UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Guía para la definición y adaptación de algoritmos para su implementación en FPGAs mediante High Level Synthesis

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Juan Quevedo, Víctor
Tutor: Andrés Martínez, David de

Curso 2021-2022

Resum

L'ús de les FPGA augmenta actualment gràcies a les seues capacitats, i ja es pot trobar en diversos llocs poden ser: en la indústria, en el camp de la medicina, centre de dades, etc.

Encara que la tecnologia FPGA tendeix a ser complexa quant a configuració i sol ser una barrera per al seu ús. Diverses marques van traure ferramentes especialitzes en la sintesi de codi escrits en llenguatges de proposit general, com: C/C++, Java, SystemC. No obstant això, el HLS o High Level Synthesis pot arribar a simplifacar el procés de disseny, però es necesiten alguns coneixements per a l'ús d'aquest, encara que aquests coneixements seran molt menys densos.

En aquest treball, ens centrarem en analitzar el flux de disseny en general de la ferramenta HLS, per a seguidament centrar-nos en el HLS de Xillinx, Vitis HLS. Per a despres poder analitzar el seu funcionament, que restriccions ens imposa o que tipus d'optimitzacions ens aporta. Per a finalment realitzar un cas practic on comprovarem com funcionament a partir de un codi sintetizable y quines optimizacions en interesen pera millorar el nostre diseny en un dels algorismes que mes popularitat esta guanyant actualment pel seu la seua aplicació en les intel·ligències artificials.

Paraules clau: HLS, Xillinx, C/C++, Algorismes, FPGA, Vivado/Vitis

Resumen

El uso de la FPGA aumenta actualmente gracias a sus capacidades, y ya se puede encontrar en diversas partes como, en la industria, en el campo de la medicina, centro de datos, etc.

Aunque la FPGA tienden a ser complejas en cuanto a su configuración y suele ser una barrera para su uso. Para ello diversas marcas sacaron herramientas especializas en la síntesis de código escritos en lenguajes de propósito general, como: C/C++, Java, SystemC. No obstante, el HLS o High Level Synthesis pueda llegar a simplificar el proceso de diseño, pero aún se necesitan algunos conocimientos para el uso de este, aunque estos conocimientos serán mucho menos densos.

En este trabajo, nos centraremos en analizar el flujo de diseño en general de la herramienta HLS, para seguidamente centrarnos en el HLS de Xillinx, Vitis HLS. Para así, analizar su funcionamiento, que restricciones encontramos o que tipo de optimizaciones nos aporta. Para finalmente realizar un caso practico donde comprobaremos el funcionamiento mediante un código sintetizable e introduciremos distintas optimizaciones para ver cual nos interesa y porque, en uno de los algoritmos que más popularidad están ganado actualmente por su aplicación en las inteligencias artificiales.

Palabras clave: HLS, Xillinx, C/C++, Algoritmos, FPGA, Vivado/Vitis

Abstract

The use of FPGA is increasing due to its capabilities, and it can already be found in various places such as: industry, the field of medicine, data centers, etc.

Although FPGA technology can be complex in terms of configurations, various companies decided to create a specialized tools that are able to synthetize code wrote in languages like C/C++, Java or SystemC as an input, and from that entry, HLS created a

design without adding more complexity to the process. Nevertheless, using HLS tools requires knowledge about some concepts, but these concepts are less dense.

Firstly, we will analyze the design flow of these tools. Secondly, we will analyze Vitis HLS and understand how it works, which restrictions there are and which optimization types there are. Finally, we will present a practical case where we prove how a synthesizable code works and we will use a different optimizations to seek the best solution.

Key words: HLS, Xillinx, C/C++, Algorithms, FPGA, Vivado/Vitis

Índice general

Índice general v
Índice de figuras ix
Índice de tablas ix

1 Introducción 1
1.1 Motivación . . . 2
1.2 Objetivos . . . 2
1.3 Impacto esperado . . . 3
1.4 Estructura de la memória . . . 3
2 FPGA 5
2.1 Arquitectura de la FPGA . . . 5
2.1.1 Cofigurable Logic Block . . . 6
2.1.2 Input/Output Block . . . 7
2.1.3 Block Random Access Memory . . . 7
2.1.4 Encaminamiento . . . 7
2.2 Arquitecturas heterogéneas . . . 8
2.2.1 CPU . . . 8
2.2.2 GPU . . . 8
2.2.3 ASIC . . . 9
2.3 FPGA vs Otras tecnologías . . . 10
2.3.1 Rendimiento . . . 10
2.3.2 Programabilidad . . . 10
2.3.3 Adaptabilidad . . . 10
2.3.4 Coste/Eficiencia . . . 11
2.3.5 Computacion paralela . . . 11
2.3.6 Aplicación en tiempo real . . . 11
2.4 Conclusión . . . 11
3 Configuración y "Workflow" en tecnologías FPGA 13
3.1 Lenguajes de descripción de hardware . . . 13
3.1.1 VHDL . . . 14
3.1.2 Verilog . . . 15
3.1.3 Flujo de diseño clásico . . . 15
3.1.4 Metodología de diseño . . . 16
3.2 Register Transfer Level . . . 17
3.3 High Level Synthesis . . . 17
3.3.1 Metodología de diseño . . . 19
3.4 Comparativa de flujo de diseño . . . 20
3.4.1 Rendimiento del diseño final . . . 20
3.4.2 Flexibilidad . . . 20
3.4.3 Aprendizaje . . . 20
3.5 Conclusión . . . 20
4 Advanced Micro-controller Bus Architecture 23

4.1 Especificación AXI 24
4.1.1 Comunicación en la especificación AXI 25
4.1.2 Señales disponibles 26
4.1.3 Handshake 28
4.1.4 Operaciones de escritura y lectura 30
4.1.5 Ráfaga en la especificación AXI 31
5 Vitis High-Level Synthesis 33
5.1 Vitis HLS Design Flow 33
5.2 Síntesis del código C/C++ 33
5.3 Xillinx AXI 34
5.3.1 Interfaz maestra AXI4 34
5.3.2 Interfaz AXI4-Lite 35
5.3.3 AXI4-Stream 36
5.4 Bundles 37
5.5 Port-level de entrada/salida 37
5.5.1 Sin protocolos 37
5.5.2 Wire handshakes 37
5.5.3 Memory inteface 38
5.6 Block-level Protocol 38
5.7 Directivas pragma 39
5.7.1 Directivas para la configuración de interfaces AXI4 39
5.8 Conclusión 40
6 Realizar un código sintetizable 41
6.1 System calls 41
6.2 Uso de la memoria dinámica 41
6.3 Punteros 42
6.4 Template metaprogamming y recursión 42
6.5 Tipos de datos sintetizables 43
6.6 Vectores 44
6.7 Conclusion 45
7 Optimización del código 47
7.1 Optimizaciones de rendimiento 47
7.2 Optimizaciones de latencia 49
7.3 Optimizaciones de área 50
7.4 Optimizaciones destinadas a la lógica 51
7.5 Otras directivas para optimización 52
7.5.1 Allocation 52
7.5.2 Loop flatten 52
7.5.3 Dataflow 52
7.6 HLS Math 53
7.7 Template metaprograming para configurar directivas 53
7.8 Conclusión 53
8 Caso práctico - Red neuronal 55
8.1 Red neuronal 55
8.2 Estructura del caso práctico 56
8.3 Entrenamiento y extracción de los datos 56
8.4 Generación de la red neuronal e inferencia 56
8.4.1 Interfaces utilizadas 57
8.4.2 Optimización/Directivas 57
8.4.3 Versión que prioriza la latencia 58
8.4.4 Versión que prioriza el área 59

8.4.5 Conclusión 60
9 Conclusiones **63**
9.1 Consideraciones finales 63
9.2 Trabajo futuro 63
Bibliografía **65**

Apéndices
A Objetivos de Desarrollo Sostenible **67**
B Código **69**
B.1 Generador y entrenamiento del modelo 69
B.2 Codigos realcionado con la inferencia 71

Índice de figuras

2.1 Arquitectura FPGA simplificada 6
2.2 Estructura general de un bloque lógico 6
2.3 Estructura simplificada del enrutado en una FPGA 7
2.4 Diseño basado en Celdas Estándares 9

3.1 Tipos de descripción 14
3.2 Herramienta de captura de esquemáticos 15
3.3 Proceso High Level Synthesis 17
3.4 Proceso de diseño en el High Level Synthesis 19

4.1 Interfaces AXI disponibles en el HLS 24
4.2 Canales de lectura 25
4.3 Canales de escritura 25
4.4 Escenario donde la señal VALID se activa antes que la señal READY . . . 29
4.5 Escenario donde la señal VALID se activa después de la señal READY . . 29
4.6 Escenario donde ambas señales se activan al mismo tiempo. 29
4.7 Proceso de escritura 30
4.8 Proceso de lectura 31

5.1 AXI en modo esclavo 35
5.2 Estructura de la interfaz AXI4-Lite 35

7.1 Pipelining aplicado a un bucle 47
7.2 Tipos de unroll en Vitis HLS 48
7.3 Fusionado de dos bucles mediante la directiva ”loop_merge”. 50
7.4 Remodelación de matriz mediante la directiva ”array_reshape”. 51
7.5 Diferencia entre un árbol sin reestructurar con uno reestructurado mediante la directiva ”expression_balance”. 51

8.1 Estructura de una red neuronal simple. 55
8.2 Comparativa de las versiones de la red neuronal. 60

Índice de tablas

2.1 Comparativa CPU, GPU, FPGA, ASIC 11

3.1 Conclusión lenguajes HDL vs HLS 21

4.1 Descripción de las señales del mecanismo de ”handshake”. 26

4.2 Descripción de las señales disponibles en el canal de escritura y lectura de dirección/control. 27
4.3 Descripción de las señales disponibles en el canal de escritura y lectura de datos 27
4.4 Descripción de las señales disponibles en el canal de escritura de respuesta. 28
4.5 Señales disponibles en AXI4-Lite 28

CAPÍTULO 1
Introducción

En 1984, Ross Freeman [4] ingeniero estadounidense eléctrico y Bernard Von Der Schmitt [5], los dos cofundadores de Xillinx, inventaron la FPGA [11], como resultado de la unión de diversas tecnologías y evolución de los CPLD [7], situando a Xillinx [34] como marca más populares en el desarrollo de esta tecnología, que actualmente podemos encontrar en diversos sectores [33], como el sector automovilístico, médico o big data, por su amplio rango de aplicación.

Actualmente esta tecnología juega un papel importante en determinadas tareas dedicadas, ya que ofrecen un mayor rendimiento que las CPUs [32] tradicionales. Esto ha hecho que la FPGA sufra un crecimiento en los últimos años, pero el desarrollo en estas necesita de conocimientos específicos en diseño de sistemas digitales y adaptarse a los lenguajes de descripción como, VHDL [20] o Verilog [19].

A consecuencia de este crecimiento experimentado las distintas marcas dedicadas al desarrollo de estas tecnologías crearon herramientas de desarrollo en lenguajes de alto nivel para resolver este problema y así mejorar la gestión del diseño y el crecimiento de la productividad en el campo del desarrollo de software para esta, High Level Synthesis o HLS, el cual aplica una capa de abstracción en el proceso de desarrollo. El HLS se encarga de interpretar la descripción de estos lenguajes que después usará para la creación de un sistema digital, el cual implementará el comportamiento descrito mediante máquinas de estado finito.

Y aunque utilizar el High Level Synthesis conlleva un aprendizaje de todas sus metodologías, directivas o librerías, serán mucho más familiares y cercanas al desarrollo en lenguajes de alto nivel. Así pues, la necesidad de tener un alto rango de conocimientos sobre esta tecnología disminuye y hace que el mundo de las FPGA más accesible a los desarrolladores, ya que solo necesitan como mínimo conocimientos de C[22]/C++[23] y acatar unas normas de estilo de código.

1.1 Motivación

Actualmente, la tecnología FPGA crece en diversas áreas de la industria, medicina, seguridad, etc. Ya que esta proporciona cualidades como: baja latencia, eficiencia, flexibilidad.

Hasta ahora, las dos formas más populares de configuración de FPGA, son el diseño a partir de los lenguajes de descripción de hardware y los diseños creados por el HLS, no obstante esta primera, requiere más aptitudes para realizar un diseño digital. Por tanto, los diversos proveedores de FPGA, empezaron a interesarse en los HLS para dar una capa de abstracción al proceso de diseño, y así que el diseñador pudiera realizar la configuración de esta con mayor facilidad o sin tener un conocimiento muy específico. Ya que, actualmente es uno de sus mayores obstáculos.

Sin embargo, las herramientas HLS, aún contienen cierta complejidad, ya que su síntesis necesita de códigos realizados en lenguajes de propósito general que cumplan con ciertas restricciones para que este pueda llegar a ser sintetizable. No obstante, estas herramientas, actualmente ya no se centran solamente en conseguir el mejor diseño posible, sino también dan la posibilidad de ajustar el comportamiento de los propios HLS mediante directivas que ayudan a la optimización del diseño.

Por tanto, el entender esta herramienta es un punto a favor en el diseño digital, ya que esta aporta mucha flexibilidad y buena adaptabilidad al cambio, propiedades que a día de hoy se pueden llegar a considerar esenciales en muchos sectores donde pueda estar aplicada esta tecnología.

1.2 Objetivos

Como objetivo principal se pretende llegar a explicar las características del High Level Synthesis desarrollado por Xillinx por tal de poder definir una guía/pautas para la definición e implementación de algoritmos escritos en C/C++ , así finalmente poder adaptar un algoritmo dedicado para el funcionamiento en CPUs, a un algoritmo sintetizable y funcional en una FPGA.

Para ello el objetivo principal esta subdividido en tres partes a conseguir:

1. Analizar y entender toda la metodología del diseño digital con el High Level Synthesis
2. Estudiar todo lo relacionado con la síntesis del HLS de Xillinx. Este apartado se divide en:
   a) Estudiar todo lo relacionado con la creación de códigos sintetizables.
   b) Realizar algoritmos sintetizables a partir de algoritmos destinados a la CPU.
3. Analizar todo el funcionamiento de las directivas de optimización de Vitis HLS [13]
4. Realizar pequeñas pruebas con las directivas de optimización.
5. Desarrollar una red neuronal como caso práctico a partir de todo lo aprendido. Este se divide en dos:
   a) Realizar la red neuronal y entrenamiento, para luego extraer el modelo.
   b) Realizar el proceso de inferencia en un código sintetizable en base al modelo importado, el cual será optimizado a partir de lo aprendido.

1.3 Impacto esperado

Actualmente el uso de las FPGA crece, ya que estas son perfectas en algunos campos en que el coste/computación es crucial. Por otra parte incorporarse en este mundo, y aunque con la aparición de los HLS facilite mucho más esto, no deja de ser necesario tener ciertos conocimientos y metodologías para llegar a tener un código útil y sintetizable. Por tanto con este trabajo se espera acercar lo máximo posible el HLS y facilitar la incorporación al HLS de Xillinx mediante una guía/pautas que ayudarán a entender el comportamiento de la herramienta y así poder empezar de cero más fácilmente.

1.4 Estructura de la memória

En este apartado, resumiremos la estructura de este trabajo de fin de grado, donde hemos optado por seguir una estructura que vaya desde los conceptos más básicos como la arquitectura de la FPGA y sus metodologías actuales hasta los conceptos para utilizar el High Level Synthesis de Xillinx.

Primeramente, detallaremos la arquitectura de la FPGA y comentaremos después sus principales competidoras para seguidamente comentarlas.

En segundo lugar, detallaremos el proceso de diseño desde el punto de vista de los lenguajes de descripción de hardware y el High Level Synthesis para finalmente ser comparados.

En tercer lugar, detallaremos todo lo relacionado con AMBA [10] y la especificación AXI [18], detallando las señales, operaciones y la transferencia en ráfaga, para así entender su funcionamiento e implementación en las interfaces disponibles en Vitis HLS, donde se detallará sus configuraciones y funciones de estas mismas.

En cuarto lugar, encontramos todo lo relacionado con la síntesis y Optimización. La primera parte, se encargará de describir todos los tipos sintetizables dentro de la herramienta HLS de Xillinx. Después una vez descritos los tipos nos centramos en detallar todo lo relacionado con la optimización mediante las directivas que aporta Vitis HLS y así poder optimizar en área, latencia, rendimiento y lógica.

Finalmente, se describirá un caso práctico donde se aplicarán los conceptos adquiridos durante el estudio de este trabajo, y así poder llegar a una conclusión de los objetivos alcanzados y posibles trabajos futuros para extender y mejorar a este.

CAPÍTULO 2
FPGA

Dentro del mercado de los circuitos integrados encontramos varios tipos de dispositivos, abarcando desde circuitos electrónicos diseñados expresamente para la problemática a dispositivos reprogramable que se adaptan en función de las necesidades. Dentro de las reprogramables encontramos la tecnología FPGA o Field Programmable Gate Array conformada por diversos bloques lógicos configurables que pueden ser modificables dependiendo la necesidad del diseño, dando como resultado un crecimiento en diferentes sectores de la industria por sus características.

En cuanto a su configuración, estas utilizan lenguajes de descripción de hardware (Hardware Descripcion Lenguaje o HDL) para implementar un diseño, normalmente estos lenguajes suelen ser VHDL o Verilog, los cuales nos permiten especificar un hardware siguiendo un estilo de descripción, que incluye tres etapas, 'dataflow', 'structural' y 'behvioural', descritas en el capítulo siguiente 3.1. Mediante estas tres etapas se genera microarquitecturas, las cuales son transferidas a la FPGA a través de un proceso de síntesis que genera un archivo Bitstream. Una vez generado, el Bitstream contiene un conjunto de bits que determina toda la configuración de la FPGA, y cada conjunto de bits contienen cada uno de los bits de configuración de los elementos que implementa la lógica combinacional, secuencial y de encaminamiento del dispositivo.

Si profundizamos en la arquitectura, esta se compone principalmente de bloques lógicos o CLB [8], los cuales esta interconectados dependiendo del comportamiento descrito por el Bitstream. No obstante esta aún necesita de una forma de comunicar su lógica con el exterior de la FPGA y aquí entran en juego la celdas de entrada y salida o IOB [17]. Finalmente con el enrutado de los CLBs y los IOBs obtenemos una matriz de enrutado que puede ser reprogramada, otorgando a la FPGA una gran flexibilidad en cuando a las especificaciones del diseño, lo que permite evolucionar a la par con la problemática encargada de tratar.

2.1 Arquitectura de la FPGA

Como se ve en 2.1, la arquitectura de una FPGA consiste en un grupo de CLB interconectados mediante canales de dirección horizontal y vertical, lo que finalmente conforma una matriz dos dimensiones y aunque esta tecnología pueda variar en sus bloques lógicos y la forma de enrutarlos, esto dependerá del fabricante, por tanto en este apartado dedicaremos a resumir la arquitectura de la FPGA desde el punto más general dentro de su arquitectura más regular.

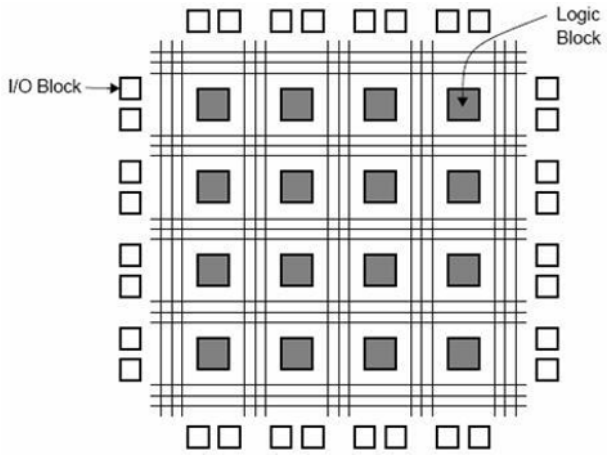


Figura 2.1: Arquitectura FPGA simplificada

2.1.1. Cofigurable Logic Block

Los bloques lógicos son los que implementan las funciones lógicas a necesidad del diseño y están compuestos por diferentes componentes para despeñar la función necesaria. El numero componentes de estos pueden variar dependiendo de las características del CLB, pero en general consiste en una o varios LUTs cuyas salidas se encuentran multiplexadas y son tratadas dependiendo de unos parámetros de configuración del CLB.

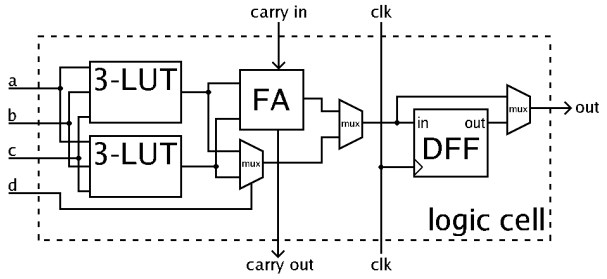


Figura 2.2: Estructura general de un bloque lógico

Su arquitectura más general, como podemos ver en 2.2, se compone de:

- **Look-up tables:** Esta determina una salida a partir de una tabla y por tanto será la que describa el comportamiento lógico. No obstante el tamaño de esta variara dependiendo de las características necesarias.
- **Full adder:** El Full Adder se caracteriza por ser una unidad aritmológica que se dedica a realizar sumas.
- **Flip-Flop tipo D:** De toda la familia existente de los biestables, este tipo suele ser útil cuando hablamos de almacenar información. No obstante esta información será únicamente un bit.

Finalmente el número de elementos dentro del bloque lógico dependerá sobre todo, de las características del producto y del propósito al que queremos llegar, por ejemplo en el número de LUTs utilizadas, ya que el uso de LUTs de mayor tamaño conlleva a mayor retardo y área utilizada obtendremos, aunque más eficiente será para implementar funciones lógicas de muchas entradas. No obstante en el momento de implementar alguna función lógica más simple, se desperdiciaría mucha área, por tanto se dividen

la LUTs para tratar de minimizar los inconvenientes que aporta una LUT monolítica de gran tamaño.

2.1.2. Input/Output Block

Esta es la parte encargada de comunicar la FPGA con el exterior y por tanto podrán tanto como enviar o recibir señales o realizar las dos acciones. Los bloques IOBs son bloques que tiene un alto rango de adaptabilidad al poder trabajar con distintas tensiones o estándares digitales, haciendo que la FPGA pueda adaptarse cómodamente a las necesidades. Cuanto a su número, existe un bloque de estos por cada terminal de I/O del FPGA, lo que posibilita la configuración independiente de cada terminal y la existencia de un buffer único para cada uno de ellos.

2.1.3. Block Random Access Memory

Los bloques de RAM o BRAM son componentes utilizados dentro de la FPGA para almacenar grandes cantidades de datos. El tamaño de estos bloques es finito, aunque el tamaño del bloque puede ser configurable según la necesidad en anchura y en profundidad, aunque normalmente estos tamaños suelen ser de 4,8,16,32 kb. La cantidad de estos bloques dependerá mayoritariamente de la necesidad de nuestra aplicación.

Por otra parte, en Xillinx también encontramos otro tipo de bloque de RAM disponible, la UltraRAM URAM, un tipo de memoria de un tamaño mayor per menos rápida.

Otro tipo de memoria dentro de las FPGA, son las LUT o "Look Up Tables", estas son pequeñas unidades de memoria que debido a su flexibilidad de su estructura en la FPGA se pueden utilizar como memoria de 64 bits y nos referimos a ellas como memorias distribuidas. También las LUT son el tipo de memoria más rápida disponible en la FPGA, ya que estas pueden ser instanciadas en cualquier parte de la estructura que mejore el rendimiento del circuito implementado.

2.1.4. Encaminamiento

Los enrutados son los encargados de conectar las distintas lógicas entre sí, creando los diferentes caminos para llegar a los diferentes bloques. Las conexiones entre si esta hechas mediante varios canales, no obstante entre para efectuar la conexión entre ellos encontramos un nuevo tipo de bloque, en este caso los bloques de conexiones.

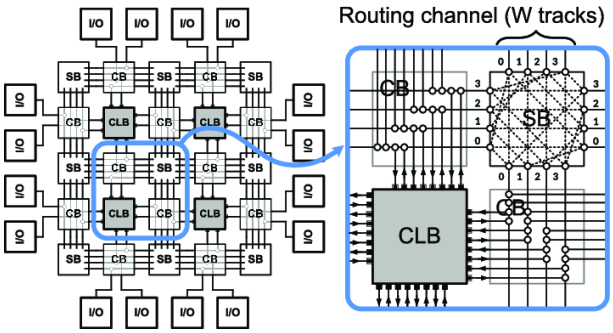


Figura 2.3: Estructura simplificada del enrutado en una FPGA

Estos bloques contienen diferentes entradas, siendo estos los encargados de redirigir la salida mediante la ruta indicada para llegar al siguiente bloque, ya sea otro bloque de conexión o los tratados anteriormente. No obstante, como se puede ver en la imagen 2.3

dentro de los bloques de conexiones encontramos switches, que son los encargados de guiar la señal por dentro del bloque de conexiones.

2.2 Arquitecturas heterogéneas

Con el tiempo la necesidad de adaptabilidad en los diseños gana importancia, ya que en la actualidad las nuevas tecnologías se incorporan a la vida diaria. Esto ha dado como resultado una cantidad de nuevas tecnologías diseñadas para suplir esta necesidad, llegando a encontrar distintas tecnologías en varios sectores, donde las distintas arquitecturas son implementadas por el diseñador según el propósito y coste de esta.

Dependiendo de la arquitectura encontramos, diseños creados expresamente para la problemática, diseños dirigidos a ser altamente configurable y flexibles con el cambio o diseños que intenta incorporar ambos en cierta medida a cambio de sacrificar otras características. Algunas de estas arquitecturas son:

2.2.1. CPU

La CPU se encarga de la coordinación de los diferentes elementos que conforman actualmente una computadora, lo que nos lleva a encontrar varios tipos dependiendo, del mercado al que vayan destinado y se componen por un único circuito integrado o varios CPUs en un mismo circuito como es el caso de los procesadores multinúcleo.

Este funciona a base de la interpretación de instrucciones, para realizar las diferentes acciones ya sean, operaciones aritméticas, manipulación de registros, decodificación de estas mismas instrucciones, etc. En cuanto a su arquitectura más simple, se compone de:

- **Registro internos:** Son registros encargados de almacenar los datos accesibles o de uso general, como contadores de programa, punteros de pila, acumuladores, etc. O datos no accesibles, como instrucciones, datos del bus de datos o de dirección.
- **Unidad aritmeticológica:** Como su nombre indica realiza las operaciones aritméticas y lógicas.
- **Unidad de control:** Esta unidad se encarga de buscar las instrucciones en la memoria principal para decodificarlas (interpretarlas), ejecutarlas.

2.2.2. GPU

Juntamente con la unidad central de procesamiento, esta es una de las arquitecturas más populares actualmente y aunque pueda llegar a ser similar, esta tiene diferentes propósitos y funcionamientos, los cuales en un principio su función principal fueron mostrar por pantalla, gráficos. No obstante hoy en día, con la evolución gráfica actual, estas optimizadas para los cálculos aritméticos de matrices dimensionales y de punto flotante, y otras tareas que requieren de un alto grado de paralelismo.

La arquitectura de la GPU[**?**] a día de hoy se encuentra segmentada con múltiples núcleos optimizados para la ejecución de procesos en paralelo.

2.2.3. ASIC

Las Application Specific Integrated Circuit o ASIC son circuitos dedicados para una sola tarea única y no está pensada para propósitos generales. Dentro de las ASIC, disponemos de los siguientes diseños:

- **Diseño basado en Celdas Estándares:** Es diseño en celdas estándares, es un método de diseño ASIC donde el diseño es encapsulado en una representación lógica abstracta para conseguir un diseño abstracto.

  Este diseño compone de celdas llamadas ”Standard Cell”, un grupo de transistores y estructuras interconectadas que proporcionan funciones de lógica booleana o de almacenamiento. Esto permite al diseñador, centrarse en aspectos de alto nivel del diseño, mientras que otro diseñador pueda centrarse en la parte de la implementación.

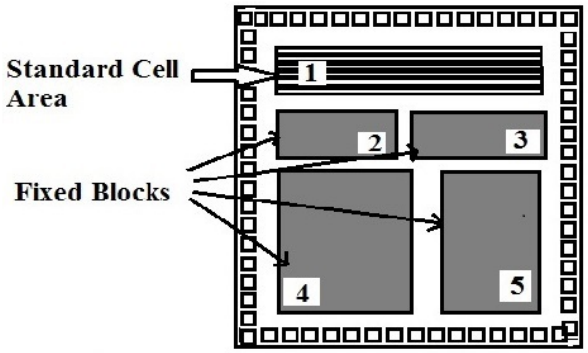


Figura 2.4: Diseño basado en Celdas Estándares

  Este diseño ASIC, como se puede ver en 2.4, esta compuesto por dos componentes con propósitos generales, primeramente encontramos la celda estándar es un bloque organizado en celdas estandarizadas en filas y su funcionamiento dependerá de las "Standard Cell Libraries". Estas son una colección de funciones lógicas electrónicas de bajo nivel como: AND, OR, INVERT, biestables y latch.

- **Diseño basado en Matriz de Puertas:** En este diseño, la placa ASIC, viene predefinida por un numero de transistores en una oblea de silicio, en esta el diseñador no puede cambiar el lugar donde se alojan estos. Esto acaba formando pequeños patrones repetitivos de "base array", los cuales son patrones de matrices lógicas donde la interconexión de estas, es donde el diseñador tiene la responsabilidad de conectar usando capas de metal y así obtener el funcionamiento lógico deseado.

  Dentro del diseño basado en Matrices de Puertas, encontramos tres tipos:

  - **Channeled Gate Array:** Se caracteriza por tener un espacio fijos entre filas de transistores.
  - **Channel Less Gate Array:** Al contrario que en el diseño comentado anteriormente, no deja espacio libre para el enrutado de los transistores. Por tanto el enrutado pasara a ser un enrutado personalizado entre el metal y el transistor.
  - **Structured Gate Array:** A diferencia de los otros tipos de estructuración, que principalmente se centraban en la interconexión de los transistores, se centra en integrar también un bloque y así poder tener bloques configurables en el diseño.

- **Diseño "Full Custom" o hecho a medida:** El diseño "Full Custom", está destinado a una acción en específica y por tanto todas sus interconexiones y capas son personalizadas para la problemática a tratar. Esto conlleva a que el propio diseñador no pueda cambiar la interconexión de estos, ya que sino obtendremos un resultado diferente al dado, haciendo que el diseñador se centre enteramente la optimización de este, construyendo distintos circuitos o optimizando las celdas de memoria.

2.3 FPGA vs Otras tecnologías

Cuando hablamos de FPGA, podemos destacar ciertas características y las cuales podemos comparar con otras arquitecturas existentes para llegar a saber cuál nos interesa más para el problema a tratar. Algunos de los detalles a comparar con otras arquitecturas de las detalladas anteriormente son:

2.3.1. Rendimiento

A términos de rendimiento, esto siempre dependerá de la necesidad a tratar, ya que si nos decantamos por procesos que siguen una esquemática muy secuencial, claramente las CPU, serán más aptas para este tipo de tareas. No obstante, cuando queramos el mayor rendimiento en calculo o procesamiento en paralelo, nos decantaremos por tecnologías como la GPU, la FPGA o el ASIC y por tanto podrán ejecutar estas funciones más rápido.

Por otra parte, como hemos mencionado anteriormente las GPU, FPGA y ASIC están más enfocadas para el procesamiento o calculo paralelo, pero cuanto a rendimiento los diseños de ASIC y FPGA serán más rápidas, ya que la GPU dependerá sustancialmente de la interacción con la CPU. Y en cuanto a mayor rendimiento entre la tecnología ASIC y FPGA, la tecnología ASIC mostrará mejor rendimiento, ya que la FPGA sacrifica un poco de este a cambio de otras características que comentaremos seguidamente.

2.3.2. Programabilidad

Esta característica hace que el diseño obtenido pueda llegar a ser reconfigurable y a su vez pueda ser adaptable al cambio, por tanto la tecnología ASIC es la más perjudicada, ya que como hemos comentado anteriormente esta se basa en diseños hecho a medida para la tarea a tratar. Sin embargo las GPU y CPU son las más beneficiadas por su configuración a bases de instrucciones, lo que permite configurarlas con con más facilidad. Mientras que la tecnología FPGA como se ha comentado anteriormente, esta sacrifica rendimiento a cambio de mejorar en el aspecto de Programabilidad. No obstante, esta sigue llegando a ser complicada, ya que hay que tener un alto conocimiento sobre el diseño y configurarla a base de máquina de estados.

2.3.3. Adaptabilidad

En cuanto adaptabilidad las CPU, GPU y FPGA son superiores en este aspecto por ser reconfigurables, lo que permite ser más flexibles frente los cambios y adaptarse a ello cos que con un dispositivo ASIC solo conseguiríamos mediante el rediseño de este.

2.3.4. Coste/Eficiencia

Cuando hablamos de coste a partir de Eficiencia, siempre se intenta hacer un estudio del caso a tratar, ya que dependiendo de las características del problema podemos acabar utilizando un sistema empotrado para tareas donde el procesamiento sea secuencial y la necesidad de núcleos no sea de extrema necesidad. Mientras que si necesitamos una respuesta rápida mediante un cambio, como podría ser la conducción autónoma, el uso de ASIC o FPGA frente la GPU es preferible, ya que estas no cuentan con tantos procesos intermedios y necesitaríamos más GPUs para llegar a suplir la necesidad, haciendo que el coste se dispare por ende el estas dos serian perfectas para esto. Sin embargo entre estas dos el ASIC puede llegar a ser muy eficiente pero al tiempo habría que rediseñarla haciendo la un proceso mucho más caro, un punto donde la FPGA no necesitaría, ya que esta se adapta al problema ya que admite la reconfiguran de ella.

2.3.5. Computacion paralela

Como se ha mencionado anteriormente, la CPU se especializa en el procesamiento secuencial, por tanto en la computación paralela es inferior a las otras tecnologías comentadas anteriormente. Por otra parte las GPU, FPGA y ASIC si están más enfocadas a este tipo de procesamiento, ya que las GPU cuenta con una gran cantidad de núcleo a comparación de la CPU y la FPGA cuenta con múltiples bloques designado para la computación en paralelo y por parte del ASIC, podemos llegar a diseñar el diseño necesario para ello. Sin embargo tanto la FPGA y la GPU llegaran a ofrecer una gran escalabilidad a comparación del diseño ASIC.

2.3.6. Aplicación en tiempo real

Como se ha descrito anteriormente, la FPGA es perfecta para dar respuestas rápidas frente a nueva entrada, por su arquitectura y configuración. Lo que la hacer perfecta para este tipo de aplicaciones. Por otra parte la tecnología ASIC, podría llegar a compararse, pero no una de las principales características de las aplicaciones en tiempo real es su adaptabilidad al cambio, lo que un ASIC no podrá suplir. Siendo la FPGA, la tecnología que obtiene un tiempo menor en el procesamiento comparado con las tecnologías tratadas.

2.4 Conclusión

Tabla 2.1: Comparativa CPU, GPU, FPGA, ASIC

Características	CPU	GPU	FPGA	ASIC
Rendimiento				x
Programabilidad	x	x		
Adaptabilidad	x	x	x	
Conste/Eficiencia			x	x
Computacion paralela		x	x	
Aplicacion en tiempo real			x	

Finalmente, viento la tabla anterior, se puede apreciar que todo rango de aplicación de estas tecnologías depende mucho de la aplicación final. La tabla muestra que a programabilidad la FPGA pierde al igual que un circuito ASIC, no obstante hoy en día la FPGA

se encuentran en mejora de este apartado con la inclusión de herramientas como el HLS, que abstrae bastante la fase de diseño. Por tanto, podemos concluir que la FPGA será en muchos casos, una buena elección sobre las otras arquitecturas y circuitos comentados.

CAPÍTULO 3
Configuración y "Workflow" en tecnologías FPGA

Actualmente existen varias formas de diseñar circuitos digitales con menos o más abstracción en el diseño, como los lenguajes de descripción o las herramientas que generan bloques a partir de lenguajes de alto nivel, el High Level Synthesis.

Algunos de estos lenguajes HDL varían en especificaciones según la marca que tenga detrás, aunque algunas de ellas con el tiempo han sido capaces de adaptarse a estándares propuestos y revisados por el Instituto de Ingenieros Eléctrico y Electrónicos o IEEE [14], dotando a esta funcionalidad en cualquier diseño independientemente de la marca.

Con el avance tecnológico, al igual que los lenguajes HDL se crearon para suplir la carencia de eficiencia del diseño tradicional, se crearon los High Level Synthesis, capaces de generar bloques RTL automáticamente. No obstante, esta herramienta lleva a replantear el flujo de diseño tradicional, descrito en 3.1.4, el cual era usado por los lenguajes de descripción HDL, lo que conlleva un nuevo esquema en cuanto a flujo de diseño en los HLS, que describiremos en 3.3.1. Por ende, en este capítulo nos centraremos en describir las características de los HDL y HLS juntamente con sus metodologías de diseño y por último, compararemos las dos tecnologías.

3.1 Lenguajes de descripción de hardware

Frente a la complejidad creciente de los circuitos digitales los diseñadores necesitaban descripciones de alto nivel que no estuvieran determinadas por tecnología electrónica, como los CMOS, un componente MOSFET (Metal-Oxide-Semiconductor Field-Effect Transistor) formado por transistores pMOS y nMOS utilizadas hoy en día para la construcción de circuitos integrados, microprocesadores, chips de memoria y otros circuitos lógicos digitales, o los transistores de unión bipolar que tiene un amplio uso en el campo de la electrónica.

Con ello se crearon los HDL, lenguajes especializados en describir estructuras y comportamiento de circuitos electrónicos y lógicos, con el propósito de dar cierta capa de abstracción entre el diseño y el diseñador, haciendo posible simular y analizar la microarquitectura descrita, llegando hasta el punto de estar la mayoría de estos dentro de los estándares ANSI[15] e IEEE lo que conlleva a los distintos diseños creados cierta independencia cuanto a tecnología empleada y fabricante.

Otra característica importante de estos lenguajes es la forma de describir un circuito, y el uso de esta dependerá mayoritariamente del diseñador, el cual elegirá dependiendo de la necesidad del diseño.

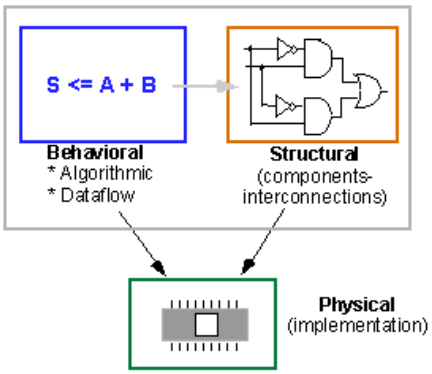


Figura 3.1: Tipos de descripción

Existen tres tipos básicos donde la abstracción varía dependiendo de cual se use, estos son:

- **Descripción comportamental:** Es la descripción de nivel más abstracto, ya que solamente describe cómo se comporta el circuito digital.

- **Descripción de flujo de datos:** La descripción de flujo de datos, es la intermedia en cuanto a nivel de abstracción y describe el modulo a partir de la perspectiva de la transformación y transmisión de datos.

- **Descripción estructural:** La forma de descripción estructural es la más cercana a la descripción de una estructura real de hardware, ya que en esta descripción se interconectan diferente componentes y puertas lógicas para obtener la función deseada, obteniendo con ello el menor nivel de abstracción de los tres tipos.

En la actualidad encontramos una gran cantidad de lenguajes de descripción de hardware, no obstante los más usados son VHDL y Verilog, los cuales detallaremos en el siguiente apartado.

3.1.1. VHDL

VHDL o VHSIC (Very High Speed Integrated Circuit) Hardware Description Lenguaje, es uno de los lenguajes dentro del estándar IEEE más popular actualmente, que surgió en los 80 por parte del departamento de defensa de Estados Unidos como alternativa a los esquemas donde los diseñadores establecían las conexiones entre bloques, lo que resultaba costoso, de difícil entendimiento y poco compatibles entre ellos.

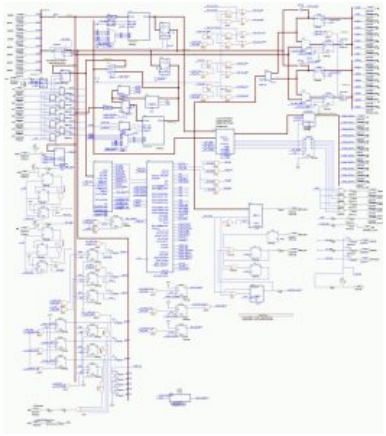


Figura 3.2: Herramienta de captura de esquemáticos

El desarrollo del VHDL sustituyó al proceso anterior 3.2, consiguiendo agilizar el proceso de diseño, hasta llegar al punto de ser aprobado por el comité de IEEE, siendo este ya un lenguaje estandarizado dentro de los estándares IEEE.

Como se ha comentado anteriormente, los lenguajes HDL tienen varias formas de describir un diseño, y VHDL puede describirlos de forma comportamental, estructural y de flujo de datos. No obstante VHDL también permite el uso mixto de esto, permitiendo combinar todas las anteriores o alguna de ellas.

3.1.2. Verilog

Verilog HDL, es otro lenguaje de descripción creado por Phil Moorby[6] con el propósito de crear un lenguaje de descripción similar a C, para que fuera más familiar para el diseñador y al igual que VHDL, los dos son lenguajes de código abierto y estandarizados en la IEEE.

Aunque Verilog, tenga similitudes en C como pueden ser algunas palabras reservadas y un preprocesador y los operadores de lenguaje, existen algunas diferencias con este, como la carencia de punteros y estructuras. También el concepto tiempo está en Verilog, ya que este concepto es muy importante en lenguajes HDL, mientras que en C no lo encontramos.

3.1.3. Flujo de diseño clásico

En este apartado, trataremos el flujo de diseño clásico, el cual se compone de varios pasos, ”Desing entry”, ”Synthesis” y ”Place and route”, que serán descritos seguidamente.

‘’Design entry”, en esta etapa se utilizan: diseños basados en esquemas, en lenguajes de descripción de hardware o una combinación de ambos. En caso, del diseñador querer ocuparse más del hardware, la entra en esquemas es la mejor opción, no obstante en caso del diseño ser complejo o querer realizar este mediante algorítmica, los lenguajes de descripción HDL, son mejor opción. No obstante, también existe un método de diseño mediante máquinas de estados, pero este suele ser poco utilizado.

Síntesis, en esta etapa se convierte un diseño de descripción de alto nivel o un diseño HDL se traduce en una representación a nivel de puerta o un componente lógico. Para realizar este proceso, se utiliza herramientas especializadas en la síntesis que recoge una

entrada escrita en lenguajes de descripción de hardware y genera una lista de conexiones sintetizadas como salida. Para ello se siguen cuatro pasos:

- **Obtención de una "netlist" de componentes básicos:** Aquí se obtiene la descripción de las conexiones entre componentes básicos en un diseño. Estos componentes básicos pueden ser como puertas lógicas, biestables, registros, contadores, etc.
- **Optimización lógica independientemente de la tecnología:** En esta fase, la herramienta agrupa todas las operaciones Booleanas en combinaciones obtenidas mediante el uso de técnicas para la minimización de la lógica.
- **Asignación de las LUTs:** Se ajustan las funciones lógicas de forma fraccionada para que puedan ser implementadas en el tamaño de LUT existentes.
- **Encapsulado de las LUT en bloques lógicos:** Se encapsulan los diferentes LUTs en bloques lógicos para intentar minimizar los retardos y ocupar los menos bloques lógicos posibles.

"Place and route", en esta etapa se compone de dos pasos, el "placement" donde se decide donde se ubicará cualquier componente en el espacio que dispongamos y el "routing" donde se decide el diseño de todas las conexiones entre los elementos ya ubicados. Esta etapa, no es realizada por el diseñador, sino una herramienta proporcionada por el proveedor de la FPGA.

3.1.4. Metodología de diseño

En cuanto a la metodología de diseño en lenguajes HDL encontramos tres etapas, las cuales son divisibles en otras tareas. Estas tres etapas son diseño lógico, implementación del diseño y finalmente verificación del diseño.

En el caso del diseño lógico, el diseñador se centra en definir el problema para generar los distintos diagramas de bloques, obtener las tablas de verdad y generar las ecuaciones lógicas que describirán el comportamiento del diseño.

Una vez hemos obtenido todo lo anterior, empieza la segunda etapa, la cual se centra en la implementación del diseño, empezando por la selección del dispositivo, el cual lo seleccionaremos dependiendo de las necesidades de la implementación, y finalmente describiremos el diseño en lenguajes HDL.

Por último, quedaría la verificación del diseño, ver si lo hecho cubre las necesidades y funciona correctamente. Esto se llevará a cabo mediante pruebas de simulación del diseño mediante salidas esperadas dependiendo de la entrada dada al componente y ejecutando "benchmarks", para comprobar su máximo potencial.

No obstante el diseñador puede orientar el flujo de diseño de dos formas:

- **Bottom-Up:** Esta metodología de diseño se centra en generar diferentes descripciones de componentes, los cuales serán agrupados en distintos de módulos, hasta formar un sistema completo. No obstante, este diseño puede llegar a ser ineficiente en diseños complejos y poco flexible al ser dependiente de la tecnología usada.
- **Top-Down:** El diseño Top-Down se centra en implementar un diseño desde la parte de más alto nivel e ir especificando el nivel de detalle según sea necesario, esto dota al diseño de la posibilidad de subdividirlo en subdiseños hasta llegar a los componentes más fáciles de tratar y a su vez aumenta la posibilidad de reutilizar el diseño.

3.2 Register Transfer Level

Dentro del diseño de circuitos digitales, el Register Transfer Level o RTL es una capa de abstracción en el diseño de circuitos, el cual modela un circuito digital síncrono en cuanto a flujo de datos entre registros físicos y operaciones lógicas. Estos circuitos síncronos están formados por dos tipos de elementos básicos, los circuitos combinacionales, los cuales se forman a partir de puertas lógicas y los circuitos secuenciales, los cuales se forman con biestables, normalmente del tipo D.

Esta abstracción que genera el RTL es aplicada en los lenguajes HDL, como Verilog o VHDL para representar el circuito en alto nivel, siendo este recurso muy usual en el diseño de circuitos actual.

3.3 High Level Synthesis

El High Level Synthesis o HLS es un proceso de diseño automático, el cual interpreta una entrada para seguidamente transformarla en una descripción HDL que implementa el comportamiento deseado de acuerdo con la arquitectura de un dispositivo lógico reconfigurable seleccionado.

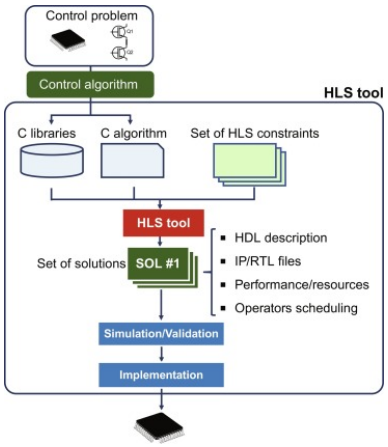


Figura 3.3: Proceso High Level Synthesis

Estas entradas suelen ser lenguajes de alto nivel, como C/C++, SystemC o Matlab, siendo estos analizados y limitados arquitectónicamente, para después ser transcompilados a una descripción RTL en un lenguaje HDL, que normalmente suelen ser VHDL o Verilog.

Como se ve en 3.3, el proceso del HLS consiste en un número de acciones, que dependiendo de la HLS usado, siguen un orden usando diferentes algoritmos y así obtener el resultado deseado. Estas actividades son:

- Procesamiento léxico.
- Optimización del algoritmo.
- Análisis de control/flujo de datos.
- Procesamiento de librerías.
- Asignación de recursos.
- Planificación.
- Enlace de la unidad funcional.

- Enlace de registro.
- Procesamiento de salida.
- Reagrupación de entrada.

No obstante, en la actualidad los lenguajes de programación que se usan como entrada, están destinados a la interpretación y manejo de una CPU, y esto provoca ciertos conflictos en el HLS a la hora de transcompilar este código a un RTL, ya que conceptos como el manejo de memoria dinamica tal y como lo conocemos en C/C++ no es sintetizable o la recursión, una técnica bastante utilizada en algoritmos que requiera de repetición de código como el Quicksort. Sin embargo en la actualidad existen varias empresas con su propio HLS y por lo cual con algunas de estas limitaciones algo diferidas enfrente otros HLS de otra marcas. Por tanto estas limitaciones podemos agruparlas en:

- Jerárquicos.
- Síntesis de interfaces.
- Manejo de memoria.
- Bucle.
- Restricciones de reloj a bajo nivel.
- Iteración.

3.3.1. Metodología de diseño

El HLS hace el proceso de diseño mucho más simplificado por el uso de lenguajes de propósitos generales, como C o C++, algo mucho más interiorizado actualmente y con ello reduciendo la dificultad de los fundamentos a número de metodologías mucho más amigables, que en los lenguajes HDL, para hacer un código sintetizable, esto permite que el proceso pueda adaptarse a metodologías de lenguajes orientados a propósitos generales.

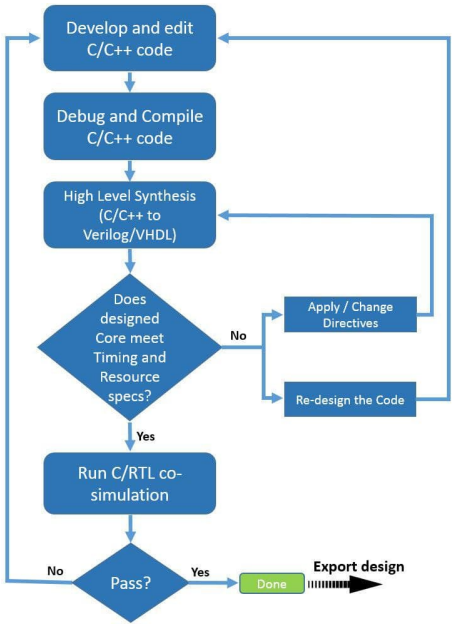


Figura 3.4: Proceso de diseño en el High Level Synthesis

Como se ve 3.4, todo el diseño se divide en diferentes etapas, la primera de todas se centra en la programación de un código en lenguajes de propósito general y a su vez sea sintetizable.

Una vez generado esto, tenemos que comprobar si la funcionalidad de este es correcta. Para comprobar su funcionalidad, en casos simples se podrá analizar el código escrito y comprobar si la salida es la deseada. Mientras que para casos donde el código tenga más complejidad podemos utilizar pruebas, como cajas negras, ya que esta se centra en analizar las salidas a partir de las entradas recibidas, algo que puede llegar a ser útil en la etapa de depuración y compilación.

Una vez ya esté todo correcto, es hora de aplicar el HLS y comprobar si no genera ningún error cuanto a síntesis, si no deberíamos volver a la primera etapa, para generar otra solución. En caso de pasar la síntesis, es hora de comprobar si el resultado dado, por el HLS es el deseado en términos de recursos utilizados, en caso el caso de no cumplir con lo esperado, tendríamos dos opciones, una sería aplicar directivas que puedan ayudar a optimizar partes de código, como dando a un array más de una interfaz de entrada o paralelizar algunos ciclos donde los datos utilizados procedan de lugar distinto. No obstante, si las directivas no fueran capaces de generar un resultado que se adapte a lo deseado, tendríamos que volver a la primera etapa para generar otra solución.

Finalmente, si todo ha ido correctamente, ya solamente nos quedaría la última etapa, donde hacemos la cosimulación de la prueba creada y el RTL generado, el cual podemos analizar seguidamente en un visor de ondas, donde veremos todas las acciones mediante ciclos de reloj. Sin embargo, esta simulación puede llegar a fallar y en caso de que

ocurriera, solo nos quedaría volver a la etapa inicial y hacer lo mismo que en los otros casos.

3.4 Comparativa de flujo de diseño

Entre los dos tipos de flujo de diseños podemos destacar ciertas características que nos pueden guiar a cuál usar. Algunos detalles que comparar podría ser:

3.4.1. Rendimiento del diseño final

En cuanto a rendimiento, describir el diseño en lenguajes HDL, de momento se obtiene mayor rendimiento, ya que tratamos la problemática de forma específica y aunque el HLS también sea una herramienta bastante óptima a la hora de la generación de RTL, esta se basa principalmente en soluciones generales, lo que puede llegar a mermar ligeramente el rendimiento, no obstante, dentro del propio HLS tiene directivas con propósitos de optimización que pueden llegar a solucionar o reducir los problemas que puedan generar un rendimiento menor. Por tanto, la elección a través de rendimiento, dependerá de cuanto se pueda adaptar el HLS al problema que queramos tratar.

3.4.2. Flexibilidad

En este aspecto, el High Level Synthesis ofrece mejor flexibilidad al cambio, ya que esta herramienta se encarga en generar un RTL en un lenguaje HDL y por tanto nosotros tratamos de cambiar o corregir el código ya escrito en un lenguaje de propósito general, por tanto la tarea de modificar la descripción ya hecha no es un problema que pensar, ya que el HLS se encargará después de volverla a generar, mientras que en los lenguajes HDL, habría que pensar en una solución y luego adaptar toda la descripción a ello, lo que puede llegar a ser una labor mucho más tardía.

3.4.3. Aprendizaje

Como hemos visto anteriormente, los lenguajes HDL, necesitan de ciertos fundamento en diseños digitales, como podrían ser los distintos tipos de descripciones, algo que incluso con la abstracción que aportan alguna de estas pueden llegar a ser costoso. Sin embargo, con la herramienta HLS el aprendizaje es mínimo y solo te basas en unas pautas para llegar a conseguir un código sintetizable en un lenguaje de propósito general, algo mucho más común en el mundo tecnológico hoy en día.

3.5 Conclusión

Finalmente, a modo de conclusión, vamos a comparar mediante la tabla 3.1, las dos formas actuales de diseñar un circuito para un propósito, en esta tabla denotamos cuál de las dos tecnologías sale ganadora para cada característica de las tratadas en el capítulo anterior, para finalmente llegar a una conclusión basándonos en el coste/eficiencia.

Analizando la tabla desde un punto coste/eficiencia, vemos que el HLS sale ganador en la tabla 3.1, ya que este tiene más características a favor que los lenguajes HDL. Esto se debe a la reducción de fundamentos para poder utilizarla y su flujo de diseño, el cual es un ciclo, donde el fallo redirige a la primera fase para volver a tratar el problema

Tabla 3.1: Conclusión lenguajes HDL vs HLS

Característica	HDL	HLS
Rendimiento del diseño final	x	
Flexibilidad		x
Aprendizaje		x

de diferente forma y dejando de lado el RTL o simplemente volver a tratar el problema por un posible cambio. No obstante, en cuanto al rendimiento, los lenguajes HLD, son más eficiente, ya que la optimización es más cercana a nivel bajo, y aunque los HLS intenten generar diseños lo más eficiente posible, para la interconexión de bloques se deben realizar múltiples interfaces y protocolos para la síntesis de cualquier diseño lo que conlleva un menor rendimiento.

Por tanto, mirando desde un punto coste/eficiencia el HLS es mejor opción frente los lenguajes HDL.

CAPÍTULO 4
Advanced Micro-controller Bus Architecture

AMBA o "Advanced Micro-controller Bus Architecture", es un conjunto de especificaciones de libre acceso, especializada en la comunicación entre bloques, que obtuvo una crecida de popularidad hasta llegar un estándar para la conexión, gestión y reutilización de bloques funcionales en SoCs[30].

A profundizando un poco más en el estándar AMBA, esta ofrece diversas características, encontramos:

- **Flexibilidad:** Frente a la amplia variedad de SoC, con diferentes características, la reutilización de bloques funcionales puede a llegar ser costosa. Por tanto, AMBA intenta cumplir con estos requisitos, mediante distintas opciones de diseño y dotando de escalabilidad a los procesadores compatibles.

- **Adopción generalizada:** AMBA, es actualmente el estándar industrial más utilizado en los estándares basados en IP[16] en todo el mundo.

- **Compatibilidad:** Hoy en día, encontramos diferentes diseños ofrecidos por diferentes proveedores. Otorgando AMBA, especificaciones que aseguran compatibilidad y escalabilidad entre componentes IP.

- **Soporte:** La adopción de la especificación de AMBA por la industria ha resultado en la creación de herramientas y productos IP por terceros. Gracias a esto AMBA, obtiene un amplio soporte en los sistemas basados en sus especificaciones.

No obstante, en la actualidad la última versión de AMBA, es AMBA5, sin embargo Xillinx solo adopta de momento AMBA4, por tanto desarrollaremos sus especificaciones a partir de la versión que utiliza Xillinx.

En cuanto a AMBA4, está compuesta por diversas especificaciones, ACE[29], destinada a compartir datos entre componentes del sistema mediante distintos protocolos dedicados a la coherencia de los datos. Y la especificación AXI, utilizada para el desarrollo de este trabajo, la cual tiene un amplio mercado, desde aplicaciones destinadas a un alto rendimiento a computación móvil, redes y sector automovilístico.

4.1 Especificación AXI

Como se ha comentado anteriormente, en este trabajo nos centraremos en las interfaces AXI, ya que Xillinx utiliza estas, para el manejo de los bloques IP. Por tanto es este apartado nos centraremos en profundizar en la especificación AXI.

La especificación AXI o "Advanced Extensible Interface", esta se une a AMBA en la versión AMBA3 en 2003 y actualmente la podemos encontrar en las versiones de AMBA4 y AMBA5.

AXI proporciona a los diseños flexibilidad e independencia en la implementación de diseños, además de funcionar en altas frecuencias sin la necesidad de crear interconexiones complejas para ellos y así conseguir bajas latencias con menor complejidad y a su vez beneficiar esas bajas latencias con un alto ancho de banda para dotar al diseño de mayor rapidez.

En cuanto a sus funciones, AXI ofrece una amplia variedad, que incluyen:

- Fases de dirección, control y datos independientes.
- Soporte para trasferencias de datos no alineados.
- Solo utiliza las transacciones basadas en ráfagas ("burst"), cuando empiezan con una dirección inicial.
- Separa los canales de escritura y lectura, para activar el DMA.
- Capacidad para emitir múltiples direcciones pendientes.
- Completado de transacciones fuera de orden.
- Soporte para operaciones atómicas.

Por otra parte AXI dispone de tres interfaces mediante la versión de AMBA4, AXI4, AXI4-Lite, AXI4-Stream.

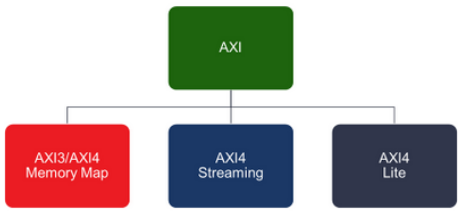


Figura 4.1: Interfaces AXI disponibles en el HLS

Cada una de estas interfaces estas destinadas a usos concretos, en el caso de AXI4, esta se utiliza para realizar las transferencias en memoria con alto rendimiento permitiendo el "burst" de datos en la misma transferencia, no obstante en la interfaz AXI4-lite, esta se trata de una simplificación de la anterior, la cual realiza lo mismo que la anterior pero solamente con trasferencia simples.

Por último, nos queda la interfaz AXI4-Stream, dedicada a la transmisión de alta velocidad, donde el maestro transfiere datos a un esclavo.

4.1.1. Comunicación en la especificación AXI

AXI, como se ha comentado anteriormente es un protocolo de transmisión de datos, y para ello tiene definido distintos canales para el funcionamiento de la comunicación entre el maestro y el esclavo.

En cuanto, al número de canales, AXI implementa en su especificación varios canales de escritura y lectura, los cuales se subdividen en dos más, que son utilizados uno para el control y los registros, mientras que los datos canales de datos se destinan para la transmisión.

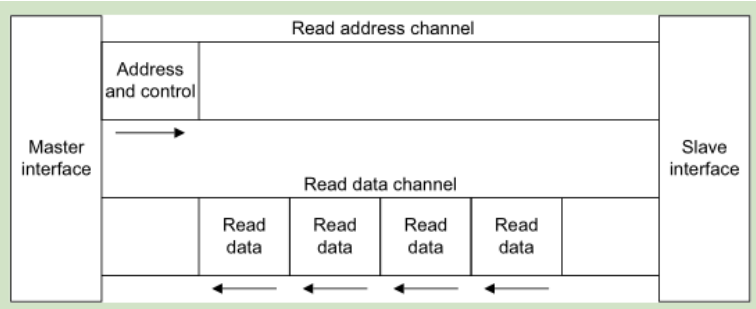


Figura 4.2: Canales de lectura

Como se puede apreciar en la 4.2, la interfaz maestra, se encarga de enviar por el canal de lectura y control las diferentes señales necesarias para indicar la dirección de lectura de datos y todas las señales de control para ello.

Seguidamente el esclavo, se encargará de enviar los datos requeridos mediante las señales recibidas, esto se llevará a cargo por el canal de lectura de datos, el cual se dedica a la transmisión de los datos pedidos.

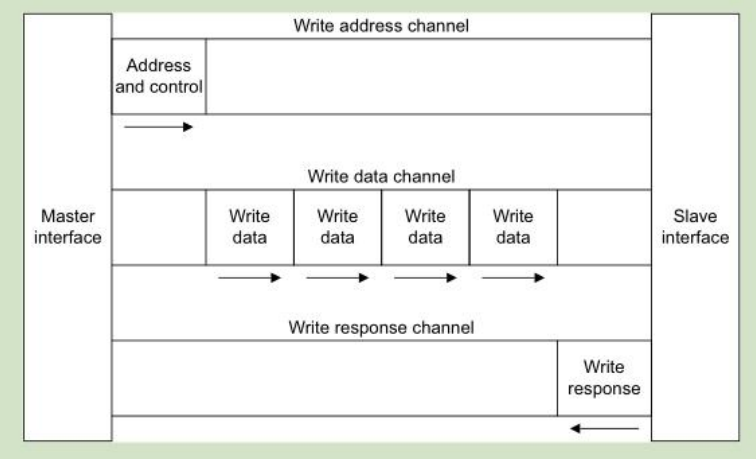


Figura 4.3: Canales de escritura

En cambio, en los canales de escritura, la interfaz esclava solamente es receptora, a diferencia que en los canales de lectura y solamente emitirá respuesta del estado de la escritura.

Como se ve en la 4.3, esta primeramente utiliza el canal de registros para enviar las señales necesarias para indicar la dirección para realizar la escritura y las señales de control. Donde seguidamente se utilizará el canal de escritura de datos, para realizar la transferencia de datos.

Finalmente para saber si todo a ido correcto o ha habido algún fallo en el esclavo en el momento de escritura, este utiliza un canal emisor, "write response", para indicar el estado.

No obstante, en la especificación AXI, a diferencia del AXI4 y AXI4-Lite, que ejecutaría las acciones de escritura y lectura de las formas vistas anteriormente, la interfaz AXI-Stream, solamente realiza escritura de datos en ráfaga. Para ello, ambas partes deberán de tener la interfaz adecuada, para así poder transmitir de maestro a esclavo por un canal unidireccional los datos necesarios.

4.1.2. Señales disponibles

En este apartado vamos a tratar todas las señales disponibles en AMBA4, ya que estas son útiles para la comprensión del proceso de trasmisión de datos y comunicación entre dos intermediarios. Estas señales, estarán divididas por canales. Por tanto, en primer lugar, encontramos las señales encargadas de efectuar la comunicación mediante el mecanismo de "handshake" [25], las cuales están disponibles en todos los canales.

Tabla 4.1: Descripción de las señales del mecanismo de "handshake".

Descripción de la señal	Escritura	Lectura
Utilizada para señalar que los datos de dirección/control del canal de escritura están disponibles	AWVALID	ARVALID
Utilizada para señalar que los datos están disponibles	WVALID	RVALID
Utilizada por el receptor para señalar los datos sobre el estado de escritura	BVALID	
Utilizada para indicar que el receptor está listo para recibir los datos de dirección/control	AWREADY	ARREADY
Utilizada para indicar que el receptor está listo para recibir datos	WREADY	RREADY
Utilizada por el emisor para señalar que está listo para recibir el estado de escritura	BREADY	

En segundo lugar, encontramos las señales de dirección/control las cuales están separadas en 4.2 por proceso de escritura y lectura. Siendo estas señales las responsables de trasmitir todo los datos necesarios para el control del proceso mismo y la ubicación de lectura o de escritura.

Tabla 4.2: Descripción de las señales disponibles en el canal de escritura y lectura de dirección/-control.

Descripción de la señal	Escritura	Lectura
Esta señal es utilizada para la identificación de múltiples transmisiones en un solo canal	AWID	ARID
Señal de dirección de lectura, esta contiene la dirección inicial del primer ”beat”	AWADDR	ARADDR
Señal el cargada de dar el número de ”beats” para la transmisión de datos	AWLEN	ARLEN
Contiene el tamaño de datos para cada ”beat”	AWSIZE	ARSIZE
Señal que selecciona el tipo de ráfaga utilizada en la transmisión	AWBURST	ARBURST
Señal encargada de definir el tipo de memoria	AWCACHE	ARCACHE
Señal encargada de definir el tipo de protección en la transacción, como por ejemplo, privilegios, configuración de los niveles de seguridad o el control de acceso a datos/instrucciones	AWPROT	ARPROT
Esta señal se envía en cada transacción de escritura o lectura, y se encarga de controlar la calidad de la transacción.	AWQOS	ARQOS
Esta señal es utilizada para la identificación de las regiones, lo que permite acceder a múltiples interfaces lógicas a partir de una sola interfaz física	AWREGION	ARREGION
Señal definida por el usuario	AWUSER	ARUSER

Seguidamente, en la siguiente tabla 4.3, tenemos todas las señales pertenecientes a al canal de escritura y lectura de datos, aquí se encuentras las señales encargadas de comunicar el tamaño de datos en la transmisión y los propios datos a transmitir.

Tabla 4.3: Descripción de las señales disponibles en el canal de escritura y lectura de datos

Descripcion de la señal	Escritura	Lectura
Señal para identificar las múltiples transacciones dentro de un canal	WID	RID
Datos de escritura/lectura	WDATA	RDATA
Señal encargada de definir el estado de a partir de la señal RDATA		RRESP
Señal encargada de indicar que bytes de la señal WDATA son validos.	WSTRB	
Señal que identifica el último identificador del ”beat”	WLAST	RLAST
Señal definida por el usuario	WUSER	RUSER

En cuanto a las señales AxREGION y xUSER o AxUSER, son señales opcionales para extender el soporte o funcionamiento. En el caso de AxREGION es una señal utilizada para soportar dos identificadores de región de 4 bits. Mientras que xUSER o AxUSER, son señales utilizadas para definir señales configuradas por el usuario, no obstante AXI4 recomienda no utilizarlas, ya que AXI no define las funciones de estas señales, porque

pueden conllevar a problemas de interoperabilidad en el caso de que dos componentes utilicen las mismas señales.

Por último, en 4.4, encontramos las señales relacionada con el canal de respuesta, el cual solamente encontramos la señales de escritura, ya que este canal solo existe para comunicar el estado de la escritura al emisor. Mientras que en el canal de lectura, se incluye la señal RRESP, vista en 4.3, para el mismo propósito.

Tabla 4.4: Descripción de las señales disponibles en el canal de escritura de respuesta.

Descripción de la señal	Escritura
Esta señal indica el estado de la transacción de escritura	BID
Esta señal indica el estado de la ráfaga de datos	BRESP
Señal definida por el usuario	BUSER

Vistas todas las señales disponibles dentro de la especificación AXI, cabe destacar, que AXI4-Lite cuenta con menos señales, ya que esta se incluye como una estructura similar a una estructura basa en registros pero con características y complejidad reducida.

Tabla 4.5: Señales disponibles en AXI4-Lite

Escritura			Lectura	
Dirección/ Control	Datos	Respuesta	Dirección/ Control	Datos
AWVALID	WVALID	BVALID	ARVALID	RVALID
AWREADY	WREADY	BREADY	ARREADY	RREADY
AWADDR	WDATA	BRESP	ARADDR	RDATA
AWPROT	WSTRB		ARPROT	RRESP

Esto causa que esta interfaz tenga diferencias notables en su versión completa AXI4, como el número de ”beat” para realizar la ráfaga y el uso de todo del ancho de banda del bus. No obstante, aunque el número de señales sea reducido, AXI4-Lite es una interfaz que hereda de AXI4 sus transacciones son completamente compatibles entre ellos, lo que permite la interoperabilidad sin conversiones lógicas adicionales.

4.1.3. Handshake

El mecanismo de ”handshake” utilizado por AXI es un proceso que ocurre al inicio de cada transacción en los cinco canales vistos anteriormente por igual. Para ello el mecanismo de ”handshake” utiliza las señales son VALID y READY, vistas en 4.1.2 de forma que permite el control del flujo de datos trasmitidos por parte de los propios intermediarios, maestro y esclavo.

El uso de estas señales, empezando por VALID, suele ser por parte del emisor, es decir, quien indica cuando los datos de dirección o los datos a transmitir están ya disponibles. Mientras que, la señal READY, es utilizada por parte del receptor para indicar que este puede aceptar esa información.

En cuanto, al inicio de transmisión, esto solamente ocurrirá cuando ambas señales este activadas, lo que posibilita tres escenarios diferentes, el primer caso, es cuando la señal VALID se indica activada, pero la señal READY aún no.

Como se aprecia en la imagen 4.4, VALID se mantendrá activa mientras READY aún no esté activada, en el momento que READY se active, las dos se mantendrán activadas

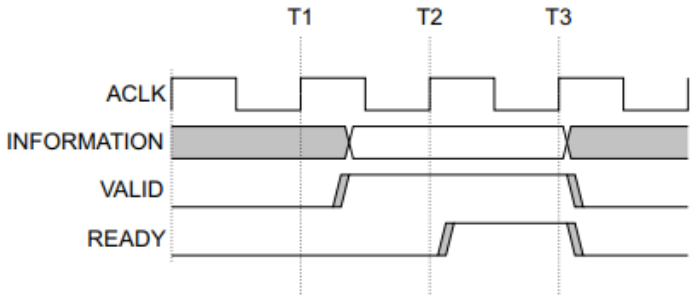


Figura 4.4: Escenario donde la señal VALID se activa antes que la señal READY

hasta el inicio de la transacción de datos. Otro caso a tener en cuenta es si el receptor está disponible para recibir datos antes de que el emisor tenga a estos disponible.

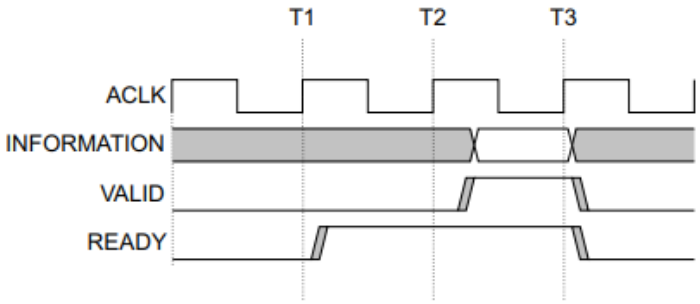


Figura 4.5: Escenario donde la señal VALID se activa después de la señal READY

En este caso 4.5, la señal READY, se indicará activa hasta que el emisor tenga todos los datos de dirección/control o datos disponibles y así poder activar la señal VALID, donde ya, en el siguiente ciclo empezará la trasmisión de los datos. Por último, encontramos el caso donde ambas señales pueden estar activadas al mismo tiempo.

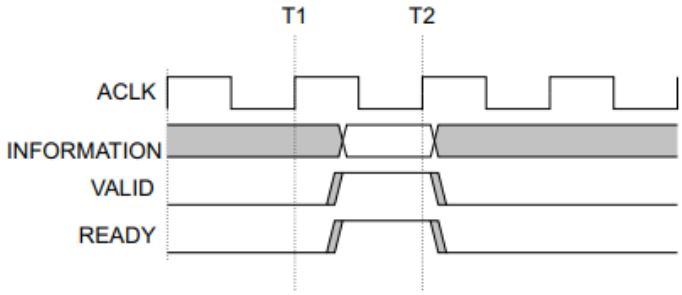


Figura 4.6: Escenario donde ambas señales se activan al mismo tiempo.

Aquí en 4.6, no habrá esperas entre emisor y receptor y solamente esperarán a que la transferencia empiece en el siguiente ciclo, ya que como se dijo anteriormente, la transmisión de datos solamente es posible con ambas señales activadas. Por último, en todos los casos disponibles, una vez la transmisión de datos de dirección/control o simplemente los datos, y ambas señales se desactivarán.

Por otra parte, como se ha mencionado anteriormente, estas señales son independientes en cada canal, no obstante los canales de escritura y lectura de datos incluyen la señal WLAST para la operación de escritura y RLAST para la operación de lectura.

Esta primera, es utilizada por parte del maestro para denotar que es el final de la transferencia de datos en ráfaga y lo mismo por parte de la señal RLAST, pero está a diferencia de la anterior, RLAST es utilizada por el esclavo para comunicar al maestro el fin de la ráfaga de datos. Mientras que los canales de lectura y escritura de dirección/control y el canal de respuesta no incorporaran ninguna señal extra.

4.1.4. Operaciones de escritura y lectura

Vista las señales disponibles en las interfaces AXI y el esquema del proceso, ya podemos profundizar en las operaciones disponibles, siendo estas, escritura y lectura.

Primeramente, en el proceso de escritura, como hemos visto anteriormente, el maestro debe proporcionar las señales necesarias en su distintos canales de transmisión, y finalmente este recibe el estado de la transmisión a través del canal de respuesta por parte del receptor.

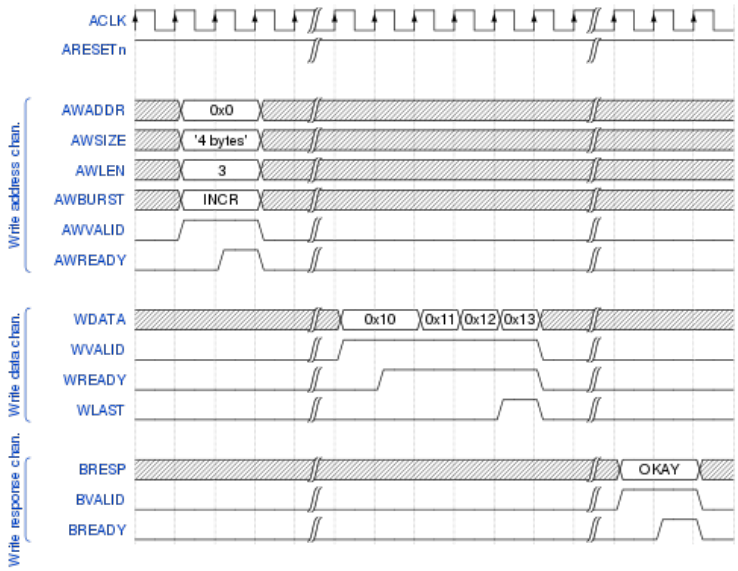


Figura 4.7: Proceso de escritura

El proceso inicia en el canal de dirección/control, donde las señales AWADDR, proporciona la dirección de escritura, ARSIZE, indicando el tamaño de los datos, ARLEN, el tamaño de la ráfaga y AWBURST como el tipo de ráfaga a utilizar.

Seguidamente con las señales de control y dirección sean validas, empezara el proceso de ”handshake”, para iniciar la trasmisión después. La transmisión de los datos se hará a través de WDATA, en el canal de datos y una vez termina la transmisión, el esclavo utilizara BRESP para señalar cual es el estado del proceso en el canal de respuesta.

En cuanto a la operación de lectura, Como se puede ver en 4.8, solamente necesita de dos canales para la lectura de datos.

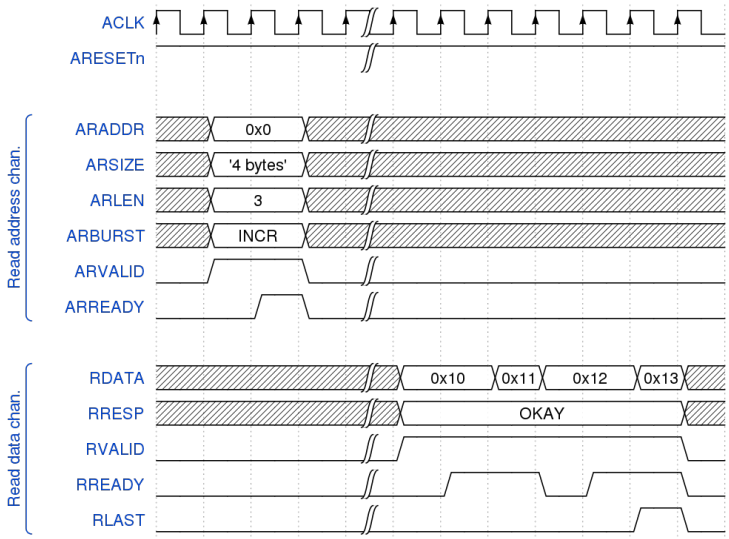


Figura 4.8: Proceso de lectura

El proceso de lectura, el cual no difiere mucho de la anterior operación, no necesita de un canal de respuesta por parte del esclavo. Por tanto, por parte del canal de lectura de dirección/control, será igual que en el canal de escritura, mientras que en el canal de lectura de datos, será el maestro quien generará la señal del estado del proceso, RRESP.

4.1.5. Ráfaga en la especificación AXI

La especificación AXI, está basada en un protocolo de ráfaga de datos, lo que proporciona la capacidad de hacer múltiples transferencias en una sola solicitud, una utilidad muy útil en caso de transmitir una gran cantidad de datos.

Actualmente, la especificación cuenta con tres tipos de ráfagas que pueden ser seleccionadas mediante la señal AWBURST o ARBURST. En primer lugar, tenemos la opción FIXED, la cual transmite la misma dirección en cada "beat". Esta opción es interesante en el caso de acceder múltiples veces a la misma dirección de memoria.

Por otra parte, encontramos la opción INCR, la cual es útil para la lectura y escritura de datos en partes colindantes en memoria.

$$\mathbf{Dirección_i} = \mathbf{DirecciónInicial} + \mathbf{i} \cdot \mathbf{TamañoTransferencia} \tag{4.1}$$

Esta se basa en la dirección inicial, donde devolverá la dirección dependiendo del tamaño del archivo y los "beats" transcurridos. Finalmente, como última opción esta WRAP, una opción similar a la anterior, no obstante, cuando esta excede el límite, se restablece la dirección a la la dirección anterior.

$$\begin{aligned}\mathbf{Dirección_i} = \mathbf{Limite} + (\mathbf{DirecciónInicial} + \mathbf{i} \cdot \mathbf{TamañoTransferencia}) \\ \mathbf{mod}(\mathbf{TamañoBurst} \cdot \mathbf{TamañoTransferencia})\end{aligned} \tag{4.2}$$

Donde,

$$\mathbf{Limite} = \left\lfloor \frac{\mathbf{DirecciónInicial}}{\mathbf{NumeroBytes} \cdot \mathbf{TamañoBurst}} \right\rfloor \cdot (\mathbf{NumeroBytes} \cdot \mathbf{TamañoBurst}) \tag{4.3}$$

CAPÍTULO 5
Vitis High-Level Synthesis

5.1 Vitis HLS Design Flow

En 3.3.1, describimos el flujo de trabajo mediante herramientas HLS, no obstante, el proceso descrito anteriormente, era un proceso generalizado, visto en 3.4, y por tanto, en Vitis HLS, el HLS proporcionado por Xillinx, tendrá pequeños cambios.

Vitis HLS, empieza con la creación del propio proyecto, donde nos propondrá la configuración de un "Testbench", dentro de este se encuentra la función encargada de ejecutar todo el test para comprobar el funcionamiento de nuestra función y el archivo donde se encuentra la función que queremos convertir en un diseño digital. Con todo esto, la propia herramienta, se encarga de crear una solución, donde encontramos todo respecto a la generación del diseño del bloque RTL.

Con toda la configuración del proyecto hecha, ya podemos empezar a desarrollar la función que necesitamos y el testeo de esta, mediante la ejecución por terminal, con el compilador que proporciona Vitis HLS, ya que en este se encuentra las determinas librería de Xillinx para el funcionamiento de diversos "pragmas" o dentro de la propia IDE.

Una vez ya comprobado el correcto funcionamiento de la función, es momento de ejecutar la síntesis de alto nivel o HLS, para generar los archivos RTL. Este paso, genera resultados sobre latencias, intervalos de ejecución, cuantos recursos hemos utilizado con ello y el propio rendimiento. Lo que nos permite tener una idea clara de si lo hecho hasta ese momento es lo que necesitábamos. En caso de los resultados obtenidos no fueran, los deseados, aplicaríamos diversas directivas o volveríamos a reescribir el código en lenguaje de propósito general.

No obstante, si el resultado es el deseado, solamente deberíamos ya ejecutar, la cosimualción C/RTL, lo que permite simular el bloque a partir de lenguajes descriptivos y obtener un resultado, que nos devolverá para poder comprobar con los resultados obtenido en la ejecución en C.

5.2 Síntesis del código C/C++

El HLS, se dedica a sintetizar código de lenguajes de propósito general a código de lenguaje descriptivo, Vitis HLS, utiliza entradas escritas en los lenguajes C y C++, para obtener una salida con los lenguajes de descripción Verilog o VHDL.

Para ello, la herramienta sintetiza el código C/C++ de la siguiente forma:

- **Síntesis de los puertos E/S del RTL:** Como se ha dicho anteriormente, la síntesis empieza con la ”top-function” escrita en C/C++, por tanto, los parámetros de la función que encontramos en la propia cabecera son sintetizados como puertos entrada o salida dentro de la RTL, no obstante el valor devuelto por esta función mediante la orden ”return” solamente podrá ser sintetizado como un puerto de salida. Los propios puertos realizados por la síntesis, dependiendo del tipo de valor a tratar son configurados como puertos AXI4, AXI4-Lite o AXI4-Stream. Aunque estos pueden ser definidos mediante las propias directivas pragma que aporta Xillinx.

- **Síntesis de las subfunciones:** Toda función llamada desde la ”top-function”, se considera una subfunción a esta. El Vitis HLS, se encarga de sintetizar estas subfunciones en bloques dentro de la propia jerarquía del diseño RTL, dando como resultado una jerarquía de módulos o entidades que se corresponden con la jerarquía original en C. Por otra parte, Vitis HLS es capaz de realizar el ”inlining”, lo que consiste en la inserción de la propia subfunción en la ”top-function” o en funciones superiores a esta, para así reducir los ciclos para la ejecución de esta.

- **Síntesis de las matrices:** La matriz, suele ser sintetizada como bloques de memoria, pueden ser RAM (BRAM), LUT RAM o UltraRAM. Y son implementados como puertos para acceder a una RAM ubicada fuera del diseño.

5.3 Xillinx AXI

En 4, se describió la especificación AXI en su versión 4, la cual es a día de hoy la utiliza por Xillinx, ya que sus interfaces de E/S funcionan mediante esta especificación. No obstante, Vitis configura estas E/S dependiendo del tipo de argumento asignado en el código en C/C++. En el caso del argumento asignado sea un escalar, una señal de control, alguna variable global o el propio valor de retorno, el HLS utilizara la interfaz AXI4-Lite, mientras que si estos argumentos son punteros o matrices será configurados con una interfaz AXI4.

Finalmente, AXI4-Stream solo será utilizada en el caso de utilizar como parámetro una asignación del tipo ”hls::stream”. Cabe destacar, que estas interfaces son configurables mediante directivas pragma, no obstante si no asignamos una directiva Vitis HLS configurara esta como se ha mencionado anteriormente.

5.3.1. Interfaz maestra AXI4

La interfaz maestra AXI4, es la encargada de leer y escribir datos en la memoria global, por parte del núcleo, lo que convierte a esta interfaz en una forma perfecta para compartir datos con diferentes elementos, además de aportar ventajas derivadas de la propia especificación AXI, como canales de escritura, vistos en, lectura separados o soporte para las transacciones pendientes o ráfagas de datos con un rendimiento de hasta 19 GB/s.

Vitis HLS configura estas interfaces por defecto, en caso de no utilizara configuraciones para estas. Estas configuraciones suelen tratar el flujo de datos, que por defecto es 64 bytes, o la longitud máxima de datos por ráfaga, siendo estos 16 bytes su longitud máxima, además de también configurar la forma de operar con la interfaz. En cuanto, a la forma de operar, la interfaz se puede configurar de tres formas distintas mediante directivas.

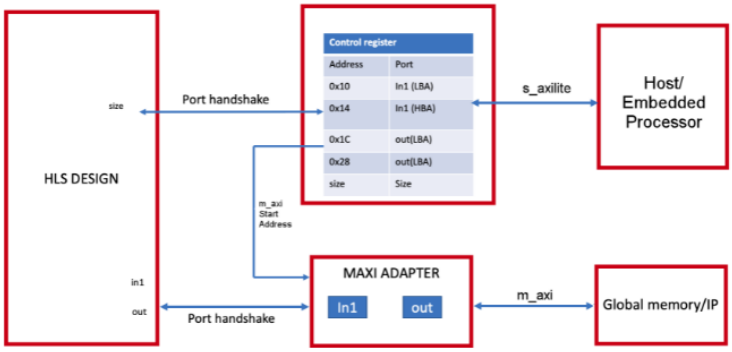


Figura 5.1: AXI en modo esclavo

En primer lugar detallaremos opción ”slave”, la cual configura Vitis HLS por defecto. Con esta opción, conseguimos que la IP se controle por el host mediante una interfaz AXI4-Lite. Dejando la parte de control y dirección a la interfaz AXI4-Lite, donde esta después pasara la dirección a la interfaz maestras, y se almacenará la dirección en una cola, para seguidamente diseño comience a leer los datos la memoria global.

En segundo lugar, encontramos la opción ”off”, que configura una dirección base a la interfaz maestra cuando el IP es utilizado, no obstante esta dirección base no puede cambiarse durante su ejecución. Algo que puede ser útil, cuando utilizaremos la orden ”memcpy”, la cual crea una ráfaga de acceso a la memoria a partir de una dirección base.

Por último, nos quedaría la opción ”direct”, la cual a diferencia de ”off”, esta si permite que se modifique la dirección configurada durante la ejecución, lo que permite poder utilizar la misma interfaz para leer o escribir en distintas ubicaciones.

5.3.2. Interfaz AXI4-Lite

AXI4-Lite es utilizada como canal para pasar los argumentos de la función, los cuales son registros, es decir, punteros que referencien a un escalar, una referencia o el paso por valor. Por otra parte, esta interfaz también actúa como una comunicación entre host o un procesador empotrado y el propio núcleo siendo capaz de iniciar o detener mediante un mecanismo de control a nivel de bloque y escribir o leer datos de el.

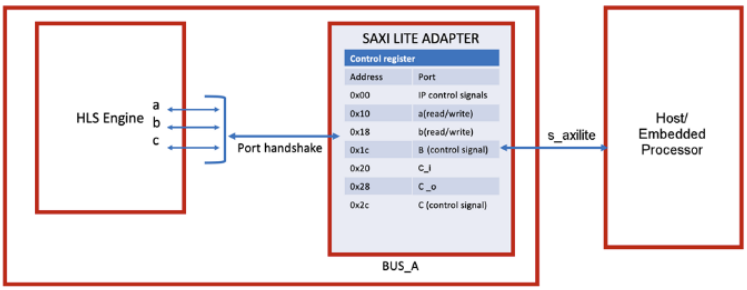


Figura 5.2: Estructura de la interfaz AXI4-Lite

Aquí, en 5.2, se puede aprecia que la interfaz aporta una tabla de registros mapeada, la cual está dividida en dos partes, una designada para las señales de control a nivel de bloque y la otra para los argumentos de la función mapeados en la interfaz.

Estos registros, son reservados por Vitis HLS desde la dirección 0x00 hasta la dirección 0x0C, para los siguientes propósitos:

- **Señales de control [0x00 - 0x03]:** Señales relacionadas con el control, ap_start, ap_done, ap_ready y ap_idle. Aunque si configuramos la interfaz con el protocolo ap_ctrl_chain, en estas señales también se incluirían la señal ap_continue.
- **Registro para activar la interrupción global [0x04 - 0x07]:** Registro de los bits encargados de la interrupción global del proceso de escritura o lectura.
- **Registro para activar la interrupción de IP [0x08 - 0x0B]:** Registros para manipular la interrupción de las señales ap_done, ap_ready.
- **Registro del estado de la interrupción de IP [0x08 - 0x0B]:** Aquí encontramos los bits para consultar el estado de interrupción de ap_done y ap_ready.

Mientras que, a partir de la dirección 0x10, encontramos todas las señales relacionadas con los parámetros, las cuales son asignadas automáticamente a partir de esta dirección. No obstante, el rango que ocupa cada argumento dependerá del tipo de argumento y del protocolo de puerto utilizado.

5.3.3. AXI4-Stream

La interfaz AXI4-Stream en Vitis HLS se encarga de transmitir los datos desde otra fuente de trasmisión al núcleo o al contrario, transmitir los datos fuera del núcleo. Esta interfaz puede aplicarse a cualquier tipo de argumento de entrada, mientras que para los argumentos de salida, se podrá aplicar a cualquier array o puntero. No obstante, al ser AXI4-Stream una interfaz que transfiere los datos de manera secuencial mediante una transmisión, los argumentos utilizados para esta no pueden ser argumentos de escritura y lectura a la vez.

La forma de transmisión en AXI4-Stream, es idéntica a la explicada en 4, aunque en la interfaz aplicada en Vitis HLS, se incluyen dos señales más.

- **TKEEP:** Utilizada para codificar añadiendo bytes nulos al final del paquete, lo que permite preservar la longitud de los paquetes luego de su conversión.
- **TSTRB:** Utilizada para codificar las transmisiones "Sparse". Estos dos puertos permiten multiplexar la posición de los datos y los propios datos de la señal TDATA.

Existe la posibilidad de que se puedan creen rutas de retroalimentación combinacionales cuando se integran múltiples bloques con interfaces AXI4-Stream en un diseño grande. Por eso la especificación registra cuatro tipos de modos para el control de los registros de la propia interfaz, siendo estos:

- **Forward:** En este modo, solamente se registran las señales TDATA y TVALID.
- **Reverse:** Solamente se registra la señal TREADY.
- **Both:** Esta es el modo que se configura por defecto, y se registran todas las señales. TDATA, TREADY y TVALID.
- **Off:** Ninguna de las señales anteriores se registra.

Por último, la interfaz AXI4-Stream permite la configuración de esta misma mediante tipos de datos mediante los tipos 'ap_axis' y 'ap_axiu', lo que permite añadir canales adicionales que incluyen las señales TUSER, TDEST y TID. No obstante, si configuramos los tipos con los parámetros inicializados a 0, estos no añadirán el canal adicional para estas señales.

5.4 Bundles

Vitis HLS, permite agrupar los puertos múltiples interfaces AXI4 y AXI4-Lite, en dos puertos único de salida y de entrada. Esta propriedad, permite optimizar el uso de recursos ya que reducimos la lógica AXI, no obstante esto puede limitar el rendimiento ya que todas las transferencias de esas interfaces pasan por un puerto de entrada o de salida.

La configuración de estas agrupaciones funciona partir de una opción de la directiva 'pragma' de la interfaz a utilizar, esta opción es 'bundle=<nombre del bundle>'.

5.5 Port-level de entrada/salida

5.5.1. Sin protocolos

La configuración 'ap_none', es el protocolo de entrada/salida más simple de todos, ya que especifica que no habrá ningún protocolo a nivel de puerto, donde el argumento implementado solamente será implementado como un puerto de datos, el cual no tiene ninguna señal de entrada y salida asociada para indicar cuando los datos son leído o escritos. En cuanto a otras otros de control, no tendrá ninguna más asociado.

5.5.2. Wire handshakes

El modo de interfaz 'ap_hs', se encarga de asignar señales para realizar el mecanismo de "handshake", visto en 4.1.3, y solo puede ser utilizado en matrices que necesiten leer o escribirse en un orden secuencial, ya que si Vitis detecta que el acceso no será de forma secuencial este detendrá la síntesis.

Dentro de esta derivan otros modos como 'ap_ovld', para el uso de argumento de entrada-salida, ya que separa en dos puertos la entrada y salida. Para la entrada asocia el modo 'ap_none' y para el puerto de salida el modo 'ap_vld'. Este último modo, es similar al modo 'ap_hs', no obstante solo incluye la señal VALID. Otro modo es 'ap_ack', que solo incluye un puerto de reconocimiento.

El modo 'ap_hs', proporciona el puerto de datos, la señal VALID, para indicar cuando los datos son válidos para poder ser leídos y la señal de reconocimiento para indicar cuando los datos han sido leídos. En cuanto a los modos derivados por el modo 'ap_hs':

Otro modo disponible, es 'ap_vld' proporciona señales para el puerto de datos y la señal VALID para indicar cuando los datos pueden ser leídos. No obstante, dependiendo de si el argumento es de entrada o salida, el diseño actuará de una forma u otra diferente. En caso del argumento ser de entrada, el diseño leerá el puerto de datos tan pronto como la señal VALID este activada. Mientras que, si el argumento es de salida, Vitis implementa un puerto para la señal VALID, para indicar cuando los datos en el puerto de salida son válidos.

Por otra parte, el modo 'ap_ovld' dispone del puerto de datos y los dos tipos de puertos mencionados anteriormente, 'ap_none' para la entrada y 'ap_vld' para la salida.

Por último, nos que el modo 'ap_ack' al igual que los anteriores dispone de un puerto de datos, aunque en vez de proporcionar la señal VALID, este proporciona una señal de reconocimiento para indicar cuando los datos son utilizados.

Los argumentos de entrada, el diseño genera una señal 'ack' en cada ciclo cuando la entrada es leída. Mientras que para los argumentos de salida, genera un puerto de entrada para la señal de reconocimiento y así confirmar si la salida fue leída.

5.5.3. Memory inteface

En Vitis, las matrices como parte de los argumentos de la función, son implementadas con una interfaz 'ap_memory' por defecto, dando como resultado una interfaz de bloque RAM estándar. Esta interfaz puede ser implementada con single-port y dual-port, aunque si Vitis determina que el uso de una interfaz con dual-port reduce el tiempo, aplicara automáticamente la interfaz con dual-port.

También existe la interfaz 'BRAM', pero es idéntica a la anterior y su única diferencia es la forma en que muestra los bloques. Por una parte, 'ap_memory' mostrar la interfaz con puertos discretos y en cuanto a 'BRAM' aparecer como un solo puerto agrupado.

5.6 Block-level Protocol

La ejecución de los modos en el núcleo de Vitis , esta especificado por el protocolo de control a nivel de bloque, y se dispone de tres tipos de modos de ejecución. El primer modo de ejecución, 'ap_ctrl_hs', configura una ejecucion segmentada que permite al kernel ejecutar operaciones tan pronto como los datos esten disponibles. Este modo de ejecución contiene las siguientes señales:

- **ap_start:** Indica cuando el bloque puede empezar a funcionar, y se mantendrá activada hasta que la señal 'ap_ready' lo este también. Esta señal, puede permanecer hasta el inició de la siguiente transacción o desactivarse, lo que conllevara a que el diseño y la transacción actual se detengan.

- **ap_idle:** Indica cuando el diseño ya no esta operando o está inactivo.

- **ap_ready:** Indica cuando el diseño inicia una operación. Esta señal permite saber si existe otra operación, ya que en caso de 'ap_start' estar activada y la señal de ready desactivada, el diseño estará a la espera hasta que llegue una operación o se active la señal 'ap_done'.

- **ap_done:** Indica cuando el bloque ha completado su función.

Por otra parte, se dispone del modo 'ap_ctrl_chain', que configura una ejecución de forma secuencial, donde el núcleo debe completar un ciclo para empezar otro. Este modo de ejecución es contiene las señales del modo anterior ademas de una adicional.

- **ap_continue:** Indica cuando el bloque descendente está listo para nuevas entradas de datos. Si la señal esta activada, mientras 'ap_done' también está activada, el diseño continua funcionando, pero en caso contrario, donde 'ap_continue' se se encuentra desactivada, el diseño deja de funcionar.

Otro aspecto que destacar es la señal 'ap_return', la cual si está definida mantendrá los datos validos en el puerto mientras la señal 'ap_done' este activada.

Por último, en el modo 'ap_ctrl_none', la forma de ejecución del bloque depende de la disponibilidad de los datos, es decir, este estará en funcionamiento mientras haya datos disponibles. Este modo de ejecución, hace que el bloque no contenga ninguna señal para el mecanismo de "handshake".

5.7 Directivas pragma

Vitis, permite la configuración mediante su propia IDE o a partir de directivas pragma, estas directivas pragma, son una construcción dentro del lenguaje, que especifica al compilador como debe procesar su entrada.

Estas directivas afectan al comportamiento global, aunque también pueden afectar solamente en una sección local, pero no realiza ninguna acción sobre el lenguaje en sí, sino solo cambia el comportamiento del compilador. Normalmente, estas directivas suelen ser prescriptivas y deben seguirse, aunque podemos encontrar directivas que pueden ser sugerencias opcionales del compilador y puede ignorarse.

Vitis HLS utiliza, las directivas pragma para implementar las diferentes configuración u optimizaciones, para luego sintetizar el código en C/C++ en base a los pragmas implementados. Aunque, en este capítulo solo veremos las directivas pragma destinadas a la configuración del RTL, ya que las directivas de optimización las veremos en 7.

5.7.1. Directivas para la configuración de interfaces AXI4

Para la configuración de las interfaces AXI, Vitis, permite configurarlas de dos formas, una mediante la propia IDE, y la segunda por las directivas pragma.

La construcción de las directivas encargadas de asignar las interfaces AXI, es la siguiente:

```
1 #pragma HLS INTERFACE mode=<tipo de interfaz> <configuraciones de la interfaz>
```

Listing 5.1: Directiva pragma para la configuración de interfaces

En primer lugar, la configuración 'mode', define que tipo de interfaz vamos a aplicar, no obstante, existen varios tipos de interfaz y dependiendo del tipo de interfaz podremos aplicar una configuración u otra.

En caso de la interfaz, 'm_axi', disponemos de las siguientes opciones:

- **depth:** Configura el tamaño máximo del adaptador FIFO.
- **offset:** Configura una de las opciones de ejecuciones comentadas en 5.3.1.
- **num_read_outstanding y num_write_outstanding:** Configura el numero máximo de solicitudes de lectura/escritura que se pueden realizar al bus AXI.
- **max_read_burst_length y max_write_burst_length:** Configuran el número de datos escritos o leídos durante una transferencia en ráfaga.

En cuanto a los tipos, 'ap_none', 'ap_vld', 'ap_ack', 'ap_hs' y 'ap_ovld', comparten el mismo número de configuraciones. 'register', la cual registra cualquier señal del protocolo que sea relevante, esta configuración se encuentra también en tipos 's_axilite' y 'axis'.

Respecto a la interfaz 'axis', la única configuración, es 'register_mode' y dispone de cuatro tipos de registros, descritos en 5.3.3. Por otra parte, en la interfaz 's_axilite', dispone de dos configuraciones únicas. 'Offset', donde se especifica la dirección en el mapa de registros y 'clock' la cual define que señal de reloj utilizará la interfaz.

En cuanto, a los tipos, 'ap_memory', 'bram', 'ap_crtl_none', 'ap_ctrl_chain', ap_ctrl_hs', no continen configuraciones unicas.

Finalmente, todas las interfaces comparten, las siguientes configuraciones, estas son: 'port' para especificar el nombre del argumento al cual asociar la interfaz y 'bundle', la cual se especifica en 5.4.

5.8 Conclusión

En conclusión, saber conocer los tipos de interfaces disponibles y sus configuraciones es útil para configurar los puertos de entrada y salida de nuestra función principal y así poder tener mayor control sobre estos o disminuir el número de señales necesarias para un proceso,

Por otra parte, también se aplica al núcleo. Por ejemplo, en caso de querer una ejecución segmentada para todo el proceso, podemos utilizar la opción 'ap_ctrl_hs', no obstante en querer segmentar solamente bloques de código o bucles, entonces utilizaríamos la directiva de optimización 'PIPELINE', detallada en 7.

Finalmente, otro aspecto a tener en cuenta es que las interfaces solamente, se podrán configurar en los argumentos o el return de la función principal, ya que Vitis HLS ignora esas interfaces en subfunciones.

CAPÍTULO 6
Realizar un código sintetizable

En este capítulo vamos a tratar todas las restricciones que tiene Vitis HLS, para así poder realizar un código sintetizable. Si bien, Vitis HLS soporta un amplio rango de lenguajes basados en C/C++, hay algunas construcciones que no pueden ser sintetizadas o producen errores más adelante en el flujo de diseño.

Por tanto, para que el código sea sintetizable en Vitis HLS, se deben cumplir las siguientes restricciones:

- La función, debe contener toda la funcionalidad del diseño.
- Ninguna de la función se puede realizar por llamadas sistema operativo.
- Las construcciones deben ser de tamaño fijo o limitado.

6.1 System calls

Vitis HLS, se encarga de ignorar las llamadas al sistema que solamente muestran datos que no tienen impacto en el funcionamiento del algoritmo, como pueden ser, 'printf' y 'fprintf', las cuales son funciones para la impresión de datos. Por tanto, estas instrucciones no serán sintetizadas, ya que estas funciones tienen relación con la funcionalidad de alguna tarea por parte del sistema operativo.

Estas órdenes deben eliminarse del código a sintetizar, ya que pueden dar problemas más adelante en el flujo de diseño, no obstante, podemos utilizar estas llamadas al sistema y hacer que no se tengan en cuenta para la síntesis. Esto, se hará mediante la macro '__SYNTHESIS__'.

6.2 Uso de la memoria dinámica

Cualquier llamada al sistema que estén destinadas al manejo de memoria dinámica dentro del sistema, como las llamadas, 'malloc', 'alloc' dedicadas al reservado de memoria o 'free' al liberado de esta misma en tiempo de ejecución. No obstante, para ser sintetizable una implementación de hardware el diseño debe tener especificados todos los recursos necesarios y por tanto, Vitis no soporta para la síntesis objectos que son creados o destruidos dinámicamente.

En 6.1, se puede aprecia que mientras la macro 'NO_SYNTH' se encarga de cambiar el tipo de reservado de memoria, es decir, si el estamos ejecutando el binario generado por el compilador, el reservado se hará mediante la llamada al sistema malloc, no obstante, si

```
#define NO_SYNTH

#ifdef NO_SYNTH
   //Reservado dinamico
   int arr = malloc(sizeof(int) * 64);
#else
   //Reservado estatico
   int arr[64];
#endif
```

Listing 6.1: Reservado de memoria dinámicamente y estáticamente

realizamos la síntesis solamente tendremos que dejar de definir la macro 'NO_SYNTH' para que el reservado de memoria sea estático y no de error en la síntesis.

Por lo tanto, todo reservado de memoria deberá ser estática, ya que esta se reservadas durante la compilación del programa y con ello la especificación de todos los recursos utilizados.

6.3 Punteros

Los punteros son variables que apuntan a una dirección de memoria, siendo estos elementos muy usados en C/C++. Vitis, permite el uso de punteros, no obstante recomienda evitar su uso en caso de escribir y leer múltiples veces in la misma función.

Estos, tienen ciertas restricciones de uso, en caso de querer generar un código sintetizable. En el lenguaje C, existe la posibilidad de generar una matriz de punteros que apunte a otros punteros, cosa que en Vitis HLS no está permitido y la síntesis producirá un error, por tanto, solamente podremos generar matrices de punteros que apunten a una escalar o una matriz de escalares. Además, estos tampoco pueden hacer referencia a una función.

Otra restricción impuesta por Vitis, para poder realizar la síntesis, es la conversión de tipos o 'casting', una propiedad de los lenguajes de propósito general para poder convertir tipos de variables con propiedades similares en otros, como puede ser la conversión de un integer a char. Vitis, permite la conversión de tipos siempre que los tipos estandarizados es decir, los tipos nativos que aporta el lenguaje C, mientras que en los tipos generalizados, es decir, tipos creados por nosotros, como una estructura, no lo permitirá, y resultará en error durante la síntesis.

Finalmente, los punteros pueden ser accedidos múltiples veces por la misma función para la escritura o lectura, puede conllevar a comportamiento anómalos después de la síntesis. Para ello, se debe utilizar la palabra reservada 'volatile'.

6.4 Template metaprogamming y recursión

La recursión, consiste en que una función sea capaz de llamarse a si misma para resolver una problemática. No obstante, esta técnica incumple una de las normas para poder sintetizar un código, ya que no permite saber el tamaño de la construcción y por tanto daría un error. Por tanto, Vitis HLS, no permite la recursión o la "tail-recursion".

No obstante, podemos generar "tail-recursión" mediante una técnica llamada, "Template metaprograming"[31], la cual es usada por el compilador para generar un código fuente temporal. Este código temporal, es combinado con el resto del código fuente para

ser compilado después. Por tanto, al ser todo producido en tiempo de compilación, esta técnica permite saber todo el tamaño de la construcción, ya que estos "templates" pueden incluir constantes, estructuras de datos y funciones completas en tiempo de compilación.

6.5 Tipos de datos sintetizables

En Vitis, encontramos varios tipos de datos que pueden ser sintetizados, como tipos estándares en C/C++, tipos compuestos o tipos definidos por el usuario con diferentes calificadores de tipo. En cuanto a los calificadores de tipo, estos tienen un gran impacto en la creación de componentes en el momento de la síntesis, ya que la propia herramienta aplica diferentes optimizaciones para intentar conseguir el diseño optimo, incluso afectan hasta el comportamiento del propio diseño. Entre los calificadores de tipo sintetizables encontramos:

- **Volatile:** Este calificador, tiene un impacto sobre cuantas veces el RTL escribirá o leerá cuando acceda a los punteros en las interfaces de la función. No obstante, estos accesos no pueden ser optimizados por el diseño. Además, no puede ser utilizado en los tipos arbitrarios para realizar operaciones aritméticas.
- **Static:** Este calificador permite a las variables mantener su valor entre funciones, y su comportamiento equivalente en un diseño de hardware es una variable registrada. Por ultimo, hay que destacar que el valor debe mantenerse entre las invocaciones de funciones y el diseño.
- **Conts:** Este calificador, especifica que el valor de la variable nunca debe ser actualizado después de su inicialización, y aunque no pueda ser escrita más veces si podrá ser leída.

En cuanto, a los tipos sintetizables en Vitis HLS, primeramente encontramos los tipos estándares que aporta el lenguaje C/C++. Que son:

- (unsigned) char
- (unsigned) short
- (unsigned) int
- (unsigned) long
- (unsigned) long long
- intN_t, solamente cuando N es 8, 16, 32, 64
- float
- double

Por otra parte, los tipos nativos del lenguaje C, tienden a tener limites de 8, 16, 32 y 64 bits, sin embargo, esto puede resultar ineficiente en una implementación de hardware, ya que en caso de utilizar un tipo que acabe ocupando 10 bits, acabaría requiriendo un tipo de mayor tamaño de los nombrados anteriormente, en este caso 16 bits. Por ello, existen los tipos de precisión arbitraria los cuales permiten especificar el tamaño exacto de bits que vamos a utilizar, lo que reduce el hardware innecesario en el diseño.

Dentro de los datos de precisión arbitraria encontramos dos tipos más, que aporta Xillinx mediante las librerías, 'ap_int.h' y 'ap_fixed.h'. La 'ap_int.h', aporta precisión arbitraria al tipo integer, y admite como tamaño máximo 1024 bits.

Mientras que, el tipo que aporta 'ap_fixed', representa un tipo dato de tamaño configurable fraccionado en dos partes, una parte de bits para representar la parte entera y la restante para la parte decimal de un valor. De esta forma, permite a este tipo ser un remplazo para los valores de coma flotante que requieren mucho más de ciclos para ser completados, haciendo que la implementación pueda ser más rápida y con menos área utilizada.

En cuanto, a los tipos compuestos, Vitis soporta estructuras, tipos enumerados y uniones. En caso de las estructuras, son un tipo de dato que a diferencia de las matrices, son capaces de almacenar distintos tipos de datos, esto conlleva a poder tener diversos alineamientos y "paddings", ya que cada dato tendrá un tamaño en bits distinto al otro en casos de que la estructura almacene distintos tipos de datos. Para ello encontramos varias configuraciones, que son:

- **Disaggregate:** Esta opción creara una estructura desagregada con elementos individuales, los cuales su número y tipo se determinarán por el contenido de la estructura misma.
- **Aggregate:** En esta opción a diferencia de la anterior, los elementos son estructurados como una sola unidad de datos, y puede implicar que para ello se utilice algún 'padding' en el proceso.
- **Aligned:** Esta opción alinea las estructuras a un determinado ancho añadiendo "padding" entre los elementos de la estructura.
- **Packed:** En esta opción, Vitis empaqueta los elementos de la estructura para que el tamaño real sea la suma de todos los elementos.

Seguidamente, encontramos los tipos enumerados, los cuales son optimizados por Vitis HLS para utilizar el numero requerido de bits, no obstante, estos ocupan 32 bits en caso de ser argumentos de la función a sintetizar.

Por último, encontramos las uniones o "unions" una estructura similar a las estructuras, pero a diferencias de estas comparten el mismo espació en memoria. No obstante, en la síntesis no se garantiza que la unión sea capaz de compartir los mismos registros, y al igual que los tipos enumerados, Vitis HLS, se encarga de optimizar este tipo de dato compuesto.

Aunque, el uso de uniones tiene ciertas restricciones, la función principal no los permite como argumentos, al igual que no pueden contener punteros de diferente tipo, acceso a una variable mediante otra o realizar "casting" entre tipos nativos y tipos definidos por el usuario.

6.6 Vectores

Los vectores son un tipo de matriz de una sola dimensión y al igual que las estructuras tipo array, este solamente es capaz de almacenar valores de un solo tipo. Dentro del lenguaje C++, dentro de la librería estándar "std" o "Standard Template Library"[3], la cual es una colección de clases y de funciones para utilizar y manipular: contenedores, funciones, cadenas de texto, flujos de E/S y también da soporte a la mayoría de características del lenguaje C++. Dentro de los contenedores disponibles encontramos "std::vector", que aporta la estructura comentada anteriormente, no obstante esta estructura reserva memoria dinámicamente, cosa que rompe con una de las normas para que el código pueda ser sintetizable, ya que toda construcción debe ser de tamaño fijo.

Para ello, Vitis HLS incluye esta construcción "hls::vector", la cual no hace uso de la reserva dinámica para que esta aportar todas las facilidades de esta estructura al diseñador y a su vez pueda ser sintetizable.

La configuración de esta construcción se produce mediante un "template", el cual dispone de dos configuraciones.

- **T:** Define el tipo del dato, este puede ser un tipo primitivo del lenguaje o un tipo definido por el usuario.
- **N:** Define el número de elementos de tipo T.

Con la siguiente configuración, garantizaremos un espació contiguo de tamaño

$$\textbf{TamañoTipo} \cdot \textbf{NumeroElementos} \tag{6.1}$$

6.7 Conclusion

En conclusión, saber hasta dónde nos restringe la síntesis es beneficioso a la hora de pensar en un diseño, ya que no contamos con muchas de las funciones que nos proporciona el lenguaje C como, el manejo dinámico de la memoria, compatibilidad total con los punteros, etc. Por tanto siempre tendremos que buscar una alternativa a la restricción

Un ejemplo sería, la recursión, la cual tiene restricciones en la síntesis. Donde, la primera opción a esta siempre sería encontrar la forma iterativa, ya que es la forma más segura de sintetizar sin tener posibles errores y en caso de no escoger la opción iterativa se podría probar con el template meta-programming.

CAPÍTULO 7
Optimización del código

En este capítulo vamos a tratar las optimizaciones que podemos aplicar a nuestro código una vez sea sintetizable. Normalmente Vitis HLS, ya aplica algunas optimizaciones automáticamente como la alineación en los tipos compuesto o el 'pipelining'. No obstante, aún podemos aplicar más optimizaciones, mediante directivas pragma. Estas optimizaciones las podemos dividir en cuatro grupos:

- **Optimizaciones para el rendimiento:** Se aplican optimizaciones como canalizar las tareas para mejorar el rendimiento, el flujo de datos entre tareas, u optimizar estructuras para perfeccionar los problemas de dirección.
- **Optimizaciones para la latencia:** Se utilizan técnicas para la restricción de la latencia, además de eliminar las transiciones del bucle para reducir los ciclos de reloj.
- **Optimizaciones para reducir el área:** Se centra en como las operaciones han sido implementadas, controlando su número y su implementación en el hardware.
- **Optimizaciones para la optimización de la lógica:** Analiza las optimizaciones que afectan a la implementación de la RTL.

7.1 Optimizaciones de rendimiento

Para optimizar el rendimiento, disponemos de la técnica de "pipelining", la cual permite que las operaciones puedan ejecutarse concurrentemente. Esta función se puede aplicar a bucles y funciones.

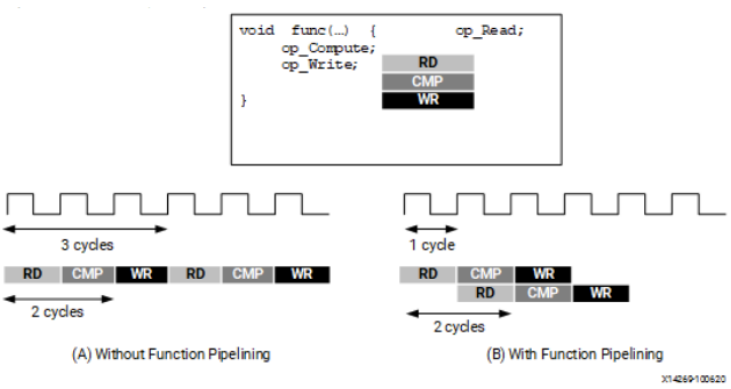


Figura 7.1: Pipelining aplicado a un bucle

El uso del ”pipelining”, se puede ver como ahora la ejecución de operaciones no tienen un orden secuencial y pueden ejecutarse sin terminar la ejecución de la anterior en el mismo ciclo, lo que conlleva a utilizar menos ciclos para completar la operación.

Los bucles a los cuales se les ha aplicado el ”pipelining”, mediante la directiva se les puede aplicar una opción llamada ”rewind”, que permite la ejecución de las operaciones de la siguiente iteración, y así no tener que esperar a que se complete el ciclo actual para ejecutar el siguiente.

No obstante, el uso del ”pipelining” tiene un comportamiento diferente en cuanto a utilizarlo en funciones, que al utilizarlo en bucles. En el caso de las funciones, el ”pipelining” se aplica y nunca termina, mientras que en el caso de los bucles el ”pipelining” termina al terminar todas las iteraciones del bucle.

Esta técnica se puede mejorar mediante la técnica de ”loop unrolling”, ya que Vitis por defecto no la aplica y genera recursos para el hardware que son utilizados en cada iteración del bucle.

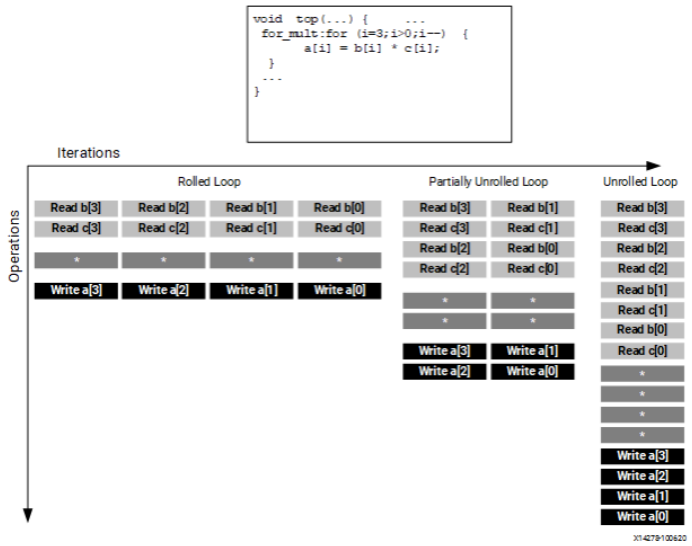


Figura 7.2: Tipos de unroll en Vitis HLS

Como se puede ver en la imagen, Vitis HLS permite alternar de tres formas para el acceso a los recursos para cada operación en cada iteración:

- **Rolled loop:** Cada iteración se realiza en ciclos de reloj separados.
- **Unrolled loop parcial:** Esta opción realiza un ”unroll” parcial, para realizar menos iteraciones.
- **Unrolled loop:** Esta opción realiza un ”unroll” total, y permite realizar todas las operaciones en una sola iteración.

Aunque la opción que realiza un ”unroll” completo, realizara todas las operaciones en paralelo siempre que todas las dependencias y recursos lo permitan, si no las operaciones se realizar tan pronto como los datos estén listos.

El ”pipelining” puede generar problemas en caso de intentar ejecutar una operación debido a las limitaciones de los puertos de memoria. Para ello Vitis HLS aporta directivas para particionar la matriz, donde cada partición contendrá sus propios puertos para su lectura y escritura.

La directiva "array_partition" dispone de tres opciones para particionar la matriz, esta son:

- **block:** Divide la matriz original en bloques consecutivos del mismo tamaño.
- **cyclic:** Divide la matriz original en bloques de igual tamaño, pero intercalando los elementos de la matriz original.
- **complete:** Divide la matriz original en bloques únicos para cada valor.

Por último, tener en cuenta de no generar dependencias en los recursos, ya que conlleva a usar más ciclos para acabar la operación. Por tanto debemos tener en cuenta las siguientes dependencias:

- **Dependencia verdadera (RAW):** Esta dependencia se produce cuando un dato intenta ser leído después de la escritura.
- **Anti-dependencia (WAR):** Esta dependencia ocurre cuando un valor intenta ser actualizado mientras está siendo leído.
- **Escritura después de una escritura (WAW):** Esta dependencia ocurre cuando un registro deba actualizarse en un orden concreto para que no afecte a otras instrucciones.

Por otra parte, también se pueden encontrar dependencias falsas, la cuales no existe en el código, pero el compilador al ser demasiado conservador puede determinarlas como dependencias. Para ello, Vitis HLS, aporta la directiva "dependence", que contiene dos opciones para tratar estas dependencias. Estas son:

- **Inter:** Esta especifica la dependencia entre diferentes iteraciones en el mismo bucle. En caso, de ser definida con FALSE, Vitis, permitirá realizar operaciones en paralelo, en caso contrario, se evitarán las operaciones concurrentes.
- **Intra:** Esta especifica la dependencia dentro de la misma iteración del bucle y cuando se especifica con valor FALSE, Vitis podría mover libremente las operaciones para potenciar el rendimiento o el uso de menos área. En caso contrario, las operaciones se realizarán en el orden especificado.

7.2 Optimizaciones de latencia

Para la optimización de la latencia disponemos de varias formas, la primera de todas es mediante la directiva "latency", que configura una latencia máxima o mínima, donde se garantiza que todas las operaciones de la función se completará dentro del rango especificado. Aunque, en caso de no poder cumplir con la restricción de latencia máxima especificada, Vitis intenta obtener el mejor resultado posible.

Mientras que, si Vitis no cumple con la latencia mínima especificada, este podrá diseñar un diseño con una latencia más baja que la mínima requerida, para seguidamente insertar varios ciclos de reloj ficticios para cumplir.

Otra forma para mejorar la latencia es fusionar bucles secuenciales. Ya que, los bucles "rolled" o enrollados, crean al menos un estado en el diseño de la máquina de estados

finitos y en varios bucles puede llegar a crearse ciclos de reloj adicionales que son innecesarios. Para ello, Vitis, aporta la directiva "loop_merge" que buscara fusionar todos los bucles que encuentre dentro de su ámbito.

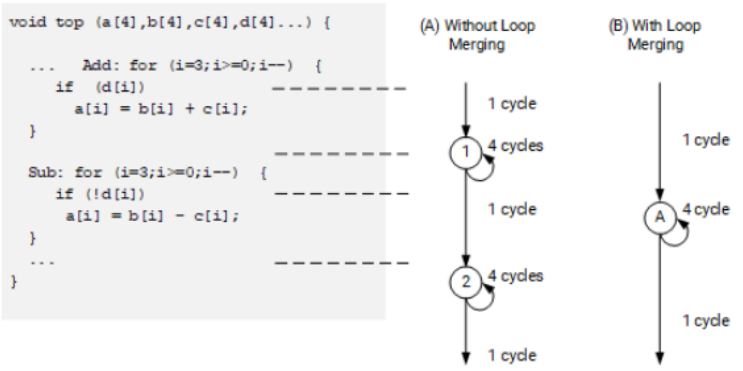


Figura 7.3: Fusionado de dos bucles mediante la directiva "loop_merge".

Como se aprecia en 7.3, al aplicar la directiva reducimos los estados y los ciclos de reloj en la máquina de estados finita, ya que esta directiva crea una nueva estructura de control con la fusión de los bucles.

No obstante, esta directiva tiene varias restricciones para poder utilizarse, como los límites del propio bucle, en caso de que estos limites sean variables, todas estas deberán tener el mismo valor, pero si los limites son constantes, la constante con mayor valor será la que establezca los limites dentro de los bucles fusionados. Por otra parte, tampoco podemos fusionar dos bucles con limites que son establecidos por variables y a su vez por algunas constantes. Otra restricción proviene de los bucles que contengan accesos FIFO, ya que el fusionado cambia el orden de escritura y lectura.

7.3 Optimizaciones de área

Al igual que en las optimizaciones para latencia, también se dispone de varias formas para optimizar el código en área. La primera que encontramos en la técnica de "inlining", la cual puede mejorar el área al permitir que los componentes dentro de la función se compartan y optimicen mejor con la lógica de la función llamada. Vitis HLS, realiza esta optimización automáticamente, pero solamente en funciones pequeñas.

El "inlining", se configura a partir de una directiva, no obstante también puede ser utilizada mediante la configuración "off" para especificar que la función no puede ser optimizada mediante esta técnica por Vitis. Por otra parte, en caso de querer que todas las funciones se inserten recursivamente, se utilizará la configuración "recursive", la cual eliminara toda la jerarquía de funciones del diseño.

Otra forma para la optimización de área es "array_reshape", una directiva que se encarga de reasignar la matriz original en forma vertical, y así reducir la cantidad de RAM consumida, además de añadir accesos paralelos a los datos.

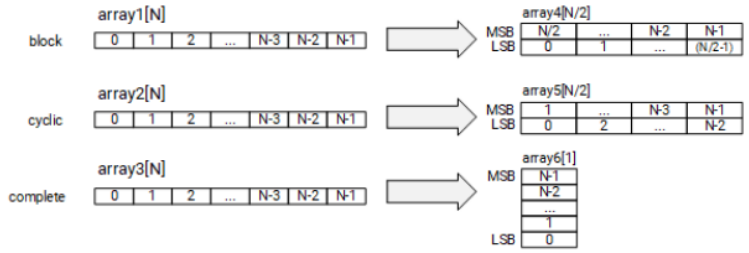


Figura 7.4: Remodelación de matriz mediante la directiva "array_reshape".

En cuanto, a las configuraciones disponibles, tiene las mismas que "array_partition" y realizan la misma función.

Por otra parte, también una forma de optimizar mediante la instanciación de funciones, una técnica que aporta beneficios a la optimización en área y a su vez mantiene la jerarquía de las funciones, además de realizar optimizaciones locales específicas. Esta técnica aporta simplicidad en la lógica de control en torno a la llamada de la función y mejoras importantes de latencia y rendimiento.

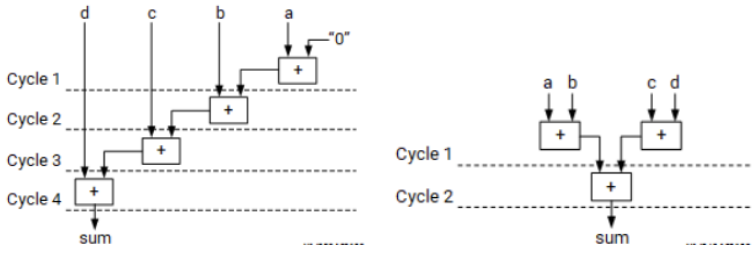
La configuración de esta técnica se hace a partir de la directiva "function_instantiate", la cual se beneficia de las entradas a una función que puedan tener un valor constante y lo utiliza a su favor para simplificar las estructuras de control, y así poder realizar bloques de la función más pequeños y optimizados. Por último, hay que aclarar que esta directiva permite optimizar cada instancia de forma independiente.

Finalmente, otra forma de optimizar el área es mediante el uso de tipos de precisión arbitraria comentados en 6.5, los cuales permiten que se pueda utilizar los bits necesarios.

7.4 Optimizaciones destinadas a la lógica

En cuanto, optimizaciones para las expresiones lógicas, encontramos la directiva "expression_balance", que activa una función para reconstruir el árbol mediante la reorganización de las expresiones para obtener un árbol más equilibrado y a su vez reducir la latencia.

Vitis HLS, para operaciones en enteros, viene activada por defecto, sin embargo, en el caso de las operaciones de coma flotante viene desactivado. No obstante, podemos activar la función mediante la directiva o desactivarla mediante la opción "off".



(a) Árbol sin reestructurar **(b)** Árbol reestructurado

Figura 7.5: Diferencia entre un árbol sin reestructurar con uno reestructurado mediante la directiva "expression_balance".

Como se puede apreciar, la directiva "expression_balance" ha visto que las operaciones de suma de 'a+b' y 'c+d', se pueden ejecutar paralelamente, y por tanto ha realizado esa acción, lo que conlleva a una reducción de la latencia. No obstante, una vez utilizada

la directiva, no se podrá compartir las expresiones, lo que supondrá en un aumento de área.

7.5 Otras directivas para optimización

En los apartados anteriores, tratamos las diferentes optimizaciones disponibles, no obstante se dispone de varias directivas más que juntamente con las tratadas en los apartados anteriores pueden generar un buen resultado, como podría ser el uso del "inlining" y la directiva "allocation", para limitar el numero de instancias y se comparta la misma instancia siempre.

7.5.1. Allocation

Esta directiva especifica un límite máximo de operaciones implementadas o funciones utilizadas, no obstante el uso de esta puede conllevar a que los recursos deban ser compartidos y por tanto puede llevar a un aumento de la latencia.

Dentro de los tipos se aceptan las siguientes configuraciones:

- **function:** Especifica que la directiva solamente se aplicará a funciones.
- **operation:** Especifica que la directiva solamente se aplica a operaciones.
- **core:** Especifica que la directiva se aplica solamente a los núcleos, en especial a los componentes de hardware que se han utilizado para crear el diseño.

7.5.2. Loop flatten

La directiva "loop_flatten", permite que los bucles anidados se junten en una jerarquía de un bucle único con una latencia mejorada.

Normalmente, para pasar del bucle al bucle anidado y viceversa, se requiere un ciclo de reloj. Para ello, la directiva optimizará estos en una sola jerarquía de bucle único ahorra ciclos de reloj, lo que permite dar una mejora de optimización de la lógica del bucle.

No obstante, la directiva tiene ciertas restricciones a la hora de poder utilizarla. Por tanto, solamente podremos utilizarla en bucles donde toda la lógica y operaciones ocurran en el bucle anidado y los limites sean constante o donde el bucle exterior solamente contenga el límite por medio de una variable.

Por otra parte, Vitis HLS, puede aplicar esta directiva en caso de ser una optimización que el crea conveniente. Por tanto, para desactivar la función que crea el "loop_flatten", añadiremos la opción off.

7.5.3. Dataflow

Esta directiva, permite la canalización a nivel de tarea, permitiendo que las funciones y bucles se superpongan en su operación, lo que aumenta la concurrencia de la implementación, lo que conlleva a una reducción de ciclos de reloj que conlleva a disminuir la latencia y a un aumento de rendimiento por parte del diseño.

Esto es posible, gracias a que la directiva "dataflow" analiza el flujo de datos entre bucles y funciones para crear canales que permiten que estos puedan acceder a los datos y puedan comenzar a funcionar, sin la necesidad de esperar a que otros bucles o funciones se hayan completado.

7.6 HLS Math

El lenguaje C/C++ aporta funciones para realizar operaciones matemáticas mediante su librería 'math.h' cómo, raíces cuadradas, potencias, funciones trigonométricas, etc. Vitis HLS creo la librería 'hls_math.h', para dotar a las funciones matemáticas la capacidad de poder operar con tipos de precisión arbitraria. Esto, conlleva que su uso pueda ser beneficioso para la optimización en rendimiento y área, ya que estas librerías tienden a implementar menos hardware con los tipos de precisión arbitraria.

No obstante, esta librería no aporta la misma precisión que las librerías matemática que aporta C, ya que estas funciones se implementan como funciones de aproximación de bits.

7.7 Template metaprograming para configurar directivas

En el apartado 6.4, introducimos el template metaprograming para crear recursión, no obstante también podemos utilizar esta técnica a nuestro favor para configurar las directivas, ya que si no están definidas en tiempo de compilación, en el momento de la síntesis se producirá un error.

```
template <class T, int N, int M, int K, int D>
void mult_matrix (T matrix_a[N][M], const T matrix_b[M][K], T matrix_sol[N][K])
{
    #pragma HLS array_partition variable=matrix_a complete dim=D
    for (int n = 0; n < N; n++)
        for (int k = 0; k < K; k++)
        {
            matrix_sol[n][k] = 0;
            for (int m = 0; m < M; m++)
                matrix_sol[n][k] += matrix_a[n][m] * matrix_b[m][k];
        }
}
```

Listing 7.1: Ejemplo de configuración de directivas mediante TMP.

Como el "template meta-programing" se concibe en tiempo de compilación, y las directivas "pragma" al igual que el TMP, también se procesa en tiempo de compilación, se podrán utilizar estas constantes para configurar las configuraciones de las directivas pragma de Vitis HLS.

7.8 Conclusión

Todo diseño busca ser óptimo para adaptarse a la necesidad lo mejor posible al mínimo coste, aunque en los HLS al final produce una salida que cree conveniente dependiendo de la entrada recibida. No obstante, a partir del diseño tradicional, podemos modificar partes del diseño para obtener un tipo de optimización u otra. Mientras que, para conseguir la optimización en Vitis HLS, es necesario conocer las directivas y el comportamiento de estas. Ya que muchas de estas generan comportamientos que pueden provocar un resultado en los cuales no llega ser óptimo o ni reflejan una mejora.

Un ejemplo sería, el caso de realizar un "unroll" de un bucle y tengamos varios accesos a una matriz, la cual es tratada como un bloque RAM por defecto en Vitis HLS. Se podría llegar al caso, en que los canales de entrada y salida quedaran ocupado y entonces habría que esperar ciclos de reloj hasta que estos puertos estuvieran libres.

Por tanto, el entender el comportamiento de nuestro código con el uso de directiva es vital, para generar una optimización que no pueda ocasionar resultados que puedan aumentar la latencia, mermar el rendimiento o llegar a utilizar más componentes que la opción sin optimizar.

CAPÍTULO 8
Caso práctico - Red neuronal

Para el estudio de este trabajo de fin de grado, se han llevado a cabo la traducción de diversos algoritmos a un algoritmo sintetizable por Vitis HLS. Como última prueba se pensó crear una red neuronal simple sintetizable, ya que actualmente es uno de los algoritmos con mayor popularidad por su capacidad de aprender, lo que permite adaptarse perfectamente al cambio que la rodea.

8.1 Red neuronal

Una red neuronal, es un modelo computacional basado en conjuntos de unidades, las cuales llamamos neuronas artificiales. Estas se interconectan entre sí para la transmisión de señales y cada una de ellas vienen con una regla o condición diferente, denominada función de activación. Dicha condición deberá cumplirse antes de continuar la propagación al resto de neuronas.

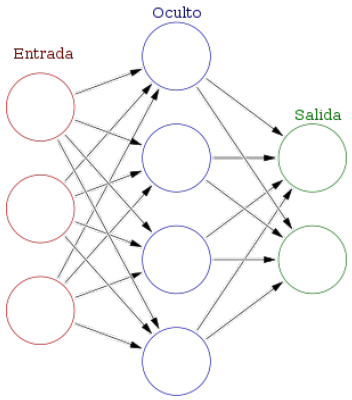


Figura 8.1: Estructura de una red neuronal simple.

Dentro de este modelo existen, diversas capas neuronales que lo conforman y podemos dividirlas en capa de entrada, capa de salida y capas ocultas. Estas ultimas, siempre están ocultas y por tanto no se ve el resultado de estas, mientras que la capa de salida y entrada sí.

Finalmente, para el entrenamiento de la red neuronal de este caso práctico, se ha utilizado el método de "backpropagation"[1]. Este método es el cálculo del error desde atrás, es decir, desde la salida hacia la entrada y a su vez actualizar los valores a las neuronas.

8.2 Estructura del caso práctico

En cuanto a su estructura se divide en dos partes, una dedicada al entrenamiento mediante códigos escritos en Python y la otra para realizar la inferencia mediante los códigos sintetizables escritos en C/C++ para poder transferir el proceso a una FPGA en un futuro.

La ejecución programada en Python, se encarga crear toda la estructura de la red neuronal, además de su entrenamiento y generación de los datos, que se pasarán a un fichero cabecera '.h' para luego ser utilizados por la síntesis para realizar el proceso de inferencia y la red neural construida en un fichero '.cpp'.

En segundo lugar, en la parte realizada en C/C++, será el código a sintetizar y encargado generar una respuesta.

8.3 Entrenamiento y extracción de los datos

Para la parte de entrenamiento, se han utilizado distintas clases creadas y una librería para la generación de matrices, llamada 'numpy'[21]. Por parte de las clases creadas, se dispone de una clase para la generación del modelo de la red y otra para entrenar el modelo, además de otra clase para extraer todos los datos del entrenamiento.

El funcionamiento de esta parte empieza en la generación del modelo de la red neuronal mediante una entrada, la cual es una matriz generada a partir de la librería 'numpy'. Una vez generado el modelo, este se da como entrada a la clase encargada del entrenamiento del modelo.

La clase 'trainer', se encargará de generar nuevos valores para las capas, mediate la función hiperbólica elegida, en este caso la tangente hiperbólica, y la última entrada de la capa anterior.

Seguidamente, una vez recorridas todas las capas mediante el "forward pass"[2], pasamos a la fase de "backpropagation", donde se calcula el error generado por la fase anterior recorriendo la red neuronal en orden inverso, es decir, de atrás hacia delante.

Este proceso se realizará en varias iteraciones, un valor configurable en el script encargado de ejecutar todo el proceso. Una vez entrenada la red neuronal, quedará la extracción del modelo, mediante un script encargado de interpretar los datos y generar un fichero de cabecera '.h' donde encontraremos los siguientes datos:

- Tamaño de la entrada y del resultado
- Tamaño de las matrices de datos
- Datos de las matrices

Y por otra parte, también se generará un fichero '.cpp' donde encontraremos la red neuronal construida a través de la información interpretada.

8.4 Generación de la red neuronal e inferencia

Para la parte de la inferencia, se utilizarán las dos partes generadas, en el proceso anterior, además de una parte dedicada a la operación de matrices. En cuanto a la parte de la red neural generada, encontramos la función a sintetizar, la cual tiene dos argumentos, uno para la entrada y el otro para la salida de función.

Luego dentro de esta función, encontramos las distintas capas existentes, las cuales dependerán del tamaño configurado en el proceso anterior.

```
1 double output_layer_0[input_n][layer_0_m];
2 mult_matrix<double, input_n, layer_0_n, layer_0_m, 2> (input, layer_0,
      output_layer_0);
3 hyperbolic_func_matrix<double, input_n, layer_0_m> (output_layer_0);
```

Listing 8.1: Ejemplo de capa de una red neuronal generada a partir del proceso anterior.

Como se ve, en cada capa se producen tres operaciones, una la creación de una matriz estática para la salida obtenida, después la multiplicación de la matriz y de la capa para generar una salida, la cual es transmitida a la matriz creada anteriormente. Y finalmente, la función hiperbólica tangencial, para la salida obtenida.

Estos bloques B.7, se generarán dependiendo del número de capas que contenga la red neuronal.

```
1 copy_arr<double, result_n, result_m> (output_layer_last_layer, output);
```

Listing 8.2: Valor a devolver mediante la función 'copy_arr'.

Finalmente, el resultado obtenido por la última capa se copia a la matriz de salida, para ser devuelto. Con esto, queda completado todo el proceso para el cálculo de una entrada.

8.4.1. Interfaces utilizadas

En cuanto a las interfaces utilizadas, se han utilizado dos interfaces "m_axi", ya que ambos argumentos de la función principal son dos matrices, por tanto imposibilita el uso de la interfaz AXI4-Lite.

En cuanto, a la configuración del argumento "input", se ha utilizado el "offset" en la opción "direct", especificada en 5.3.1, que permite la lectura y escritura en distintas ubicaciones, consiguiendo así disminuir el número de ciclos.

No obstante, para el argumento de salida de la función principal, se utilizado el "offset", con la opción "off", ya que se produce una copia de la matriz de la última capa, la cual contine la solución, a la matriz de salida. Esto se debe a que la opción "off" solamente configura una dirección base y es perfecta para las copias secuenciales ya que solamente hay que incrementar el valor de la dirección base para obtener el valor.

8.4.2. Optimización/Directivas

En este apartado vamos a detallar todo lo relacionado con la optimización del modelo, no obstante, antes de empezar hay que tener claras las propiedades que vamos a modificar para optimizar el proceso.

Anteriormente en el capítulo 7, vimos como podíamos dirigir nuestra optimización en cuatro formas diferentes, siendo estas: rendimiento, área, latencia y lógica. Por tanto, deberemos saber asociar estos al resumen del resultado final de la síntesis.

- **Optimización en área y lógica:** Para optimizar en estos apartados, nos centraremos en reducir el número de componentes, los cuales en el resumen son: LUTs, BRAMs o URAM, Biestables.
- **Optimización en rendimiento:** Para optimizar en este apartado, deberemos centrarnos subir la frecuencia máxima estimada en el resumen.

- **Optimización en latencia:** Para optimizar la latencia, en el resumen, nos aparecerán esta de dos formas distintas. La primera muestra los ciclos de reloj utilizados y otra con el tiempo final.

Toda la optimización añadida en la parte de la inferencia está en las operaciones para el cálculo de la solución. Estas operaciones son:

- Función para la multiplicación de matrices, la cual contiene directivas para su optimización.
- Función para aplicar la tangente hiperbólica a los valores de la matriz, la cual contiene una función de la librería 'hls_math'.

Por otra parte, las optimizaciones mediante directivas las vamos a destinar según la necesidad, es decir, un caso donde nos interesa la optimización en área y otro caso nos interese tener la menor latencia posible. En cuanto, al rendimiento, siempre intentaremos obtener el máximo posible, sin sacrificar el área o la latencia.

8.4.3. Versión que prioriza la latencia

Para mejorar la latencia se han probado varias directivas nombradas del capítulo 7, no obstante, la dos con un resultado notable, han sido las directivas 'PIPELINE' y 'DATAFLOW'.

En cuanto a la directiva 'DATAFLOW', esta ha obtenido buenos resultados a la hora de mejorar la latencia, no obstante 'PIPELINE' ha mostrado mejores resultados finalmente, además de ser más flexible con su uso.

```
template <class T, int N, int M, int K, int Dims>
void mult_matrix (T matrix_a[N][M], const T matrix_b[M][K], T matrix_sol[N][K])
{
   #pragma HLS PIPELINE II=1
   for (int n=0; n<N; n++)
        for (int k=0; k<K; k++)
        {
            matrix_sol[n][k] = 0;
            for (int m=0; m<M; m++)
                matrix_sol[n][k] += matrix_a[n][m] * matrix_b[m][k];
        }
}
```

Listing 8.3: Función para la multiplicación de matrices optimizada para obtener la mejor latencia.

Esta flexibilidad se debe a que dependiendo del bucle donde se ubique la directiva 'PIPELINE' realizará un desenroscado del bucle o no, en el caso de aplicar la directiva al bucle más externo, los bucles serán desenroscados haciendo que aumente el número de componentes a utilizar. No obstante, sí usamos la directiva en el bucle más interno, no desenroscara ningún bucle. Por tanto, esta directiva nos permite en esta función tener más o menos latencia a cambio de utilizar más o menos componentes.

Aunque, en el caso de 'DATAFLOW', no ocurre lo mismo y siempre se obtiene más o menos la misma latencia y numero de componentes. Pero si lo utilizamos juntamente con la directiva 'PIPELINE' en el bucle interior, se consigue disminuir la latencia y con un número menor de elementos utilizados. Este se debe a que la directiva 'DATAFLOW' activa la segmentación a nivel de tarea, lo que conlleva a incrementar la concurrencia de la implementación RTL y con ello un menor número de ciclos utilizados.

Finalmente, hemos aplicado la directiva 'PIPELINE' en el bucle exterior, ya que es la mejor opción encontrada para conseguir obtener la mayor latencia posible.

```
template <class T, int N, int M>
void hyperbolic_func_matrix (T matrix[N][M])
{
    #pragma HLS PIPELINE II=1
    for (int i=0; i<N; i++)
        for (int j=0; j<M; j++)
            matrix[i][j] = tanh(matrix[i][j]);
}
```

Listing 8.4: Función para el cálculo hiperbólica de una matriz optimizada para obtener la mejor latencia.

En el caso, de la función encargada de aplicar la función hiperbólica, hemos concluido que utilizar el 'PIPELINE' también, por los benéficos que aporta a la latencia.

8.4.4. Versión que prioriza el área

Para la optimización en área, se han probado las diversas directivas que se mencionan en el capítulo 7. Y en primer lugar se ha intentado disminuir el tamaño de los bloques de RAM mediante la directiva 'ARRAY_RESHAPE', aunque al aplicarla no se ha notado un cambio sustancial, y además aumentaba la latencia, ya que el propio diseño creado por el HLS ya optimizo los bloques RAM.

Por otra parte, también probamos a utilizar la directiva 'FUNCTION_INSTANTIATE' para ver si podíamos optimizar los argumentos de salida de las subfunciones, aunque el cambio también era mínimo.

Y por último, se utilizó la directiva 'INLINE' para ver como era de beneficioso en ambas funciones, siendo esta directiva la que mejor resultado dio, en cuanto la reducción de componentes utilizados. Ya que estas funciones se insertan dentro de la sección de código desde son llamadas, lo que conlleva eliminarse de la jerarquía de la RTL y a eliminar los componentes que estaban relacionados con la llamada a esta función.

```
template <class T, int N, int M, int K, int Dims>
void mult_matrix (T matrix_a[N][M], const T matrix_b[M][K], T matrix_sol[N][K])
{
    for (int n=0; n<N; n++)
        for (int k=0; k<K; k++)
        {
            matrix_sol[n][k] = 0;
            for (int m=0; m<M; m++)
                matrix_sol[n][k] += matrix_a[n][m] * matrix_b[m][k];
        }
}
```

Listing 8.5: Función para la multiplicación de matrices optimizada para obtener la mejor optimización en área.

En cuanto, a 8.5, ningunas de las directivas comentadas anteriormente, ha dado un resultado significativo en este bloque de código, así que hemos optado por dejar las optimizaciones que aplica Vitis HLS.

```
template <class T, int N, int M>
void hyperbolic_func_matrix (T matrix[N][M])
{
    #pragma HLS INLINE
    for (int i=0; i<N; i++)
        for (int j=0; j<M; j++)
```

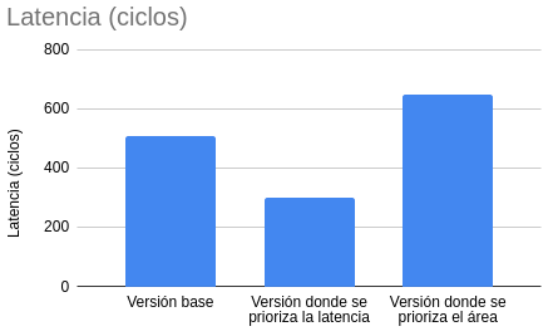
```
7             matrix[i][j] = tanh(matrix[i][j]);
8 }
```

Listing 8.6: Función para el cálculo hiperbólica de una matriz optimizada para obtener la mejor optimización en área.

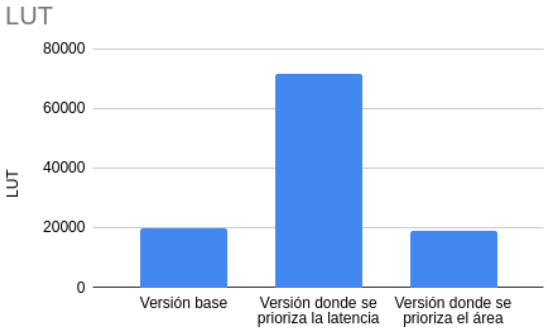
Finalmente, para la función hiperbólica, hemos optado por convertirla en una función insertada o 'inline'.

8.4.5. Conclusión

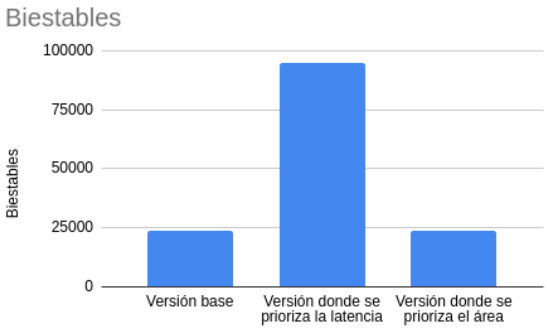
Para el este apartado, hemos realizado varias gráficas a partir de los datos obtenidos de los reportes de la síntesis. Donde seguidamente vamos a comparar su cambio con la versión base del modelo para obtener una conclusión de la herramienta Vitis HLS.



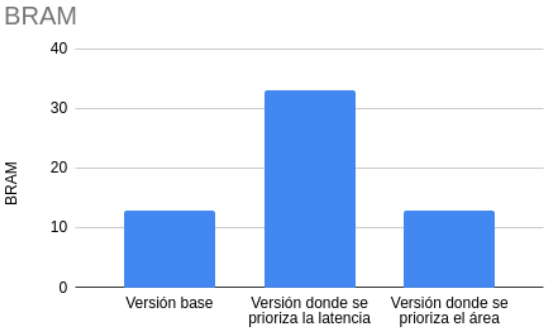
(a) Latencia



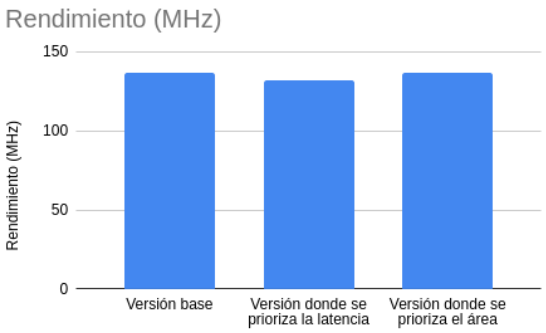
(b) Número de LUTs



(c) Número de Biestables



(d) Número de bloques RAM



(e) Rendimiento

Figura 8.2: Comparativa de las versiones de la red neuronal.

A partir de los gráficos, vemos que ambas versiones dedicadas al área y la latencia, obtienen el resultado deseado. No obstante, en la optimización dedicada a la latencia, se nota una diferencia considerable en cuanto, al número de componentes que utilizará. Por tanto, podríamos mejorar esta sin afectar mucho a la latencia mediante tipos de precisión arbitraria, ya que solamente utilizarían en numero de bits necesarios.

Por otra parte, en la versión dedicada a la optimización en área, obtiene resultados muy similares a la versión base, debido a que Vitis HLS ya aplica muchas optimizaciones para mejorar en este apartado de manera automática. Como por ejemplo, en cada argumento de alguna subfunción, donde se ha aplicado automáticamente la función 'array_partition' en la configuración complete o también el empaquetado de alguna subfunción en alguna iteración.

Otro aspecto a tener en cuenta es el parecido que va teniendo la herramienta HLS en los compiladores actuales de lenguajes de propósito general, en cuanto a optimización. Hoy en día estos compiladores suelen aplicar diversas técnicas para obtener un compilado lo más optimizado posible, y lo mismo ocurre con Vitis HLS, la cual contempla ya muchos aspectos para aplicar alguna optimización si la herramienta lo cree conveniente. Por tanto, visto en las gráficas, para casos donde el código que sintetizar sea una pequeña estructura, el diseño base que produce puede ser la mejor opción.

CAPÍTULO 9
Conclusiones

Para finalizar el trabajo, vamos a presentar algunas consideraciones acerca de este, como objetivos alcanzados y planteados. Además también hablaremos sobre cómo se podría extender para mejorar aún más a este.

9.1 Consideraciones finales

Finalmente, podemos concluir que todos los objetivos se han llevado a cabo y hemos llegado a descubrir los límites de la herramienta dentro de las restricciones establecidas.

Primeramente, hemos analizado todo su flujo de diseño en general de realizar un diseño mediante High Level Synthesis, además de ver todo la interconexión de las interfaces mediante la especificación AMBA4, la cual hemos detallado para poder finalmente entender los aspectos más específicos de la especificación implementada en Vitis HLS.

Seguidamente, hemos visto todos los aspectos para generar un código sintetizable y optimizado mediante las directivas que aporta Vitis HLS, además de poder concluir a estos dos capítulos con conclusiones sobre las experiencias obtenidas la síntesis de algoritmos dirigidos a la CPU y de la prueba hecha después de probar las directivas de optimización.

Por último, se ha realizado el caso práctico con toda la estructura propuesta en los objetivos.

9.2 Trabajo futuro

Aunque todos los objetivos se hayan cumplido y hemos llegado a analizar y entender la herramienta, aún se puede llegar más lejos con esta.

Para empezar, el template metaprogramming, es una técnica que puede dar muchas posibilidades dentro de la síntesis, ya que todo ocurren en tiempo de compilación. No obstante, se debería probar que limites tiene con el HLS, para así poder aprovechar al máximo su funcionamiento.

Otros aspectos que poder estudiar, sería recrear la recursión mediante lenguaje específico de dominio, el cual es una especificación dedicada a resolver problemas en particular, que juntamente con template y funciones lambda se podría crear un código sintetizable. Y finalmente estudiar hasta qué punto resulta conveniente.

Finalmente, poder llegar a crear la parte de entrenamiento en código sintetizable también sería un buen trabajo de futuro, para así poder comprobar hasta qué punto de mejora

obtenemos a diferencia de entrenar un modelo de red neuronal frente al modelo entrenado en el caso práctico.

Bibliografía

[1] Algoritmo backpropagation. Consultado en https://empresas.blogthinkbig.com/como-funciona-el-algoritmo-backpropagation-en-una-red-neuronal/

[2] Algoritmo fordward pass. Consultado en https://towardsdatascience.com/forward-propagation-in-neural-networks-simplified-math-and-code-version-bbcfef6f9250

[3] Biblioteca estándar o "std". Consultado en https://en.cppreference.com/w/cpp/header

[4] Cofundador de Xillinx, Ross Freeman. Consultado en https://en.wikipedia.org/wiki/Ross_Freeman

[5] Cofundador de Xillinx, Bernard Valentine Vonderschmitt. Consultado en https://en.wikipedia.org/wiki/Bernard_V._Vonderschmitt

[6] Creador de Verilog, Philip Moorby. Consultado en https://computerhistory.org/profile/philip-moorby/

[7] Complex Programmable Logic Device Consultado en https://es.wikipedia.org/wiki/CPLD.

[8] Configurable logic block Consultado en https://www.ni.com/documentation/en/labview-comms/latest/fpga-targets/configurable-logic-blocks/.

[9] Distribución de las FPGA en la industria Consultado en https://www.generatecnologias.es/aplicaciones_fpga.html.

[10] Documentación AMBA4. Consultado en https://developer.arm.com/architectures/system-architectures/amba/amba-4.

[11] Field-Programmable Gate Array Consultado en https://es.wikipedia.org/wiki/Field-programmable_gate_array.

[12] High Level Synthesis. Consultado en https://en.wikipedia.org/wiki/High-level_synthesis

[13] High Level Synthesis creado por Xillinx. Consultado en https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf.

[14] Institute of Electrical and Electronics Engineers. Consultado en https://www.ieee.org/.

[15] Instituto Nacional Estadounidense de Estándares Consultado en https://ansi.org/.

[16] Intellectual Property Consultado en https://www.xilinx.com/products/intellectual-property.html.

[17] Input Output Block Consultado en https://www.fpgakey.com/wiki/details/50.

[18] Interfaz AXI Consultado en https://www.xilinx.com/products/intellectual-property/axi.html

[19] Lenguaje de descripción de hardware, Verilog. Consultado en https://es.wikipedia.org/wiki/Verilog

[20] Lenguaje de descripción de hardware, VHDL. Consultado en https://es.wikipedia.org/wiki/VHDL

[21] Librería numpy Consultado en https://numpy.org/

[22] Lenguaje C Consultado en https://es.wikipedia.org/wiki/C_(lenguaje_de_programaci%C3%B3n).

[23] Lenguaje C++ Consultado en https://es.wikipedia.org/wiki/C%2B%2B.

[24] Lenguaje Python Consultado en https://es.python.org/

[25] Mecanismo de apretón de manos o "handshake" Consultado en https://es.wikipedia.org/wiki/Establecimiento_de_comunicaci%C3%B3n.

[26] Naciones unidas. Consultado en https://www.un.org/es/

[27] Objetivos de Desarrollo Sostenible. Consultado en https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/

[28] Objetivos de Desarrollo del Milenio. Consultado en https://www1.undp.org/content/undp/es/home/sdgoverview/mdg_goals.html

[29] Protocolo ACE en AMBA. Consultado en https://developer.arm.com/documentation/ihi0022/e/ACE-Protocol-Specification/About-ACE/Protocol-overview/About-the-ACE-protocol

[30] System on Chip. Consultado en https://es.wikipedia.org/wiki/System_on_a_chip

[31] Template metaprograming y reglas de uso Consultado en https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-template-metaprogramming

[32] Unidad Central de Procesamiento Consultado en https://concepto.de/cpu/

[33] Unidad Gráfica de Procesamiento Consultado en https://hardzone.es/reportajes/que-es/gpu-caracteristicas-especificaciones/.

[34] Xillinx Inc. Consultado en https://www.xilinx.com/

APÉNDICE A
Objetivos de Desarrollo Sostenible

En cuanto, a los Objetivos de Desarrollo Sostenible [27] son una iniciativa impulsada por las Naciones Unidas [26] para dar continuidad tras los Objetivos de Desarrollo del Milenio [28] o ODM. El ODS esta compuesto por 17 objetivos establecidos, no obstante nosotros nos centraremos en los siguientes:

- **Objetivo 9:** Agua, industria, innovación e infraestructura.
  Actualmente crece el uso de tecnologías para realizar procesos en diversos campos de la industria. Por tanto, entender el uso del High Level Synthesis, puede ayudar a tratar la problemática con mayor facilidad, esto puede dar más uso a esta tecnología y así mejorar la resolución de problema en esta.
- **Objetivo 11:** Ciudades y comunidades sostenibles.
  A día de hoy, encontramos cada vez más tecnologías al nuestro alrededor, incluso alguna de ellas llega a interactuar en nuestras vidas, como podría ser un coche de conducción automática. Entender la configuración de las FPGA, puede llevar a usar más a esta en nuestros ámbitos y así poder resolver algunas problemáticas de nuestros entornos.
- **Objetivo 13:** Acción por el clima.
  La herramienta HLS aporta distintas formas de optimización, lo que permite reducir en varios aspectos, como podría ser la optimización de área o en rendimiento, lo que nos permitiría crear diseños más sostenibles, ya que haríamos un gasto menor en componentes y con ello se reducirá el consumo energético.

APÉNDICE B
Código

B.1 Generador y entrenamiento del modelo

```
import numpy as npy

class neural_layer:

    def __init__(self, current_layer, next_layer):
        self.weight = npy.random.rand(current_layer, next_layer) * 2 - 1


class neural_network():

    def __init__(self, layers, seed=1):
        self._neural_network = []
        self.layers = layers
        self.seed = seed
        self._init_neural_network()


    def get_neural_network(self):
        return self._neural_network


    def set_neural_network(self, new_neural_network):
        self._neural_network = new_neural_network


    def _init_neural_network(self):
        npy.random.seed(self.seed)
        for i in range(len(self.layers[:-1])):
            self._neural_network.append(neural_layer(self.layers[i], self.
                layers[i+1]))
```

Listing B.1: Clases para la generación del modelo.

```
import numpy as npy


def tanh(x):
    return npy.tanh(x)


def derived_tanh(x):
    return 1.0-x**2


```

```
def train(neural_network, input, output, ratio=0.1):
    delta = []
    trained_output = [input]
    last_weight = None

    #Forward pass
    for i, layer in enumerate(neural_network):
        trained_output.append(tanh(npy.dot(trained_output[-1], layer.weight)))

    #Back propagation
    for i in range(0, len(neural_network))[::-1]:
        current = trained_output[i+1]
        if i == len(neural_network)-1:
            delta.insert(0, (current - output) * derived_tanh(trained_output
                [-1]))
        else:
            delta.insert(0, npy.dot(delta[0], last_weight.T) * derived_tanh(
                current))

        #Gradient descent
        neural_network[i].weight = neural_network[i].weight - (npy.dot(
            trained_output[i].T, delta[0]) * ratio)
        last_weight = neural_network[i].weight

    return neural_network


def get_solution(neural_network, input):
    output = [input]
    for layer in neural_network:
        output.append(tanh(npy.dot(output[-1], layer.weight)))
    return output[-1]
```

Listing B.2: Script encargado del entrenamiento del modelo.

```
import numpy as npy


def _weight_to_string(weight):
    str_weight = "{"
    for n in range(0, len(weight)):
        str_weight += "{"
        for m in range(0, len(weight[n])):
            if m != len(weight[n]) - 1:
                str_weight += " %s," %(weight[n][m])
            else:
                str_weight += " %s" %(weight[n][m])

        if n != len(weight) - 1:
            str_weight += "},\n"
        else:
            str_weight += "}"

    str_weight += "};"
    return str_weight


def _generate_layer_variables(id_layer, weight):
    # Array dimensions
    n, m = len(weight), len(weight[0])
    content = "#define layer_%s_n %s\n" %(id_layer, n)
    content += "#define layer_%s_m %s\n" %(id_layer, m)
    # Array content
    string_weight = _weight_to_string(weight)
```

```
    content += "const double layer_%s[layer_%s_n][layer_%s_m] =\n%s\n\n" %(
        id_layer,
        id_layer,
        id_layer,
        string_weight,
    )
    return content


def _generate_layer(id_layer):
    str_layer = ""
    if id_layer == 0:
        str_layer = f"""
    //LAYER {id_layer}
    double output_layer_{id_layer}[input_n][layer_{id_layer}_m];
    mult_matrix<double, input_n, layer_{id_layer}_n, layer_{id_layer}_m, 2> (input, layer_{id_layer}, output_layer_{id_layer});
    hyperbolic_func_matrix<double, input_n, layer_{id_layer}_m> (output_layer_{id_layer});
        """
    else:
        str_layer = f"""
    //LAYER {id_layer}
    double output_layer_{id_layer}[input_n][layer_{id_layer}_m];
    mult_matrix<double, input_n, layer_{id_layer}_n, layer_{id_layer}_m, 2> (output_layer_{id_layer-1}, layer_{id_layer}, output_layer_{id_layer});
    hyperbolic_func_matrix<double, input_n, layer_{id_layer}_m> (output_layer_{id_layer});
        """
    return str_layer


def get_string_file(neural_network, input_n, result_m):
    str_layer_weights = "#ifndef LAYERS_H\n#define LAYERS_H\n"
    str_layer_weights += "\n#define input_n %s\n" %(input_n)
    str_layer_weights += "#define input_m %s" %(
        str(len(neural_network[0].weight)) + "\n" * 2
    )
    str_layer_weights += "#define result_n %s\n" %(input_n)
    str_layer_weights += "#define result_m %s" %(str(result_m) + "\n" * 3)

    str_nn = (
        '#include "neural-network.h"'
        + ' \n\n\nvoid get_solution(double input[input_n][input_m],'
        + ' double output[result_n][result_m]) \n{'
    )

    for i, layer in enumerate(neural_network):
        str_layer_weights += _generate_layer_variables(i, layer.weight)
        str_nn += _generate_layer(i)

    str_nn += "\n\tcopy_arr<double, result_n, result_m> (output_layer_%, output); \n}" %(len(neural_network) - 1)
    str_layer_weights += "#endif"
    return str_layer_weights, str_nn
```

Listing B.3: Script encargado de la generación del modelo en un código sintetizable y los datos relacionados con las capas.

B.2 Codigos realcionado con la inferencia

```
#ifndef LAYERS_H
```

```
#define LAYERS_H

#define input_n 1
#define input_m 3

#define result_n 1
#define result_m 1


#define layer_0_n 3
#define layer_0_m 5
const double layer_0[layer_0_n][layer_0_m] =
{{ -0.2613329510861131, 1.856077937631827, -1.260519685404232,
    -0.7953275588594104, -0.8826187401942487},
{ -1.0049621922153, 0.07484860413395591, -0.0681367647971603,
    1.4964464281138201, 0.2551633918357144},
{ -0.21909198063957608, -0.6708090255169676, -1.397274636284205,
    0.3607515010494793, -1.132090958873656}};

#define layer_1_n 5
#define layer_1_m 4
const double layer_1[layer_1_n][layer_1_m] =
{{ 0.66705128169126, -0.7324143689658317, -0.3505506539077308,
    -0.12476803829039575},
{ -0.8231233010757667, 0.791227710816027, 1.4003849014701852,
    -0.7412736749942941},
{ 0.6886223034359052, 0.6135615519650327, 1.1918269078129113,
    -0.7709370181772116},
{ -0.6868252061464787, -0.16447886788931096, 1.071175996766034,
    -1.002780178466279},
{ 0.011228557559429241, 0.8617335645432478, 0.44354195997778256,
    0.3424271092565365}};

#define layer_2_n 4
#define layer_2_m 1
const double layer_2[layer_2_n][layer_2_m] =
{{ -1.3686723105048437},
{ 0.33177768793488227},
{ 1.4148874048353874},
{ -0.7837359810574125}};

#endif
```

Listing B.4: Ejemplo del fichero de cabecera con los datos de las capas neuronales.

```
#ifndef NEURAL_NETWORK_H
#define NEURAL_NETWORK_H

#include "generated-layers.h"
#include "utils/matrix-operations.h"
#include <iostream>

void get_solution(double input[input_n][input_m], double output[result_n][
    result_m]);

#endif
```

Listing B.5: Fichero cabecera del modelo sitetizable generado.

```
#include "neural-network.h"


void get_solution(double input[input_n][input_m], double output[result_n][
    result_m])
```

```
{
    //LAYER 0
    double output_layer_0[input_n][layer_0_m];
    mult_matrix<double, input_n, layer_0_n, layer_0_m, 2> (input, layer_0,
        output_layer_0);
    hyperbolic_func_matrix<double, input_n, layer_0_m> (output_layer_0);

    //LAYER 1
    double output_layer_1[input_n][layer_1_m];
    mult_matrix<double, input_n, layer_1_n, layer_1_m, 2> (output_layer_0,
        layer_1, output_layer_1);
    hyperbolic_func_matrix<double, input_n, layer_1_m> (output_layer_1);

    //LAYER 2
    double output_layer_2[input_n][layer_2_m];
    mult_matrix<double, input_n, layer_2_n, layer_2_m, 2> (output_layer_1,
        layer_2, output_layer_2);
    hyperbolic_func_matrix<double, input_n, layer_2_m> (output_layer_2);

  copy_arr<double, result_n, result_m> (output_layer_2, output);
}
```

Listing B.6: Ejemplo del modelo sintetizable generado.

```
#ifndef MATRIX_OPERATIONS_H
#define MATRIX_OPERATIONS_H

#include "hls_math.h"
#include <iostream>

template <class T, int N, int M>
void hyperbolic_func_matrix (T matrix[N][M])
{
    for (int i=0; i<N; i++)
        for (int j=0; j<M; j++)
            matrix[i][j] = tanh(matrix[i][j]);
}


template<class T, int N, int M>
void copy_arr(T matrix[N][M], T new_matrix[N][M])
{
    for (int i=0; i<N; i++)
        for (int j=0; j<M; j++)
            new_matrix[i][j] = matrix[i][j];
}

template <class T, int N, int M, int K>
void mult_matrix (T matrix_a[N][M], const T matrix_b[M][K], T matrix_sol[N][K])
{
    for (int n=0; n<N; n++)
        for (int k=0; k<K; k++)
        {
            matrix_sol[n][k] = 0;
            for (int m=0; m<M; m++)
                matrix_sol[n][k] += matrix_a[n][m] * matrix_b[m][k];
        }
}

#endif
```

Listing B.7: Fichero cabecera con las operaciones necesaria para realizar el proceso de inferencia.