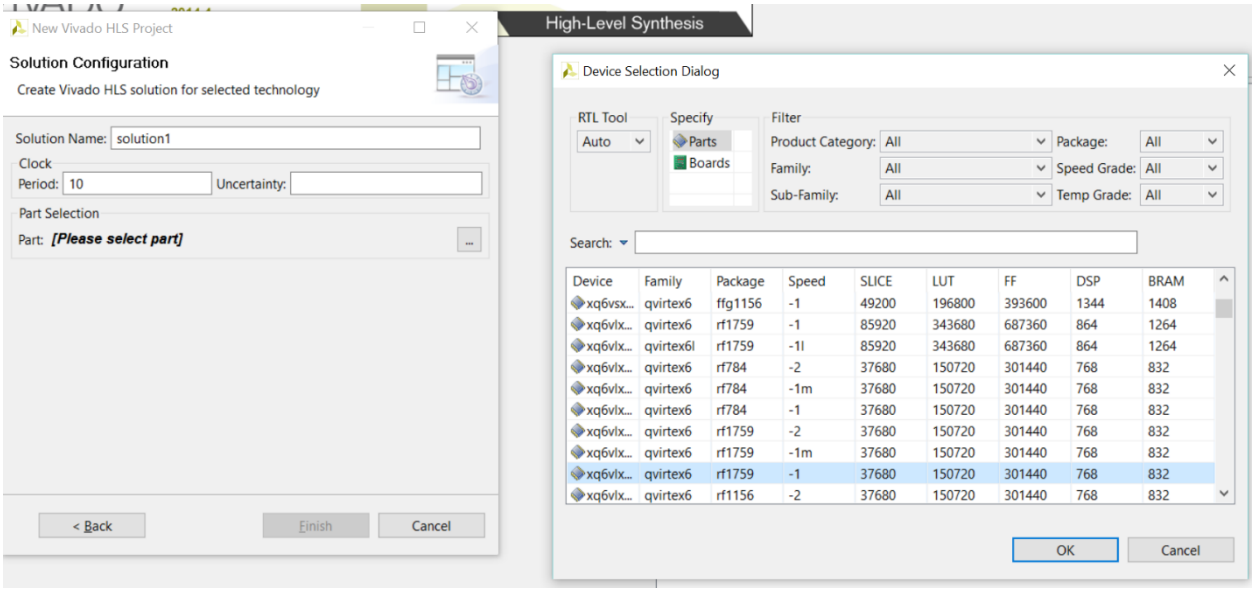


Figura 2.13. Elección de la tarjeta en Vivado HLS

Seleccionamos la tarjeta para la que va dirigido el diseño. Vivado HLS tiene incluidas las tarjetas más comunes. También podemos fijar el período de reloj objetivo para nuestro sistema (también se puede hacer mediante la directiva Clock). Hacemos clic en Finish.

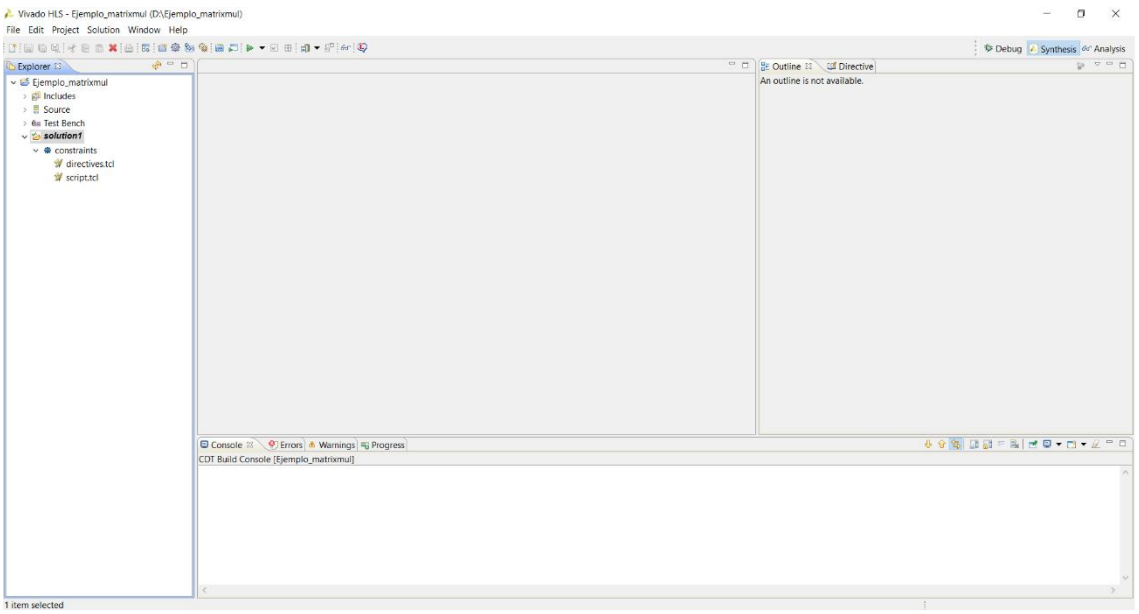


Figura 2.14. Vista entorno de trabajo en Vivado HLS

Por último, se abrirá la siguiente ventana, que es donde vamos a trabajar a la hora de desarrollar nuestro código. Hacemos clic en el apartado de Sources y doble clic en cada uno de los archivos para abrirlos y poderlos modificar. Para este ejemplo vamos a centrarnos en matrixmul.cpp para generar distintas soluciones y poder compararlas:

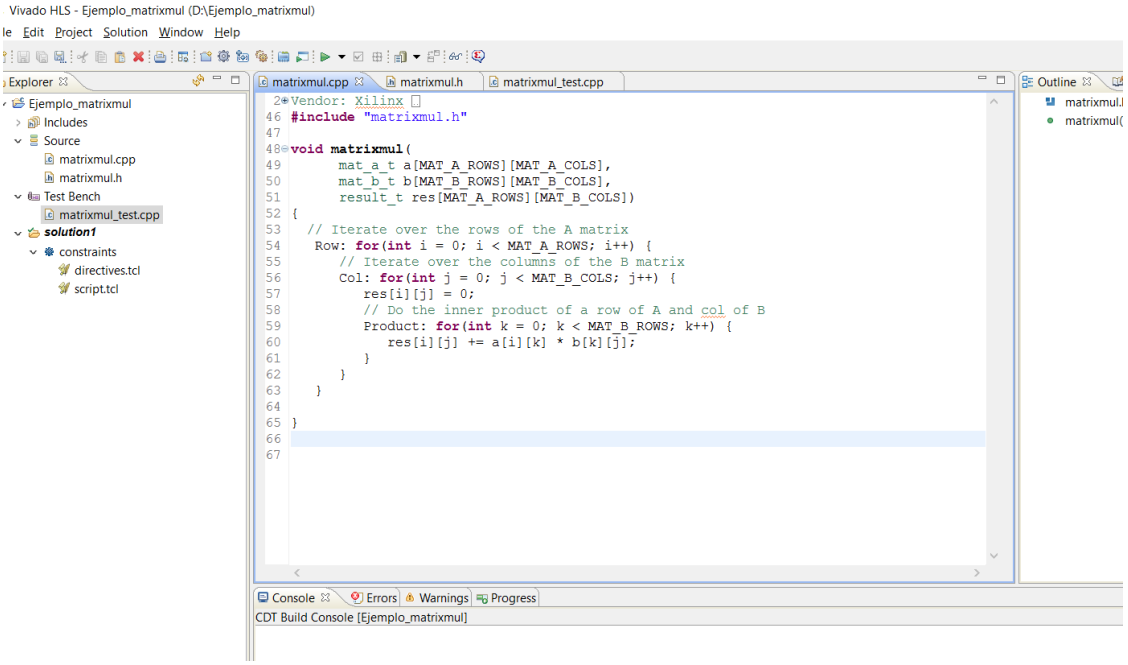


Figura 2.15. Vista perspectiva Synthesis en Vivado HLS

En primer lugar, vamos a generar una solución con el código tal y como está, sin aplicar ninguna directiva de optimización y así poder valorar posteriormente las mejoras que

éstas suponen. Para ello clicamos sobre , que es el equivalente a hacer la implementación en Vivado HLS. Cuando finalice nos mostrará la siguiente ventana:

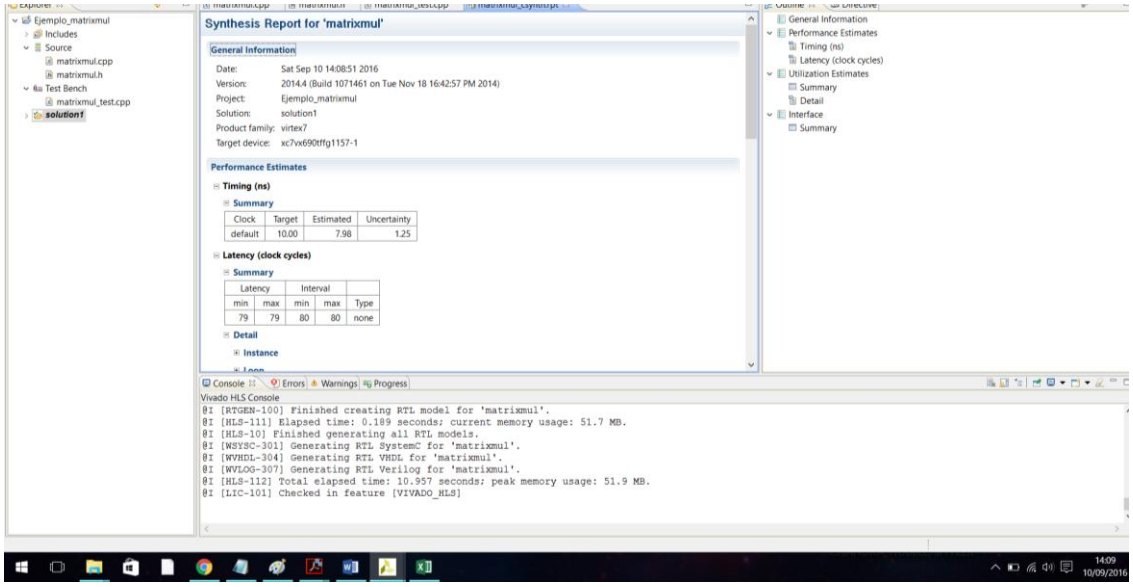


Figura 2.16. Vista de reportes de la síntesis en Vivado HLS

En la carpeta solution1 -> syn -> report -> matrixmul_csynth.rpt podemos encontrar el reporte que nos aparece en pantalla. En él podemos observar todos los detalles que se han comentado en apartados anteriores, tales como el ciclo de reloj estimado del sistema, el cual se obtiene durante la fase de síntesis en la cual Vivado HLS analiza el diseño y selecciona el ciclo de reloj estimado para un correcto y óptimo funcionamiento del mismo. También obtenemos el valor de la latencia de nuestro sistema, el cual viene desglosado en función de las etiquetas de nuestro programa con el fin de localizar aquellas zonas donde existe una mayor dependencia de datos, mayores retardos, etc. Lo mismo ocurre con los recursos consumidos, nos aparece en principio una tabla resumen, pero si deseamos conocer exactamente en qué zona de código o parte de la ejecución se consumen, basta con clicar en la pestaña “Detail”.

A continuación, vamos a generar más soluciones y finalmente realizaremos una comparación de todas ellas.

Para introducir las directivas, debemos clicar en la pestaña Directives de la parte superior derecha de nuestra pantalla, y a continuación hacer clic derecho sobre cualquiera de los elementos que aparezcan -> Insert Directive (Como se ha comentado anteriormente, las directivas se pueden aplicar tanto a funciones, como a variables globales y valores que devuelven las funciones), y seleccionamos la que deseemos.

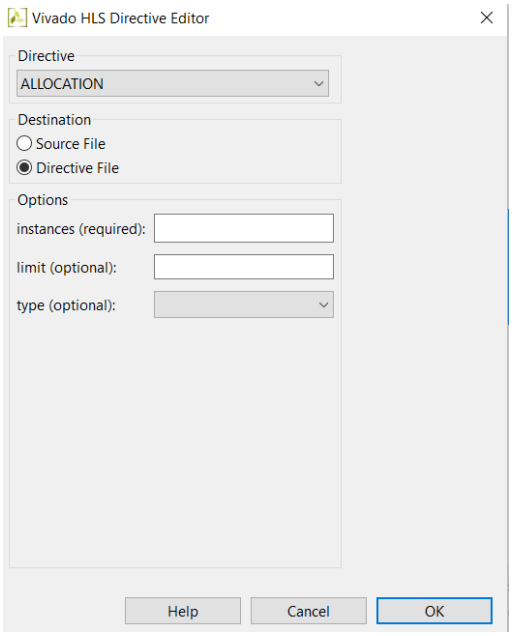


Figura 2.17. Insertar una directiva en Vivado HLS

Empíricamente, se han analizado las distintas alternativas posibles, introduciendo fundamentalmente las directivas Pipeline y Unroll ya que son las que más nos convienen en este caso dado que lo que buscamos es una reducción de la latencia sin que haya una excesiva penalización en recursos consumidos, como información adicional podríamos incluir Array_reshape para verificar que efectivamente se produce una reducción de la latencia al aumentar el ancho de palabra. A continuación, se muestran las gráficas correspondientes a la latencia obtenida, el uso de DSP48 y de LUTs[13]:

[13] Los valores de utilización de DSP48 y LUTs vienen expresados en tanto por cien,

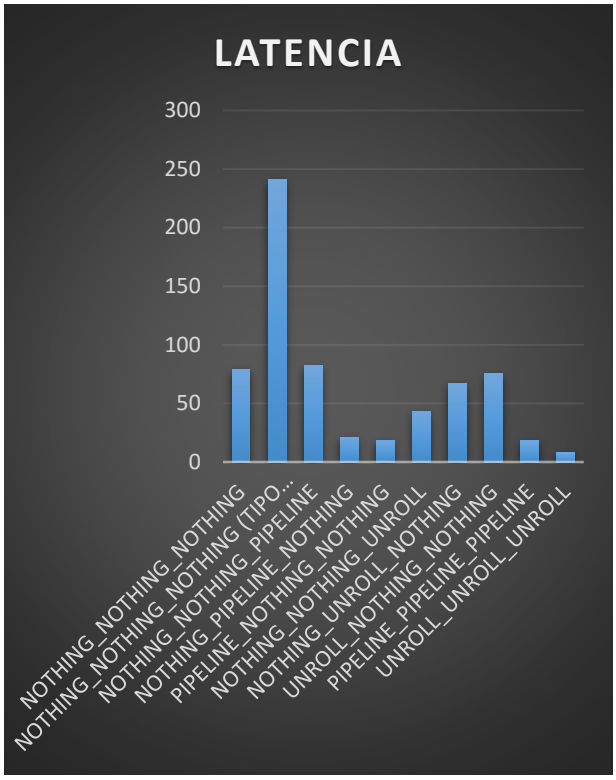


Figura 2.18. Tabla análisis de la latencia de las distintas alternativas

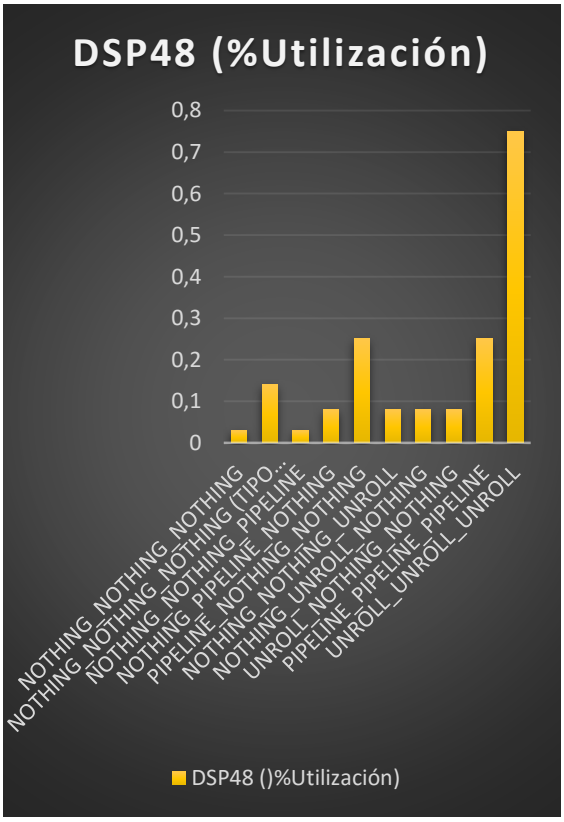


Figura 2.19. Tabla análisis de los DSP48 consumidos por las distintas alternativas

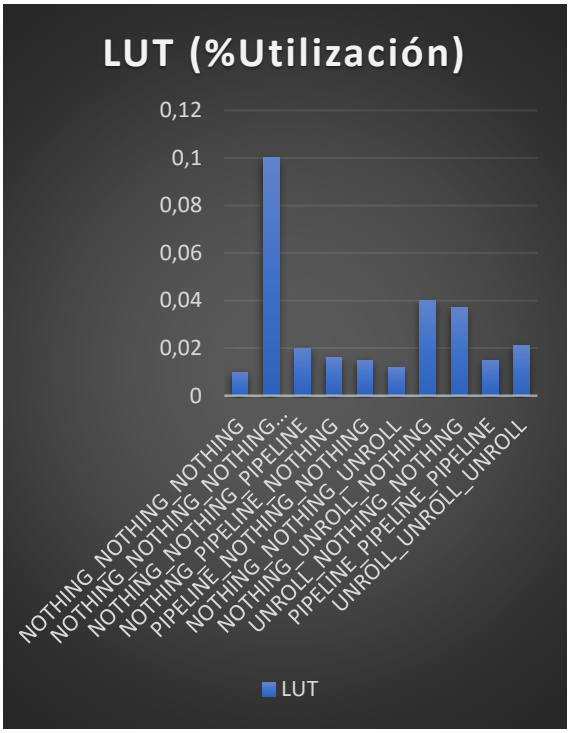


Figura 2.20. Tabla análisis de los LUTs consumidos por las distintas alternativas

Cada una de las opciones corresponden a aplicar la directiva Pipeline o Unroll sobre alguno de los tres bucles for de nuestro código (Row, Col o Product), de esta forma, de izquierda a derecha se muestran las directivas aplicadas del lazo más externo al más interno. Los dos primeros casos, en principio parecen iguales, dado que en ninguna de ellas se aplican directivas. La diferencia está en que el tipo de datos en el primer caso es char y en el segundo es de tipo float, de forma que la primera conclusión que podemos extraer es que a pesar de que el tipo float supone tener una altísima precisión, en aplicaciones donde no lo requiera, es mejor trabajar con datos enteros para evitar el incremento de la latencia y de los recursos hardware empleados ya que trabajar en formato de coma flotante supone trabajar con datos más grandes. Por último, tal y como se puede observar en las gráficas anteriores, podemos concluir con que, en caso de que nuestros diseños tengan bucles for anidados, como es este caso, la mejor alternativa es únicamente aplicar la directiva Pipeline en el segundo de ellos, consiguiendo una latencia bastante pequeña, junto con un consumo de recursos hardware ínfimo comparado con el resto de alternativas.

También es interesante abrir la perspectiva de Analysis para observar los procesos que se llevan a cabo en cada flanco de reloj, así como todos aquellos que se paralelizan gracias a la directiva Pipeline:

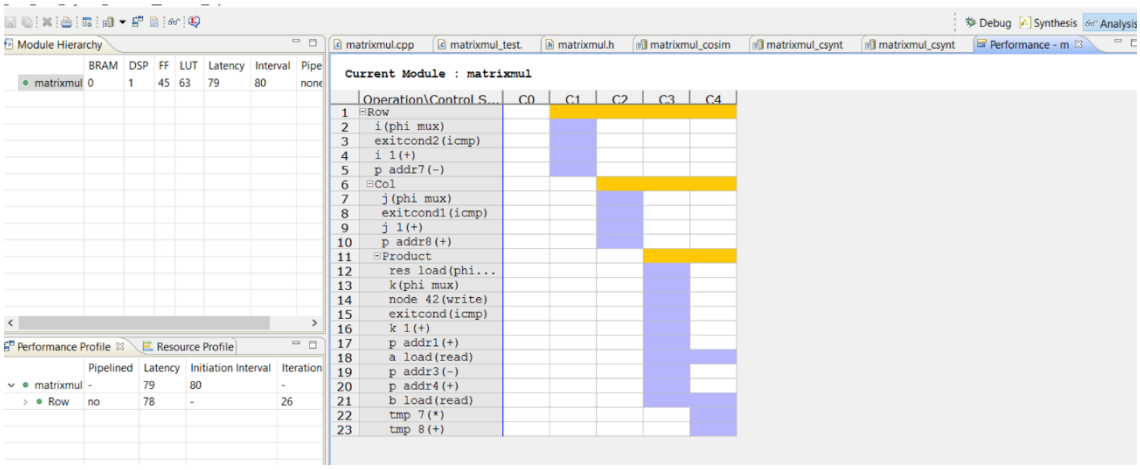


Figura 2.21. Vista de la perspectiva Analysis sin aplicar directivas de optimización en Vivado HLS

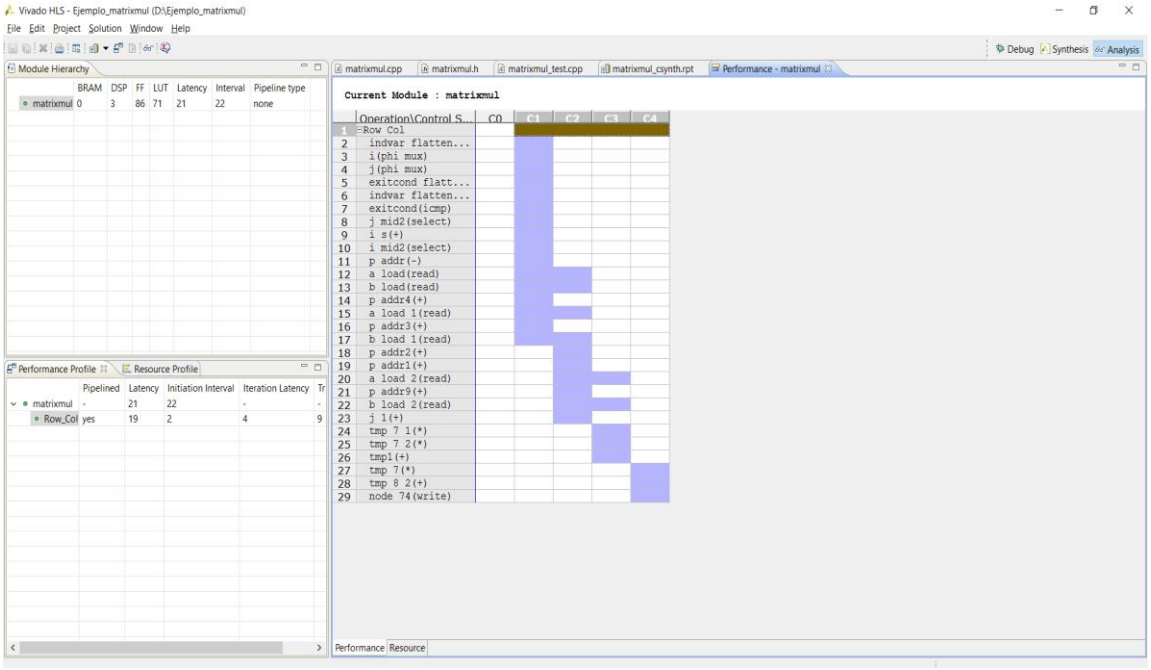
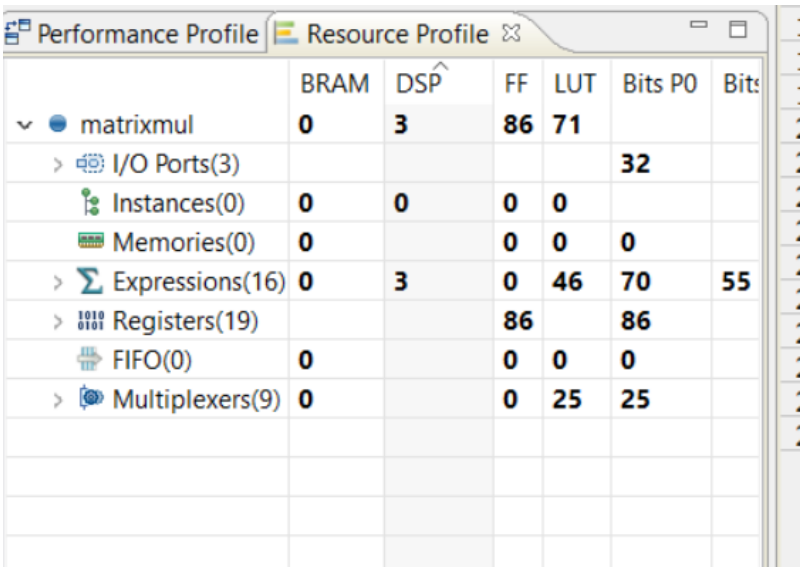


Figura 2.22. Vista de la perspectiva Analysis aplicando directivas de optimización en Vivado HLS

Como podemos observar al comparar las imágenes superiores, al realizar pipeline en el bucle correspondiente a la etiqueta Col, lo que Vivado HLS ha hecho por dentro ha sido "desenrollar" los tres bucles en uno con el fin de paralelizar al máximo las operaciones. Podemos observar todas las operaciones que han podido paralelizarse en el estado 1 de la FSM que genera de forma automática. También en la columna de la izquierda de la ventana principal viene el nombre resumido de la operación que se está llevando a cabo, como evaluar las condiciones de los bucles, leer o escribir sobre los puertos de entrada/salida, etc.

Si clicamos en la zona inferior izquierda en la pestaña de Resource Profile, podremos ver un reporte detallado de los recursos hardware consumidos, así como de las zonas donde se emplean:



	BRAM	DSP	FF	LUT	Bits P0	Bits
matrixmul	0	3	86	71		
I/O Ports(3)					32	
Instances(0)	0	0	0	0		
Memories(0)	0		0	0	0	
Expressions(16)	0	3	0	46	70	55
Registers(19)			86		86	
FIFO(0)	0		0	0	0	
Multiplexers(9)	0		0	25	25	

Figura 2.23. Vista de la perspectiva Analysis en cuanto a recursos consumidos en Vivado HLS

En el momento en que tengamos que generar el código nuevo de cero, el primer paso a realizar es validar nuestro diseño software, por lo que debemos hacer clic sobre el icono  . A continuación, se nos abrirá la perspectiva Debug, que es básicamente un simulador software para la validación:

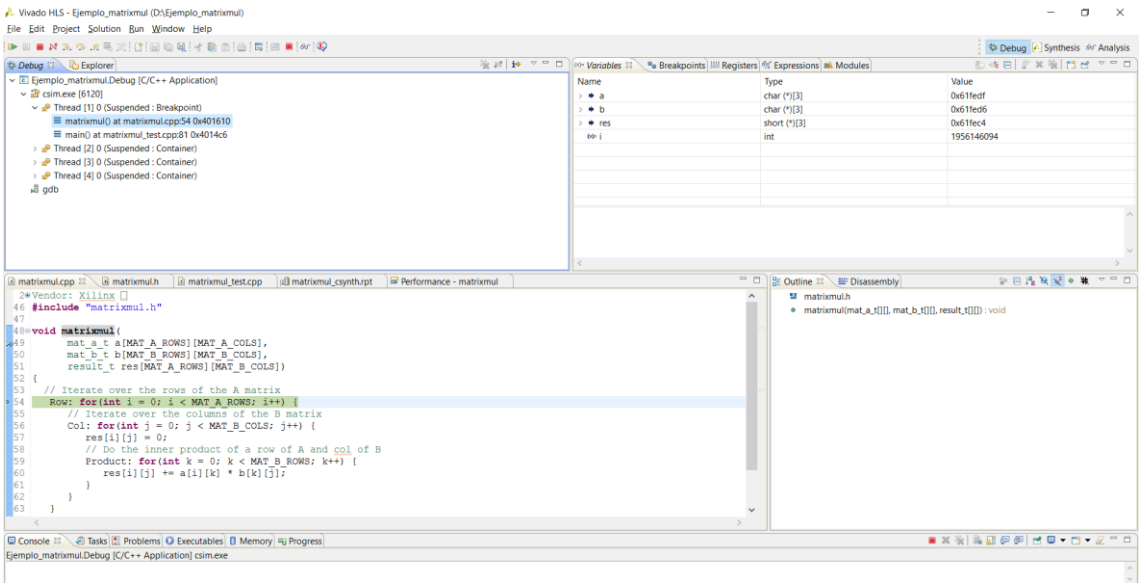


Figura 2.24. Vista de la perspectiva Debug en Vivado HLS

Una vez tenemos nuestro diseño software validado y con las directivas seleccionadas, pasamos a realizar la simulación RTL. Este es el último paso para validar nuestro diseño antes de exportarle. Para ello, hacemos clic sobre el icono y nos aparecerá la siguiente pantalla:

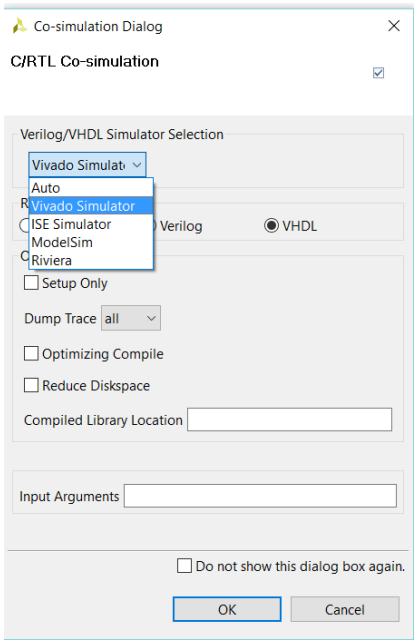


Figura 2.25. Vista de la pestaña de Cosimulacion en Vivado HLS

Donde la opción más sencilla a la hora de seleccionar un simulador es elegir Vivado Simulator, es decir, el simulador de Vivado, ya que para los otros hay que realizar una serie de acciones para "conectar" Vivado HLS con ellos, sin embargo, el simulador de Vivado viene ya conectado por defecto. La cosimulación es por así decirlo, una simulación funcional que genera automáticamente Vivado HLS a partir del testbench en C/C++ que hayamos empleado para su diseño, y de la implementación RTL del sistema que dicha herramienta realiza por sí misma a la hora de lanzar dicha simulación. Es muy importante ya que da al usuario información específica sobre el comportamiento hardware que tendrá su módulo software una vez im

Un error típico y difícil de depurar es realizar la cosimulación y para una tarjeta de una familia anterior a la Serie 7 de Xilinx, ya que la realizará y en el momento de exportarla no se nos permitirá hacerlo al catálogo de IPs y sólo se nos dará la opción de hacerlo para un "System Generator for DSP (ISE) y como un "Pcore for EDK". Esto sucede porque las versiones anteriores ya no están soportadas en esta versión de Vivado.

Una vez finalizada la cosimulación nos aparecerá la siguiente ventana:

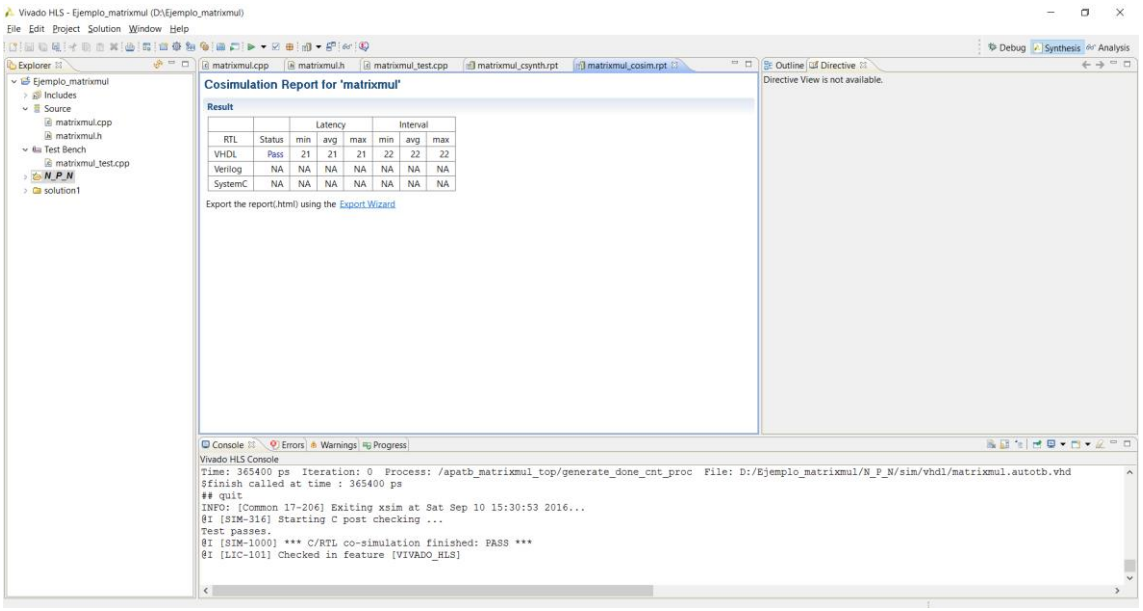


Figura 2.26. Reportes Cosimulacion en Vivado HLS

Además, en la pestaña de la solución -> sim -> vhdl, podemos encontrar los archivos generados para la simulación, así como los archivos .vhd implementados para nuestro diseño, así como de un test_bench.vhd para la simulación. Cabe destacar que, si deseamos crear un diseño nuevo en Vivado para formar parte de un sistema mayor o simplemente a modo de testeo de nuestro diseño, este testbench que genera Vivado HLS no puede ser reutilizado, por lo tanto se debe crear un testbench nuevo para que Vivado puedo simularlo. Quizás esta sea uno de los mayores puntos negativos de la herramienta, ya que, para diseños sencillos, generar un testbench nuevo es trivial, pero para sistemas más complicados puede resultar complicado generar un testbench para simular nuestro diseño.

Esto último es lo que ha ocurrido a la hora de simular el periférico para la generación de matrices de covarianza y la forma en que se ha solucionado, ha sido estudiando la co-simulación para sacar conclusiones de los comportamientos de las señales, así como de los diferentes datos de entrada junto con sus salidas correspondientes.

Para poder ver la co-simulación, seguiremos los siguientes pasos:

- Abrimos Vivado, para ello: Inicio -> Todas las aplicaciones -> Xilinx Desing Tools -> Vivado 2014.4
- En la consola de comandos Tcl introducimos la siguiente secuencia:
    - cd D:/Ejemplo_matrixmul/N_P_N/sim/vhdl
    - current_fileset
    - open_wave_database matrixmul.wdb
    - open_wave_config matrixmul.wcfg

- Por último, se abrirá la siguiente ventana:

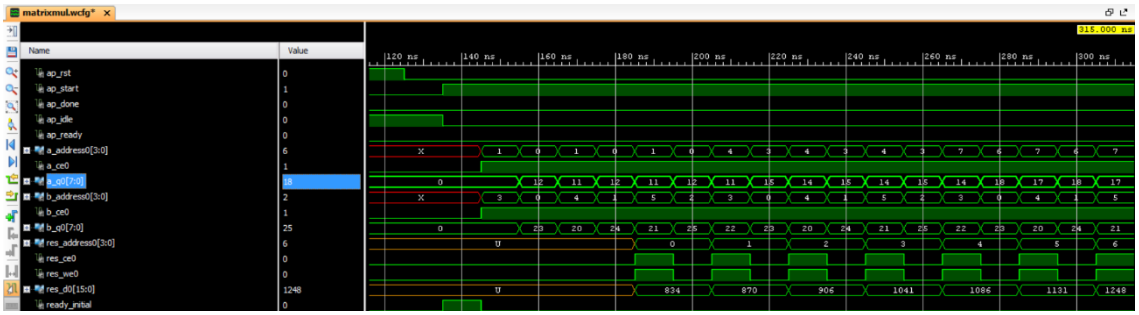


Figura 2.27 Vista cosimulación en Vivado

Como podemos observar en la imagen superior, el comportamiento de las señales de control sigue el protocolo handshaking, explicado en secciones anteriores, donde nuestro módulo solo se pone a procesar datos una vez ap_start y ap_idle pasan a nivel alto y bajo respectivamente y conforme va habiendo datos válidos en el puerto de salida, se van poniendo a nivel alto las señales res_ce0 y res_we0. Dado que en este diseño se ha aplicado la directivas de optimización, los datos de entrada entran según una secuencia específica, de ahí que la cosimulación sea tan importante.

Por último, en el momento en que estamos seguros de que nuestro diseño funciona, viene el paso de exportarlo para emplear en otros sistemas. Para ello, clicamos en el icono  y nos aparecerá la siguiente ventana:

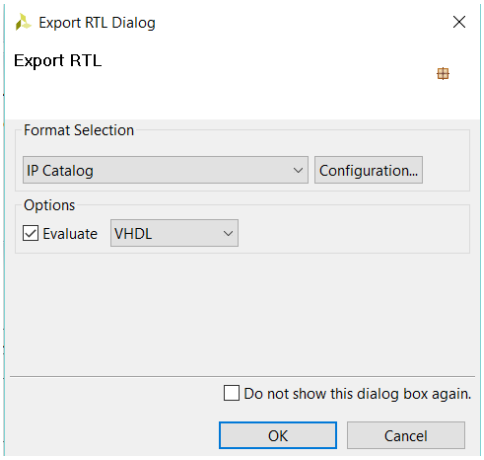


Figura 2.28. Vista de la pestaña para exportar el periférico implementado en Vivado HLS

Donde podemos seleccionar cualquiera de las siguientes opciones ya explicadas anteriormente: IP Catalog, System Generator for DSP, System Generator for DSP (ISE), Pcore for EDK y Synthetised Checkpoint.

Aquí concluiría la parte de diseño en Vivado HLS.

2.4. Implementación del Generador de Matrices de Covarianza.

Introducción

Una vez hemos realizado un análisis exhaustivo de Vivado HLS, hemos realizado un ejemplo sencillo para conocer las diferentes opciones que tenemos y hemos aprendido a verificarlo, validarlo y finalemente, exportarlo, el siguiente paso que debemos dar es comenzar a realizar nuestro diseño, el generador de matrices de covarianza.

Como ya se comentó en el apartado 1.3, este proyecto surge a partir del proyecto del Dr. Bravo, así que lo más conveniente es que partamos del diagrama del sistema completo, y lo particularicemos para este proyecto:

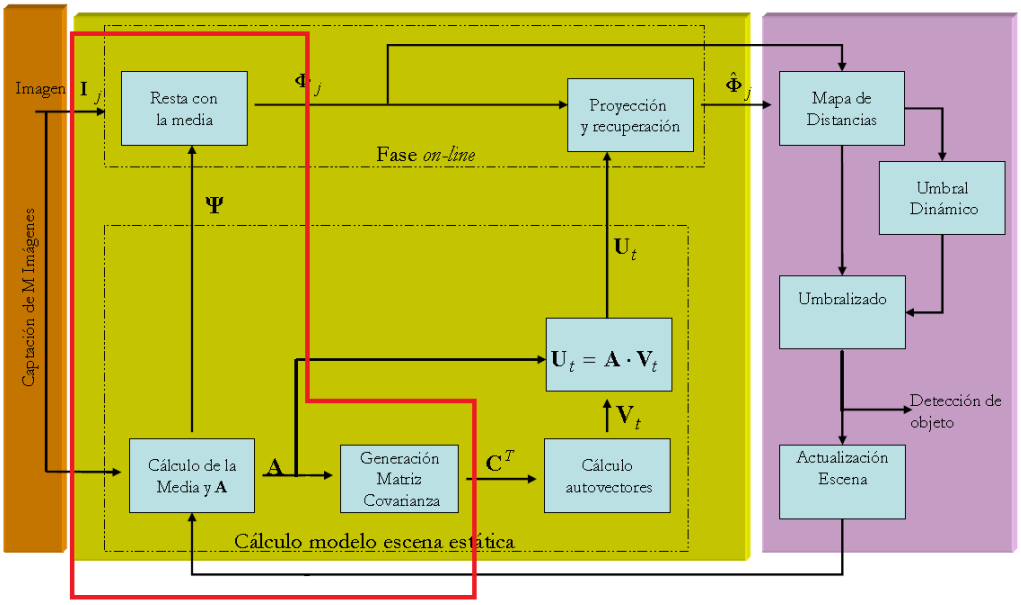


Figura 2.29. Diagrama de bloques del algoritmo PCA implementado por el Dr. Bravo

En la figura 2.29, podemos observar cual es el esquema funcional del sistema implementado por el Dr. Bravo. Inicialmente se propuso abarcar todo el sistema, pero

las dificultades encontradas provocaron centrar este TFG en la zona señalada de color rojo. De todas formas, la flexibilidad del diseño realizado, permitiría en un futuro expandirlo hacia el sistema completo.

De esta forma, para nuestro diseño, contamos con que nuestra entrada es un paquete de M imágenes, al cual debemos hacer la resta con la media, para poder generar la matriz A y finalmente, que en la salida se observen los valores correspondientes a la matriz de covarianza transpuesta.

Para llevarlo a cabo, la metodología de diseño realizada ha sido puramente ingenieril, es decir, se ha cogido un problema grande y se ha dividido en diferentes problemas más pequeños, quedando los siguientes bloques funcionales:

- Cálculo de la media.
- Resta de la media.
- Cálculo de Ct (Matriz de Covarianza Traspuesta).

Cabe destacar que, dado que se trabaja con imágenes, que se codifican como matrices, la dificultad de realizar este diseño empleando lenguajes de descripción hardware es altísima. El principal problema se centra en la manera óptima de implementar un multiplicador de matrices, que ocupe y tarde lo menos posible, sin penalizar excesivamente el número de recursos de la FPGA. Sin embargo, empleando Vivado HLS, todo se basa en saber trabajar con arrays bidimensionales en lenguaje C/C++.

Diseño realizado

La primera dificultad a la que se ha hecho frente fue el tratamiento de los datos de entrada, es decir, teóricamente sabemos que la entrada es un paquete de datos que en este caso particular son imágenes, pero se debía conseguir la manera de que, partiendo de un paquete de imágenes conseguir las matrices equivalentes. Una solución para ello es MATLAB.

MATLAB posee una toolbox con funciones específicas para trabajar con imágenes llamada Image Processing Toolbox. En ella podemos encontrar funciones para leer imágenes, para modificar su nivel de contraste, saturación, brillo, etc. Para este proyecto se han empleado las siguientes:

- imread: se emplea para leer una imagen a partir del nombre del archivo donde se encuentra. Este comando devuelve una matriz.
- mat2gray: está función convierte una matriz correspondiente a una imagen, en una matriz en escala de grises, es decir, normaliza todos los elementos para que

su parte entera sea 0 o 1, (blanco o negro) y con el resto de decimales indicar la escala de grises de ese pixel específico.

El código empleado para ello (Adquisicion_de_imagenes.m) se encuentra en el apartado 3. Planos y diagramas. Una vez hemos conseguido obtener las matrices correspondientes a las imágenes, las escribimos sobre un fichero (imágenes_input.txt) para poder leerlas desde código en Vivado HLS.

Otro de los problemas que han surgido durante el desarrollo de este proyecto es que, dado que el tamaño de las imágenes es de 256 x 256, y que el paquete de entrada está compuesto por M imágenes, la idea más inmediata es que leamos el fichero que contiene dichas imágenes, guardemos cada una en un array bidimensional de 256 x 256 posiciones y pasándolas por referencia, trabajemos con ellas. Sin embargo, al intentar hacerlo, en la parte de simulación daba un error que es provocado porque estamos intentando reservar más memoria (un array es una dirección de memoria reservada, que en hardware se traduce en el uso de bloques BRAM de la tarjeta) de la que laFPGA tiene disponible.

La alternativa que se hizo fue leer solamente una imagen cada vez, de forma, que leemos la imagen del fichero, la empleamos en los procesos que sean necesarios y finalmente sobrescribimos la información de una imagen, con la siguiente. Esto genera nuevamente varios problemas, el más importante es la sincronización de todo el sistema ya que al no tener disponibles las M imágenes a la vez, la multiplicación de la ecuación 2.13 de $A^T \cdot A$ se convierte en crítica porque al tener solamente una imagen en cada iteración, sólo tenemos disponibles una fila/columna de A^T y de A. Por lo tanto, la forma en que se ha realizado el diseño ha sido el siguiente:

En primer lugar, dado que las imágenes se cogen a través de un fichero de texto, lo más sencillo es coger cada una de forma cíclica, es decir, desde la primera hasta la última, para a continuación, volver a empezar. Una vez se ha fijado eso, tenemos que asumir que habrá situaciones en las que nuestro sistema no haga nada, ya que tiene que esperar hasta la iteración en que llegue la imagen adecuada. En la siguiente imagen se puede observar a qué se debe este momento de inactividad:

$$C^T = \frac{1}{M} \cdot A^T \cdot A = \begin{bmatrix} \Phi_{1_{(1,1)}} & \Phi_{1_{(2,1)}} & \dots & \Phi_{1_{(N^2,1)}} \\ \Phi_{2_{(1,1)}} & \Phi_{2_{(2,1)}} & \dots & \Phi_{2_{(N^2,1)}} \\ \dots & \dots & \dots & \dots \\ \Phi_{M_{(1,1)}} & \Phi_{M_{(2,1)}} & \dots & \Phi_{M_{(N^2,1)}} \end{bmatrix}_{MxN^2} \cdot \begin{bmatrix} \Phi_{1_{(1,1)}} & \Phi_{2_{(1,1)}} & \dots & \Phi_{M_{(1,1)}} \\ \Phi_{1_{(2,1)}} & \Phi_{2_{(2,1)}} & \dots & \Phi_{M_{(2,1)}} \\ \dots & \dots & \dots & \dots \\ \Phi_{1_{(N^2,1)}} & \Phi_{2_{(N^2,1)}} & \dots & \Phi_{M_{(N^2,1)}} \end{bmatrix}_{N^2xM} \quad \textbf{(2.14)}$$

Como podemos ver, para calcular el primer elemento de nuestra matriz C^T, necesitamos tener almacenada la primera fila de la matriz A^T y la primera columna de la matriz A. Por lo tanto, para calcular la primera fila de C^T, almacenamos la primera fila de A^T y en cada iteración de nuestro bucle cíclico de imágenes, calculamos la columna de A, obtenemos el elemento correspondiente, y en la siguiente, sobrescribimos el array donde almacenamos la columna de A, con la siguiente.

Una vez obtenemos la primera fila de C^T, comenzaríamos a calcular la segunda. Aquí es donde se da el "retardo" o momento de inactividad, ya que para calcular el primer elemento de la segunda fila de C^T, necesitamos la segunda fila de A^T, que corresponde con la segunda imagen, por lo tanto, el primer ciclo de M imágenes lo utilizamos para calcular la fila de A^T que necesitemos y en el siguiente calculamos la fila correspondiente de C^T. Podemos finalizar con que el principal problema de este sistema es que hay una fuerte dependencia entre datos.

Por lo tanto, siguiendo este planteamiento, hemos codificado cuatro funciones distintas, una para calcular la media durante el primer ciclo de imágenes, otra para que cada vez que vayamos a obtener valores correspondientes a C^T, reste, primeramente, a la imagen entrante su media (Tal y como vimos en el apartado 2.1 Fundamentos Teóricos), otra para generar cada elemento de C^T cuando sea necesario y por último, la función top de nuestro sistema que es donde se lleva a cabo la sincronización del conjunto.

A continuación, se muestra una imagen con el proceso secuencial que sigue nuestro sistema:

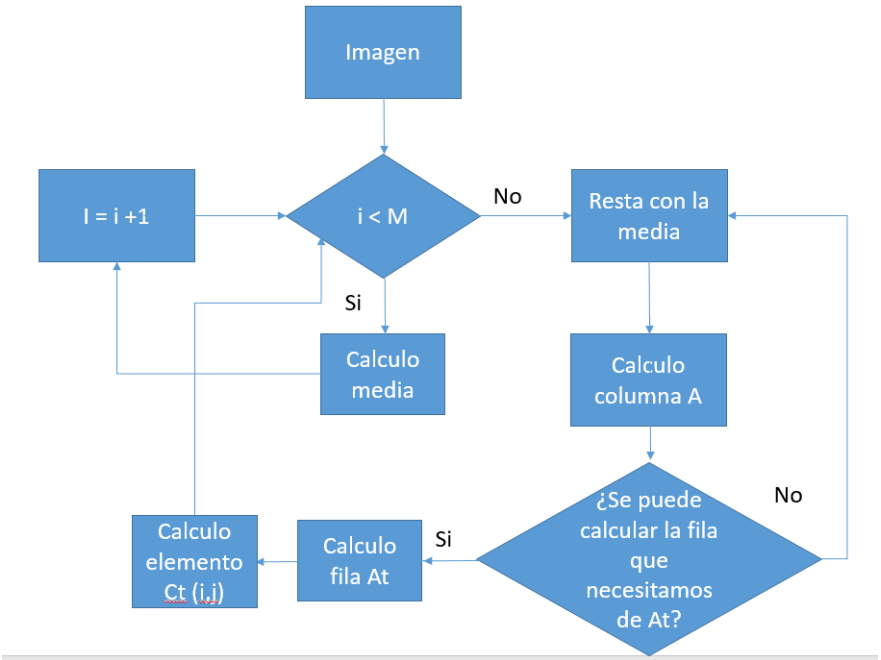


Figura 2.30. Flujo del programa.

Una vez conseguimos que el diseño anterior funcionase correctamente, se hizo una optimización del mismo, basándonos en las características de las matrices de covarianza, llegando hasta el diseño definitivo que hemos empleado en nuestro Generador de Imágenes de Covarianza.

La optimización surge a raíz de las características de las matrices de covarianza, tal y como se explicó en el apartado 2.1 Fundamentos Teóricos, ya que son matrices simétricas por definición, por lo tanto, basta con calcular la triangular superior o inferior ya que el resto de valores son valores repetidos. Gracias a ello, conseguimos una gran optimización de nuestro sistema, por tanto, al tener que calcular menos valores, es más rápido y, además, emplea menos recursos hardware, tal y como veremos en el apartado de Resultados y conclusiones.

En cuanto a las directivas de optimización empleados, se ha aplicado la directiva Pipeline a todos los bucles for. Estos están anidados tal y como se puede ver en el apartado 3. Planos y diagramas, además es la mejor forma de trabajar en C/C++ con arrays bidimensionales. Siempre, al segundo for, o al más interno en caso de que haya solo dos , puesto que esta es la forma de lograr una mayor optimización tal y como se explicó en el apartado 2.3 Vivado HLS. Además, se le ha aplicado tanto al puerto de entrada, correspondiente a la entrada de imágenes, como al de salida, correspondiente a la matriz de covarianza traspuesta, la directiva Interface, para que se implemente una interfaz AXI, con vistas a trabajos futuros, lo cual, se explicará más adelante.

En cuanto al tipo de datos, durante el desarrollo de nuestro sistema se han empleado datos en coma fija (fixed point) con un ancho de palabra de 30 bits, reservando 20 para la parte decimal. El motivo por el cual se ha elegido este tamaño se explica en profundidad en el siguiente apartado.

En el apartado 3. Planos y diagramas, se encuentra el código empleado para conseguirlo (PCA.cpp, PCA.h y PCA_test.cpp).

Análisis de la precisión

Como hemos observado en el apartado 2.3. Vivado HLS, emplear datos de tipo float supone un incremento muy importante de la latencia y del uso de recursos hardware, por lo tanto, se debe optar por el paso intermedio entre los números enteros y los flotantes (floating point), los de coma fija (fixed point).

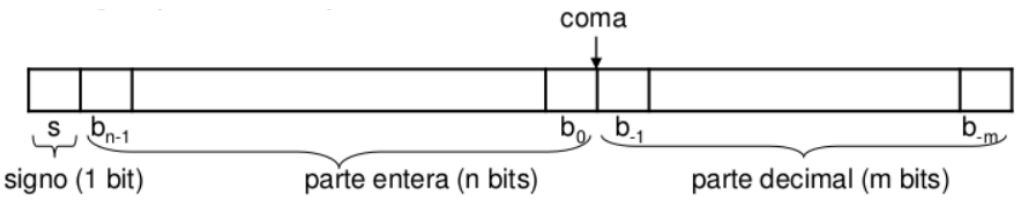


Figura 2.31. Formato en coma fija.

Los datos codificados en coma fija, acorde a la figura 2.28,son muy versátiles, ya que nos permiten fijar tanto la parte entera, como la decimal de nuestro tamaño de palabra.

Para realizar el estudio del error que cometemos al restringir el número de decimales se ha empleado MATLAB, ya que tiene integrado una toolbox específica (fixed point)para trabajar con datos de tipo fixed.

Dentro de la función fi de MATLAB, podemos fijar, además del ancho de palabra, tanto la parte entera como la decimal, también podemos fijar el tipo de redondeo, las acciones en caso de overflow y si deseamos que se codifique con signo o sin él, entre otras cosas. Tal y como podemos encontrar en Matlab al introducir el comando “help fi”

En este caso, para realizar el estudio, hemos partido de 1 decimal, y se ha ido aumentando hasta llegar a 16 bits, dejando la parte entera con 1 sólo bit. El resultado obtenido se ve reflejado en la siguiente gráfica:

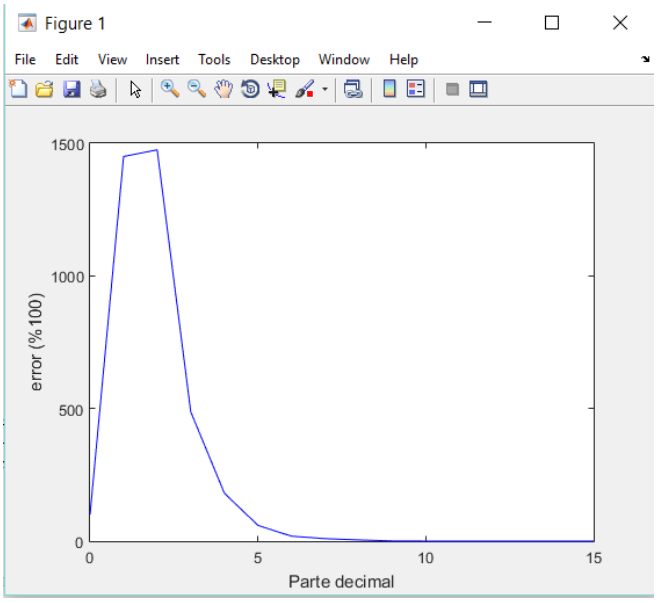


Figura 2.32. Diagrama de bloques del algoritmo PCA implementado por el Dr. Bravo

Esta gráfica surge a partir de codificar todo el diseño realizado en Vivado HLS, pero en MATLAB, siendo el código el mismo, pero adaptándolo a la sintaxis necesaria para programar en MATLAB. Para realizarla se ha ido calculado la matriz Ct a partir de ir transformando todos los datos que teníamos en formato double a formato fixed point, dejando a 1 bit la parte entera y añadiendo parte decimal en cada iteración.

Como podemos observar, el error inicialmente tiende a hacerse muy grande, lo cual es lógico ya que prácticamente no tiene parte decimal, y conforme vamos añadiendo decimales va tendiendo a hacerse 0.

Este estudio es meramente orientativo, ya que para fijar el ancho final de palabra se ha realizado de forma empírica, escribiendo los resultados de C^T sobre un fichero de texto y estudiando el error medio en MATLAB. El código empleado para ello (Ver_error_medio.m) se muestra en el apartado 3. Planos y diagramas.

Tal y como comentamos en el apartado 2.3., Vivado HLS posee una librería para trabajar con datos de tipo fixed. Es muy importante hacer el análisis de la precisión una vez se ha validado el código, ya que el modo debug de Vivado HLS funciona con datos de tipo fixed, pero sin embargo, no muestra correctamente este tipo de variables al realizar ejecuciones paso por paso, por lo tanto, lo más conveniente es que estemos seguros de que el diseño funciona correctamente antes de evaluar la precisión de los datos.

La clase fixed que lleva incorporada Vivado HLS es muy similar a la de MATLAB. Los parámetros que podemos fijar son los siguientes:

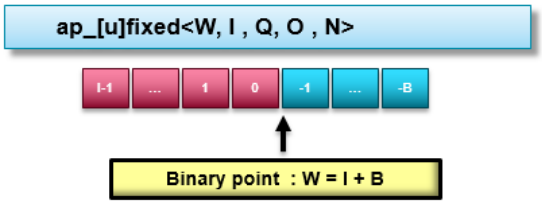


Figura 2.33. Clase ap_fixed de Vivado HLS.

Los parámetros de la clase son:

- W: ancho de la palabra.
- I: ancho de la parte entera. Si se utiliza *"ap_ufixed"* (unsigned), I es el ancho de la parte entera.
- Q: define el modo de redondeo. Tenemos varias opciones:
  - AP_RND: redondeo hacia +∞. El equivalente en MATLAB es "ceiling". Este es el que se ha empleado en este proyecto.

- AP_RND_ZERO: redondeo hacia 0. El equivalente en MATLAB es “fix”.
- AP_RND_MIN_INF: redondeo hacia -∞. El equivalente en MATLAB es “floor”.
- AP_RND_CONV: redondeo al valor cercano. El equivalente en MATLAB es “round”
- AP_RND_INF: redondeo a ∞.
- AP_TRN: truncamiento.
- AP_TRN_ZERO: truncamiento a 0. (por defecto)

- O: determina como debe comportarse la clase cuando el número de bits necesarios para representar un dato excede el especificado (overflow). Tenemos varias opciones:
  - AP_SAT: saturación, hacia el valor mínimo o máximo.
  - AP_SAT_ZERO: saturación a 0.
  - AP_SAT_SYM: saturación simétrica.
  - AP_WRAM: Valor por defecto.
  - AP_WRAP_SM.
- N: determina el número de bits que se utilizan cuando el modo de overflow es envoltorio (wrapped).

Finalmente, el tamaño del dato que se ha elegido para este proyecto han sido 30 bits, 10 para la parte entera y 20 para la decimal, valor para el cual, el error obtenido es menor del 5%. Todo ello ha sido posible gracias a que una vez calculábamos la matriz Ct en Vivado HLS, escribíamos el resultado en un fichero de texto para realizar el estudio del error en MATLAB tal y como se ha comentado anteriormente.

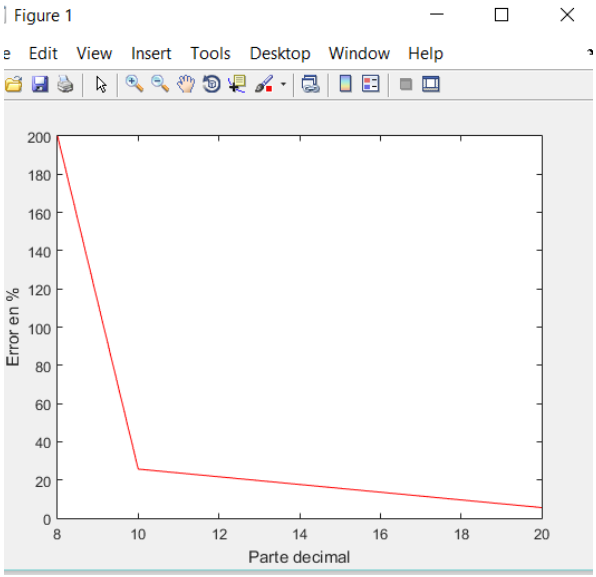


Figura 2.34. Estudio del error de Ct del sistema en Vivado HLS.

La gráfica anterior muestra el estudio del error medio en función de la cantidad de bits de la parte decimal, esto es porque la parte entera ya se fijó a 10 bits con anterioridad ya que el error antes de fijarlo se hacía infinito para cualquier valor de la parte decimal. La razón porque se tuvo que ampliar el número de bits de la parte entera ha sido que hay zonas en el código donde se realizan operaciones de multiplicación y suma sobre una misma variable, dando lugar situaciones de overflow, para evitarlas, se incrementó el número de bits. El caso más crítico, para el paquete de imágenes analizado es para la primera posición de la matriz de covarianza, ya que, el valor llegaba a ser del orden de 120, lo cual son 8 bits más el de signo. Por precaución se fijó el valor a 10 bits para la parte entera.

Implementación RTL y exportación del módulo.

Una vez hemos validado el diseño tanto en coma fija como en coma flotante, procederemos a realizar la cosimulación del sistema en coma fija, el cual es el diseño definitivo y seleccionando 8 imágenes de entrada de 256x256 (M = 8), obteniendo la siguiente gráfica donde se ve el comportamiento de todas las señales que gobiernan el sistema y que se explicarán a continuación:

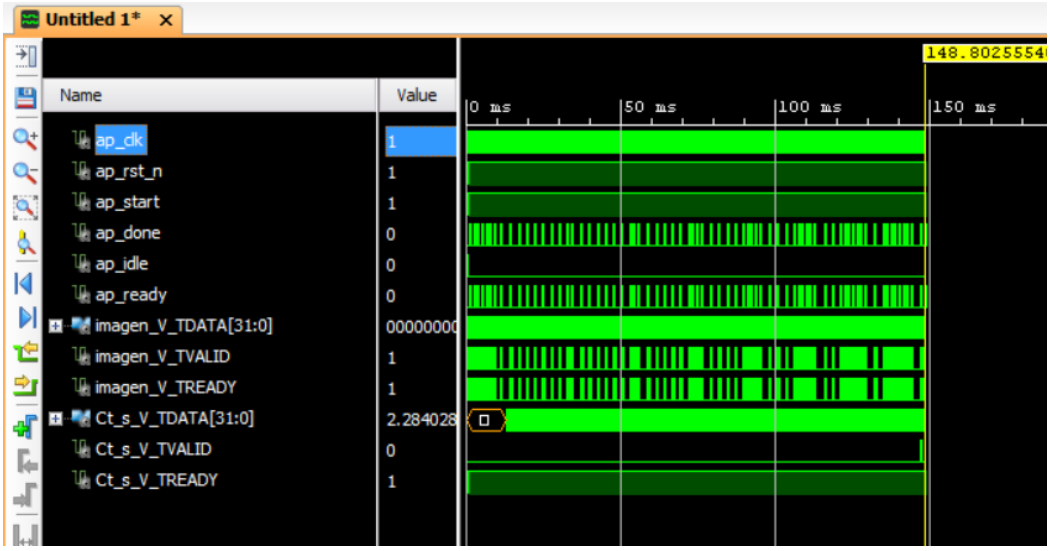


Figura 2.35. Cosimulación de Vivado HLS.

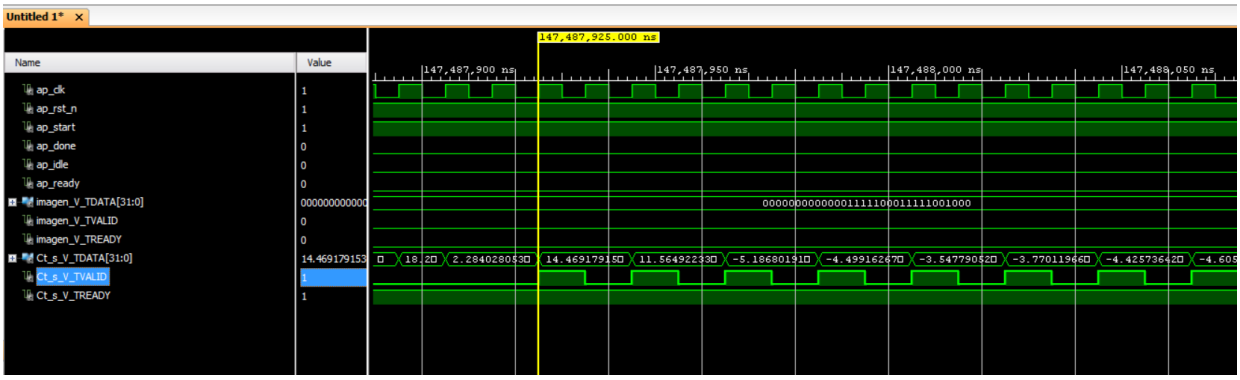


Figura 2.36. Cosimulación en detalle de Vivado HLS.

Tal y como podemos observar, los datos de salida válidos se obtienen cuando Ct_s_V_TVALID se pone a 1. Por otra parte, cada una de las veces que las señales imagen_V_TVALID e imagen_V_TREADY se ponen a 1, son cada una de las veces que entra una imagen, es decir, cada una de las iteraciones. Las demás señales son las mismas que las de una interfaz handshaking ya explicada en apartados anteriores.

Una vez hemos comprobado que los valores de salida obtenidos son los correctos, procedemos a exportar el módulo RTL generado y ya estaría listo para usarse en sistemas más complejos. Para ello, seleccionamos el botón desde Vivado HLS y seleccionamos las opciones de IP Catalog y clicamos la opción de Evaluate -> VHDL. Una vez finalice el proceso de exportación, se habrá creado una carpeta dentro de la de la solución que hayamos exportado con el nombre de impl, tal y como se ha explicado en la sección 2.3. Vivado HLS.

A continuación abrimos Vivado y para añadir nuestro IP basta con clicar la pestaña de Tools -> Project Settings -> IP y seleccionar el repositorio de nuestro IP que se encontrará en solutionXXX -> impl -> ip. Con ello ya podemos buscar nuestro IP por el nombre de la función top de Vivado HLS y emplearlo como un core más.

Nuevamente, con el fin de verificar nuestro módulo y aprender a trabajar con él, se ha realizado un proyecto en Vivado que tiene por finalidad realizar la misma simulación que durante la cosimulación que hace Vivado HLS.

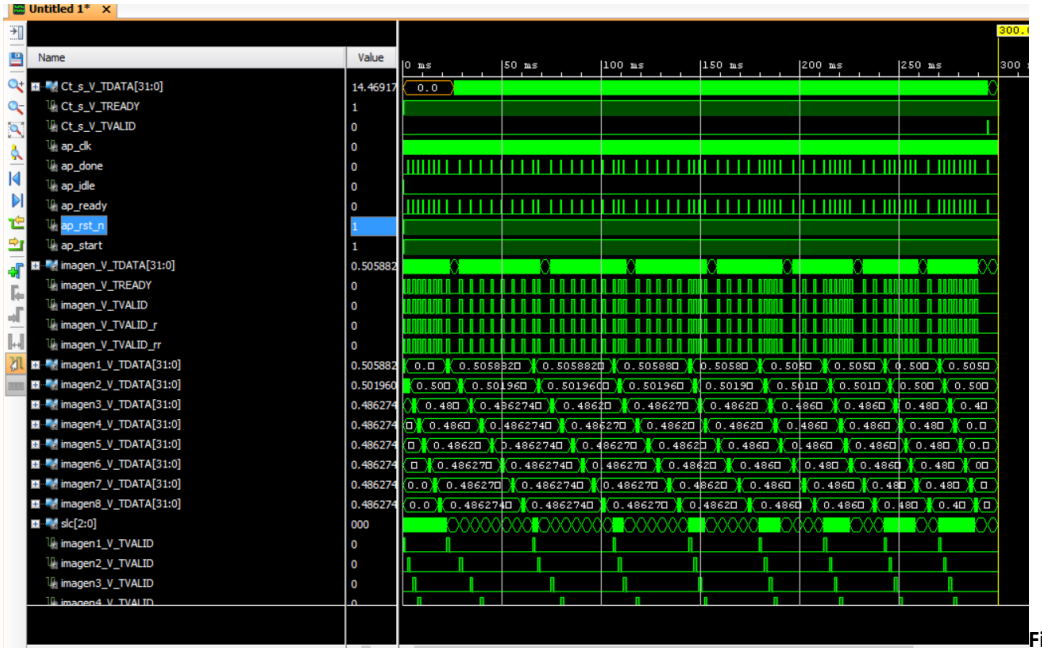


Figura 2.37. Simulación del generador de covarianza en Vivado

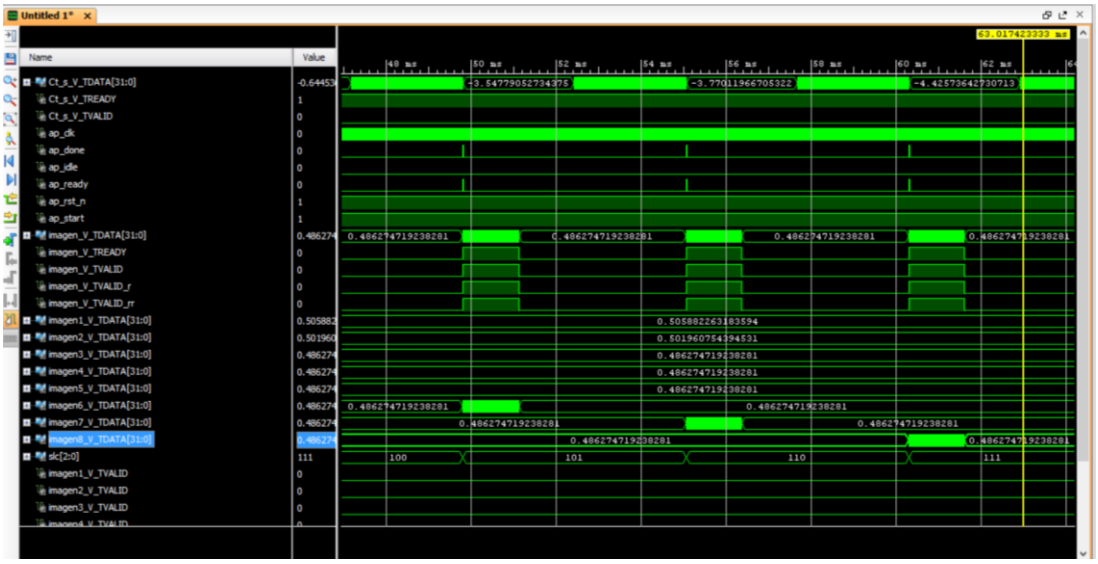


Figura 2.38. Simulación del generador de covarianza en Vivado

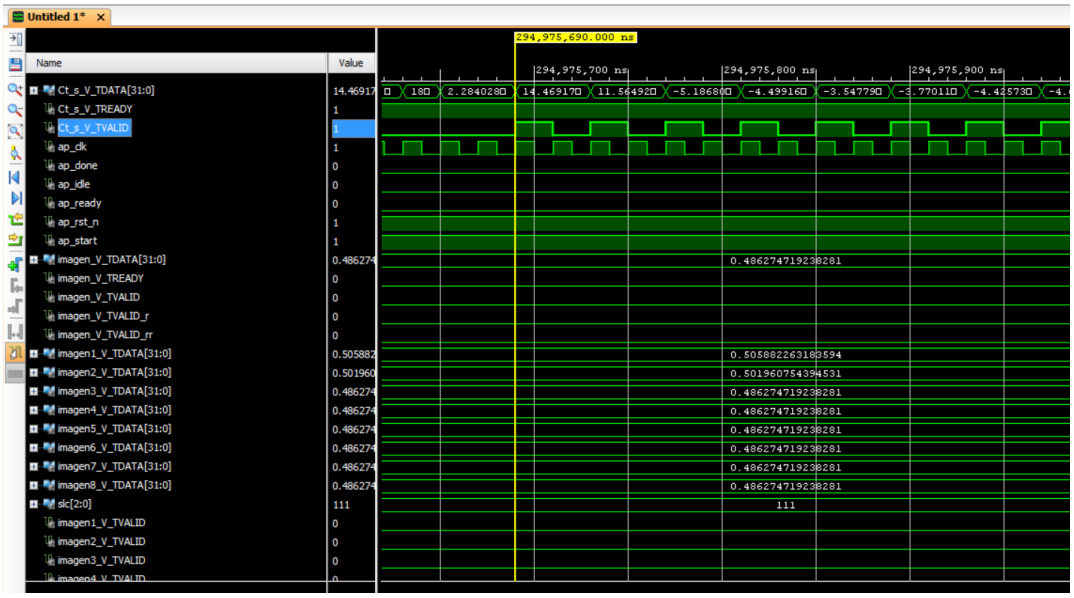


Figura 2.39. Vista detallada del puerto de salida del generador de covarianza en Vivado

En las gráficas anteriores se muestra la simulación llevada a cabo en Vivado a partir de un testbench generado en vhdl. En ellas, aparecen las mismas señales que en la cosimulación de Vivado HLS, junto con algunas nuevas que son han sido necesarias para generar dicho testbench. El código que se ha empleado para ello queda recogido en la sección 3. Planos y diagramas, pero básicamente es un multiplexor donde en el momento en que corroboramos que se ha introducido una imagen, y que nuestro sistema está listo para recibir una nueva (flanco de subida de ap_done), hacemos que se lea la siguiente desde un fichero y se introduzca por la entrada del sistema.

Cabe destacar que la principal diferencia entre la cosimulación de Vivado HLS y la que hemos generado nosotros es que el tiempo de ejecución en el mismo PC en Vivado HLS es la mitad que en Vivado. Una casusa es el buffer de memoria del PC. En vhdl metes muchas más muestras que en Vivado HLS, ya que tienes más señales individuales y menor período de muestreo. Además, esto puede ser debido a que Vivado HLS realiza optimizaciones a la hora de generar los datos de entrada, como por ejemplo implementar una RAM interna para que los datos vayan entrando a golpe de flanco de reloj.

También cabe destacar que los datos de entrada, tanto para Vivado HLS, como para Vivado, son mediante un fichero de texto. La principal diferencia es que en el primer caso se pueden escribir los valores de entrada en el formato double de MATLAB, ya que Vivado HLS tiene funciones de conversión para transformar este tipo de datos en tipo fixed, sin embargo, en Vivado, dado que es un entorno puramente hardware, tenemos que pasarle los datos codificados directamente en formato fixed. El problema es que MATLAB no tiene funciones para escribir sobre fichero en formato fixed, asique la alternativa que se eligió para solventarlo fue escribir los datos en formato binario, el

cual sí está soportado tanto en MATLAB como en VHDL, tal y como se muestra a continuación:

```
a = imagen (i,j,k);
dlmwrite('imagenes_32bits_8.txt',a.bin,'-append','delimiter','','roffset',1);
```

Figura 2.40. Código empleado para escribir datos de tipo fixed en un fichero desde MATLAB

Dicho código se muestra en el Apartado 3. Planos y diagramas.

Resultados y conclusiones

A continuación, se adjuntan capturas de los distintos reportes obtenidos durante la síntesis de las distintas alternativas que se han seguido durante este proyecto:

En las siguientes gráficas se muestran las distintas alternativas de diseño para nuestro Generador de Matrices de Covarianza, en la parte izquierda, la diferencia entre emplear datos de tipo fixed con 10 bits de parte entera y 20 de parte decimal frente al uso de datos de tipo float. En la parte derecha se muestra una gráfica comparativa entre estos tipos de datos sin aplicar ninguna directiva de optimización y aplicando la directiva pipeline en los bucles for anidados (en el más interno de ellos) ya que es la forma en que conseguimos una mejor optimización en cuanto a latencia y a recursos consumidos tal y como vimos en secciones anteriores.

Timing (ns)

Clock		Fixed_N_N_N	Float_N_N_N
default	Target	10.00	10.00
	Estimated	7.68	8.44

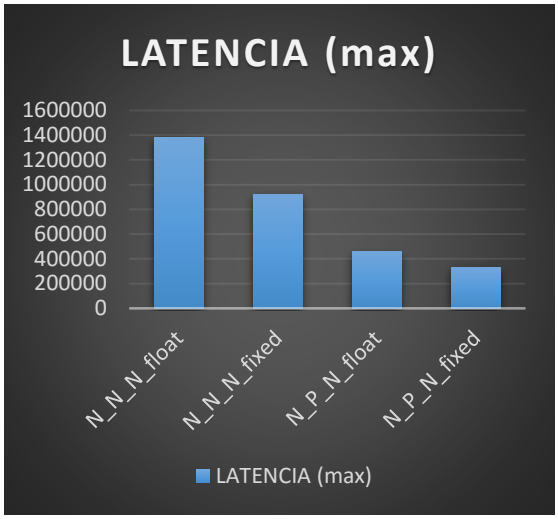
Latency (clock cycles)

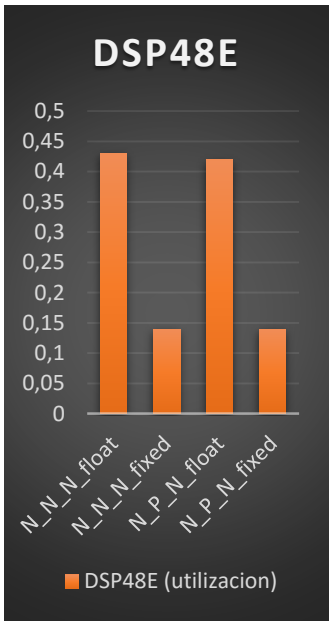
		Fixed_N_N_N	Float_N_N_N
Latency	min	131589	131589
	max	919738	1378493
Interval	min	131590	131590
	max	919739	1378494

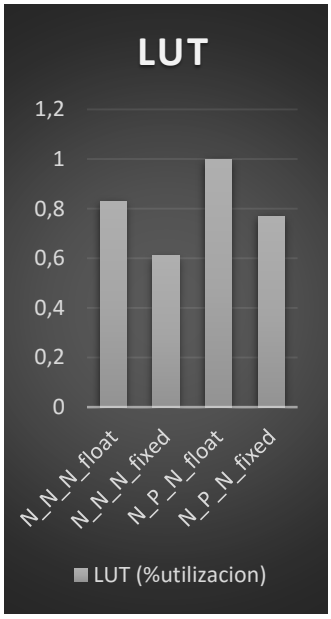
Utilization Estimates

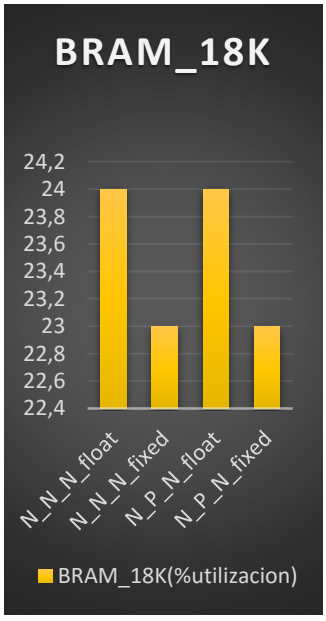
	Fixed_N_N_N	Float_N_N_N
BRAM_18K	481	513
DSP48E	4	12
FF	1078	2327
LUT	1876	2523

Export the report(.html) using the Export Wizard









Como se puede deducir de las tablas anteriores, el coste de emplear una codificación en formato float es bastante superior que en formato fixed point. En primer lugar, en cuanto a la latencia, podemos ver que es superior en todos los casos al emplear datos de tipo float, lo cual es bastante lógico al ser datos con un ancho de palabra más grande que los de tipo fixed. De igual forma sucede con el gasto de recursos en cuanto al uso de DSP48, LUTs y bloques BRAM, al ser más datos los que se deben de operar, para intentar realizarlos en el mismo tiempo lo más lógico es que se empleen más recursos para ello.

Por otra parte, se ha llegado a la conclusión de que la principal diferencia entre el diseño en el que calculamos la matriz de covarianza completa, y la alternativa donde se calcula la triangular superior es el tiempo en el que se lleva a cabo, lo cual es lógico, ya que en una de ellas se calculan más valores que en la otra, y más importante, el consumo de bloques BRAM de la primera es superior al de la segunda y dado que tanto en este diseño como en los futuros que le seguirán, el uso de recursos de memoria es la restricción más

crítica, puede determinar el uso de una alternativa por encima de la otra. Los reportes que lo justifican son los siguientes:

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	670
FIFO	-	-	-	-
Instance	120	4	451	1185
Memory	241	-	0	0
Multiplexer	-	-	-	343
Register	-	-	557	-
Total	361	4	1008	2198
Available	2060	2800	607200	303600
Utilization (%)	17	~0	~0	~0

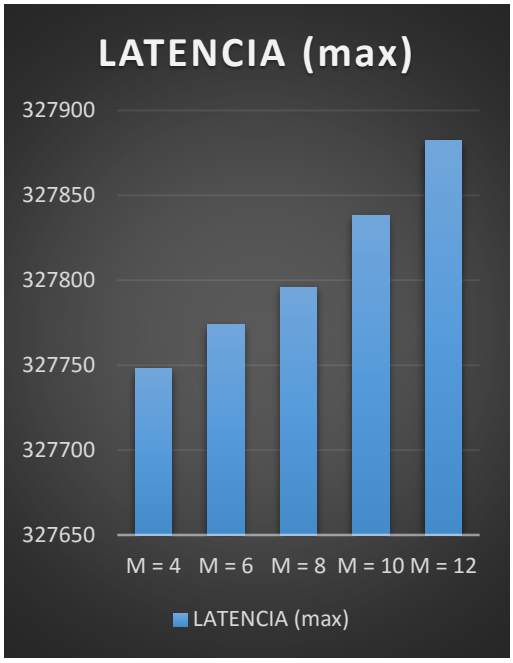
Utilization Estimates

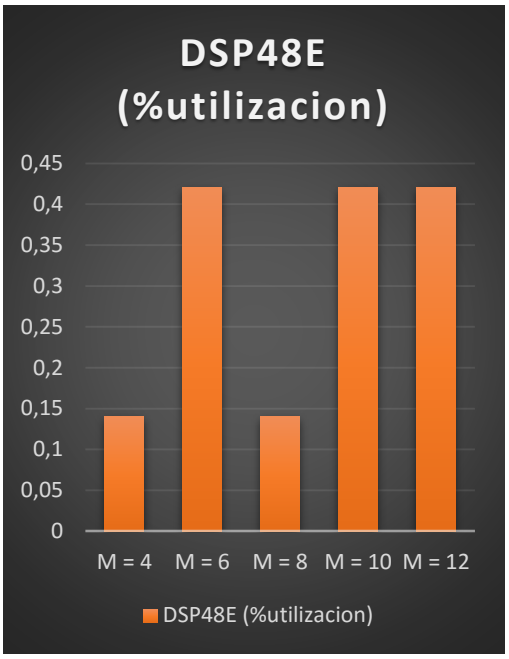
Summary

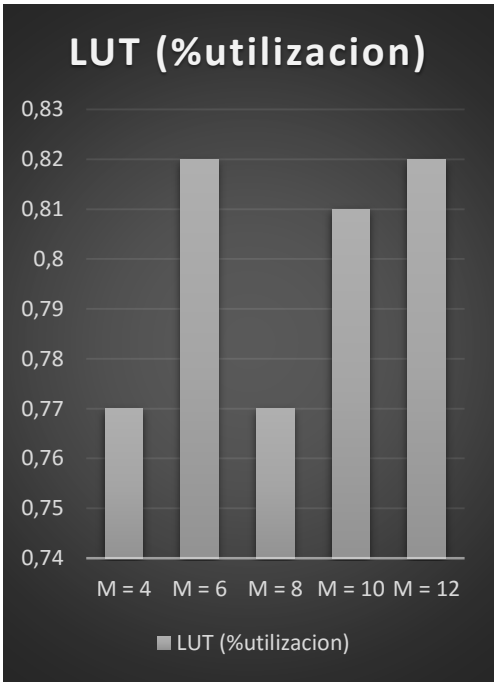
Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	463
FIFO	-	-	-	-
Instance	120	4	612	1250
Memory	237	-	0	0
Multiplexer	-	-	-	252
Register	-	-	401	-
Total	357	4	1013	1965
Available	2060	2800	607200	303600
Utilization (%)	17	~0	~0	~0

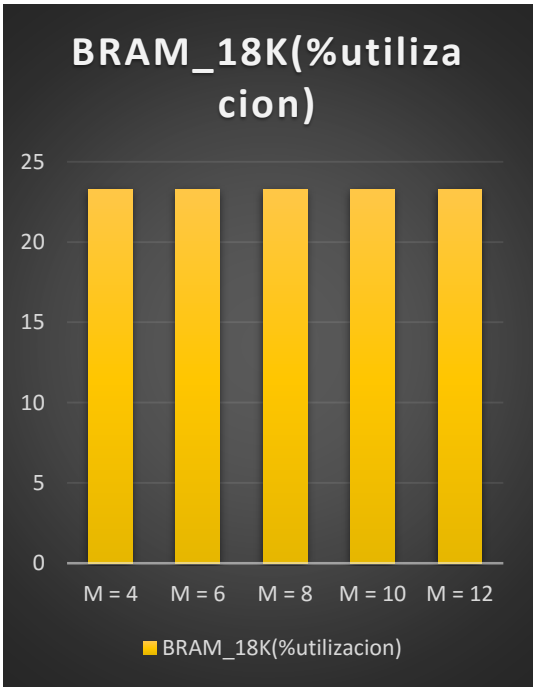
Donde la figura de la izquierda es el reporte correspondiente al diseño principal y el de la derecha, corresponde con la alternativa optimizada. Como podemos observar, al calcular la matriz completa gastamos 4 bloques BRAM más, los mismos DSP48 (lo cual tiene lógica porque se emplean los mismos recursos, pero durante más tiempo) y se incrementa el uso de LUTs. Hablando en tiempo, el tiempo de ejecución de la cosimulación para la opción donde se calcula la matriz completa es de unos 180 ms y en el caso en el que se calcula la triangular es de unos 148 ms.

También, dado que nuestro sistema es altamente parametrizable, se han realizado las síntesis para distintos paquetes de imágenes, obteniendo las siguientes gráficas:









La conclusión más inmediata que podemos sacar es que el aumento del número de imágenes provoca un aumento progresivo de la latencia. Sin embargo, si M es una potencia de 2, podemos observar que no aumenta ni el consumo de LUTs ni el de DSP48. Y, en cualquier caso, el número de bloques BRAM se mantiene constante para cualquiera de las distintas alternativas, dado que el tamaño de Ct que es donde realmente influyen los cambios de M, es prácticamente el mismo.

Por último, para terminar el apartado de resultados, se van a mostrar a continuación los reportes de la solución final implementada, es decir, con formato fixed point, M igual a 8, aplicando la directiva Pipeline a los bucles for internos y aplicando la directiva Interface a los puertos de entrada y salida, específicamente para que se implemente una interfaz AXI, ya que, tal y como se ha comentado en secciones anteriores, se implementa esta interfaz con vistas a un diseño futuro en el que esta interfaz es necesaria para conectar con un sistema de controlador de memoria DDR:

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	PCA	return value
ap_rst_n	in	1	ap_ctrl_hs	PCA	return value
ap_start	in	1	ap_ctrl_hs	PCA	return value
ap_done	out	1	ap_ctrl_hs	PCA	return value
ap_idle	out	1	ap_ctrl_hs	PCA	return value
ap_ready	out	1	ap_ctrl_hs	PCA	return value
imagen_V_TDATA	in	32	axis	imagen_V	pointer
imagen_V_TVALID	in	1	axis	imagen_V	pointer
imagen_V_TREADY	out	1	axis	imagen_V	pointer
Ct_s_V_TDATA	out	32	axis	Ct_s_V	pointer
Ct_s_V_TVALID	out	1	axis	Ct_s_V	pointer
Ct_s_V_TREADY	in	1	axis	Ct_s_V	pointer

Figura 2.41. Vista detallada del reporte de Vivado HLS en cuanto a Interfaz

Como podemos observar en la figura anterior, al aplicar la directiva Interface (Axis), se implementa un sistema mixto, donde el protocolo handshaking se emplea para controlar el funcionamiento básico del módulo, y el protocolo AXI para la entrada/salida de datos. Además, a pesar de que el tamaño de nuestras variables es de 30 bits, Vivado HLS implementa los puertos en el tamaño estándar más cercano sin que haya pérdida de información, probablemente porque es más óptimo y sencillo crear un puerto de 32 bits que uno de 30.

A continuación, se muestran los detalles del consumo específico de cada función en cuanto a latencia y recursos hardware consumidos.

Instance	Module	min	max	min	max	Type
grp_PCA_Generar_Matriz_Covarianza_fu_350	PCA_Generar_Matriz_Covarianza	2	196617	2	196617	none
grp_PCA_Calcular_Media_fu_364	PCA_Calcular_Media	1	131077	1	131077	none

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L_PCA_label6	65536	65536	1	1	1	65536	yes
- L_Resta_Media_label0	65536	65536	2	1	1	65536	yes
- L_PCA_label7	64	64	2	1	1	64	yes

Figura 2.42. Vista detallada del reporte de Vivado HLS en cuanto a latencias de las funciones internas

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
grp_PCA_Calcular_Media_fu_364	PCA_Calcular_Media	0	0	178	553
grp_PCA_Generar_Matriz_Covarianza_fu_350	PCA_Generar_Matriz_Covarianza	120	4	243	631
PCA_srem_32ns_5ns_32_36_seq_U11	PCA_srem_32ns_5ns_32_36_seq	0	0	256	304
Total	3	120	4	677	1488

Memory

Memory	Module	BRAM_18K	FF	LUT	Words	Bits	Banks	W*Bits*Banks
Ct_V_U	PCA_Ct_V	1	0	0	64	30	1	1920
M_V_U	PCA_Generar_Matriz_Covarianza_At_V	120	0	0	65536	30	1	1966080
imagen_aux_V_U	PCA_imagen_aux_V	120	0	0	65536	30	1	1966080
media_V_U	PCA_media_V	120	0	0	65536	30	1	1966080
Total	4	361	0	0	196672	120	4	5900160

Figura 2.43. Vista detallada del reporte de Vivado HLS en cuanto a recursos consumidos de las funciones internas

Por último, podemos concluir con que la herramienta Vivado HLS, supone un gran avance en el campo del diseño hardware, ya que ahorra mucho tiempo de diseño y generando unos resultados muy buenos e incluso mejorables teniendo un control avanzado de las directivas de optimización.

Es cierto que el defecto fundamental que tiene, que más bien es una limitación, es que en el momento en que finalizas tu diseño en Vivado HLS tienes que averiguarqué tipo de módulo te ha creado y lo más importante, cómo emplearlo. A pesar de ello, cuenta con la cosimulación del diseño RTL, que sirve como apoyo para el diseñador para entender la implementación que se ha realizado.

Otra ventaja muy importante de esta herramienta es que nos permite realizar diseños parametrizables, forma que si queremos cambiar cualquier parámetro de nuestro sistema lo podemos hacer en un corto período de tiempo. Sin embargo, si se tratase de un diseño puramente hardware, una modificación de este tipo podría suponer tener que realizar casi totalmente el diseño de nuevo.

Trabajos futuros

La continuación de este proyecto podríamos dividirla en diferentes partes:

- En primer lugar, sería conveniente realizar un estudio mucho más exhaustivo de las directivas de optimización a aplicar en Vivado HLS, como por ejemplo las directivas Dependence, Array_Reshape, etc. También sería conveniente explorar nuevos algoritmos para multiplicar matrices o intentar optimizar el código empleado para implementarlo.

- También, aunque no afecta para nada al comportamiento del diseño, se debería de indagar en la forma de generar un tipo de dato, fix por ejemplo, con el cual poder parametrizar también los datos fixed point.

- Por otra parte, se debería analizar exhaustivamente el tamaño máximo que puede llegar a tener una variable, ya que durante este proyecto, para simplificarlo, se ha puesto a todas las variables con un ancho de palabra de 30 bits, con 20 bits para la parte decimal.

- En segundo lugar, en caso de que el usuario necesite que la matriz de covarianza esté completa, se podría generar un pequeño módulo nuevo que genere la matriz completa a partir de la triangular que genera el módulo implementado en este proyecto.

- Otra alternativa interesante, sería implementar el mismo módulo en un SystemOnChip (SOC) para poder comparar ambas plataformas hardware y sacar conclusiones sobre la que éstas últimas brinda.

- También se debería de cambiar el diseño realizado durante este proyecto, se debería crear un puerto nuevo para la media de las imágenes, para que se asemeje al sistema completo diseñado por el Dr. Bravo.

- Por último, relacionado con el punto anterior, se debe investigar el periférico Memory Interface Generator (MIG) ya que es el IP que Vivado nos brinda a modo de controlador de memoria DDR, lo cual es muy importante para el correcto desarrollo de este proyecto debido a que en una FPGA los recursos hardware relacionados con la memoria son extremadamente limitados y lo más conveniente sería guardar elementos como la media y la matriz de covarianza en memoria, para poder utilizarlas en la fase online sin necesidad de consumir más memoria.

3. Planos y diagramas

Introducción:

Esta sección está reservada para mostrar aquellos códigos fuentes que se ha creído necesario, con el fin de clarificar tanto el diseño implementado durante este proyecto, como la forma en que se ha realizado.

Escritura de las imágenes en un fichero de texto y análisis del error:

Entrada de datos en Vivado HLS:

MATLAB:

Como se ha comentado en secciones anteriores, Vivado HLS posee librerías para trabajar en formato fixed point. En ellas, existe una función que convierte automáticamente y de forma transparente para el usuario los datos de tipo double en tipo fixed según el ancho de palabra que hayamos fijado. Por lo tanto, basta con escribir en un fichero los datos en formato double, el cual es el más inmediato de hacer en MATLAB, para, a continuación, que Vivado HLS los lea. El código empleado para ello ha sido el siguiente:

Adquisicion_de_imagenes.m:

```
1  -    imag(:,:,1) = imread('D:\TFG\Fotos\image019_00000.bmp');
2  -    imag(:,:,2) = imread('D:\TFG\Fotos\image019_00001.bmp');
3  -    imag(:,:,3) = imread('D:\TFG\Fotos\image019_00002.bmp');
4  -    imag(:,:,4) = imread('D:\TFG\Fotos\image019_00003.bmp');
5  -    imag(:,:,5) = imread('D:\TFG\Fotos\image019_00004.bmp');
6  -    imag(:,:,6) = imread('D:\TFG\Fotos\image019_00005.bmp');
7  -    imag(:,:,7) = imread('D:\TFG\Fotos\image019_00006.bmp');
8  -    imag(:,:,8) = imread('D:\TFG\Fotos\image019_00007.bmp');
9
10 -    ima(:,:,:)  = double(imag(:,:,:));
11 -    im(:,:,:)   = mat2gray(imagen(:,:,:));
12
13 -  for k= 1:N_IMAGENES
14 -      for i = 1:MAT_IMAGEN_ROWS
15 -          for j = 1:MAT_IMAGEN_COLS
16 -              dlmwrite('imagenes_input.txt',imagen(i,j,k),'-append','delimiter','','roffset',0);
17 -          end
18 -      end
19 -  end
```

Figura 3.44. Vista detallada del código para escribir las imágenes en un fichero en formato double MATLAB

Vivado HLS:

Una vez conseguimos escribir todos los valores de las M imágenes, el siguiente paso es generar un testbench en Vivado HLS para poder trabajar con ellas. EL código empleado ha sido el siguiente:

PCA_test.cpp:

```
#include <iostream>
#include "PCA.h"
#include <fstream>
#include <string>

using namespace std;

int main(int argc, char **argv)
 {
    int i=0,j=0,k=0;
    ap_fixed<30,10,AP_RND> imagen [MAT_IMAGEN_ROWS][MAT_IMAGEN_COLS];
    ap_fixed<30,10,AP_RND> Ct [N_IMAGENES][N_IMAGENES];

    ifstream leer_fichero_1("D:\\TFG\\imagenes_input.txt");
    for (i=0;i<((N_IMAGENES*N_IMAGENES)+N_IMAGENES+1);i++)
    {
                if(!leer_fichero_1)
                    cout<<"No se ha podido abrir el fichero";
                else{
                    for (j=0;j<MAT_IMAGEN_ROWS;j++){
                        for (k=0;k<MAT_IMAGEN_COLS;k++){
                                leer_fichero_1>>imagen[j][k];
                        }
                    }
                }
        PCA (imagen,Ct);
    }
    leer_fichero_1.close();
```

Figura 3.45. Adquisición de imágenes en Vivado HLS

En él, leemos una imagen en cada iteración del bucle for y la enviamos como entrada al top de nuestra función, así como la salida donde va a escribirse el resultado final.

Por último, una vez se han calculado los valores de Ct, se ha empleado el siguiente código para escribirlo en un fichero y realizar el estudio del error cometido al emplear datos de tipo fixed, tal y como se ha comentado en secciones anteriores:

```
  ofstream escribir_imagen("D:\\TFG\\Ct_output_fixed.txt");
   if(!escribir_imagen)
           cout<<"No se ha podido abrir el fichero";
   else{
          for(int i = 0; i < N_IMAGENES; i++) {
              for(int j = 0; j < N_IMAGENES; j++) {
                  escribir_imagen<<Ct[i][j];
                  escribir_imagen<<" ";
                  }
              escribir_imagen<<endl;
          }
          escribir_imagen<<endl;
   }
     escribir_imagen.close();
return 0;
}
```

Figura 3.46. Vista detallada del código para escribir los elementos de C^T en un fichero en Vivado HLS

Dicho fichero se leerá en MATLAB empleando el siguiente código:

Ver_error_medio.m

```
Ct_read=dlmread('Ct_output_fixed.txt');
error_Ct_fixed=abs((Ct)-Ct_read)./abs((Ct));
err_max_Ct_fixed=max(max(error_Ct_fixed))*100;
```

Figura 3.47. Vista detallada del código para leer la matriz C^T en MATLAB y obtener el error

Diseño altamente parametrizable:

Vivado HLS:

Como podemos observar en el siguiente .h, Vivado HLS permite generar diseños parametrizables, es decir, cambiando el valor de un variable podemos modificar el tamaño de las imágenes de entrada, así como el del número de imágenes de entrada:

PCA.h:

```
#include <cmath>
#include <iostream>
#include "ap_fixed.h"

#define MAT_IMAGEN_ROWS 256
#define MAT_IMAGEN_COLS 256
#define N_IMAGENES 8

using namespace std;

typedef ap_fixed<30,10,AP_RND> imagen_t;
typedef ap_fixed<30,10,AP_RND> Ct_t;
typedef ap_fixed<30,10,AP_RND> fix;

// Prototype of top level function for C-synthesis
void PCA(
        imagen_t imagen[MAT_IMAGEN_ROWS][MAT_IMAGEN_COLS],Ct_t Ct_s[N_IMAGENES][N_IMAGENES]);
```

Figura 3.48. Alto grado de parametrización en Vivado HLS

PCA.cpp:

```
66 void PCA(imagen_t imagen[MAT_IMAGEN_ROWS][MAT_IMAGEN_COLS],Ct_t Ct_s[N_IMAGENES][N
67 )
68 {
69     ap_fixed<30,10,AP_RND> M[MAT_IMAGEN_ROWS][MAT_IMAGEN_COLS];
70     static ap_fixed<30,10,AP_RND> media [MAT_IMAGEN_ROWS][MAT_IMAGEN_COLS];
71     static ap_fixed<30,10,AP_RND> Ct [N_IMAGENES][N_IMAGENES];
72     static ap_fixed<30,10,AP_RND> imagen_aux [MAT_IMAGEN_ROWS][MAT_IMAGEN_COLS];
73     int i=0,j=0;
74     static int aux_1,aux_2,aux_3,t1=0,t2=0,calcula_at=1,calcula_ct=1;
75
76     aux_1++;
77     if (aux_2==0)
78     {
79         Calcular_Media (imagen_aux,aux_1,media);
80     }
81     if (aux_2==1)
82     {
83       Resta_Media (imagen_aux,media,M);
84       if ((aux_3%(N_IMAGENES+1)==0)&&(aux_3!=0))
85       {
86             calcula_at=1;
87             t1++;
88             t2=t1;
89             calcula_ct=1;
90       }
91       if ((aux_3%N_IMAGENES==0)&&(aux_3!=0))
92         {
93             calcula_ct=0;
94
95         }
96         Generar_Matriz_Covarianza (M,Ct,calcula_at,calcula_ct,t1,t2);
97         if (calcula_ct==1)
98         {
99             t2++;
```

Figura 3.49. Código función top

```
97          if (calcula_ct==1)
98          {
99              t2++;
100         }
101         calcula_at=0;
102         aux_3++;
103     }
104     if (aux_1==N_IMAGENES)
105     {
106         aux_2=1;
107     }
108     if (aux_1 == ((N_IMAGENES*N_IMAGENES)+N_IMAGENES+1))
109     {
110         for (i=0;i<N_IMAGENES;i++)
111         {
112             PCA_label7:for (j=0;j<N_IMAGENES;j++)
113             {
114                 Ct_s[i][j]=Ct[i][j];
115             }
116         }
117     }
118 }
```

Figura 3.50. Código función top

Fase de debug en Vivado:

Tal y como hemos comentado anteriormente, Vivado HLS resulta muy útil para diseñar módulos con un alto nivel de complejidad a partir de lenguaje C/C++, sin embargo, lo que ocurre de puertos para fuera sigue siendo hardware, así que un punto importante de este proyecto ha sido comprobar cómo trabajar en un entorno hardware a partir de un módulo generado por software. Para ello se decidió emplear el mismo módulo generado en Vivado HLS, como si fuera un IP del catálogo de cores de Vivado y generar la misma simulación que la cosimulación que genera Vivado HLS.

El principal problema al que se ha hecho frente es a la entrada de datos, ya que, mientras Vivado HLS tiene funciones para transformar datos de tipo doublé en fixed, así como MATLAB puede escribir datos en formato double en ficheros, en Vivado todas estas comodidades desaparecen. Por lo que se llegó a la siguiente solución:

Escritura en MATLAB de datos de tipo fixed point en ficheros:

MATLAB:

Para realizarlo se empleó el siguiente código:

```
50 -  for k = 1:N_IMAGENES+1
51 -      for i = 1:MAT_IMAGEN_ROWS
52 -          for j = 1:MAT_IMAGEN_COLS
53
54 -              a = imagen (i,j,3);
55 -              dlmwrite('imagen3_32bits_8.txt',a.bin,'-append','delimiter','','roffset',0);
56
57 -          end
58 -      end
59 -  end
```

Figura 3.51. Escritura de datos de tipo fixed point en MATLAB

Donde cada imagen se guarda en un fichero distinto un número M de veces, de forma que en el testbench que se explicará a continuación podamos realizar la sincronización completa de los datos de entrada para conseguir la secuencia cíclica de imágenes que conseguíamos en el testbench de Vivado HLS.

La forma en que se solucionó el problema de escribir en formato fixed point fue la siguiente: escribir el dato en el formato internacional de la programación, en binario. De esta forma a nuestro módulo le entran un conjunto de ceros y unos y ya se encarga él de trabajar con ellos. Este razonamiento parte del hecho de que el formato fixed point no es más que una reinterpretación del código binario, donde nosotros colocamos la

coma donde nos conviene. Es decir, lo importante es el contenido, no cómo lo interpretemos.

Lectura y escritura sobre ficheros de datos en formato fixed point en VHDL y testbench empleado en Vivado:

Vivado:

El código para lograr lo expuesto en la sección anterior es el siguiente:

```
22  library ieee;
23  use ieee.std_logic_1164.all;
24  use ieee.numeric_std.all;
25  library std;
26  use std.textio.all;
27  ----------------------------------------------------------------------
28
29  entity design_2_wrapper_tb is
30
31  end entity design_2_wrapper_tb;
32
33  ----------------------------------------------------------------------
34
35  architecture skeleton of design_2_wrapper_tb is
36
37    -- component ports
38    signal Ct_s_V_TDATA    : STD_LOGIC_VECTOR (31 downto 0);
39    signal Ct_s_V_TREADY   : STD_LOGIC;
40    signal Ct_s_V_TVALID   : STD_LOGIC;
41    signal ap_clk          : STD_LOGIC := '0';
42    signal ap_done         : STD_LOGIC;
43    signal ap_idle         : STD_LOGIC;
44    signal ap_ready        : STD_LOGIC;
45    signal ap_rst_n        : STD_LOGIC;
46    signal ap_start        : STD_LOGIC;
47    signal imagen_V_TDATA  : STD_LOGIC_VECTOR (31 downto 0);
48    signal imagen_V_TREADY : STD_LOGIC;
49    signal imagen_V_TVALID : STD_LOGIC;
50    signal imagen_V_TVALID_r : STD_LOGIC;
51    signal imagen_V_TVALID_rr : STD_LOGIC;
52    signal imagen1_V_TDATA : STD_LOGIC_VECTOR (31 downto 0);
53    signal imagen2_V_TDATA : STD_LOGIC_VECTOR (31 downto 0);
54    signal imagen3_V_TDATA : STD_LOGIC_VECTOR (31 downto 0);
55    signal imagen4_V_TDATA : STD_LOGIC_VECTOR (31 downto 0);
56    signal imagen5_V_TDATA : STD_LOGIC_VECTOR (31 downto 0);
57    signal imagen6_V_TDATA : STD_LOGIC_VECTOR (31 downto 0);
58    signal imagen7_V_TDATA : STD_LOGIC_VECTOR (31 downto 0);
59    signal imagen8_V_TDATA : STD_LOGIC_VECTOR (31 downto 0);
60    signal slc : unsigned (2 downto 0);
61    signal imagen1_V_TVALID : STD_LOGIC;
62    signal imagen2_V_TVALID : STD_LOGIC;
63    signal imagen3_V_TVALID : STD_LOGIC;
64    signal imagen4_V_TVALID : STD_LOGIC;
65    signal imagen5_V_TVALID : STD_LOGIC;
```

Figura 3.52. Testbench para la simulación en Vivado

```
74    -- component instantiation
75    DUT: entity work.design_2_wrapper
76      port map (
77        Ct_s_V_TDATA    => Ct_s_V_TDATA,
78        Ct_s_V_TREADY   => Ct_s_V_TREADY,
79        Ct_s_V_TVALID   => Ct_s_V_TVALID,
80        ap_clk          => ap_clk,
81        ap_done         => ap_done,
82        ap_idle         => ap_idle,
83        ap_ready        => ap_ready,
84        ap_rst_n        => ap_rst_n,
85        ap_start        => ap_start,
86        imagen_V_TDATA  => imagen_V_TDATA,
87        imagen_V_TREADY => imagen_V_TREADY,
88        imagen_V_TVALID => imagen_V_TVALID);
89
90    -- clock generation
91    ap_clk <= not ap_clk after 10 ns;
92
93    -- waveform generation
94    WaveGen_Proc: process
95    begin
96      -- insert signal assignments here
97      ap_rst_n <= '0';
98      wait for 125 ns;
99      ap_rst_n <= '1';
100     Ct_s_V_TREADY <= '1';
101     ap_start <= '1';
102     wait;
103     wait;
104   end process WaveGen_Proc;
105
106   process (ap_clk, ap_rst_n) is
107   begin  -- process
108     if ap_rst_n = '0' then                -- asy
109       slc <= "000";
110     elsif ap_clk'event and ap_clk = '1' then
111       if ap_done = '1' then
112         slc <= slc + 1;
113       end if;
114     end if;
115   end process;
116
```

Figura 3.53. Testbench para la simulación en Vivado

```
106   process (ap_clk, ap_rst_n) is
107   begin  -- process
108     if ap_rst_n = '0' then              -- asynchronous
109       slc <= "000";
110     elsif ap_clk'event and ap_clk = '1' then  -- rising
111       if ap_done = '1' then
112         slc <= slc + 1;
113       end if;
114     end if;
115   end process;
116
117   process (ap_clk, ap_rst_n) is
118     file f : TEXT open read_mode is "D:\TFG\Imagen1_32bi
119     variable linea  : line;
120     variable entrada : bit_vector (31 downto 0);
121     variable aux : unsigned(16 downto 0)  := "0000000000
122   begin  -- process
123     if ap_rst_n = '0' then              -- asynchronous
124       imagen1_V_TDATA <= X"00000000";
125       imagen1_V_TVALID <= '0';
126     elsif ap_clk'event and ap_clk = '1' then  -- rising
127       if slc = "000" then
128         if not(endfile(f)) then
129           imagen1_V_TVALID <='1';
130          readline(f,linea);
131          read (linea,entrada);
132          imagen1_V_TDATA <= to_stdlogicvector(entrada);
133          aux := aux + 1;
134         end if;
135         if aux = 65536 then
136           imagen1_V_TVALID <= '0';
137           aux := "00000000000000000";
138         end if;
139       end if;
140     end if;
141   end process;
142
143   process (ap_clk, ap_rst_n) is
144     file f : TEXT open read_mode is "D:\TFG\Imagen2_32bi
145     variable linea  : line;
146     variable entrada : bit_vector (31 downto 0);
147     variable aux : unsigned(16 downto 0)  := "0000000000
148   begin  -- process
149     if ap_rst_n = '0' then              -- asynchronous
```

Figura 3.54. Testbench para la simulación en Vivado

```
324
325     imagen_V_TDATA <=  imagen1_V_TDATA when slc = "000" else
326                        imagen2_V_TDATA when slc = "001" else
327                        imagen3_V_TDATA when slc = "010" else
328                        imagen4_V_TDATA when slc = "011" else
329                        imagen5_V_TDATA when slc = "100" else
330                        imagen6_V_TDATA when slc = "101" else
331                        imagen7_V_TDATA when slc = "110" else
332                        imagen8_V_TDATA when slc = "111";
333
334     imagen_V_TVALID_r <= imagen1_V_TVALID when slc = "000" else
335                          imagen2_V_TVALID when slc = "001" else
336                          imagen3_V_TVALID when slc = "010" else
337                          imagen4_V_TVALID when slc = "011" else
338                          imagen5_V_TVALID when slc = "100" else
339                          imagen6_V_TVALID when slc = "101" else
340                          imagen7_V_TVALID when slc = "110" else
341                          imagen8_V_TVALID when slc = "111";
342
343      process (ap_clk, ap_rst_n) is
344      begin  -- process
345        if ap_rst_n = '0' then               -- asynchronous reset
346          imagen_V_TVALID_rr <= '0';
347        elsif ap_clk'event and ap_clk = '1' then  -- rising clock
348           imagen_V_TVALID_rr <= imagen_V_TVALID_r;
349        end if;
350      end process;
351
352      process (ap_clk, ap_rst_n) is
353        file f1 : TEXT open write_mode is "D:\TFG\Ct_salida.txt";;
354       variable linea  : line;
355       variable dato1 : integer;
356       variable entrada : bit_vector (31 downto 0);
357      begin  -- process
358        if ap_rst_n = '0' then               -- asynchronous reset
359         entrada <= X"00000000";
360        elsif ap_clk'event and ap_clk = '1' then  -- rising clock
361           if ap_none = '1' then
362             entrada <= to_bitvector (Ct_s_V_TDATA);
363             write (linea, entrada);
364             writeline(f1,linea);
365           end if;
366        end if;
367      end process;
```

Figura 3.55. Testbench para la simulación en Vivado

4. Pliego de condiciones

En este capítulo se muestran los requisitos hardware y software que han sido necesarios para la realización de este proyecto.

4.1. Requisitos hardware

- Ordenador personal (portátil) Asus GL552J
  - Procesador Intel CORE i7-4720 HQ.
  - 16Gb de memoria RAM.
  - Sistema operativo de 64 bits.

4.2. Requisitos Software

Requisitos generales:

- Sistema operativo Windows 10.
- Editor de textos NotePad++ o alguno similar.

Requisitos específicos:

- Vivado Desing Suite 14.4, incluyendo la herramienta de síntesis de alto nivel Vivado HLS
- Matlab 2015a, incluyendo la toolbox de coma fija y la de procesamiento de imágenes.

Para la redacción de la memoria:

- Word 2016.
- Adobe Acrobat Reader DC.

5. Presupuesto

Podemos dividir el coste total en los siguientes grupos:

Recursos software				
Concepto	Coste	Amortización (años)	Tiempo de uso (meses)	Total
Matlab 2015a	2.000€	4	5	208,33€
Vivado System Edition	3.495€	4	5	364,06€
NotePad++	0€	-	5	0€
Word 2016 (Licencia Universitaria)	0€	-	5	0€
Adobe Acrobat Reader DC	0€	-	5	0€
Total				572,39€

Tabla 5.1. Coste de los recursos hardware.

Recursos hardware				
Concepto	Coste	Amortización (años)	Tiempo de uso (meses)	Total
Asus GL552J	1.200€	4	5	125€
Total				125€

Tabla 5.2. Coste de los recursos hardware.

Mano de obra			
Concepto	Coste por hora	Horas	Total
Ejecución	30€/h	200	3.000€
Redacción	12€/h	100	120€
Total			3.120€

Tabla 5.3. Coste de la mano de obra.

Material fungible y otros costes	
Concepto	Coste
DVD-RW	1,5€
Impresión y encuadernación de libros	80€
Total	81,5€

Tabla 5.4. Coste del material fungible.

A partir de los datos expuestos, se procede a calcular el coste total del proyecto. El *coste de ejecución por contrata* se calcula a partir de la siguiente ecuación:

$$PEC = CM + b_{\%}CM$$

Donde PEC es el *presupuesto de ejecución por contrata*, CM es el coste de ejecución material del proyecto, tomado a partir de la suma de los totales de los costes de los recursos hardware, software, mano de obra y material fungible, y $b_{\%}$ el porcentaje del coste material del proyecto tomado como beneficio industrial.

El coste total de ejecución resulta:

$$CM = 572{,}39 + 125 + 3.120 + 81{,}5 = 3898.89€$$

Suponiendo que el $b_{\%}$ es un 30%, el presupuesto de ejecución por contrata será:

$$PEC = 5068.56\ €$$

El cose final del proyecto asciende a CINCO MIL SESENTA Y OTRO CON CINCUENTA Y SEIS CENTIMOS**.**

6. Bibliografía

- Minitab 17. “Tutorial sobre matrices de covarianza”. http://support.minitab.com/es-mx/minitab/17/topic-library/modeling-statistics/anova/anova-statistics/what-is-the-variance-covariance-matrix/ [última visita septiembre 2016]
- J.M. Marín. “Tema 2. Estadística Descriptiva Multivariante”. Universidad Carlos 3 Madrid. http://halweb.uc3m.es/esp/Personal/personas/jmmarin/esp/AMult/tema2am.pdf [última visita septiembre 2016]
- National Instruments. “Introducción a la Tecnología FPGA”. http://www.ni.com/white-paper/6984/es/ [última visita septiembre 2016]
- Xilinx.” Virtex-7 FPGAs Product Advantage” http://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html [última visita septiembre 2016]
- Xilinx. “7 Series FPGAs Overview“ http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf [última visita septiembre 2016]
- Mathworks. “Image Processing Toolbox” http://es.mathworks.com/products/image/ [última visita septiembre 2016]
- Xilinx. “Vivado Design Suite User Guide. High-Level Synthesis” http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug902-vivado-high-level-synthesis.pdf [última visita septiembre 2016]

7. Manual de usuario

A continuación se muestran todos los pasos necesarios para llevar a cabo la ejecución del Generador de Matrices de Covarianza.

MATLAB

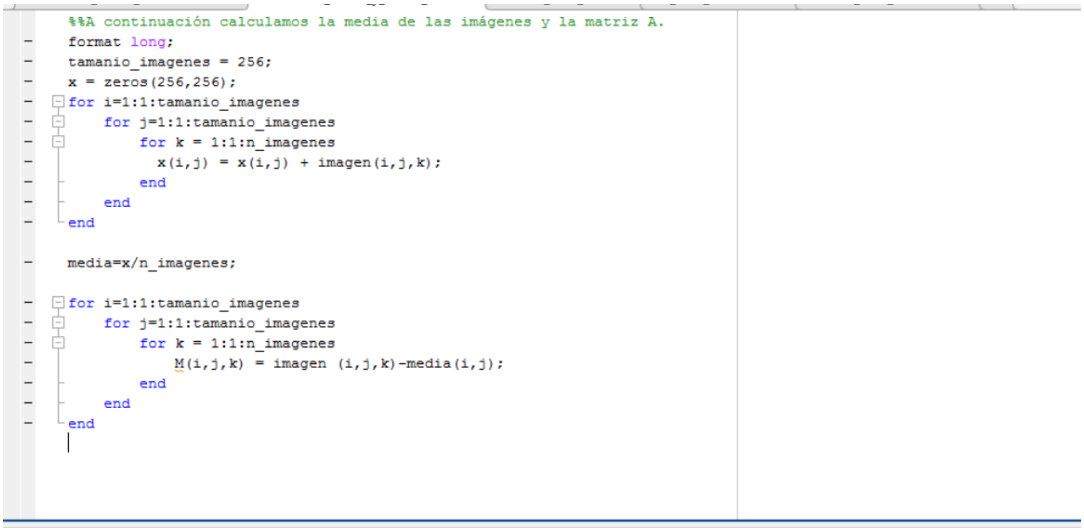
Dentro del directorio de MATLAB se encuentran los siguientes scripts, además de los archivos que se generan con ellos:

- Adquisición_de_imágenes: en él, leemos las imágenes a través de la ruta donde se encuentren en el ordenador y a continuación generamos M ficheros de texto donde guardamos cada una de las imágenes un número M de veces, en formato fixed point:

```
clear all;
%%En primer lugar obtenemos las imagenes de entrada:
n_imagenes = 8;
imag(:,:,1) = imread('D:\TFG\Fotos\image019_00000.bmp');
imag(:,:,2) = imread('D:\TFG\Fotos\image019_00001.bmp');
imag(:,:,3) = imread('D:\TFG\Fotos\image019_00002.bmp');
imag(:,:,4) = imread('D:\TFG\Fotos\image019_00003.bmp');
imag(:,:,5) = imread('D:\TFG\Fotos\image019_00004.bmp');
imag(:,:,6) = imread('D:\TFG\Fotos\image019_00005.bmp');
imag(:,:,7) = imread('D:\TFG\Fotos\image019_00006.bmp');
imag(:,:,8) = imread('D:\TFG\Fotos\image019_00007.bmp');

ima(:,:,:) = double(imag(:,:,:));
im(:,:,:)=mat2gray(ima(:,:,:));
imagen (:,:,:) = im (:,:,:);

imagen(:,:,:) = fi(im(:,:,:),1,32,20);
for k = 1:N_IMAGENES+1
    for i = 1:MAT_IMAGEN_ROWS
        for j = 1:MAT_IMAGEN_COLS

            a = imagen (i,j,1);
            dlmwrite('imagen3_32bits_1.txt',a.bin,'-append','delimiter','','roffset',0);

        end
```

Figura 7.56. Adquisición_de_imágenes

- Obtención_media_y_matriz_A: donde obtenemos el valor de la media y generamos el paquete inicial de imágenes de entrada una vez los restamos con la media:



```
%%A continuación calculamos la media de las imágenes y la matriz A.
format long;
tamanio_imagenes = 256;
x = zeros(256,256);
for i=1:1:tamanio_imagenes
    for j=1:1:tamanio_imagenes
        for k = 1:1:n_imagenes
          x(i,j) = x(i,j) + imagen(i,j,k);
        end
    end
end

media=x/n_imagenes;

for i=1:1:tamanio_imagenes
    for j=1:1:tamanio_imagenes
        for k = 1:1:n_imagenes
            M(i,j,k) = imagen (i,j,k)-media(i,j);
        end
    end
end
```

Figura 7.57. Obtención_media_y_matriz_A

- Obtención_matriz_covarianza: en el obtenemos los valores de Ct:

```
%%Obtenemos la matriz de Covarian:
M1 = M (:,:,1);
M2 = M (:,:,2);
M3 = M (:,:,3);
M4 = M (:,:,4);
M5 = M (:,:,5);
M6 = M (:,:,6);
M7 = M (:,:,7);
M8 = M (:,:,8);

At(1,:)=M1(:);
At(2,:)=M2(:);
At(3,:)=M3(:);
At(4,:)=M4(:);
At(5,:)=M5(:);
At(6,:)=M6(:);
At(7,:)=M7(:);
At(8,:)=M8(:);
A = At';

Ct = 1/n_imagenes *(At*A);
```

Figura 7.58. Obtención_matriz_covarianza

- Estudio_precisión: en el realizamos un estudio inicial del error al emplear datos de tipo fixed point:

```
a=[1:16];
b=[0:15];

for i=1:16
    x_fi = fi(x,1,b(i)+5,b(i));
    media_fi= fi(x_fi/n_imagenes,1,b(i)+1,b(i));

    for l = 1:1:n_imagenes
        M_fi(:,:,l) = fi(imagen(:,:,l)-media_fi(:,:),1,b(i)+1,b(i));
    end

    M1_fi = M_fi (:,:,1);
    M2_fi = M_fi (:,:,2);
    M3_fi = M_fi (:,:,3);
    M4_fi = M_fi (:,:,4);
    M5_fi = M_fi (:,:,5);
    M6_fi = M_fi (:,:,6);
    M7_fi = M_fi (:,:,7);
    M8_fi = M_fi (:,:,8);

    At_fi(1,:)=M1_fi(:);
    At_fi(2,:)=M2_fi(:);
    At_fi(3,:)=M3_fi(:);
    At_fi(4,:)=M4_fi(:);
    At_fi(5,:)=M5_fi(:);
    At_fi(6,:)=M6_fi(:);
    At_fi(7,:)=M7_fi(:);
    At_fi(8,:)=M8_fi(:);
    A_fi = At_fi';
    Ct_fi = fi(1/n_imagenes *(At_fi*A_fi),1,b(i)+5,b(i));
    error(:,:,i) =double(abs(Ct-Ct_fi)./abs(Ct));
    error_max(i) = max(max(error(:,:,i)))*100;
    clear M_fi
    clear At_fi
end
plot (b,error_max,'b')
```

Figura 7.59. Estudio_precision

- Ver_error_medio: Con él obtenemos el error medio empleando la clase fixed de Vivado HLS:

Ver_error_media.m

```
Ct_read=dlmread('Ct_output_fixed.txt');
error_Ct_fixed=abs((Ct)-Ct_read)./abs((Ct));
err_max_Ct_fixed=max(max(error_Ct_fixed))*100;
```

Figura 7.60. Ver_error_media

Vivado HLS

En Vivado HLS se ha desarrollado íntegramente nuestro Generador de Matrices de Covarianza. Para ello, se han seguido los siguientes pasos:

En primer lugar, iniciamos Vivado HLS, para ello, hacemos doble clic sobre el acceso directo que se debe haber creado durante el proceso de instalación, o bien, hacemos doble clic en: Inicio -> Todos los programas -> Xilinx Design Tools -> Vivado HLS 2014.4. Se abrirá la siguiente ventana:

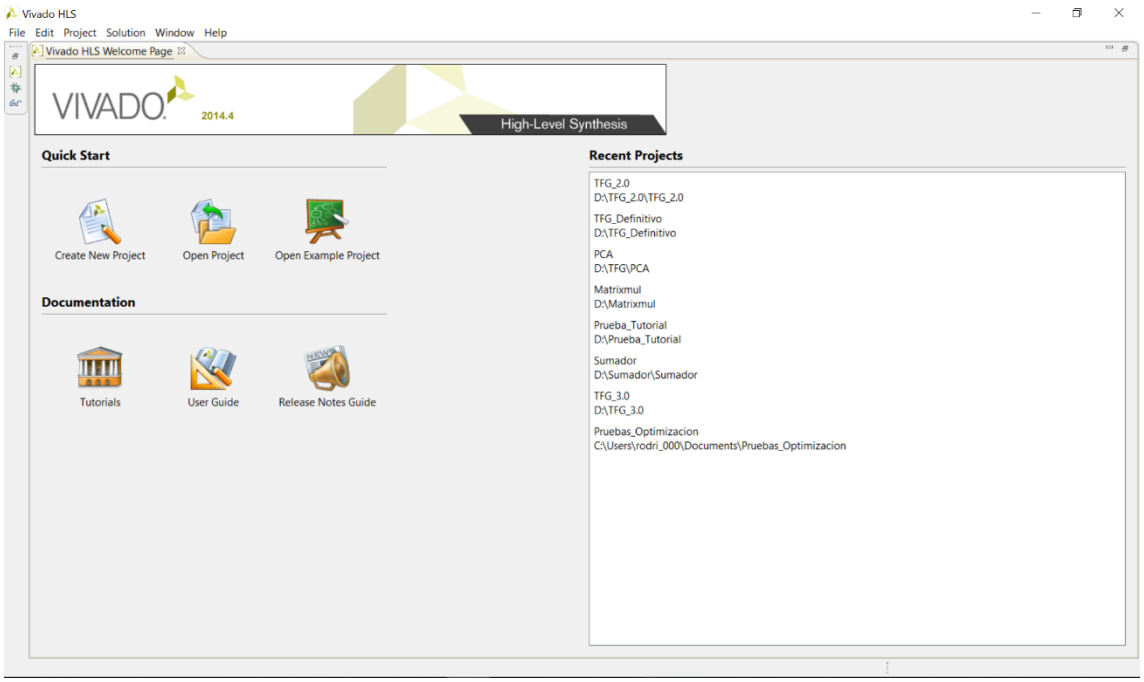


Figura 7.61. Pantalla de inicio de Vivado HLS

A continuación, hay que hacer clic en Create New Project y se nos abrirá la siguiente ventana:

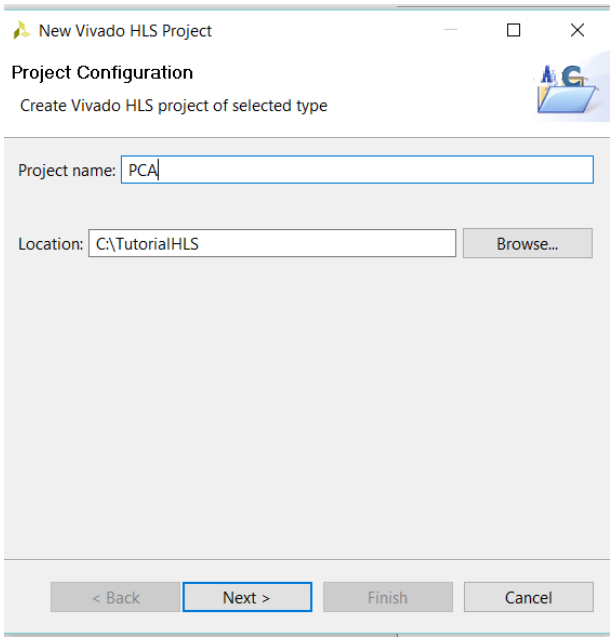


Figura 7.62. Creación de un proyecto en Vivado HLS

Seleccionamos las fuentes que encontraremos en el directorio raíz de nuestro proyecto y especificamos como función top del diseño PCA:

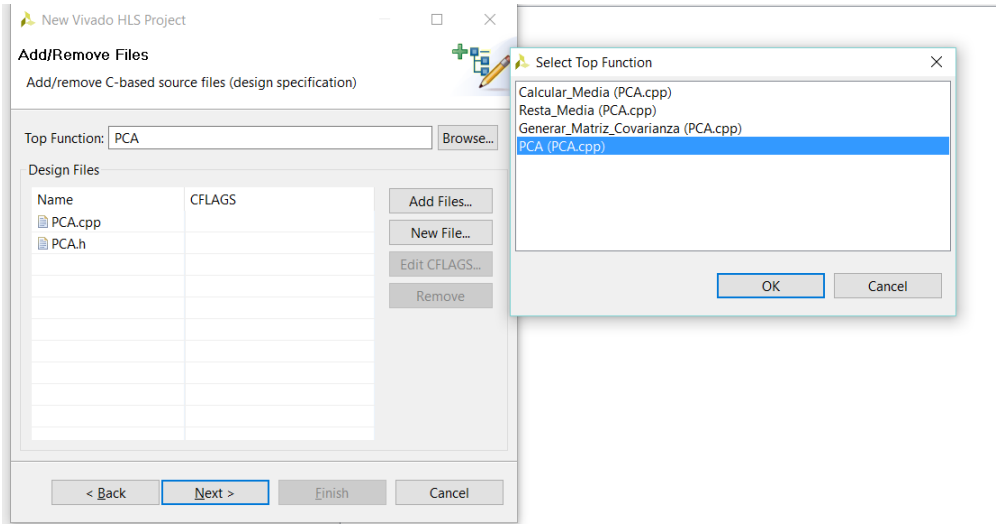


Figura 7.63. Incluimos el código fuente en Vivado HLS

Y añadimos el archivo que hará de testbench:

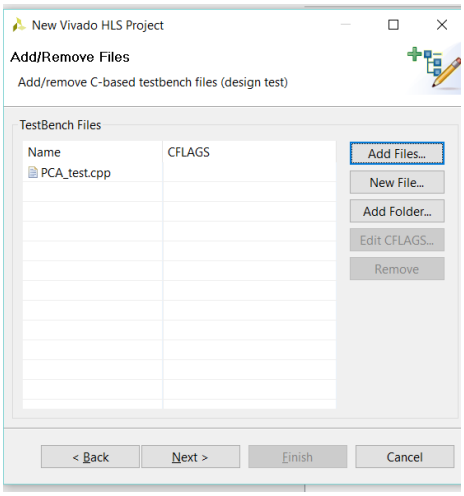


Figura 7.64. Incluimos el testbench en Vivado HLS

En la siguiente ventana, especificamos el período de reloj deseado a 10 ns y fijamos el dispositivo para el que va a ser implementado, en nuestro caso es una Virtex 7, modelo: xc7vx485tffg1927-1.

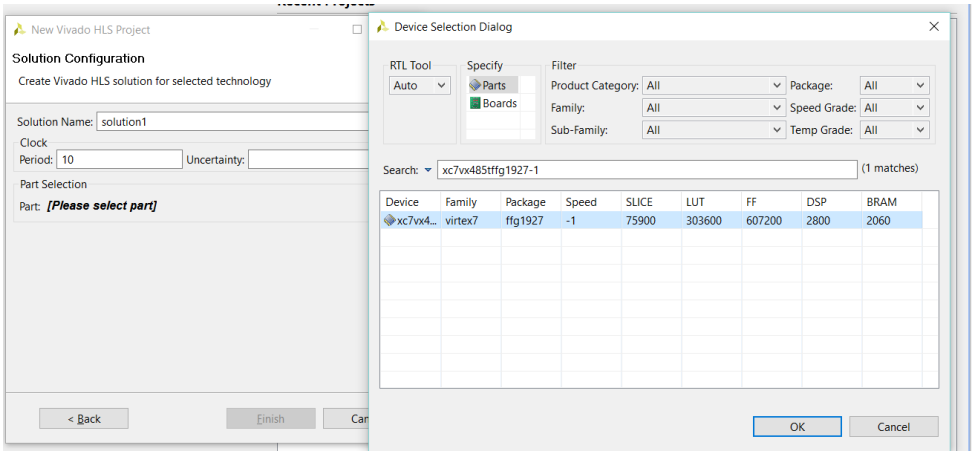


Figura 7.65 Seleccionamos el tipo de tarjeta para la que queremos realizar la implementación

En este momento ya tenemos nuestro proyecto creado y listo para trabajar con él. El primer paso aconsejable es que realicemos una compilación de nuestro código fuente, para ello, seleccionamos el icono . Se abrirá la perspectiva de Debug y bastará con clicar en el icono para iniciarla. Un apunte importante es que el testbench lee las imágenes a partir de un fichero de texto, especificando su ruta y es posible que ésta deba ser modificada por el usuario en función de dónde haya depositado dicho archivo.

Una vez finalizada la fase de depuración, pasamos a la de síntesis, para ello, clicamos en el botón para iniciarla. De forma opcional, se puede generar una nueva solución haciendo clic en Project -> New Solution e introduciendo las directivas de optimización que el usuario desee tal y como se explicó en la sección 2.3. Vivado HLS.

Una vez hemos realizado la síntesis de nuestro diseño, pasamos a realizar o bien la cosimulación (con el fin de observar el comportamiento hardware de nuestro sistema) clicando en y a continuación seleccionamos como simulador “Vidado Simulato” de la siguiente forma:

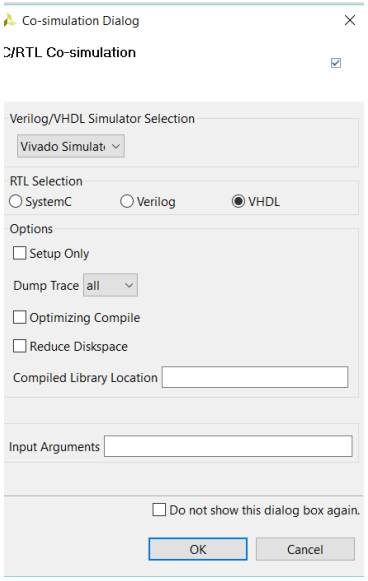


Figura 7.66 Cosimulación en Vivado HLS

Por otro lado, podemos exportar nuestro periférico como un IP, clicando en y seleccionando lo siguiente:

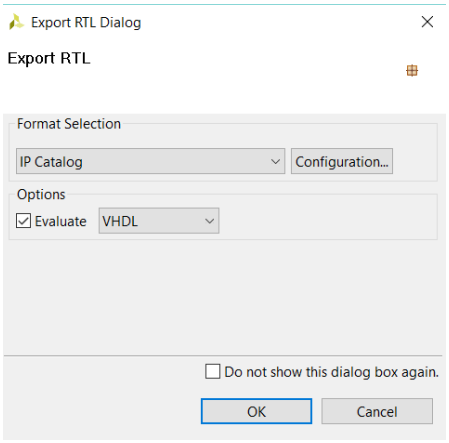


Figura 7.67 Exportar como IP nuestro diseño en Vivado HLS

Una vez finalizada la exportación, se debe haber generado un archivo comprimido que contiene la información del IP, dentro de la carpeta solutionX -> impl -> ip.

Vivado

A continuación, se muestran los pasos para incluir dicho IP que hemos exportado en la sección anterior, en un sistema más grande. Se realiza de la siguiente manera. Abrimos Vivado:

Seleccionamos Create New Project:

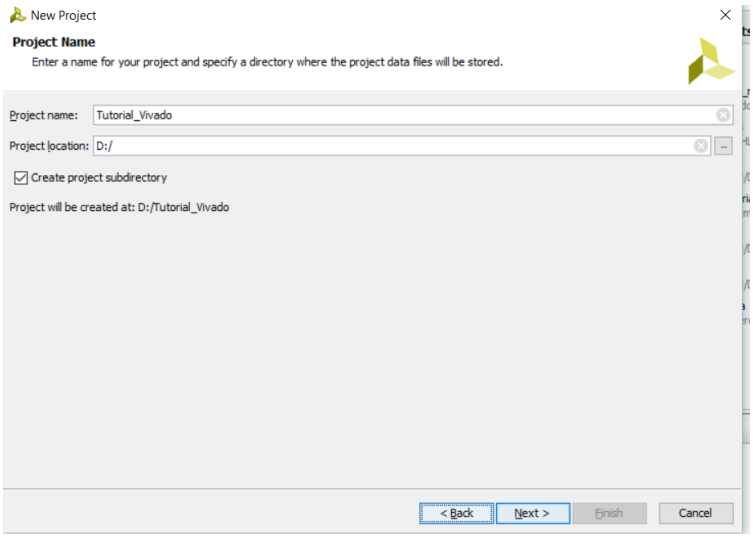


Figura 7.68 Crear un proyecto nuevo en Vivado

Clicamos Next, dejamos las opciones por defecto y clicamos Next nuevamente. Seleccionamos la tarjeta para la que va destinado nuestro diseño:

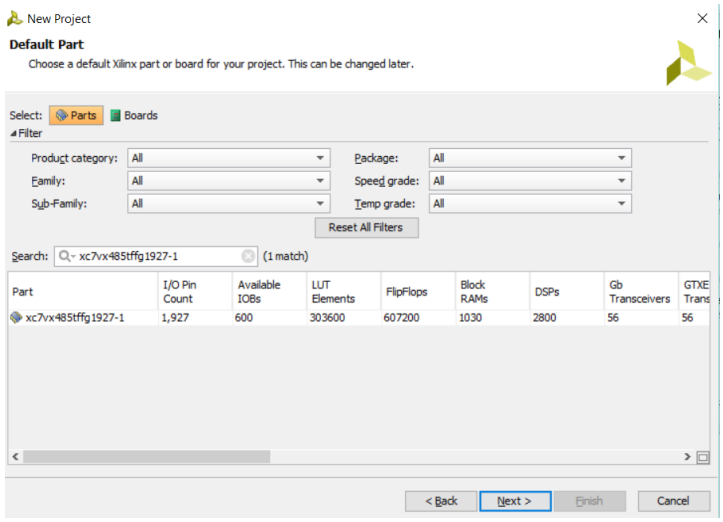


Figura 7.69 Seleccionar el tipo de FPGA en Vivado

Clicamos nuevamente en Next y Finish. Se nos abrirá la siguiente ventana:

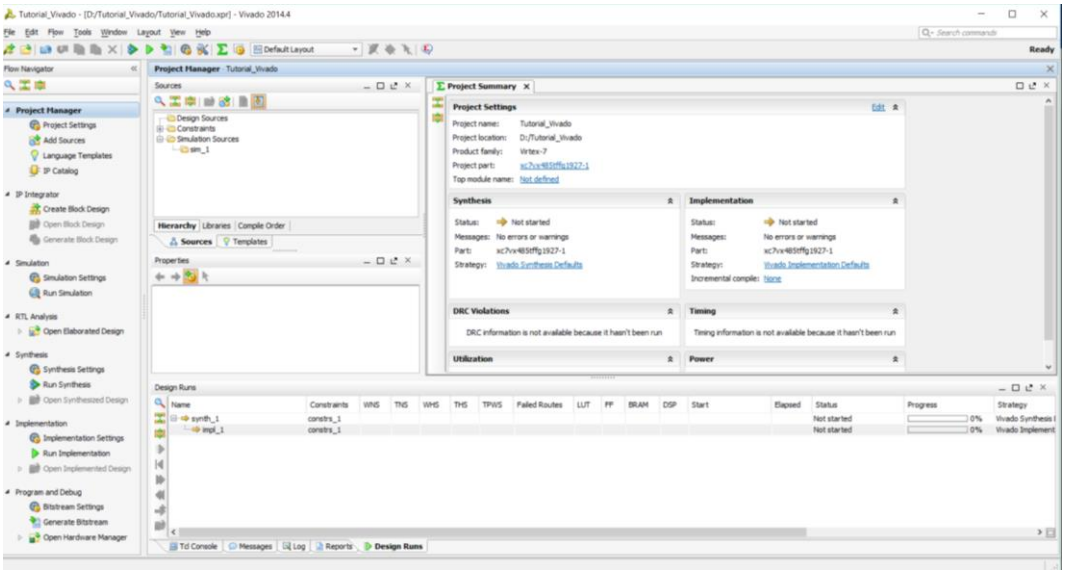


Figura 7.70 Vista general de un proyecto en Vivado

Clicamos en la columna izquierda -> Create Block Design:

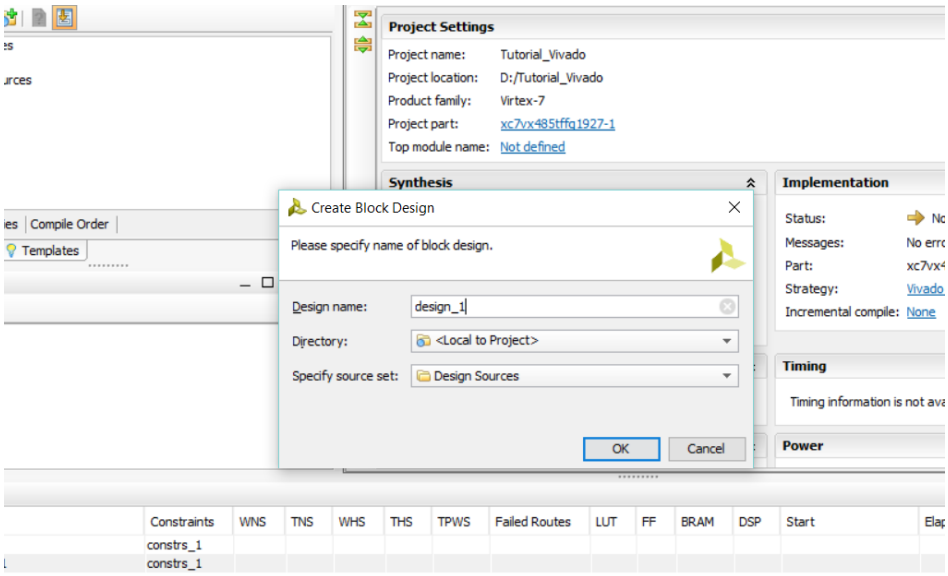


Figura 7.71 Generar un diseño nuevo en un proyecto en Vivado

Clicamos en la pestaña Tools -> Proyect Settings -> IP -> Add Repository. Y seleccionamos la ruta donde se encuentra el IP que hemos exportado desde Vivado HLS. A continuación, clicamos en Add IP y lo añadimos:

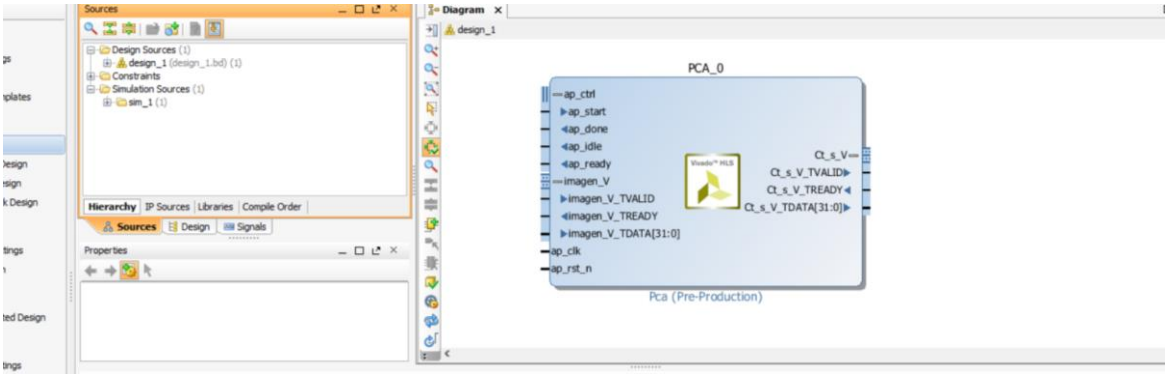


Figura 7.72 Área de trabajo de Vivado con nuestro IP de Vivado HLS

Y ya podemos incluir este IP en cualquier otro sistema o diseño que deseemos.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá