UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Implementación de búsquedas regulares en una plataforma para el procesamiento de paquetes de red en FPGA Xilinx Zynq

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Mención: Sistemas Electrónicos

Autor: D. Yúbal Barrios Alfaro

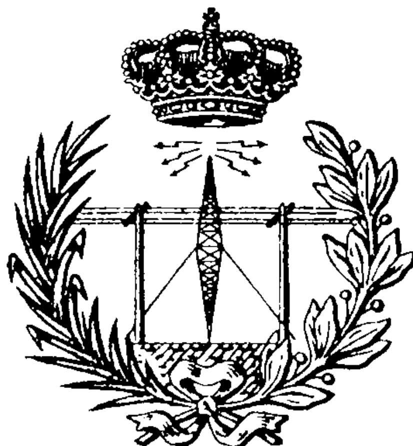
Tutores: D. Pedro Pérez Carballo

D. Adrián Domínguez Hernández

Fecha: junio de 2016

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

Implementación de búsquedas regulares en una plataforma para el procesamiento de paquetes de red en FPGA Xilinx Zynq

HOJA DE FIRMAS

Alumno/a

Fdo.: Yúbal Barrios Alfaro

Tutor/a

Fdo.: Pedro Pérez Carballo

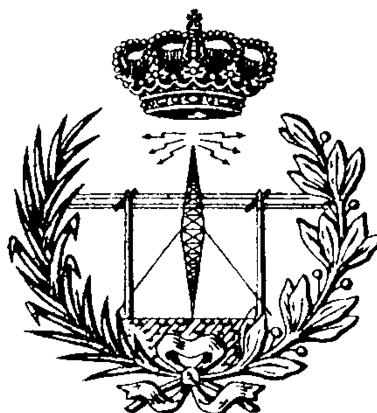
Tutor/a

Fdo.: Adrián Domínguez Hernández

Fecha: junio de 2016

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

Implementación de búsquedas regulares en una plataforma para el procesamiento de paquetes de red en FPGA Xilinx Zynq

HOJA DE EVALUACIÓN

Calificación: ______________________________

Presidente

Fdo.:

Vocal	**Secretario/a**
Fdo.:	Fdo.:

Fecha: junio de 2016

AGRADECIMIENTOS

En primer lugar, me gustaría hacer una mención especial a mis tutores, Pedro Pérez Carballo y Adrián Domínguez Hernández, quienes con su dedicación, tiempo y esfuerzo han contribuido de forma determinante a que este trabajo se realizase satisfactoriamente.

Seguidamente, me gustaría agradecer a los encargados de soporte del Instituto Electrónico de Microelectrónica Aplicada, la ayuda proporcionada para solventar cualquier tipo de problema surgido con cualquiera de las herramientas *software* empleadas en el transcurrir de este proyecto, así como a mis compañeros de grupo de trabajo por la colaboración establecida entre todos para lograr llevar a cabo todos nuestros trabajos de forma exitosa, contribuyendo a enriquecer esta experiencia única.

Finalmente, no puedo acabar estos párrafos sin resaltar el apoyo más importante que he tenido durante estos años, mi familia, a quienes quiero agradecer el esfuerzo que han realizado para que yo finalice mis estudios, así como la paciencia que han tenido conmigo cuando más cuesta arriba se han puesto las circunstancias.

RESUMEN

En este trabajo se realiza el diseño e implementación de un *firewall* de última generación compuesto por un detector de direcciones IP origen, el cual está basado en filtros Bloom con contador sobre una plataforma configurable Xilinx Zynq, con la finalidad de detectar y bloquear aquellas conexiones que se consideren una amenaza para el correcto funcionamiento del equipo empleado por el usuario.

De forma inicial, se hace referencia al catálogo de algoritmos disponible para cumplir con esta aplicación y se justifica la selección de los filtros Bloom con contador como la solución más óptima. A continuación, se describen las herramientas tanto *hardware* como *software* que se utilizaron durante el transcurso de dicho trabajo. Una vez analizadas, se pasa a describir con detalle las fases de diseño e implementación del proyecto, tanto del bloque IP creado siguiendo una metodología de síntesis de alto nivel como su posterior integración en la plataforma que conforma el sistema completo.

Finalmente, se realiza un banco de pruebas con el objetivo de validar el correcto funcionamiento del sistema en términos de tiempo de ejecución y consumo de recursos, calculando la latencia del filtro desde que recibe un paquete de datos hasta que devuelve una respuesta, tanto en *hardware* como en *software*, y calculando el factor de utilización de las *slices*, respectivamente. En función de los resultados obtenidos, se alcanza la conclusión de que el empleo de filtros Bloom constituye una buena solución para la implementación de un *firewall hardware* para velocidades de Gigabit.

ABSTRACT

This paper describes the design and implementation of a next-generation firewall, which consists in a source IP address detector based on Counting Bloom Filters (CBF) on a Xilinx Zynq configurable platform, in order to detect and block connections that are considered a threat to the user.

Initially, we make reference to the algorithms catalogue available and we justify the selection of the CBF algorithm as the best solution. Next, the hardware and software tools used during this work are described. Once they are analysed, we describe the design and implementation phases of the project, both the IP block created following a High-Level Synthesis Methodology and its subsequent integration into the platform that forms the whole system.

Finally, a test is performed in order to validate the correct operation of the system in terms of execution time and resource consumption, calculating the latency of our filter when it receives an IP packet until it returns a response in both, hardware and software implementations, and calculating the slices utilization factor. According to the results, we conclude that the use of Bloom filters is a good solution for implementing a hardware firewall in Gigabit environments.

TABLA DE CONTENIDO

TABLA DE CONTENIDO **13**

ÍNDICE DE FIGURAS **19**

ÍNDICE DE TABLAS **23**

ACRÓNIMOS **25**

Capítulo 1. Introducción **29**

1.1. Antecedentes 29

1.2. Objetivos 34

1.3. Peticionario 34

1.4. Estructura del documento 35

Capítulo 2. Algoritmos de búsqueda de expresiones regulares **37**

2.1. Introducción 37

2.2. Algoritmos de búsqueda de cadenas fijas 37

2.2.1. Boyer-Moore (BM). 38

2.2.2. Knuth-Morris-Pratt (KMP). 39

2.2.3. Karp-Rabin. 40

2.2.4. Aho-Corasick. 41

2.3. Algoritmos de búsqueda de patrones regulares 42

2.3.1. Autómatas finitos. 43

2.3.2. Filtros Bloom. 44

2.4. Conclusiones 47

Capítulo 3. Estudio de la plataforma Xilinx Zynq **49**

3.1. Introducción 49

3.2. Dispositivo XC7Z020 51

3.3. Bloque PS 53

3.4. Mecanismos de comunicación PS-PL 58

3.5. Placa de protipado ZedBoard 59

3.6. Conclusiones 61

Capítulo 4. Entorno de diseño Xilinx Vivado 63

4.1. Flujo de diseño e interfaz de usuario 63

4.1.1. Flujo de diseño de Xilinx Vivado 63

4.1.2. Interfaz de usuario 65

4.1.3. Diseño basado en plataformas y bloques IPs 68

4.1.4. Uso de restricciones 70

4.2. HLS y herramientas para síntesis de alto nivel 72

4.2.1. Flujo de diseño HLS 72

4.2.2. Vivado HLS 73

4.3. Herramientas de síntesis lógica e implementación 75

4.3.1. Síntesis lógica e implementación en Xilinx Vivado 75

4.3.2. Estrategias de optimización 77

4.3.3. Monitorización y visualización de recursos 78

4.4. Prototipado hardware y desarrollo software 79

4.4.1. Opciones de depurado hardware 79

4.4.2. Generación del bitstream 81

4.4.3. Xilinx SDK 82

4.5. Conclusiones 83

Capítulo 5. Descripción de la solución propuesta 85

5.1. Descripción del diseño 85

5.2. Arquitectura y estructura sobre la plataforma configurable 86

5.3. Arquitectura del software empotrado 88
5.4. Diseño de la plataforma 90
5.5. Conclusiones 90
Capítulo 6. Capítulo 6. Diseño del bloque IP 91
6.1. Introducción 91
6.2. Diseño hardware del filtro 91
6.2.1. Implementación 92
6.2.2. Test, depurado y verificación 95
6.3. Modelado de interfaces orientado a hardware 97
6.4. Síntesis, análisis de resultados y optimización 100
6.4.1. Análisis del diseño sobre Zynq sin directivas de optimización 101
6.4.2. Análisis del diseño sobre Zynq aplicando *pipeline* 102
6.4.3. Análisis del diseño sobre Virtex-7 104
6.5. Verificación RTL y exportación del IP 105
6.6. Conclusiones 109
Capítulo 7. Integración en la plataforma 111
7.1. Introducción 111
7.2. Primera iteración: diseño de plataforma básica 111
7.2.1. Arquitectura de la plataforma 111
7.2.2. Bloques IP 112
7.2.3. Métodos de comunicación. Interfaces y buses 116
7.2.4. Diseño del software empotrado 118
7.3. Segunda iteración: plataforma final 123

7.3.1. Modificación de la plataforma hardware 123
7.3.2. Síntesis e implementación de la plataforma 128
7.3.3. Modificación del software empotrado 132
7.4. Conclusiones 138
Capítulo 8. Fase de validación 141
8.1. Prestaciones del diseño a obtener 141
8.2. Configuración del banco de pruebas 142
8.2.1. Diagrama de conexión 142
8.2.2. Metodología de verificación 143
8.3. Descripción de los equipos empleados 143
8.4. Resultados obtenidos 145
8.5. Conclusiones 149
Capítulo 9. Conclusiones y trabajos futuros 151
9.1. Conclusiones del proyecto 151
9.2. Trabajos futuros 152
Referencias 155
Presupuesto 161
1. Recursos Hardware 161
2. Recursos Software 162
3. Recursos Humanos 162
4. Material fungible 163
5. Coste de edición del proyecto 163
6. Coste total del proyecto 163
Pliego de condiciones 165
1. Recursos hardware 165

2. Recursos software 165

3. Recursos de edición del proyecto 165

ÍNDICE DE FIGURAS

Figura 1. Dispositivos conectados a Internet por tipos. 29

Figura 2. Incremento de ataques malintencionados a través de la red 31

Figura 3. Ámbito de actuación de un NGFW [10] 32

Figura 4. Relación flexibilidad-eficiencia entre distintas soluciones electrónicas 33

Figura 5. Ejemplo de funcionamiento del algoritmo Boyer-Moore 38

Figura 6. Ejemplo de funcionamiento del algoritmo KMP 40

Figura 7. Ejemplo de funcionamiento del algoritmo Karp-Rabin para un índice hash de 97 41

Figura 8. Ejemplo de algoritmo Aho-Corasick. (a) Entrada al algoritmo. (b) Función de fallo. (c) Función de salida [17]. 42

Figura 9. Ejemplo de DFA para patrones HIS y HHK (Adaptada de [12]) 44

Figura 10. Ejemplo de filtro Bloom con m=16 bits y dos elementos dentro del vector 46

Figura 11. Ejemplo de CBF con dos elementos 47

Figura 12. Diagrama de la familia Zynq-7000 [25] 50

Figura 13. Diagrama de bloques a alto nivel [24] 50

Figura 14. Slice de un CLB de la familia Artix-7 de Xilinx [26] 52

Figura 15. Diagrama de bloques de un SoC de la serie Xilinx Zynq-7000 [24] 54

Figura 16. Diagrama de bloques de la distribución de reloj en el PS [28] 56

Figura 17. Arquitectura de la distribución de relojes en la familia Xilinx Zynq-7000 [28] 57

Figura 18. Interfaz AXI para comunicaciones entre el PL y las memorias RAM del PS [24] 59

Figura 19. Diagrama de bloques de la ZedBoard [30] 60

Figura 20. Localización de los principales componentes de la placa ZedBoard [30] 61

Figura 21. Flujo de diseño partiendo de una especificación de alto nivel en Xilinx Vivado 64

Figura 22. Ventana de trabajo de Xilinx Vivado IDE [32] 66

Figura 23. Flujo de diseño IP 69

Figura 24. Ejemplo de sistema basado en Zynq en IP Integrator 70

Figura 25. HLS en el flujo de diseño de la FPGA 73

Figura 26. Flujo de diseño de Vivado HLS 74

Figura 27. Entorno de diseño Vivado HLS 75

Figura 28. Resultados de implementación en Vivado 77

Figura 29. Layout de consumo de recursos de la FPGA 79

Figura 30. Ventana de trabajo del Hardware Manager 81

Figura 31. Opción de generar el bitstream en Xilinx IDE 81

Figura 32. Exportación del hardware y lanzamiento posterior de Xilinx SDK 82

Figura 33. Localización de la dirección IP origen en un frame Ethernet 86

Figura 34. Arquitectura del sistema empotrado 87

Figura 35. Flujo de diseño del programa 89

Figura 36. Diagrama de E/S del bloque IP 97

Figura 37. Resultados temporales de la síntesis de alto nivel del diseño 101

Figura 38. Consumo de recursos lógicos del diseño IP realizado 101

Figura 39. Planificación de las operaciones para la inserción de direcciones 102

Figura 40. Planificación de las operaciones para la consulta sobre un payload 102

Figura 41. Resultados temporales aplicando la directiva de pipeline 103

Figura 42. Consumo de recursos aplicando la directiva de pipeline 103

Figura 43. Resultados temporales sobre Virtex-7 104

Figura 44. Consumo de recursos en Virtex-7 105

Figura 45. Resultados de la cosimulación 105

Figura 46. Cuadro de diálogo para la exportación del IP 106

Figura 47. Consumo de recursos calculados durante la evaluación RTL 107

Figura 48. Informe temporal del diseño tras evaluar la descripción RTL 107

Figura 49. Consumo de recursos para la evaluación del RTL con 4.5 ns 108

Figura 50. Información temporal para la evaluación del RTL con 4.5 ns 108

Figura 51. Diagrama de bloques de la arquitectura de la plataforma básica de prueba 112

Figura 52. Diagrama de bloques de la plataforma 114

Figura 53. Bloque cbf_top_0 desarrollado mediante metodología HLS 116

Figura 54. Diagrama de bloques del AXI DMA 117

Figura 55. Plataforma completa incluyendo ILAs de depuración 125

Figura 56. Slack para una frecuencia de reloj para el PL de 150 MHz 129

Figura 57. Slack para una frecuencia de reloj para el PL de 180 MHz 130

Figura 58. Recursos utilizados por la plataforma 131

Figura 59. Layout de la plataforma 131

Figura 60. Consumo de potencia de la plataforma 132

Figura 61. Esquema de bifurcación de paquetes Ethernet sin análisis en el PS............................ 133

Figura 62. Interacción entre el hardware y el software del sistema 135

Figura 63. Consulta con respuesta negativa.. 137

Figura 64. Consulta con respuesta positiva... 138

Figura 65. Esquema de conexión para validación del sistema empotrado 143

Figura 66. Instante del proceso de verificación del software con ILA en Vivado.......................... 144

Figura 67. Cálculo de la latencia en hardware 146

Figura 68. Verificación del funcionamiento de la interfaz AXI4-Lite 147

ÍNDICE DE TABLAS

Tabla 1. Características relevantes de la familia de FPGAs Artix-7 51

Tabla 2. Estrategias de optimización proporcionadas por Xilinx 78

Tabla 3. Listado de señales del protocolo AXI-Stream.. 99

Tabla 4. Consumo de recursos del bloque IP y la plataforma.. 130

Tabla 5. Costes de recursos hardware 161

Tabla 6. Coste de recursos software.. 162

Tabla 7. Coste de recursos humanos 162

Tabla 8. Coste total del proyecto 164

ACRÓNIMOS

ACP *Accelerator Coherency Port*

AHB *Advanced High-performance Bus*

AMBA *Advanced Microcontroller Bus Architecture*

APB *Advanced Peripheral Bus*

API *Application Programming Interface*

APU *Application Processor Unit*

ARM *Advanced RISC Machine*

ARP *Address Resolution Protocol*

ASCII *American Standard Code for Information Interchange*

AXI *Advanced Extensible Interface*

BSP *Board Support Package*

CAN *Controller Area Network*

CBF *Counting Bloom Filter*

CLB *Configurable Logic Block*

CPU *Central Processing Unit*

DDR *Double Data Rate*

DFA *Deterministic Finite Automata*

DHCP *Dynamic Host Configuration Protocol*

DMA *Direct Memory Access*

DNS *Domain Name System*

DPI *Deep Packet Inspection*

DSP *Digital Signal Processor*

EITE *Escuela de Ingeniería de Telecomunicaciones y Electrónica*

EMIO *Extendable Multiplexed I/O*

FF *Flip-Flop*

FIFO *First In-First Out*

ACRÓNIMOS

FMC *FPGA Mezzanine Card*

FPGA *Field Programmable Gate Array*

FSBL *First Stage BootLoader*

GCC *GNU Compiler Collection*

GDB *GNU Debugger*

GIC *Generic Interrupt Controller*

GPIO *General Purpose Input/Output*

GUI *Graphical User Interface*

HDMI *High-Definition Multimedia Interface*

HDL *Hardware Description Language*

HLS *High-Level Synthesis*

I2C *Inter-Integrated Circuit*

ICMP *Internet Control Message Protocol*

IDE *Integrated Development Environment*

ILA *Integrated Logic Analyzer*

I/O *Input/Output*

IOP *Input-Output Peripheral*

IoT *Internet of Things*

IP *Intellectual Property o Internet Protocol*

IPS *Intrusion Prevention System*

IUMA *Instituto Universitario de Microelectrónica Aplicada*

JTAG *Join Test Action Group*

LVCMOS *Low-Voltage Complementary Metal Oxide Semiconductor*

LVDS *Low-Voltage Differential Signalling*

LUT *Look-Up Table*

LWIP *Light-Weight IP*

MAC *Media Access Control*

MMCM *Mixed-Mode Clock Manager*

NFA *Non-deterministic Finite Automata*

NGFW *Next-Generation Firewall*

NIDS *Network Inspection Detection System*

OCM *On-Chip Memory*

OLED *Organic Light-Emitting Diode*

PL *Programmable Logic*

PLL *Phase-Locked Loop*

PS *Processing System*

QoS *Quality of Service*

RAM *Random Access Memory*

RGMII *Reduced Gigabit Media-Independent Interface*

RTL *Register-Transfer Level*

SD *Secure Digital*

SDRAM *Synchronous Dynamic RAM*

SDK *Software Development Kit*

SICAD *Sistemas Industriales y CAD*

SLL *Super Long Line*

SLR *Super Logic Region*

SoC *System-on-Chip*

SPI *Serial Peripheral Interface*

SSL *Secure Sockets Layer*

SSH *Secure Shell*

TCL *Tool Command Language*

TCP/IP *Transmission Control Protocol/Internet Protocol*

TEMAC *Tri-Mode Ethernet MAC*

TLM *Transaction-Level Modelling*

UART *Universal Asynchronous Receiver-Transmitter*

UDP *User Datagram Protocol*

URL *Uniform Resource Locator*

USB *Universal Serial Bus*

VGA *Video Graphics Array*

VHDL *Very High speed integrated circuit hardware Description Language*

Capítulo 1. Introducción

En este capítulo inicial se realizará una breve exposición de las razones y necesidades que surgen para el desarrollo de este proyecto, especificando posteriormente cómo se estructura y los objetivos básicos a llevar a cabo.

1.1. Antecedentes

En la actualidad, la seguridad y la privacidad al estar conectado a Internet se han convertido en una prioridad para numerosas áreas de investigación relacionadas con la ciencia y la tecnología, debido al considerable incremento del uso de dispositivos electrónicos en nuestro día a día y, por consiguiente, a la cantidad de información personal que se almacena y expone en la red (Figura 1) [1].

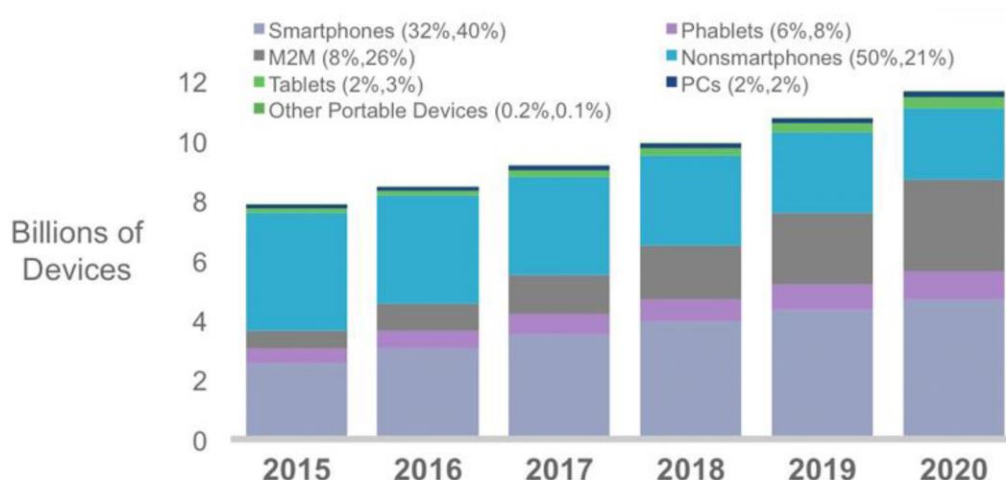


Figura 1. Dispositivos conectados a Internet por tipos.

Debido a este aumento de la tecnología tanto en la vida cotidiana como en el ámbito industrial y empresarial, los administradores de redes tienen que gestionar cada

vez una mayor cantidad de amenazas a nivel web y de aplicación que tienen como objetivo el robo de información de los usuarios, ya sea a nivel personal o corporativo, para su propio beneficio económico [2][3]. Según un estudio sobre la seguridad en Internet llevado a cabo por la empresa de seguridad informática Symantec en el año 2016 [4], durante el 2015 se registró un millón de ataques web por día, descubriendo sus sistemas 430 millones de tipos nuevos de *malware* durante el mismo año (Figura 2). En palabras del experto en cibercrimen y ciberseguridad John Lyons en una entrevista concedida al diario *El País* el 29 de mayo de este año, *"En 2020 ya no podremos proteger nuestras redes frente a los ataques, especialmente con el desarrollo del IoT, que hará que incluso nuestro frigorífico esté conectado a la red. Por eso, es necesario invertir en ciberseguridad en lugar de dar dinero a los criminales para recuperar datos a posteriori. Debe convertirse en una prioridad de los Gobiernos occidentales, la guerra de hoy en día se libra en el ciberespacio"* [5].

Además, este aumento de los dispositivos electrónicos conectados a Internet requiere de una gran capacidad de procesamiento para analizar toda la información procedente de la red, que lleva consigo un consumo de energía cada vez mayor por parte de los equipos encargados de realizar las tareas de monitorización, detección y corrección[6]. Fruto de esta preocupación por preservar la intimidad de las personas, nacen lo que se conocen como NIDS (*Network Intrusion Detection System* – Sistema de detección de intrusos en la red) [7], sistemas digitales encargados de asegurar una correcta experiencia de uso en el acceso a Internet. El principal procedimiento que llevan a cabo este tipo de sistemas es lo que se ha definido como DPI (*Deep Packet Inspection* – Detección Profunda de Paquetes) [8] ,que consiste en el análisis de todos y cada uno de los paquetes de información que recibe un equipo conectado a Internet, con el objetivo de detectar en ellos ciertos patrones identificadores previamente conocidos y realizar las operaciones que se consideren pertinentes con ellos. Este tipo de sistemas está siendo acogido cada vez con más interés por mayor número de entidades, ya sean cuerpos de seguridad nacionales e internacionales o empresas privadas que buscan monitorizar el tráfico de datos recibido con el fin de detectar amenazas virtuales potencialmente peligrosas para su infraestructura de red.

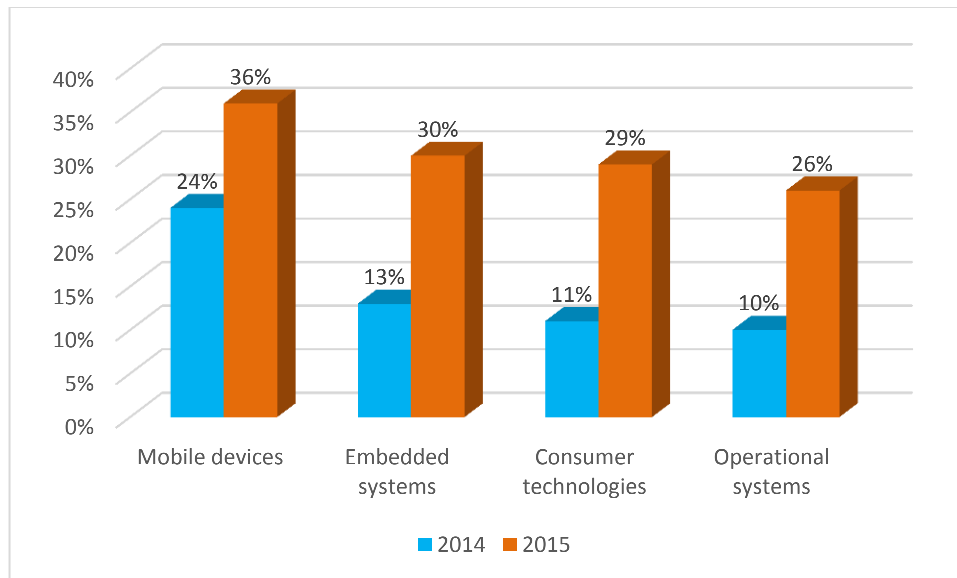


Figura 2. Incremento de ataques malintencionados a través de la red

En los últimos tiempos, se encuentra en una tendencia claramente ascendente el desarrollo de un nuevo tipo de DPI denominado NGFW (*New Generation FireWall* – FireWall de Nueva Generación), cuyo funcionamiento es similar al de un cortafuegos tal como lo conocemos hasta ahora (*software* de protección frente a posibles amenazas externas mediante bloqueo de puertos y filtrado de paquetes por protocolo), pero extendiendo su campo de actuación hasta el nivel de aplicación (Figura 3) e incluyendo funcionalidades tales como sistemas de prevención de intrusiones, intercepción de amenazas vía SSL o SSH, filtrado web o inspección del funcionamiento del antivirus. Mediante el empleo de este tipo de herramientas, se consigue una protección bastante más profunda frente a los nuevos ataques malintencionados que surgen en la actualidad, en comparación con el *software* tradicional empleado en las últimas décadas [9].

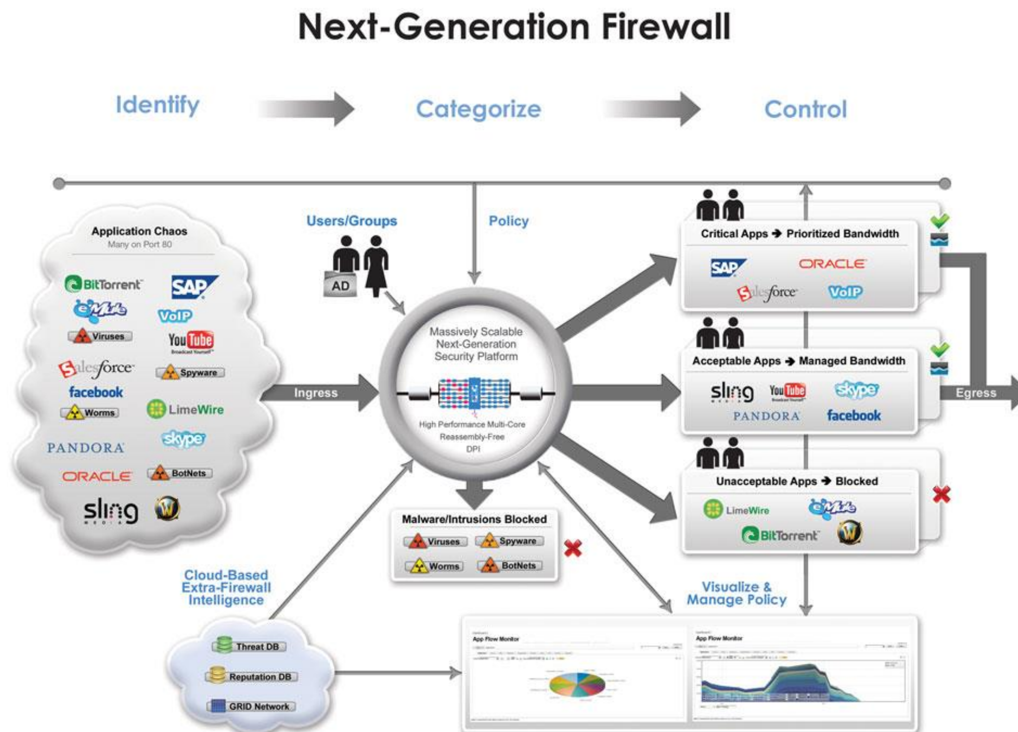


Figura 3. Ámbito de actuación de un NGFW [10]

El principal problema que presenta este tipo de sistemas es que, al analizar cada uno de los paquetes de información que llegan a un equipo a través de la red, llevan consigo un consumo considerable de tiempo, memoria y potencia; por tanto, se hace necesario el diseño y desarrollo de algún tipo de acelerador *hardware* que permita reducir sustancialmente estas desventajas.

En este sentido, como resultado de los trabajos de investigación llevados a cabo, se ha concluido que las estructuras que incluyen conjuntamente núcleos procesadores y dispositivos programables, en su mayor caso FPGAs, en una plataforma de desarrollo permite abordar este problema de forma eficiente, permitiendo una alta capacidad de cómputo, una reducción significativa de la potencia consumida por el sistema y la posibilidad de reconfigurar la plataforma en el instante en el que sea necesario, permitiendo a la misma adaptarse a nuevos condicionantes que pueden aparecer con el transcurrir del tiempo (Figura 4) [11].

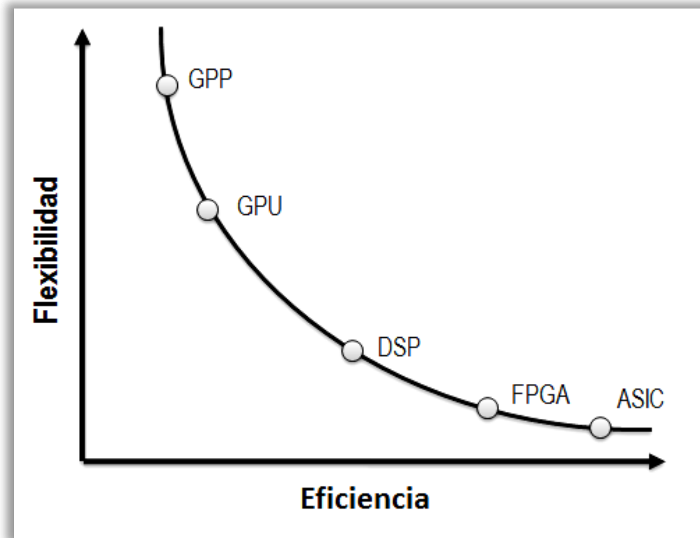


Figura 4. Relación flexibilidad-eficiencia entre distintas soluciones electrónicas

En este tipo de dispositivos electrónicos convergen la capacidad de cómputo de los núcleos procesadores y la flexibilidad y reconfigurabilidad de las FPGAs. El flujo de diseño del SoC ahora incluye las siguientes etapas:

1. Análisis de las características de la aplicación, perfilado y partición *hardware/*software del sistema
2. Diseño *hardware* a medida mediante el empleo de una metodología de diseño de síntesis de alto nivel
3. Diseño e integración de la plataforma basada en bloques IP y en los buses mediante los cuales se establecen las comunicaciones pertinentes entre los mismos
4. Programación del *software* empotrado, el cual puede incluir sistema operativo o no en función de la aplicación final
5. Integración *hardware/software* y verificación del sistema

De esta forma, nos encontramos con un flujo de diseño que, además de ser abordado en el dominio *hardware*, precisa del desarrollo *software* empotrado necesario para el correcto funcionamiento del sistema electrónico completo.

1.2. Objetivos

Este trabajo fin de grado consiste en la implementación sobre FPGA de un *firewall* de última generación que permita, mediante un algoritmo de búsqueda de patrones específico, la detección de determinadas direcciones IP origen de los paquetes de datos recibidos a través de una red TCP/IP, realizando a partir de dicha consulta las operaciones que se consideren oportunas según la respuesta proporcionada y en función de si dichas direcciones IP se encuentran en nuestra lista negra o no (igualmente se pueden gestionar listas blancas). Para ello, se describirán los algoritmos en lenguaje de alto nivel (en este caso, C/C++), que se verificará y se transformará en *hardware* mediante herramientas de síntesis de alto nivel.

Este trabajo utiliza la plataforma de referencia desarrollada en la División SICAD del IUMA, basada en un SoC programable (Zynq 7000) que permite extraer información de un paquete de datos de la red y analizar su contenido[12].

Los objetivos propuestos son los siguientes:

1. Elección y modelado de los algoritmos apropiados para que nuestro sistema cumpla con la funcionalidad deseada

2. Síntesis, optimización e implementación del sistema sobre la plataforma Zynq de Xilinx

3. Validación del sistema

1.3. Peticionario

La realización de este trabajo se realiza por petición de la División de Sistemas Industriales y CAD (SICAD), perteneciente al Instituto Universitario de Microelectrónica Aplicada (IUMA), instituto de investigación perteneciente a la Universidad de Las Palmas de Gran Canaria.

La implementación del bloque IP desarrollado se considera necesaria con el fin de ser incluido como filtro de preprocesado a la entrada de la plataforma de referencia desarrollada con anterioridad por la división, para evitar de esta forma que los paquetes deseados no alcancen el sistema de análisis detallado y así mejorar de forma notable las prestaciones globales del sistema al no ser preciso analizar paquetes desechables con la información de su cabecera.

Asimismo, hay que incluir como solicitante de este proyecto a la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE), con el fin de cubrir los créditos y requerimientos de la asignatura Trabajo Fin de Grado dentro del plan de estudios del Grado en Ingeniería en Tecnologías de la Telecomunicación.

1.4. Estructura del documento

En este primer capítulo se relatan los antecedentes que derivan en la realización de este trabajo, además de exponer los objetivos a alcanzar, el solicitante (o solicitantes) del mismo y cómo se estructura este documento. En el capítulo 2 se describen los algoritmos de búsqueda de patrones más utilizados actualmente en nuestro ámbito de trabajo, analizando qué opciones se ajustan mejor a la aplicación que se desea implementar. Por otro lado, en los capítulos 3 y 4 se describe la arquitectura de la plataforma Zynq de Xilinx y el flujo de diseño Xilinx Vivado, respectivamente. Seguidamente, en el capítulo 5 se realiza una aproximación a la solución que se ha considerado óptima para abordar el problema planteado.

En el capítulo 6 se describe en detalle el proceso de diseño y síntesis de alto nivel del bloque IP desarrollado, mientras que en el capítulo 7 se detalla la integración del mismo en la plataforma para posteriormente verificar el funcionamiento del sistema electrónico completo.

Para finalizar, en el capítulo 8 se realiza la validación del sistema y la recogida de los resultados obtenidos para su análisis, mientras que en el capítulo 9 se citan las conclusiones alcanzadas tras finalizar la realización de este trabajo.

Capítulo 2. Algoritmos de búsqueda de expresiones regulares

2.1. Introducción

Centrándonos a continuación en los algoritmos a implementar en la plataforma con el objetivo de detectar un determinado patrón previamente conocido que nos permita identificar con precisión un tipo de paquete de datos específico, debemos tener en cuenta esencialmente dos factores: el consumo de recursos de cómputo y almacenamiento de la implementación de dicho algoritmo sobre la FPGA y el tiempo de ejecución total del mismo, ya que el objetivo establecido es el de diseñar un sistema acelerador lo suficientemente rápido en cuanto a cómputo se refiere consumiendo la menor cantidad de recursos posible [13].

Existen dos grandes clasificaciones en la literatura actual sobre este tipo de algoritmos, las cuales introduciremos brevemente a continuación, describiendo aquellos algoritmos más empleados dentro de cada grupo en el ámbito tecnológico actual.

2.2. Algoritmos de búsqueda de cadenas fijas

Este tipo de algoritmos se centran en encontrar cadenas de caracteres fijas e inmodificables en el tiempo y conocidas con anterioridad. Los algoritmos más usados de este tipo realizan la búsqueda de cada carácter de la cadena de forma individual, con el consiguiente alargue del tiempo de ejecución del mismo. Dentro de este apartado cabe destacar las cuatro siguientes implementaciones:

2.2.1. Boyer-Moore (BM).

Este algoritmo se basa en la comprobación de que el último carácter de un *string* introducido como entrada coincide con el último carácter de nuestra cadena objetivo. Si coinciden, se retrasa una posición el puntero y se vuelve a realizar la comparación con el carácter anterior. Esta comprobación se realizará de forma iterativa tantas veces como caracteres tenga la palabra (Figura 5).

El algoritmo precalcula dos tablas (llamadas tablas de salto) para procesar la información que obtiene en cada verificación fallida:

- una primera tabla calcula cuántas posiciones hay por delante en la siguiente búsqueda basada en el valor del carácter que no coincide, y
- una segunda tabla hace el cálculo similar basado en cuántos caracteres coincidieron satisfactoriamente antes del intento de coincidencia fallido.

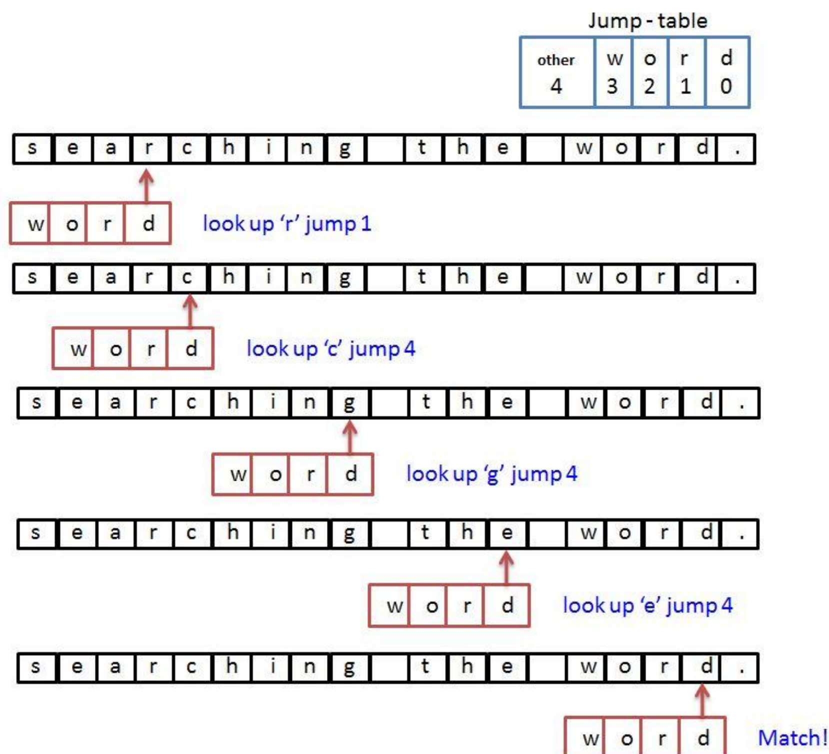


Figura 5. Ejemplo de funcionamiento del algoritmo Boyer-Moore

Este tipo de búsqueda presenta un tiempo de ejecución más bajo que otros algoritmos de este tipo ya que, en caso de que alguno de los caracteres del *payload*

entrante no coincidan con los de la palabra a buscar, se saltan tantos caracteres como indiquen las tablas de salto en función del último carácter del *payload* con el que se ha realizado la última comparación satisfactoria.

El rendimiento de este algoritmo para un *payload* de longitud n y un patrón de longitud m, es n/m; en el mejor de los casos, solo uno de m caracteres necesita ser comprobado (en caso de que el último carácter del patrón, el primero que se examina, no sea coincidente). El caso peor para encontrar todas las coincidencias en un *payload* dado es de aproximadamente $3n$ comparaciones [14].

2.2.2. Knuth-Morris-Pratt (KMP).

Al igual que el algoritmo BM, el algoritmo KMP también tiene como objetivo encontrar un determinado patrón dentro de una cadena de caracteres de mayor longitud que el mismo, aunque la forma de proceder es bastante distinta. Lo primero que hace este algoritmo es calcular una tabla, conocida como tabla de fallos, a partir de la palabra objetivo, que servirá para hacer saltos cuando se localice un fallo en alguna de las búsquedas, evitando de esta forma volver a analizar un *string* que ya ha sido comparado con el patrón a buscar. Esta tabla se elabora con la distancia que existe desde una ocurrencia en la palabra (un carácter) hasta la posición en la que vuelve a repetirse y, mientras sigan coincidiendo, se marca la distancia (cuando haya una ruptura de coincidencia se indica con un 0 generalmente).

Una vez calculada la tabla, empezamos a realizar comparaciones con el patrón a buscar en el *payload* usando un puntero de avance de modo que, si ocurre un fallo, en vez de volver a la posición siguiente a la primera coincidencia, se salta hacia donde indique la tabla de fallos (Figura 6).

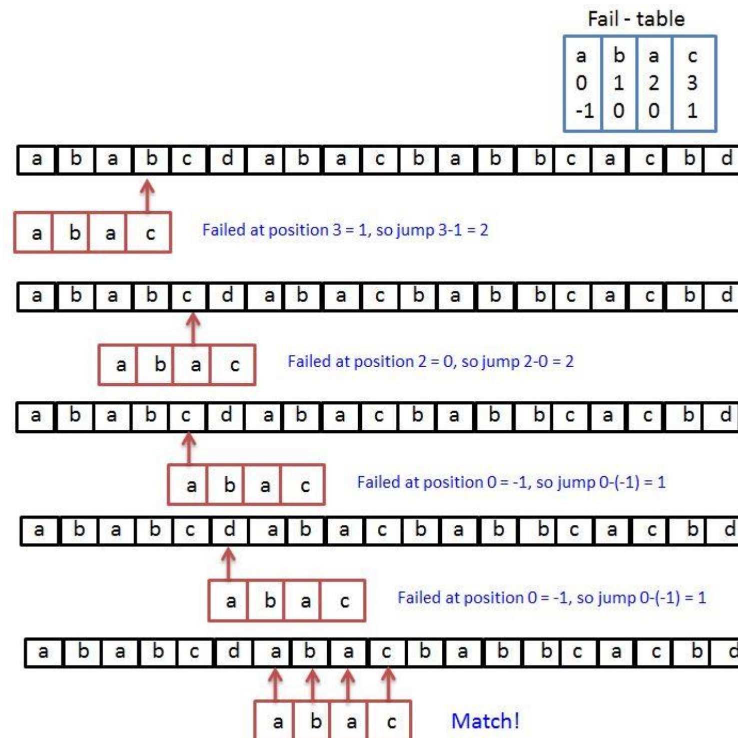


Figura 6. Ejemplo de funcionamiento del algoritmo KMP

El rendimiento de este algoritmo es algo más lento que el de BM, ya que es de *m+n*, siendo *m* la longitud del *payload* y *n* el número de caracteres del patrón a buscar [15].

2.2.3. Karp-Rabin.

Su funcionamiento se basa esencialmente en tratar cada una de las palabras de *m* caracteres del *payload* a examinar como un índice de una tabla de valores *hash* calculado mediante una operación de módulo, resultante de dividir el número entero equivalente a la cadena de caracteres objeto de la consulta con un número previamente definido de manera que, si la función *hash* de los *m* caracteres de una palabra concreta coincide con la del patrón, es posible que hayamos encontrado un acierto. Para verificarlo hay que comparar la palabra coincidente con el patrón, pues puede que la función *hash* aplicada sea propensa a devolver falsos positivos.

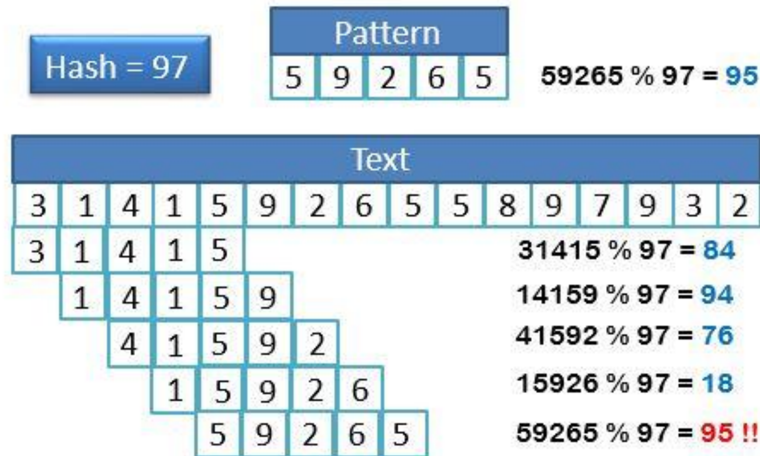
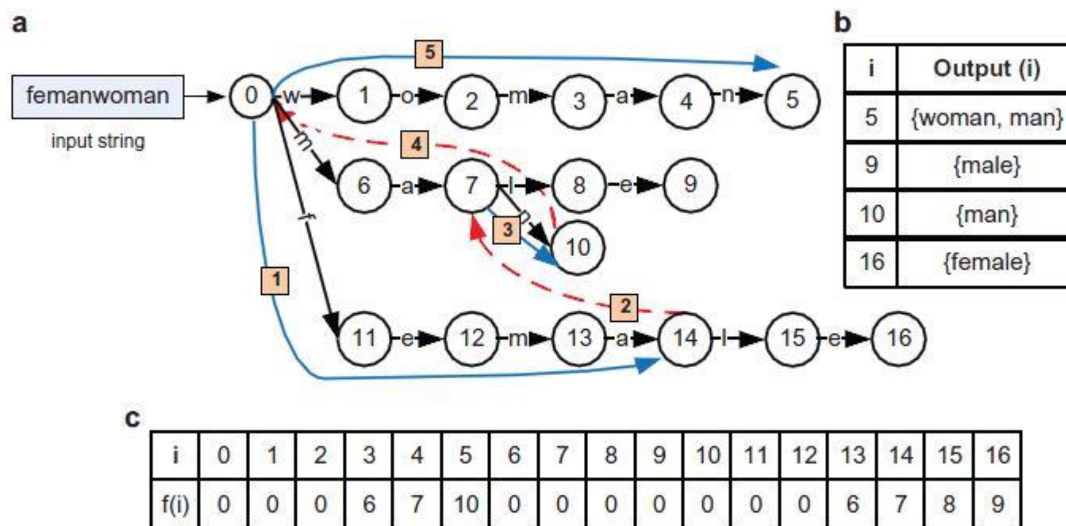


Figura 7. Ejemplo de funcionamiento del algoritmo Karp-Rabin para un índice *hash* de 97

A pesar de ser un algoritmo bastante eficiente en cuanto a precisión de la búsqueda se refiere (siempre y cuando la función *hash* empleada también lo sea), no es ampliamente utilizado porque su tiempo de ejecución para el mejor caso es de *m+n*, donde *m* es el número de caracteres del patrón a buscar y *n* la longitud del *payload* a analizar [16].

2.2.4. Aho-Corasick.

Capaz de realizar la búsqueda de varios patrones de forma simultánea dentro de un *payload* dado, aunque solo procesa un carácter por ciclo de ejecución y una sola vez durante el ciclo temporal global de funcionamiento, este algoritmo permite la detección de palabras incluidas en su diccionario, ubicando todos y cada uno de los patrones objeto de búsqueda en una estructura en forma de árbol que se comporta como un autómata finito [17]. Su funcionamiento se basa en las transiciones entre los nodos que componen dicho árbol, con enlaces adicionales entre nodos (cada nodo representa un carácter de alguno de los patrones almacenados) con alguna relación de prefijo, permitiendo transiciones más rápidas en caso de encontrar una correspondencia fallida entre caracteres de una determinada palabra (Figura 8). Por tanto, este algoritmo resulta bastante eficaz cuando los patrones a buscar presentan caracteres similares en alguna ubicación de su longitud [18].



i	Output (i)
5	{woman, man}
9	{male}
10	{man}
16	{female}

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
f(i)	0	0	0	6	7	10	0	0	0	0	0	0	0	6	7	8	9

Figura 8. Ejemplo de algoritmo Aho-Corasick. (a) Entrada al algoritmo. (b) Función de fallo. (c) Función de salida [18].

La complejidad del algoritmo viene dada por la suma de las longitudes de los patrones a buscar más la longitud del *payload*. Como desventajas principales presenta, además de un considerable tiempo de ejecución, numerosos accesos a memoria (en la actualidad existen diferentes modificaciones que consiguen optimizar este parámetro) y la dificultad de reconfigurar la FPGA de forma dinámica, ya que incluir una nueva palabra en el diccionario supone el diseño e implementación de nuevos nodos [19].

2.3. Algoritmos de búsqueda de patrones regulares

Con el crecimiento incontrolable que está sufriendo el número de dispositivos conectados a Internet y el consiguiente incremento de los ataques malintencionados, es preciso realizar la búsqueda de expresiones más complejas compuestas por dos o más de este tipo de cadenas de forma conjunta, pudiendo incluir distintos símbolos además de caracteres alfabéticos o una serie de operadores especiales que tienen como fin la creación de cadenas aún más complejas; a este conjunto de *strings* de mayor complejidad es a lo que conocemos como expresiones regulares. Los algoritmos existentes encargados de la detección de este tipo de patrones pueden agruparse en dos grupos bien diferenciados:

2.3.1. Autómatas finitos.

Esta clase de algoritmos están basados en máquinas de estados, que son modelos secuenciales síncronos basados en un conjunto de estados posibles y de las transiciones entre ellos, dando un resultado específico para cada entrada aplicada [18]. La detección de un determinado patrón se lleva a cabo mediante transiciones entre los distintos nodos que componen el autómata, representando cada uno de ellos a un carácter del patrón a buscar (de forma similar a como pasaba en el algoritmo Aho-Corasick), a partir de una cadena dada como entrada. Dentro de esta categoría, podemos distinguir entre dos subtipos: autómatas finitos no deterministas (NFA) y deterministas (DFA).

Un NFA es un grafo con un único estado inicial pero varios posibles estados finales, de forma que no se puede deducir el comportamiento del autómata, no funcionando de forma lineal. Este grave inconveniente se puede solucionar mediante la implementación en *hardware* de varios autómatas en paralelo ejecutándose concurrentemente y centrándose cada uno de ellos en un grupo de expresiones en concreto, consiguiendo de esta forma también una mayor velocidad de ejecución pero con el consiguiente gasto en recursos *hardware*. Este algoritmo dará como positiva una coincidencia cuando, a partir de una palabra aplicada como entrada, el autómata alcance alguno de los estados definidos como posibles finales. El espacio de memoria que ocupa un NFA es directamente proporcional a la longitud de la expresión a buscar, procesando un solo carácter en un tiempo igual al número de caracteres de la palabra objetivo; esto da como resultado un tiempo de búsqueda de $m \cdot n$, siendo m la longitud del *payload* y n el número de caracteres del patrón a buscar [20].

Por su parte, un DFA es un autómata con un estado inicial y un único estado final para cada posible patrón. Por tanto, podemos afirmar que se comporta de forma lineal, lo que propicia que sea bastante más rápido y eficiente que un NFA (Figura 9). Este autómata es capaz de procesar un carácter en cada ciclo de reloj, por lo que el tiempo de ejecución será equivalente a la longitud m del *payload* a analizar. Además, es posible la obtención de DFAs equivalentes a partir de un único NFA para su implementación en

hardware empleando técnicas de *pipelining* [18]. Sin embargo, el inconveniente principal de este algoritmo viene dado por la cantidad de memoria que ocupa, que es del orden de 2^n, siendo n la longitud del patrón a buscar. Esto ha hecho que en los últimos años las investigaciones sobre este algoritmo se centren en reducir considerablemente el espacio de memoria que ocupa su implementación [20].

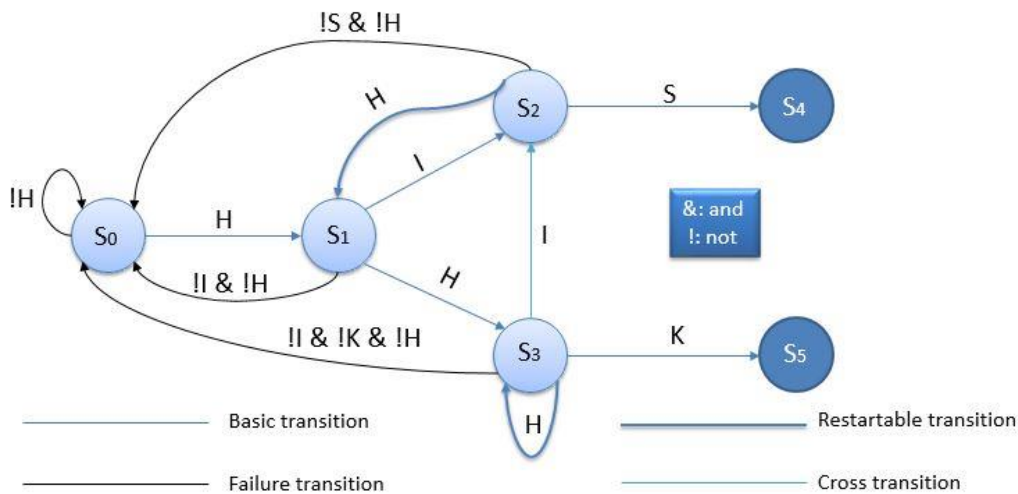


Figura 9. Ejemplo de DFA para patrones HIS y HHK (Adaptada de [13])

2.3.2. Filtros Bloom.

Se trata de una estructura de datos que, mediante el uso de métodos probabilísticos, permite saber si un determinado elemento pertenece a un conjunto. Empleando este tipo de metodología estadística, se consigue reducir el tiempo de análisis de la información entrante en el equipo a través de la red.

Además, esta técnica es muy recomendada en aquellas aplicaciones en las que se necesita de un consumo mínimo de recursos, el cual dependerá del tiempo de respuesta que deseemos para nuestro sistema y del número de elementos que lo integran. Además, se trata de un procedimiento bastante eficiente, ya que siempre tarda lo mismo en responder a una petición de búsqueda, independientemente del número de elementos que integran el conjunto. Los tiempos de inserción de un patrón en el conjunto también son siempre estables.

El principal inconveniente de estas estructuras radica en que es propicia a dar "falsos positivos", es decir, puede indicar que el elemento buscado está dentro del conjunto cuando realmente no es así. Esto ocurre cuando los *bits* se han puesto a 1 durante la inserción de nuevos elementos en el conjunto. Sin embargo, la tasa de este tipo de error suele ser comúnmente del orden del 1% o menor (directamente proporcional al número de elementos de nuestro conjunto), siendo configurable mediante parámetros del filtro. Por otro lado, la probabilidad de que el filtro nos dé un resultado de "falso negativo" es nula.

La probabilidad de "falsos positivos" viene determinada por la expresión:

$$f = (1 - e^{-\frac{kn}{m}})^k \qquad (1)$$

donde *k* es el número de funciones *hash*, *m* el número de posiciones del *array* y *n* el número de *ítems* que integran el conjunto.

Dicho de otra forma, el número *k* de funciones *hash* necesarias para obtener una baja tasa de "falsos positivos" dependerá del tamaño *m* del *array* y de la dimensión *n* del conjunto:

$$k = \ln(2) \cdot {}^{m}/_{n} \qquad (2)$$

Por tanto, tal como se observa en la expresión anterior, la probabilidad de falsos positivos únicamente dependerá de la relación bit por elemento *m/n*.

Las operaciones básicas que se realizan empleando este tipo de estructuras son la inserción de elementos y la consulta de si algún patrón específico se encuentra dentro de la "base de datos" del filtro. Inicialmente, un filtro Bloom no es más que un vector de *n* *bits* con todos y cada uno de ellos a valor 0. En cada una de estas posiciones se irán mapeando los patrones introducidos mediante funciones *hash* que realizan esta distribución de una manera aleatoria uniforme (cálculo de múltiples funciones *hash* o de la misma función varias veces para cada miembro del conjunto). Cuando se inserta un elemento en el vector, el *hash* correspondiente cambiará su *bit* de estado a 1. Cada

patrón introducido en el conjunto cambiará a 1 tantas posiciones del vector como funciones *hash* hayamos decidido ejecutar.

En la Figura 10 se puede observar un ejemplo en el que se define un vector de *m*=16 bits y en el que se introducen 2 elementos (*a* y *b*) empleando *k*= 2 funciones *hash*; al realizar la consulta sobre si un tercer elemento *c* se encuentra en el conjunto, el filtro da una respuesta negativa, ya que una de las posiciones del vector que se le asigna mediante las funciones *hash* se encuentra a 0 [21].

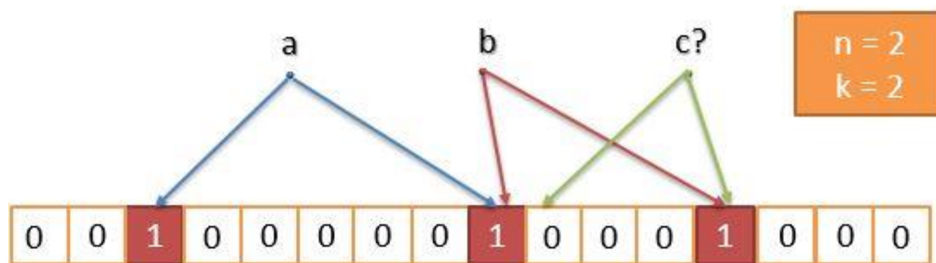


Figura 10. Ejemplo de filtro Bloom con m=16 bits y dos elementos dentro del vector

A la hora de realizar una consulta sobre un elemento en concreto sobre una implementación *hardware* de este tipo de filtros, lo que se hará es instanciar todas las funciones *hash* de forma concurrente para preguntar sobre dicho patrón. Si la AND resultante devuelve un 0, se descartará la consulta y se ahorrará en accesos innecesarios a memoria, ya que el elemento no está en nuestra estructura de datos con un 100% de seguridad. Pero si todos y cada uno de los bloques que calculan el hash devuelven direcciones del mapa de *bits* en las que nos encontramos los *bits* a 1 (AND=1), nuestro elemento puede formar parte del *array* o no, debido a que quizás un *hash* en particular se encuentra a 1 por otro miembro del conjunto distinto al que estamos introduciendo como objetivo de la consulta [22].

Al no permitirse “falsos negativos”, eliminar un elemento del vector resulta prácticamente imposible, ya que al poner a 0 una determinada posición del vector se puede estar incidiendo sobre más de un elemento, lo que podría llevar a eliminaciones erróneas.

Si queremos introducir esta opción, debemos optar por lo que se conoce como Filtro Bloom con Contador (CBF por sus siglas en inglés), que introduce un contador para cada *bit* del *array*, el cual se incrementa o decrementa en función de si se añade o se elimina, respectivamente, un elemento en dicha posición. Cuando un contador cambia de 0 a 1, el *bit* correspondiente en el vector de bit se establece, mientras que cuando cambia de 1 a 0, el *bit* correspondiente en el vector de *bits* se borra (Figura 11).

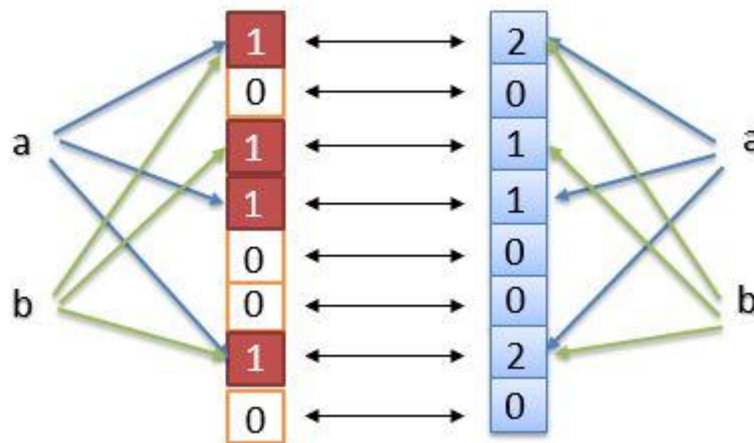


Figura 11. Ejemplo de CBF con dos elementos

Esta variación es la más empleada en el ámbito de la monitorización y enrutado de redes de comunicación aunque, adicionalmente, existen distintas opciones basadas en filtros Bloom y que están destinadas a optimizar distintos parámetros del sistema final, destacando especialmente en este punto el consumo de recursos del sistema (especialmente memoria) y de energía, parámetros clave en el diseño de sistemas electrónicos empotrados [21][23][24].

2.4. Conclusiones

En este capítulo se han estudiado los distintos algoritmos que se utilizan para el análisis de expresiones, ya sea para la búsqueda en cadenas fijas como para la búsqueda de patrones regulares. Se hace un estudio detallado de su complejidad y se plantean algunos aspectos más significativos en cuanto a su implementación hardware.

Una vez realizado dicho estudio, se opta por la selección del algoritmo basado en Filtros Bloom con contador para su implementación *hardware* en el sistema final.

Capítulo 3. Estudio de la plataforma Xilinx Zynq

3.1. Introducción

Para incrementar las prestaciones de un NGFW se hace necesario el diseño de un acelerador *hardware*, con el fin de realizar búsquedas acerca de las direcciones IP origen de los paquetes de datos entrantes a través de la red de la forma más rápida y eficiente (en términos de tiempo y consumo) posible.

Se ha optado por un sistema tipo SoC compuesto por núcleos procesadores que permitan alcanzar las altas tasas de cómputo exigidas, junto con lógica programable (una FPGA) que ofrece la posibilidad de reconfiguración y de flexibilidad para una aplicación específica. Xilinx ofrece la solución Zynq que incluye estos requerimientos.

Todos los SoC que integran la familia Zynq-7000 de Xilinx están compuestos por dos bloques básicos: un bloque de procesamiento (PS) y otro lógico (PL), tal como se observa en la Figura 12.

De forma más simplificada, el diagrama de bloques del SoC se muestra en la Figura 13. Como se puede apreciar, el PS o unidad de procesamiento está formado por dos *cores* ARM Cortex-A9, sus correspondientes unidades NEON DSP/FPU (una por *core*) las memorias caché, distintos *timers* y unidades de control, así como las interfaces de memoria, los periféricos de entrada/salida y un conjunto de buses en chip basados en AMBA AXI para la interconexión de todos los bloques y para la realización de la interfaz con el bloque lógico (PL). Por su parte, el bloque PL incluye los bloques lógicos programables (CLBs), así como bloques de memoria RAM dedicada y DSPs, entre otros [25].

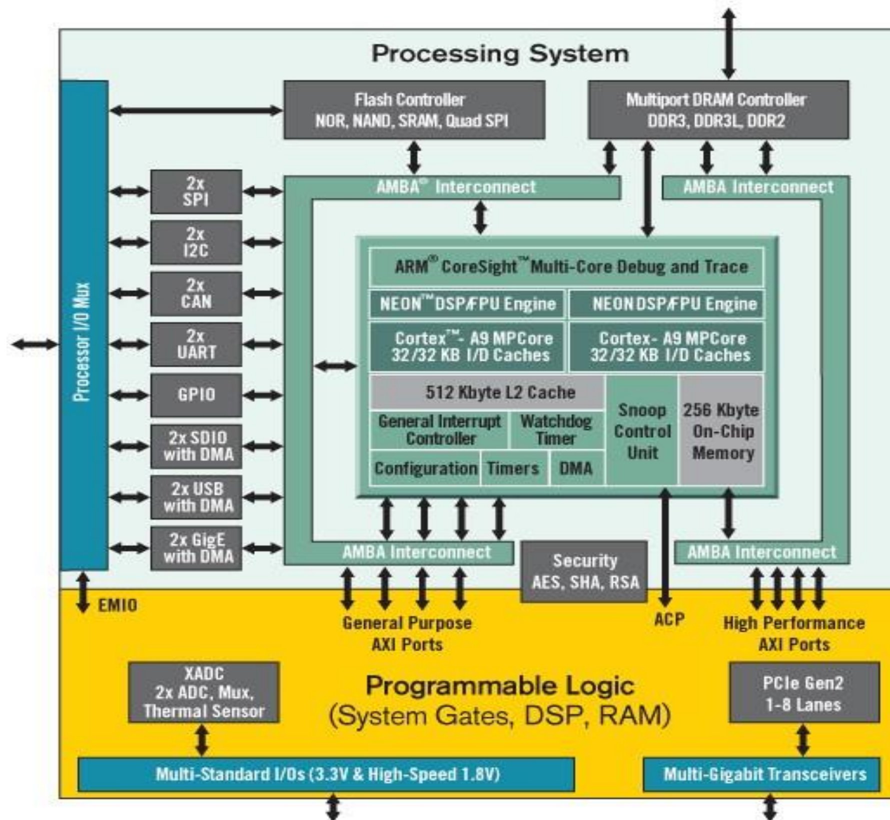


Figura 12. Diagrama de la familia Zynq-7000 [26]

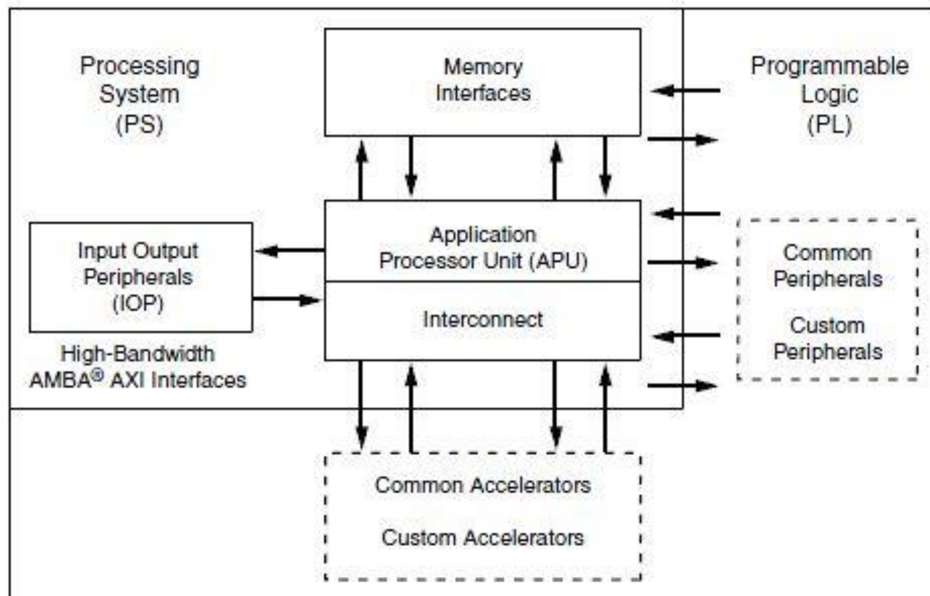


Figura 13. Diagrama de bloques a alto nivel [25]

3.2. Dispositivo XC7Z020

Dentro de la gama de dispositivos de la familia Zynq que ofrece Xilinx a sus usuarios, trabajaremos con el XC7Z020, un SoC que integra una FPGA de la familia Artix-7 y dos núcleos procesadores ARM Cortex-A9, además de una serie de interfaces que permiten la comunicación con periféricos o el depurado *in situ* del chip.

Entre las características de la familia Artix-7 de FPGAs caben destacar las mencionadas en la Tabla 1.

Tabla 1. Características relevantes de la familia de FPGAs Artix-7

CLBs	215K
Block RAM	13 Mb
DSP *slices*	740
Transceivers	16
Velocidad máxima de transferencia de los *transceivers*	6.6 Gbps
Ancho de banda máximo (Full duplex)	211 Gbps
***Pines* de E/S**	500
Tensiones de E/S	1.2V, 1.35V, 1.5V, 1.8V, 2.5V, 3.3V

Cada CLB, por su parte, se compone de 2 *slices,* cada uno con 4 LUTs (*LookUp Tables*) de hasta 6 entradas cada una de ellas, dando lugar a un total de 8 LUTs por CLB, las cuales proporcionan la posibilidad de ser configuradas como memoria RAM distribuida (de 64x1 o 32x2 *bits*) o como desplazador de *bits* a la izquierda (SRL). Además, cada bloque lógico incluye un total de 16 *flip-flops* (Figura 14)[27].

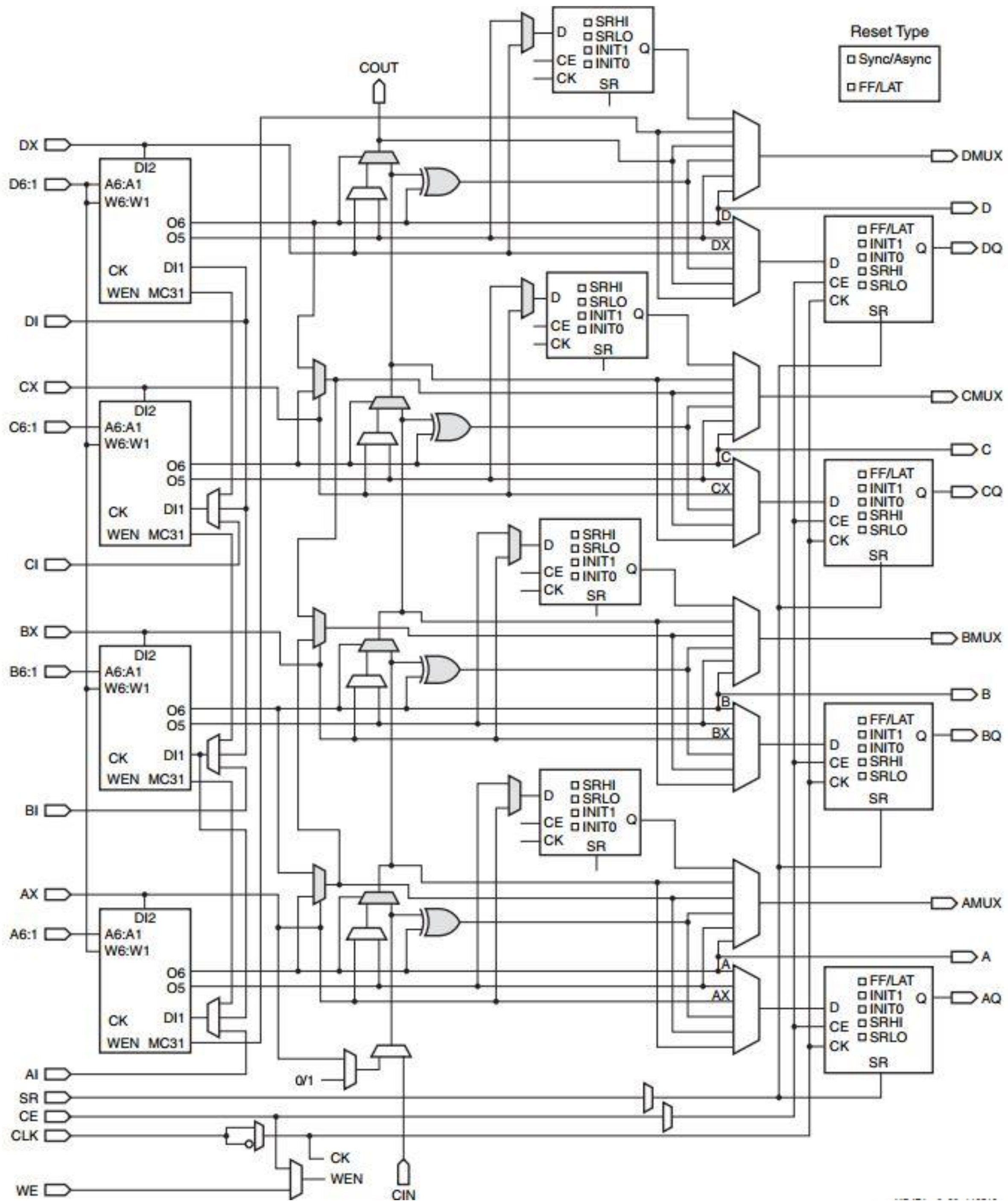


Figura 14. *Slice* de un CLB de la familia Artix-7 de Xilinx [27]

Los bloques de memoria RAM dedicada son de doble puerto y tienen una capacidad de 36 Kb (posibilidad de configuración 18x2 Kb). Como elemento adicional relevante, cuentan con lógica para gestionar sistemas de colas FIFO, usadas cuando la escritura y la lectura de la memoria se realizan a diferente tasa binaria.

Las interfaces de E/S soportan tecnologías como pueden ser LVCMOS, LVDS y SSTL. Según la tecnología empleada, tendremos una u otra tensión de alimentación de las especificadas en la Tabla 1, que será aportada al periférico asociado a una determinada interfaz para su funcionamiento. El PL puede acceder a puertos de E/S a través de los EMIO sin necesidad de que estos periféricos se encuentren mapeados en la memoria del PS [25][28].

3.3. Bloque PS

Tal como se ha hecho mención con anterioridad, el sistema de procesamiento (PS) está conformado por cuatro bloques diferentes (Figura 15):

- la unidad de procesamiento de aplicación (APU),
- las interfaces de memoria,
- los periféricos de entrada/salida (IOP), y
- las interconexiones con el PL.

El bloque principal o APU contiene en su interior, además de los dos *cores* ARM Cortex-A9 de 32 bits, una memoria RAM *on-chip* de doble puerto con una capacidad de 256KB (que proporciona baja latencia), y la memoria caché del sistema de 512KB. Cada núcleo es capaz de alcanzar una frecuencia de funcionamiento de 866 MHz en el modo más rápido y cuenta con dos memorias caché de 32KB separadas para instrucciones y datos.

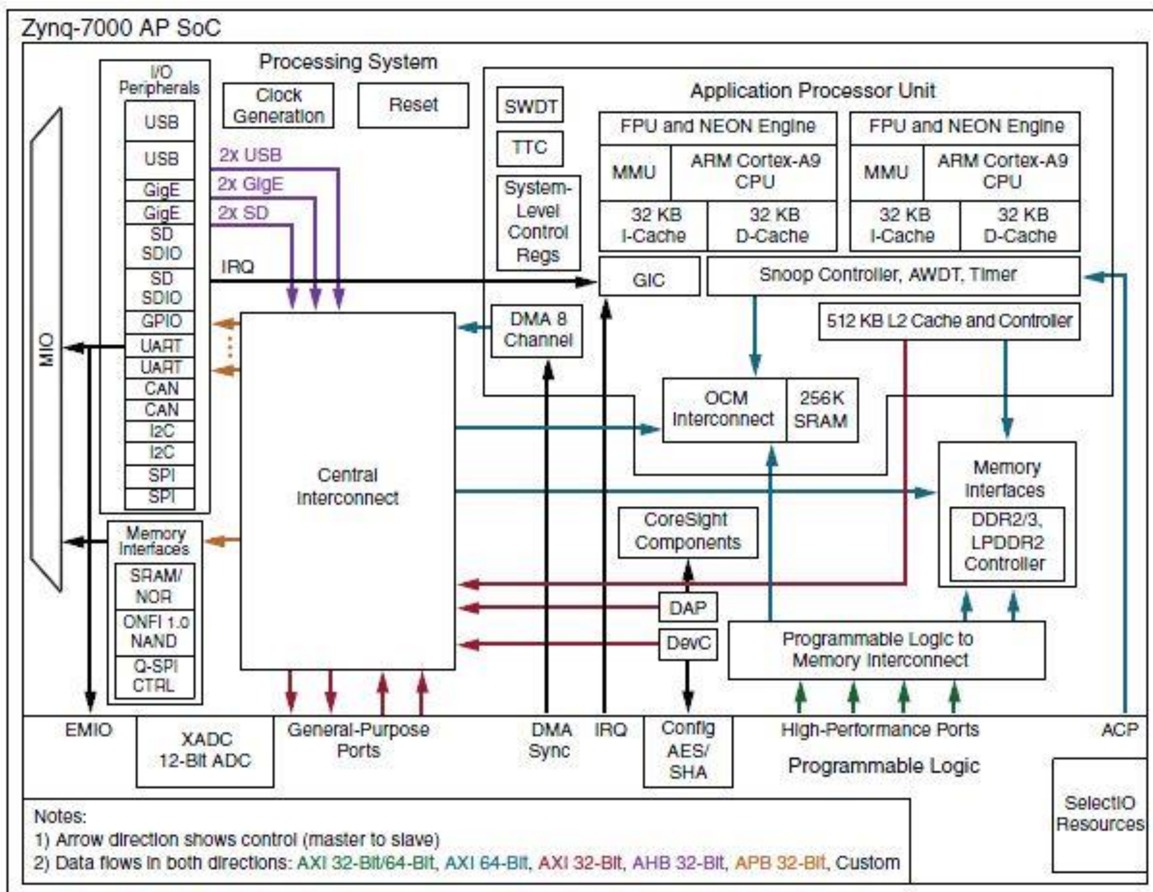


Figura 15. Diagrama de bloques de un SoC de la serie Xilinx Zynq-7000 [25]

Otros elementos integrados en la APU son los siguientes:

a. El bloque *Accelerator Coherency Port* (ACP), interfaz que permite que el PL realice accesos coherentes a la memoria de la CPU.
b. El *Generic Interrupt Controller* (GIC), encargado de gestionar todas las interrupciones procedentes de las interfaces de entrada/salida.
c. Canales de DMA, 8 en total, 4 de ellos dedicados exclusivamente al PL, que permiten la escritura y lectura de la memoria del sistema de ciertos bloques o periféricos sin necesidad de realizar interacciones con la CPU, empleando una interfaz AMBA AXI de 64 bits.
d. Adicionalmente, cuenta con una serie de sistemas de interrupción y *timers* (como contadores o *watchdogs*), útiles para indicar la realización de ciertos eventos a los procesadores, así como con la interfaz CoreSight de ARM para el depurado *in situ* de los *cores* [29].

La unidad de interfaces de memoria incluye módulos de control tanto de memorias dinámicas como estáticas. El controlador de memorias dinámicas soporta memorias de los tipos DDR3, DDR3L, DDR2 y LPDDR2, pudiendo proporcionar accesos de 16 o 32 *bits* a una memoria DRAM de como máximo 1GB de capacidad y alcanzando velocidades máximas de transferencia de 1.333 Mbps para memorias DDR3; el controlador de memorias estáticas, por su parte, permite el empleo de memorias NAND *flash*, Quad-SPI *flash*, bus de datos paralelo y NOR *flash* paralela.

El bloque IOP incluye un conjunto de periféricos de comunicaciones: controladores Ethernet, CAN, USB 2.0, tarjetas SD, I2C, UART, puertos SPI para comunicación *full-duplex* y hasta 118 GPIO.

Finalmente, el bloque de interconexión, que comunica todas las partes del PS con el bloque lógico, está basado en buses del estándar AMBA AXI de ARM, protocolo orientado a comunicaciones a altas tasas de transferencia de datos en diseños con baja latencia temporal. AMBA AXI permite transacciones de datos entre múltiples dispositivos maestros y esclavos de forma simultánea, incluyendo mecanismos de arbitraje para otorgar el uso del bus en cada momento. Igualmente, posee la capacidad de adaptar el ancho de banda entre maestros y esclavos que trabajen a diferentes tasas binarias o la posibilidad de gestionar el tráfico de datos que circula por el bus mediante parámetros de calidad de servicio. Además, este bloque actúa de *bridge* para la comunicación desde el PL hacia periféricos como UART y GPIO mediante buses APB y desde periféricos de comunicación a altas tasas binarias, como por ejemplo Ethernet o USB, al propio PL, mediante buses AHB, ambos de 32 *bits* e incluidos también en el estándar AMBA de ARM [25].

En cuanto a la gestión de los relojes del sistema se refiere, el XC7Z020 cuenta con 4 *Mixed-Mode Clock Manager* (MMCM) y 4 PLL. En el bloque PS de nuestro SoC contamos con un total de 3 PLL, uno asignado a la APU, otro al controlador de memoria DDR y uno más destinado a controlar la frecuencia de reloj de los periféricos de E/S, con los que el diseñador puede asignar diferentes frecuencias de funcionamiento a cada uno de estos

bloques de forma independiente. La comunicación entre zonas del sistema con diferentes frecuencias de funcionamiento se ajusta mediante el uso de FIFOs y de circuitería de adaptación (*Cross Clock Domain Circuits*) y metodologías automáticas de CDC.

Con el objetivo de reducir el consumo de energía, podemos reducir la frecuencia de salida de los PLL (el consumo es directamente proporcional a la frecuencia) o podemos desactivar los PLL de la APU y de los IOP, dejando únicamente activo el PLL del controlador DDR, el cual puede gestionar todos los generadores de reloj del PS (Figura 16). Para el PL, aunque se puede gestionar de forma independiente, también contamos con 4 señales de reloj asíncronas procedentes del PS [29].

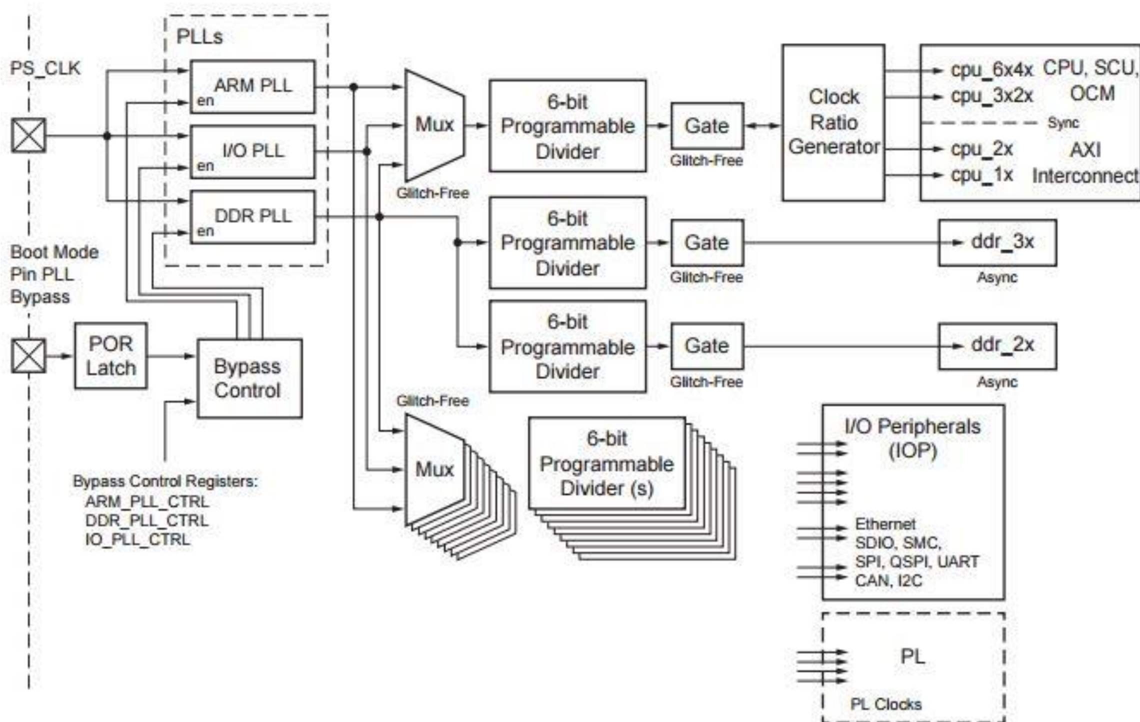


Figura 16. Diagrama de bloques de la distribución de reloj en el PS [29]

Una vez introducida la gestión de las señales de reloj para la familia Zynq-7000 de Xilinx, pasaremos a comentar la distribución de los mismos en el SoC. Dentro de cada SoC existen 32 líneas de reloj globales con alto *fanout* y que llegan a todos las señales CLK del sistema, a los *clock enable* y a los *set/reset* de los *flip-flops* ubicados dentro de nuestro dispositivo. Además, hay 12 líneas de reloj globales horizontales, pudiéndose activar cada

una de ellas de forma independiente con el objetivo de establecer diferentes frecuencias de reloj en cada región o de consumir menos energía al mantener apagadas aquellas que no se necesiten en un determinado momento (Figura 17) [30].

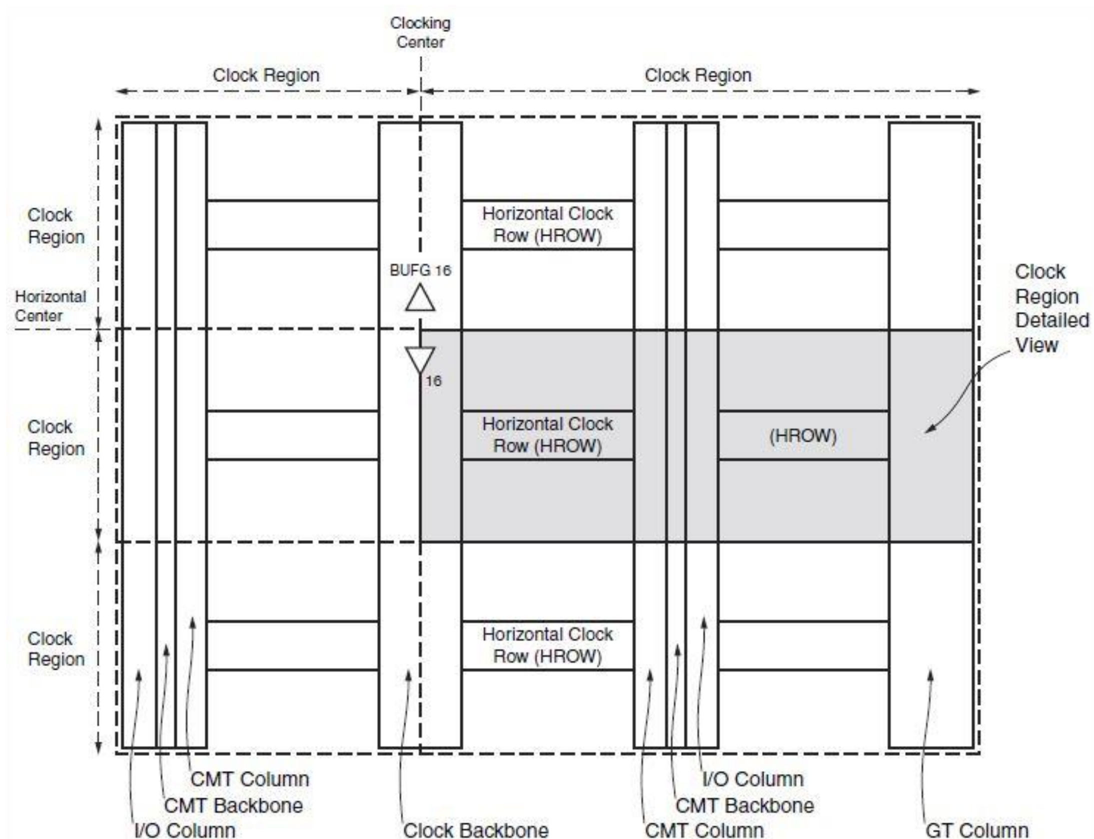


Figura 17. Arquitectura de la distribución de relojes en la familia Xilinx Zynq-7000 [29]

Por otro lado, tenemos las líneas de reloj regionales, que se encargan de la distribución de la señal de reloj en una determinada región, definida comúnmente como un área de 50 E/S y 50 CLBs. Cada dispositivo de la familia de SoCs Zynq-7000 suele contar con entre 4 y 16 regiones, habiendo aproximadamente unas 4 líneas de reloj regionales en cada una de ellas. Finalmente, los relojes de entrada-salida tienen como característica principal que son especialmente rápidos y se emplean exclusivamente para lógica de I/O y para circuitos serializadores/deserializadores (SerDes) de datos.

3.4. Mecanismos de comunicación PS-PL

Una vez analizados los dos bloques de que consta la plataforma, a continuación se hace necesario conocer de qué forma se comunican estos dos sistemas entre ellos para el intercambio de datos.

Se hace preciso establecer una arquitectura de comunicación que realice las transferencias de información adaptando las altas frecuencias de funcionamiento de los *cores* con la de los elementos lógicos, principalmente la FPGA; para cumplir con esta función, se emplean diferentes interfaces del estándar AXI4, que permiten transferir palabras de tamaño variable y con modos de direccionamiento de 32 o de 64 *bits*.

En Zynq, se dispone de dos interfaces AXI maestras y otras dos esclavas de 32 *bits*, además de otros 4 configurables a 32 o a 64 *bits* para una comunicación directa entre el PL y la memoria RAM de tipo DDR y la OCM ubicadas en el bloque de procesamiento, adaptando las distintas velocidades de funcionamiento mediante colas FIFO de 1 KB de capacidad (Figura 18).

Adicionalmente, contamos también con una interfaz esclava de 64 *bits* destinada a realizar accesos coherentes a la memoria caché de la CPU a nivel *hardware*; esta interfaz se conoce como ACP y su función principal es reducir la latencia de la transferencia al valor más bajo posible cuando la FPGA se emplea como acelerador *hardware* para aplicaciones específicas y realiza comunicaciones frecuentes con la APU.

Para finalizar, indicar que el PS dispone de cuatro salidas de reloj hacia el PL con opciones de *start/stop*, así como otras cuatro salidas de *reset* [25].

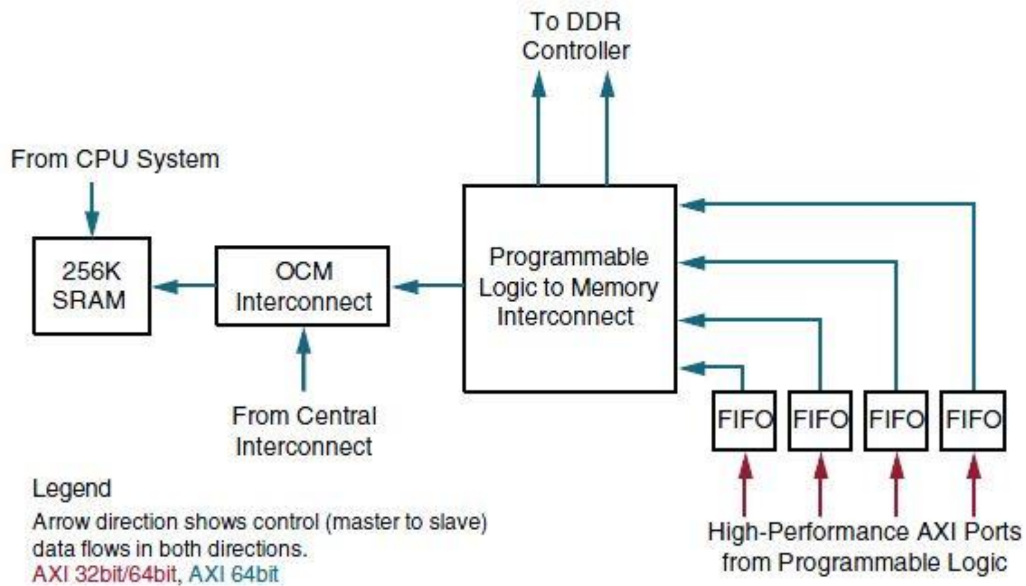


Figura 18. Interfaz AXI para comunicaciones entre el PL y las memorias RAM del PS [25]

3.5. Placa de protipado ZedBoard

Para prototipar el diseño realizado hemos empleado la placa de prototipado ZedBoard (*Zynq Evalutation and Development Board*), que integra un dispositivo XC7Z020 de la familia Zynq-7000 descrito con anterioridad. Además del SoC en sí, proporciona un conjunto de herramientas para interactuar con otros dispositivos externos, así como para realizar una depuración del diseño. Entre los periféricos incluidos encontramos, por citar algunos relevantes, puertos USB 2.0, lector de tarjetas SD para almacenar y cargar desde la misma un sistema operativo en nuestra plataforma, interfaz JTAG para realizar tareas de depuración, salida de video HDMI, OLED y VGA; interfaz para memoria RAM de tecnología DDR3 de hasta 512 MB de capacidad, un bridge de USB a UART para comunicaciones seriales, una serie de LEDs para comprobar el correcto funcionamiento del sistema durante su ejecución o un conjunto de botones para interactuar con la placa, entre otros. En la Figura 19 se muestra el diagrama de bloques de la placa en cuestión:

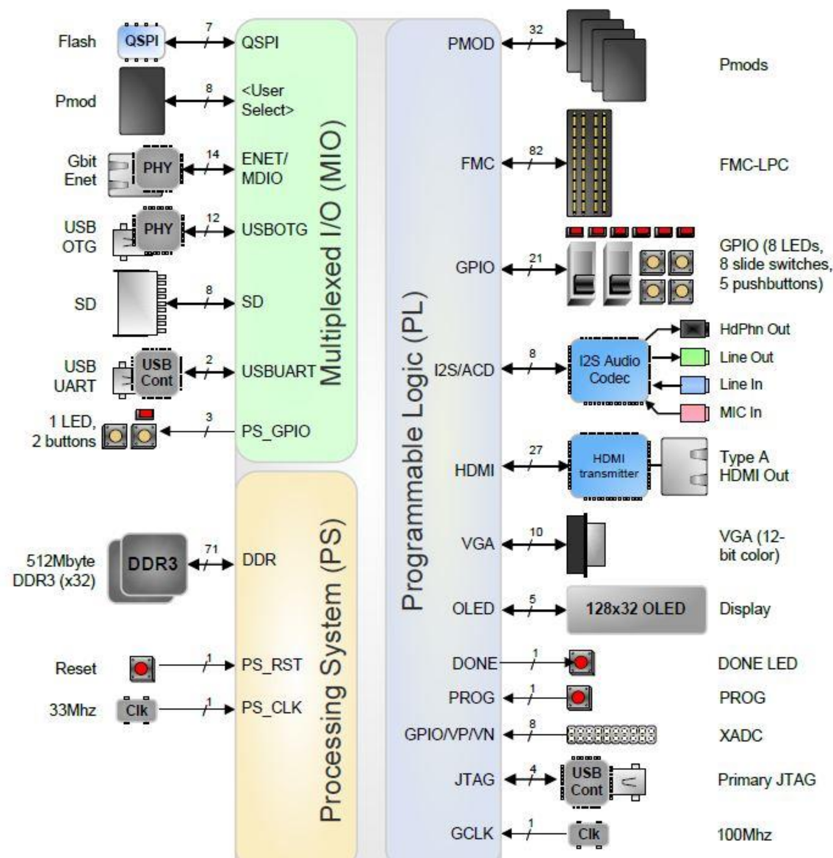


Figura 19. Diagrama de bloques de la ZedBoard [31]

Debido a la aplicación que tenemos como objetivo implementar para nuestro proyecto, cabe destacar el puerto de la interfaz Gigabit Ethernet para comunicaciones a velocidades de 10/100/1000 Mbps empleando el modo RGMII, que permite reducir a la mitad de pines (de 24 a 12) la interfaz entre la MAC y la capa física; otra característica necesaria para nuestra aplicación es que estos controladores se pueden conectar al PL directamente a través de EMIO, permitiendo conexiones directamente con la FPGA. Un segundo conjunto de interfaces son los conectores de expansión FMC, que permiten conectar distintas interfaces E/S directamente al bloque lógico de nuestro SoC (es decir, a la FPGA).

Finalmente, indicar que la placa incluye una serie de *jumpers* y *switches* con los que se puede seleccionar el modo inicial de arranque y configuración del sistema, partiendo desde cualquiera de las interfaces de memoria disponibles o desde el puerto JTAG (configuración por defecto) [25][31].

En la Figura 20 se puede observar la apariencia real de la placa de prototipado, con todos los componentes e interfaces integrados en la misma, destacando aquellos más relevantes:

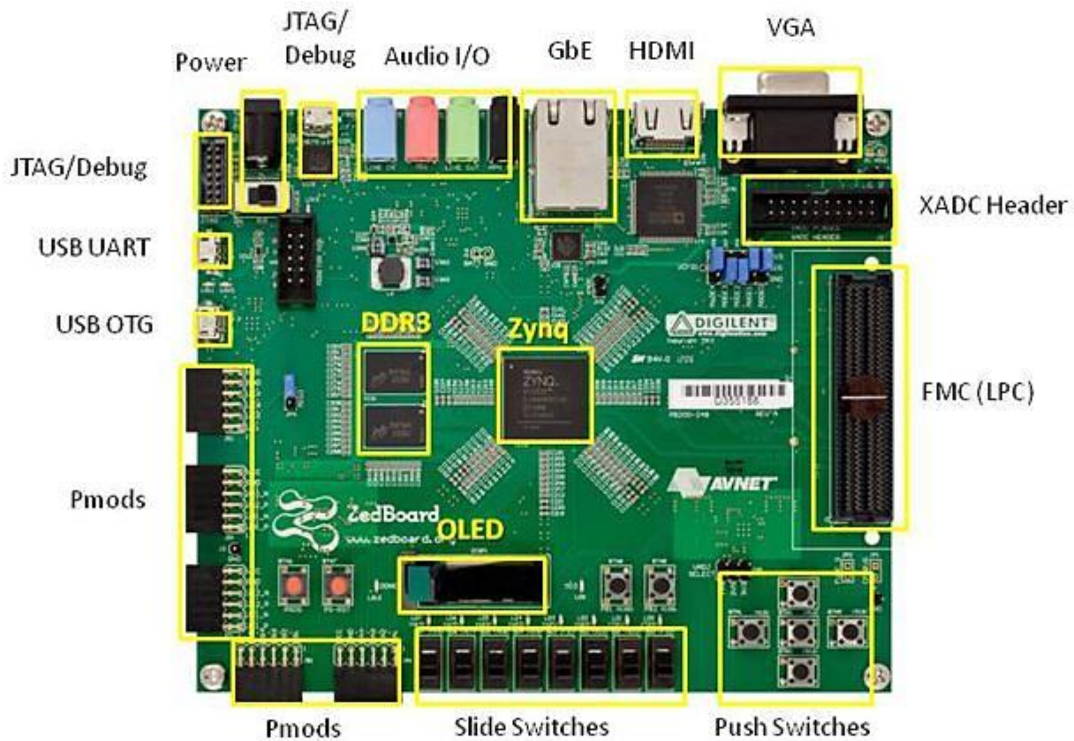


Figura 20. Localización de los principales componentes de la placa ZedBoard [31]

3.6. Conclusiones

En este capítulo se ha analizado la plataforma configurable Xilinx Zynq sobre la que se implementa el sistema final, describiendo sus bloques principales: el sistema de procesamiento (PS) y la lógica programable (PL). Además, se han detallado los mecanismos de comunicación entre ambas partes de la plataforma.

Finalmente, se ha descrito la placa de prototipado ZedBoard, la cual se emplea para la implementación física del sistema.

Capítulo 4. Entorno de diseño Xilinx Vivado

Xilinx Vivado Suite es un entorno de herramientas de diseño que facilita la realización de los diferentes pasos necesarios para implementar un sistema completo en una plataforma *hardware* basada en FPGA de Xilinx a partir de un programa descrito en un lenguaje de alto nivel. Su objetivo es acortar el tiempo de diseño y, por tanto, acelerar la implementación de sistemas con parte lógica programable, proporcionando un entorno en el que se pueden optimizar todos y cada uno de los parámetros clave en función de su aplicación concreta, ya sea el uso de recursos, el consumo de potencia, restricciones temporales y de área, o la planificación de las señales de reloj y de las interfaces E/S. Por tanto abarca desde la partición *hardware/software*, el diseño de la plataforma *hardware,* del *software* empotrado de la plataforma y la integración entre ambas, así como la programación y depuración del sistema completo [32].

En los siguientes apartados se analiza con detalle el flujo de diseño, haciendo hincapié en cada una de sus etapas, así como las funcionalidades que adquieren especial relevancia en la implementación de la aplicación.

4.1. Flujo de diseño e interfaz de usuario

4.1.1. Flujo de diseño de Xilinx Vivado

Vivado Design Suite ofrece la posibilidad de realizar diseños basándose en metodologías de diseño distintas, como pueden ser RTL, TLM, basados en *netlist* o en una planificación de las E/S, por citar algunos ejemplos, variando las herramientas empleadas así como el flujo de diseño en función de la metodología empleada.

En nuestro caso, nos centraremos en el flujo de diseño para SoCs desde una especificación en un lenguaje de alto nivel, donde se integran conjuntamente descripciones RTL y TLM con núcleos IP propiedad de Xilinx o desarrollados por el usuario, pudiendo analizar de forma independiente los resultados proporcionados por cada una de las partes que integran la plataforma, para finalmente generar un *bitstream* que permita programar y depurar el sistema implementado sobre la plataforma *hardware* de prototipado ZedBoard.

En la Figura 21 queda reflejado de forma completa el flujo de diseño de Xilinx Vivado, partiendo de una descripción de la aplicación en un lenguaje de alto nivel como pueden ser C, C++ o SystemC, hasta su implementación final y depuración sobre una plataforma *hardware* física.

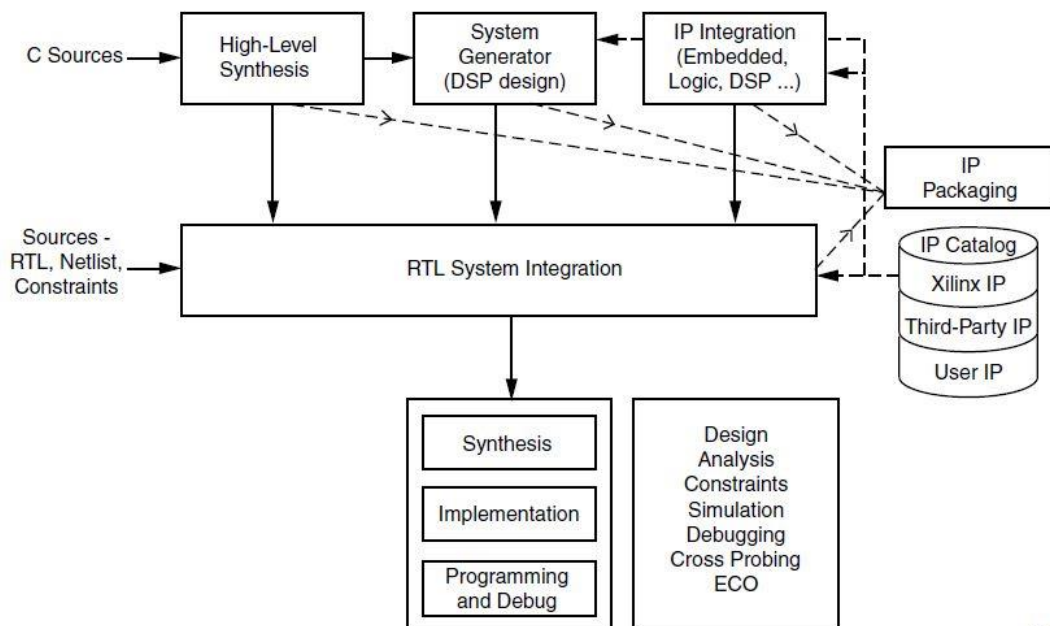


Figura 21. Flujo de diseño partiendo de una especificación de alto nivel en Xilinx Vivado

Con esta metodología, podemos integrar en un mismo diseño la síntesis de un diseño en un lenguaje de alto nivel convertida a una descripción RTL con bloques IP del catálogo de Xilinx o de propiedad intelectual del propio diseñador y descripciones RTL o *netlist* previamente creadas.

Si partimos de una descripción del sistema a alto nivel destinada a la implementación de aceleradores *hardware* como es nuestro caso, se hace necesario el uso de las herramientas HLS de Vivado, que analizaremos al detalle en el siguiente apartado de este documento.

4.1.2. Interfaz de usuario

Vivado permite la interacción con sus herramientas del IDE así como con la configuración del proyecto en una etapa concreta o del proyecto en su conjunto, ya sea empleando la interfaz gráfica o mediante escritura de comandos Tcl mediante su aplicación nativa desde la consola, proporcionando al diseñador total flexibilidad en lo que a metodología de trabajo se refiere. Además, ofrece la posibilidad de simular y depurar cada paso del proceso de diseño de forma independiente, lo que permite identificar con total precisión en qué parte exacta de nuestro sistema ha surgido un problema o la posibilidad de introducir modificaciones en etapas concretas del flujo de diseño.

4.1.2.1. Entorno gráfico

En la Figura 22 se muestra la ventana de trabajo de la interfaz de usuario de Xilinx Vivado. Como se puede observar, a la izquierda se sitúa el *Flow Navigator*, que nos permite acceder a etapas básicas del flujo de diseño para realizar las modificaciones que se consideren oportunas. Entre las opciones predefinidas a las que se pueden acceder se incluyen la configuración general del proyecto, el acceso al catálogo de IPs, la modificación del diseño y la realización de la simulación de dicha etapa, el análisis de la descripción RTL generada de nuestro diseño, así como funcionalidades asociadas a la síntesis e implementación, y a la programación y depurado sobre una plataforma *hardware*.

En el resto de la ventana de trabajo se pueden tener diferentes vistas de diseño del sistema en función de la etapa concreta (configuración y ejecución de la síntesis lógica, integración y conexión de bloques IP en la plataforma, análisis de recursos previo a

la implementación o la edición del código HDL en la etapa de análisis RTL, por citar algunos ejemplos) [33]. Asimismo, la plataforma soporta la realización de los procesos intensivos en cómputo aprovechando la capacidad multiprocesamiento de los recursos *hardware* disponible.

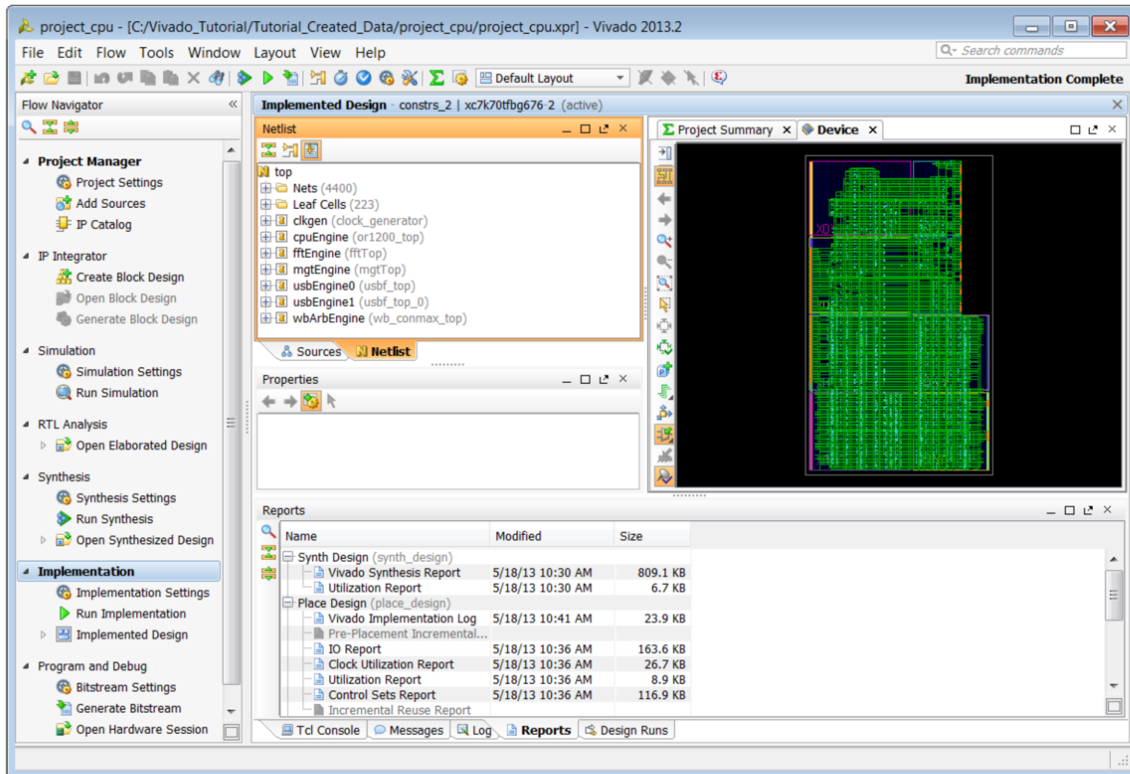


Figura 22. Ventana de trabajo de Xilinx Vivado IDE [34]

4.1.2.2. Comandos Tcl

Xilinx integra Tcl como lenguaje de comandos para controlar los procesos relacionados con su plataforma de diseño.. Tcl permite al diseñador ejecutar acciones que formen parte del flujo de diseño del sistema, así como lanzar *scripts* automatizados que realizan funciones generales necesarias en todo diseño de un sistema electrónico. Además, ofrece la posibilidad de realizar consultas a bases de datos integradas en el *software* empleado acerca del estado y de la configuración del diseño. Como ejemplos de utilización de los comandos Tcl en un diseño electrónico como en el que nos incumbe en nuestro proyecto cabe mencionar la consulta de un análisis temporal del diseño,

aplicación de *constraints* o la ejecución de consultas para verificar que el sistema se ha comportado como cabía esperar sin necesidad de realizar procesos de *re-running*. La estructura general de un comando Tcl en Vivado es la que se muestra a continuación:

```
command [optional_parameters] required_parameters
```

Los comandos se agrupan mediante prefijos según la funcionalidad que desempeñen. Los prefijos más empleados generalmente son los siguientes:

- *get_:* comandos utilizados para obtener algún dato o valor relacionado con el diseño.
- *set_:* se emplean para cambiar el valor de un parámetro específico.
- *report_:* generan informes relacionados con cualquiera de las etapas del flujo de diseño del sistema.

Estos prefijos deben ir acompañados de algún tipo de objeto de los definidos por Xilinx Vivado, sobre el cual se quiere realizar la opción concreta especificada mediante el comando. Los tipos básicos de objetos definidos y que son los que más ampliamente se utilizan son los indicados a continuación:

- *Cell:* hace referencia a cualquier elemento lógico o instancia de nuestro diseño, como pueden ser *flip-flops*, LUTs, *buffers* I/O, *blocks* RAM o DSPs.
- *Pin:* es el punto de conexión lógica con una *cell*; permiten la abstracción de la estructura interna de una *cell* para simplificar y hacer su uso más fácil.
- *Port:* se trata de un tipo especial de *pin*, que representa la entrada o salida de un módulo del sistema; normalmente son conectados a *pads* de I/O para establecer relaciones con dispositivos externos a la FPGA.
- *Net:* conexión o conjunto de ellas que permiten la interconexión de los elementos internos de la FPGA.
- *Clock:* señal periódica que se propaga por toda la lógica secuencial del diseño para gobernar el correcto funcionamiento del sistema y controlar las condiciones temporales del mismo [35].

4.1.3. Diseño basado en plataformas y bloques IPs

El diseño basado en plataformas es una alternativa de diseño que permite su reutilización mediante el uso intensivo de bloques IP y la estandarización de la arquitectura de comunicaciones. Generalmente incluye uno o varios bloques procesadores de propósito general, bloques especializados y periféricos.

En caso de que sea preciso incluir en el sistema bajo diseño diferentes bloques IP, Vivado incluye la posibilidad de insertar módulos en la plataforma desde distintas fuentes, como pueden ser desde el catálogo propio de Xilinx, que contiene módulos ya implementados para facilitar la creación de un diseño por parte del usuario, desde un tercero o creados por el propio diseñador con una finalidad específica para la aplicación que está siendo diseñada, y que le aporta el valor añadido a la plataforma. Estos bloques se integran con el código fuente de alto nivel sintetizado (que puede encapsularse como un bloque IP para su integración en la plataforma), así como con otras descripciones de modelos de simulación y *testbenches* que son convertidas a bloques IPs mediante la herramienta IP Packager, para formar el diseño completo de la plataforma del usuario (Figura 23).

La integración de IPs en la plataforma acelera el proceso de creación del sistema final ya que el catálogo de periféricos estandarizados disponibles es amplio, debiéndose únicamente personalizar los parámetros que permitan configurar el bloque IP para una aplicación en particular. Con ello se consigue disminuir los costes de puesta en el mercado del producto final. Xilinx ha estandarizado AMBA AXI [36] como protocolo de referencia de su librería de IPs. El acceso al catálogo de los bloques IP estandarizados que proporciona Xilinx se realiza a través de la herramienta *Flow Navigator* del Vivado IDE.

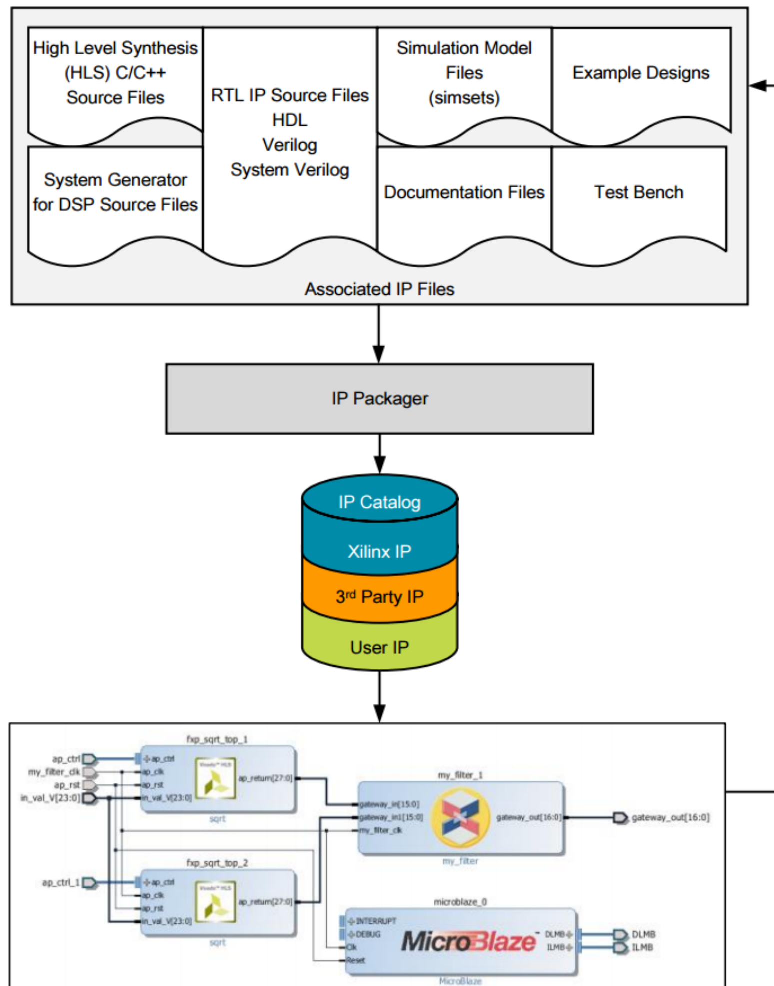


Figura 23. Flujo de diseño IP

Una vez que se disponga de los bloques IP necesarios para nuestra plataforma, estén disponibles en el catálogo que proporciona Xilinx o que hayan sido creados por el diseñador con anterioridad (por ejemplo, desde síntesis de alto nivel), el siguiente paso será realizar su personalización y establecer los canales de comunicación entre ellos para realizar el diseño de nuestro sistema específico completo. Para ello, Xilinx proporciona la herramienta IP Integrator, un entorno basado en una ventana de trabajo con bloques IP y sus interconexiones, el cual permite la configuración a medida tanto de la parte de procesamiento (PS) de nuestro sistema como de la parte lógica (PL) que conforma la FPGA (Figura 24), ya sea desde el entorno GUI o mediante la ejecución de comandos Tcl [37].

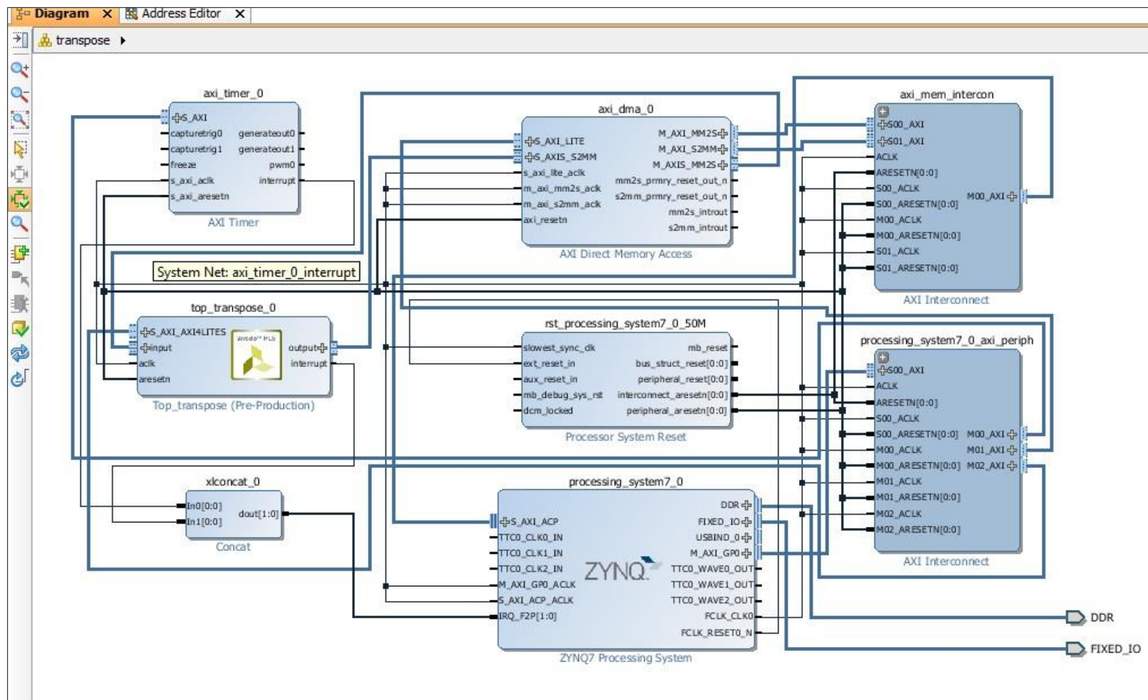


Figura 24. Ejemplo de sistema basado en Zynq en IP Integrator

4.1.4. Uso de restricciones

El empleo de restricciones (*constraints*) permite definir condiciones para la optimización por las herramientas de síntesis e implementación para lograr los objetivos deseados en término de Prestaciones, Potencia y Área (PPA) que nuestro sistema implementado en *hardware* debe cumplir. Se trata por tanto de modelar las especificaciones no funcionales del sistema.

Cada etapa del flujo de diseño tiene sus propias *constraints* en función de las tareas asociadas. Las restricciones deben ser transmitidas a través del flujo de diseño para mantener su coherencia ya que posee una influencia directa sobre las tareas que siguen en las etapas definidas. Este es el caso por ejemplo del ciclo de reloj del diseño. Por ejemplo, si durante la etapa de síntesis definimos una *constraint* que fije un *delay* máximo entre dos componentes de nuestro sistema, este parámetro tendrá una influencia directa sobre la etapa de implementación pues habrá que situar dichos elementos con la proximidad necesaria de tal forma que se consiga contener el retardo dentro del rango definido.

Aparte del ciclo de reloj que es necesario definirlo para todos los diseños síncronos, no existen un conjunto de *constraints* único que debamos usar en todos nuestros diseños, pues cada sistema a desarrollar precisará de unas *constraints* determinadas en función de la aplicación que se desea implementar con nuestro diseño.

Para nuestro en diseño en concreto, cobran especial relevancia dentro de este grupo de *constraints* temporales las que se indican a continuación:

- *create_clock*. Define la señal de reloj, incluyendo los principales parámetros del sistema que gobernará el funcionamiento de todos los elementos síncronos incluidos dentro de nuestro diseño. Esta restricción debe ser definida en primer lugar pues condicionará la correcta ejecución del resto de etapas posteriores del flujo de diseño.
- *set_multicycle_path*. Define el número de ciclos de reloj requeridos para propagar un dato a través de una ruta lógica.
- *set_false_path*. Se utiliza para indicar que una parte en concreto del diseño no debe ser analizada por el analizador temporal debido a que una determinada condición lógica nunca se producirá.
- *set_min/max_delay*. Se emplea para fijar el rango de retardos (máximo y mínimo) que presenta la transmisión de información entre dos elementos lógicos de nuestro sistema.

Las restricciones de E/S permiten configurar los puertos, así como celdas conectadas a ellos. Algunas de las más empleadas son *IOSTANDARD*, que cambia el estándar de E/S con el que está funcionando una o un banco de E/S en concreto. *IOB* le indica a la herramienta de *place* que en lugar de utilizar la salida lógica estándar de cada *slice* intente ubicar previamente a una E/S un *flip-flop*, lo que puede ser útil para establecer etapas de *pipeline* y adaptar las velocidades entre la lógica de una *slice* y una determinada entrada/salida.

Por su parte, las *constraints* de *place & route* se aplican para controlar la ubicación de los componentes lógicos que conforman el sistema en los recursos físicos de la FPGA,

con el objetivo de realizar una distribución lo más óptima posible en función de determinados parámetros, como por ejemplo el retardo de la ruta crítica que determina la frecuencia máxima de funcionamiento del sistema. Algunas restricciones utilizadas son *PBLOCK,* que se aplica para situar un bloque lógico en una región concreta del dispositivo, *PACKAGE_PIN,* para especificar la ubicación de un puerto del diseño en un pin concreto del dispositivo *hardware* lógico; y *LOC,* que permite localizar un elemento lógico en concreto de la *netlist* en un lugar concreto dentro de los recursos del dispositivo [38].

4.2. HLS y herramientas para síntesis de alto nivel

4.2.1. Flujo de diseño HLS

La síntesis de alto nivel, o HLS (*High-Level Synthesis*) por sus siglas en inglés, es una técnica de diseño orientada al desarrollo del *hardware* del sistema desde un nivel de abstracción mayor y, por tanto, mayor facilidad de implementación y menor tiempo requerido para el diseño que si se realizara directamente la especificación a nivel de transferencia de registros o RTL como comúnmente se hacía hasta hace unos años en el ámbito de la electrónica digital.

Esta metodología parte de una especificación algorítmica en un lenguaje de alto nivel, normalmente en C o en alguna de sus variedades (C++ o SystemC), en la que se describe una funcionalidad específica del sistema a nivel de transacciones o TLM, lo que permite separar el comportamiento individual de cada bloque del sistema de las comunicaciones entre ellos (Figura 25).

La metodología TLM declara los medios de comunicación entre dos bloques funcionales o módulos como canales, los cuales son accesibles por los módulos mediante sus propias interfaces, centrándose en la transferencia de datos entre dos componentes del sistema sin tener en cuenta su implementación final, lo que permite experimentar con diferentes arquitecturas de buses para comprobar cuál se adapta mejor a cada situación de intercambio de información dentro de una plataforma específica (la implementación de algoritmos en esta forma de modelado se suele realizar en SystemC).

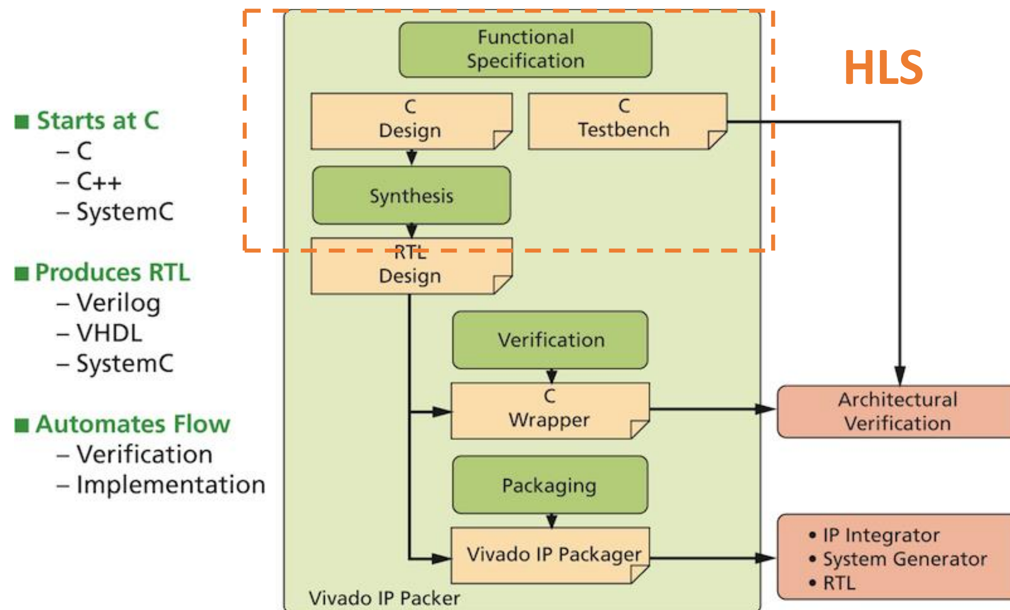


Figura 25. HLS en el flujo de diseño de la FPGA

4.2.2. Vivado HLS

A partir de esta descripción algorítmica se genera, mediante la herramienta de Xilinx Vivado HLS, la correspondiente arquitectura del sistema a nivel RTL, generalmente compuesta de una ruta de datos y una unidad de control, expresada en un lenguaje de descripción *hardware* tal como VHDL o Verilog, o incluso en descripciones mixtas según sea especificado por parte del diseñador en la configuración del proceso de síntesis de alto nivel. Finalmente se encapsula la implementación obtenida como un nuevo bloque IP que pueda ser integrado en la plataforma a través de la herramienta IP Integrator.

Además del código C/C++/SystemC, Vivado HLS precisa que se le proporcionen como entradas, mediante el mecanismo de restricciones, las condiciones de reloj, el tipo de FPGA y directivas para dirigir la herramienta con objeto de obtener la arquitectura deseada y permitir la exploración del espacio de diseño para realizar su optimización bajo alguna de las restricciones de prestaciones, potencia o recursos. Este paso es previo a la generación de la descripción a nivel RTL (Figura 26).

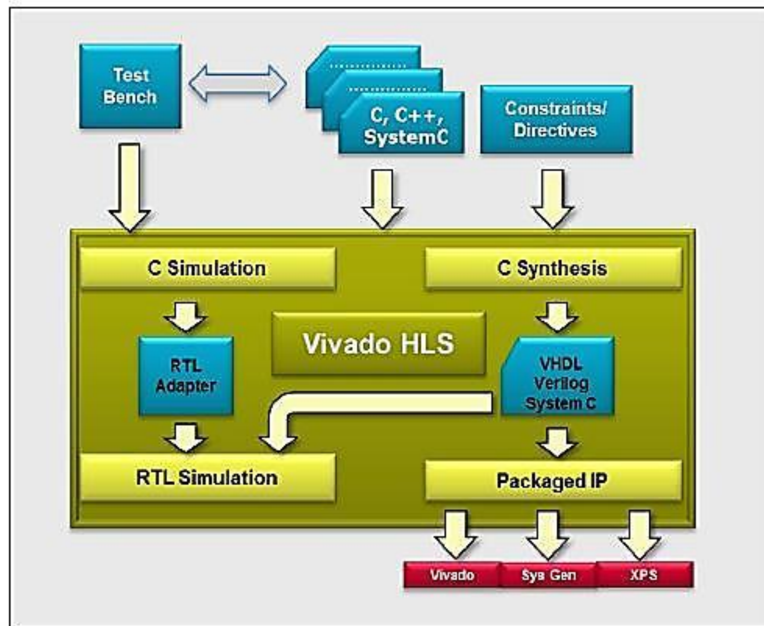


Figura 26. Flujo de diseño de Vivado HLS

Vivado HLS permite, además de sintetizar el algoritmo de alto nivel como una implementación RTL, compilar, simular (incluyendo los correspondientes *testbenches*) y depurar previamente el algoritmo para comprobar que no contiene ningún tipo de error que pueda ser propagado a las siguientes etapas del flujo de diseño. Como ventaja adicional verifica de forma automática la descripción RTL generada, además de proporcionar funciones de análisis de la misma por si el diseñador considera que tiene que realizar alguna modificación final en el código HDL. A pesar de ello, es aconsejable si se decide realizar algún cambio volver atrás en el proceso y hacerlo en el lenguaje de alto nivel para mantener la integridad del diseño.

El entorno de diseño en HLS está basado en el IDE de Eclipse que usa diferentes perspectivas (depurado, síntesis y análisis), tal como se muestra en la Figura 27. Como se aprecia, está formada por una ventana de desarrollo que incluye la barra de navegación del proyecto, la ventana de variables y directivas, la barra de herramientas, el editor de código y la consola. De igual forma, incluye un entorno de depuración al cual hay que proporcionarle un punto de entrada al programa (normalmente, una función *main()*) y permite analizar línea a línea la funcionalidad del código desarrollado, pudiendo observar

el estado de las variables en tiempo real y acceder a partes concretas del programa mediante puntos de ruptura.

Finalmente, la otra opción y la más característica que proporciona la herramienta Vivado HLS es la síntesis de alto nivel que proporciona la descripción RTL generada en el lenguaje *hardware* que se haya seleccionado, así como un informe de resultados en el que se reflejan datos relacionados con parámetros como el rendimiento del algoritmo (PPA) y la latencia [39].

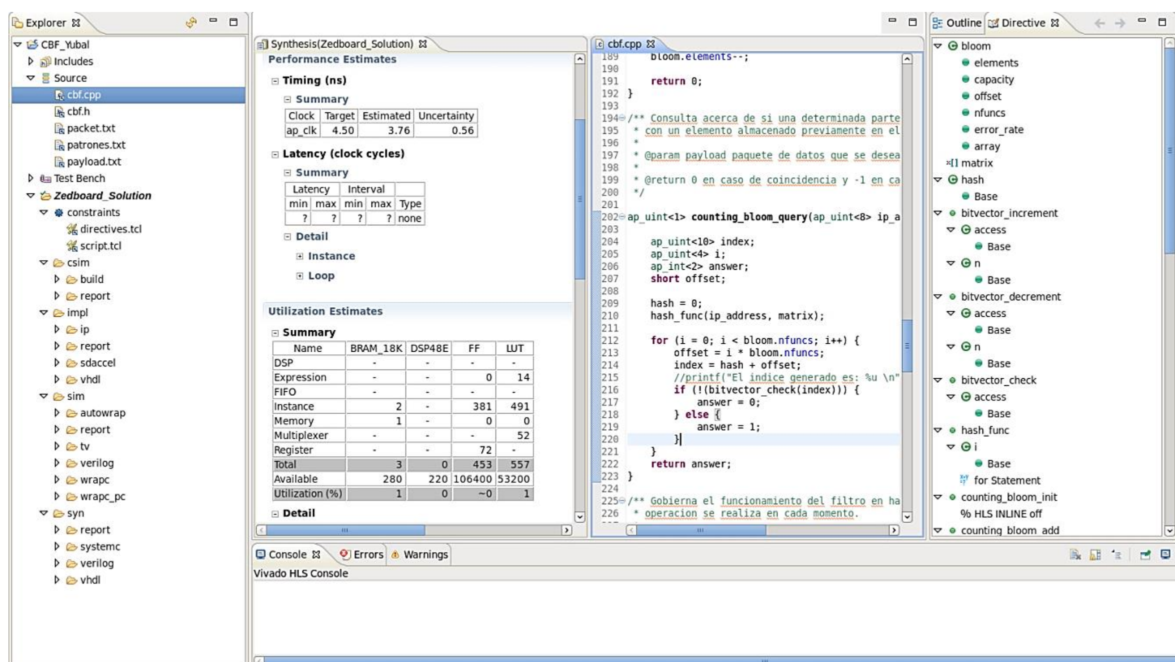


Figura 27. Entorno de diseño Vivado HLS

4.3. Herramientas de síntesis lógica e implementación

4.3.1. Síntesis lógica e implementación en Xilinx Vivado

Además de la síntesis a alto nivel realizada por la herramienta de Xilinx Vivado HLS, durante el flujo de diseño de la FPGA es necesario un segundo paso de síntesis, denominada síntesis lógica, cuya misión es transformar la descripción RTL en lenguaje HDL del sistema a un *netlist* a nivel lógico o de puertas, en el caso de las FPGAs de Xilinx utilizando las primitivas básicas disponibles, listas para seguir hacia el proceso de

implementación. Al final de esta fase se realiza un análisis temporal para verificar el cumplimiento de las restricciones temporales, informando del *slack* disponible y de la ruta donde este *slack* se minimiza.

Por su parte, la etapa de implementación consiste, una vez realizada la síntesis lógica, en seleccionar y distribuir sobre los recursos *hardware* necesarios que posee el dispositivo físico el *netlist* obtenido para cumplir las restricciones de tipo temporal, de rendimiento o de consumo de recursos, principalmente. Generalmente esta fase se compone de las etapas de mapeado, asignación o *placement* e interconexionado o *routing*. Al igual que anteriormente, se realiza un análisis temporal para verificar que se continúan cumpliendo las restricciones temporales del diseño implementado.

Vivado integra en su IDE una herramienta de síntesis e implementación (Figura 28) que permite realizar dichos procesos pudiendo optimizar los parámetros que se consideren claves en cada diseño específico, pudiendo almacenar configuraciones personalizadas en cuanto a optimización de parámetros se refiere para futuros diseños o teniendo la posibilidad de emplear estrategias de optimización ya definidas por Xilinx, las cuales se comentarán en el apartado posterior de este capítulo [39].

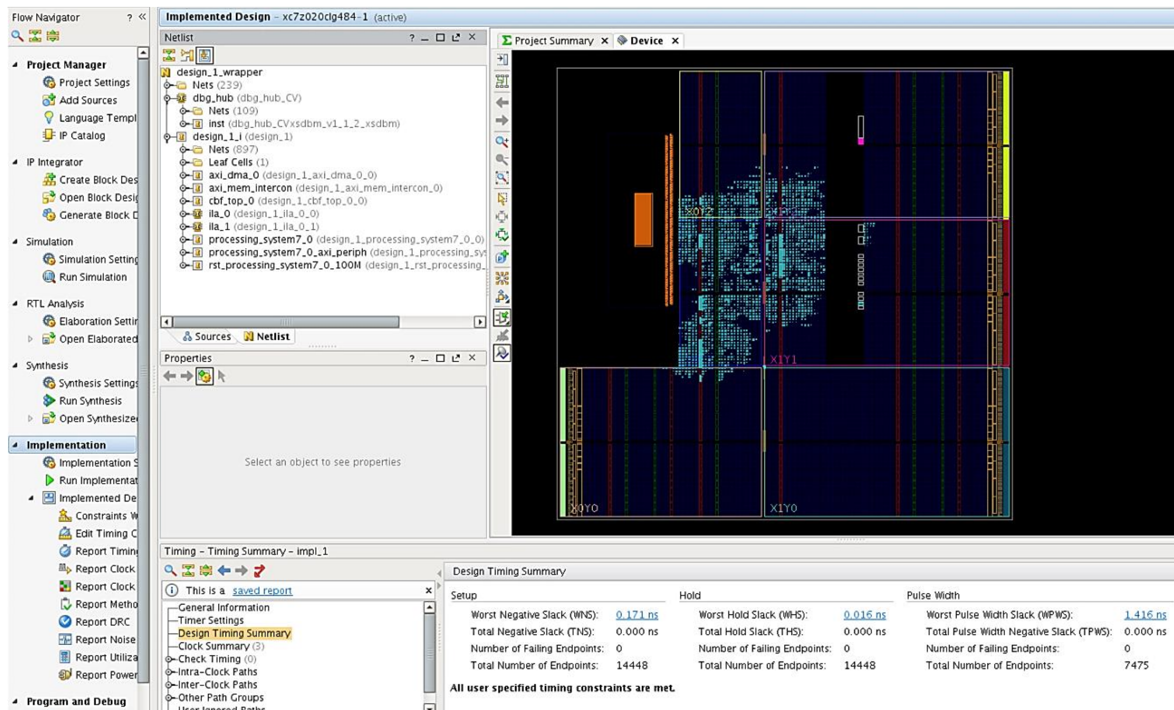


Figura 28. Resultados de implementación en Vivado

4.3.2. Estrategias de optimización

Dependiendo de la aplicación interesará optimizar unos parámetros en concreto durante las fases de síntesis e implementación. En caso de que contemos con más de un parámetro que sea crítico o especialmente relevante optimizar (tiempo, recursos, potencia…) para nuestro proyecto en concreto y como el valor de uno puede condicionar directamente los otros parámetros, se hará necesario llegar a una situación de compromiso en la que se encuentre un punto de equilibrio.

Como se ha indicado, los parámetros que se permiten optimizar desde Vivado son el área (recursos consumidos en la FPGA), el rendimiento o prestaciones, la potencia, el tiempo de ejecución del flujo de diseño y la congestión de la interconexión en la FPGA. En la Tabla 2 se enumeran las estrategias de optimización predefinidas más utilizadas de las que Xilinx Vivado proporciona al diseñador y que están relacionadas con alguno de los parámetros citados con anterioridad [40]:

Tabla 2. Estrategias de optimización proporcionadas por Xilinx

Estrategia	Descripción
Performance_Explore	Optimización mediante diversos algoritmos del proceso de *place & route* para obtener mejores resultados
Performance_ExplorePostRoutePhysOpt	Similar a la anterior, pero permite además la optimización lógica
Performance_RefinePlacement	Incrementa los esfuerzos en la optimización *post-placement.*
Performance_WLBlockPlacement	Prioriza minimizar la longitud de los buses al realizar el *place* de los *block* RAM y de los DSPs.
Performance_WLBlockPlacementFanoutOpt	Ignora las *constraints* temporales para los bloques anteriores y prioriza reducir de forma radical *fanout* altos.
Performance_NetDelay_high/medium/low	Añade un retardo extra a costa de aumentar la longitud de los buses y de elevar el *fanout* de las conexiones.
Performance_ExploreSLLs	Reasigna SLRs con el objetivo de aumentar la holgura temporal.
Performance_Retiming	Combina técnicas de *retiming* con optimización del *placement* a cambio de un retardo en el ruteo
Area_Explore	Emplea múltiples algoritmos para compactar el uso de LUTs
Power_DefaultOpt	Añade optimización de potencia para reducir el consumo del sistema
Flow_RunPhysOpt	Similar a la ejecución por defecto pero activando la optimización a nivel físico
Flow_RunPostRoutePhysOpt	Igual a la anterior salvo en que permite la optimización a nivel físico tanto antes como después del ruteo
Flow_RuntimeOptimized	Cada etapa de la implementación optimiza su tiempo de ejecución
Flow_Quick	Sólo se ejecutan las etapas de *place & route*, con todas las optimizaciones y condiciones temporales desactivadas. Útil para estimación de recursos.
Congestion_SpreadLogic_high/medium/low	Duplica la lógica en el dispositivo para evitar la congestión de regiones determinadas
Congestion_SpreadLogicSLLs	Distribuye las SLLs de forma que se pueda extender la lógica a través de todas las SLRs para evitar que éstas se congestionen
Congestion_BalanceSLLs	Asigna SLLs de tal forma que no existan SLRs que precisen un elevado número de ellas, reduciendo de esta forma la congestión en las SLRs.
Congestion_BalanceSLRs	Divide cada SLR en dos de similar área, para evitar la congestión en las SLRs.
Congestion_CompressSLRs	Parte solo las SLRs con mayor utilización con el objetivo de reducir SLLs.

4.3.3. Monitorización y visualización de recursos

Tanto Vivado HLS como Vivado IDE proporcionan las opciones de monitorizar el proceso de síntesis. Si estas etapas se realizan satisfactoriamente, Vivado lanza al finalizar

una ventana de resultados en la que el diseñador puede analizar si las prestaciones de su diseño se encuentran dentro del rango deseado una vez el sistema *hardware* ha sido implementado en la plataforma física, así como ventanas de consumo de recursos donde se pueden ver los componentes de la FPGA consumidos para implementar la aplicación desarrollada, pudiendo visualizarse de forma esquemática o en forma de *layout* (Figura 29) [40] [41].

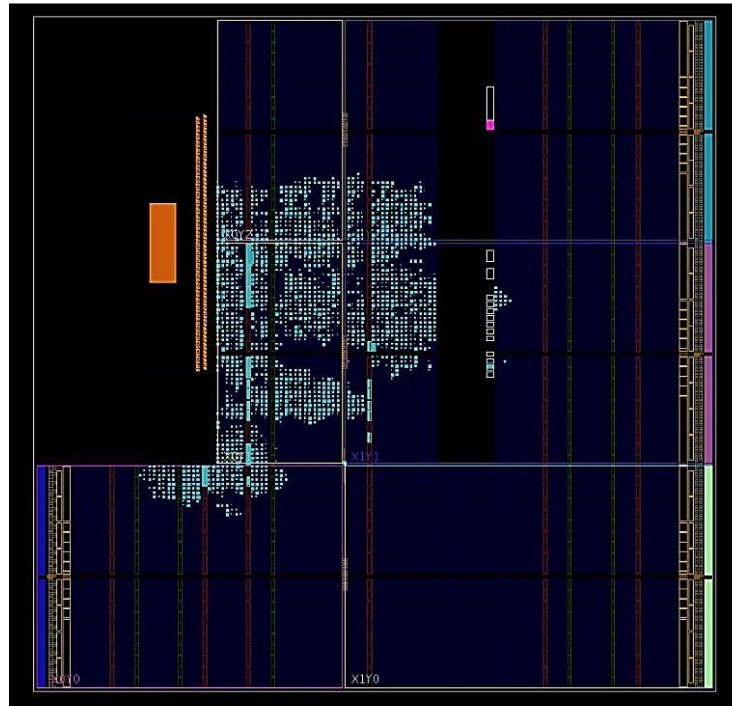


Figura 29. *Layout* de consumo de recursos de la FPGA

4.4. Prototipado hardware y desarrollo software

4.4.1. Opciones de depurado hardware

Para depurar la plataforma *hardware* con el objetivo de comprobar su correcto funcionamiento, Xilinx proporciona dos herramientas: el Virtual I/O y los ILA. Ambos mecanismos son proporcionados por Xilinx en forma de bloques IP para ser conectados en nuestra plataforma en aquel punto del sistema que desee monitorizar [42].

En este caso, la opción de depurado que vamos a emplear para visualizar el estado de las señales internas del diseño es el LogiCORE IP *Integrated Logic Analyzer*, comúnmente conocido como ILA.

Se trata de un entorno visual que permite monitorizar una interfaz de comunicación del diseño como si de un analizador lógico físico se tratara, pudiendo observar el estado de determinadas señales en un periodo de tiempo determinado. Como todo analizador lógico, incluye opciones de *trigger* y determinadas condiciones de disparos en flancos concretos de la señal de reloj, así como el análisis y visualización de hasta un total de 1024 señales por ILA, pudiendo especificar para cada una de ellas la muestra a partir de la cual se empieza a capturar la señal y durante cuántos ciclos de reloj a partir de la misma.

Al tratarse de un elemento síncrono, es necesario aplicarle las restricciones de reloj concretas que le permitan trabajar a la misma frecuencia que nuestro diseño. La comunicación con este módulo desde la placa de prototipado se realiza mediante la interfaz de comunicación JTAG[43].

Este módulo se comunica con la placa de prototipado a través de la herramienta Hardware Manager de Vivado, para lo cual habrá que seleccionar previamente el dispositivo *hardware* sobre el cual se va a descargar nuestro archivo de programación (*bitstream*). En este entorno se lanza una ventana en la que se muestran los ILA que se encuentran en ejecución, las condiciones de *trigger* y de visionado de la ventana de forma de onda establecidas, así como la propia ventana de representación de las formas de onda de las señales que se consideren relevantes por el diseñador para comprobar el correcto funcionamiento del sistema sobre la plataforma *hardware* final (Figura 30) [42].

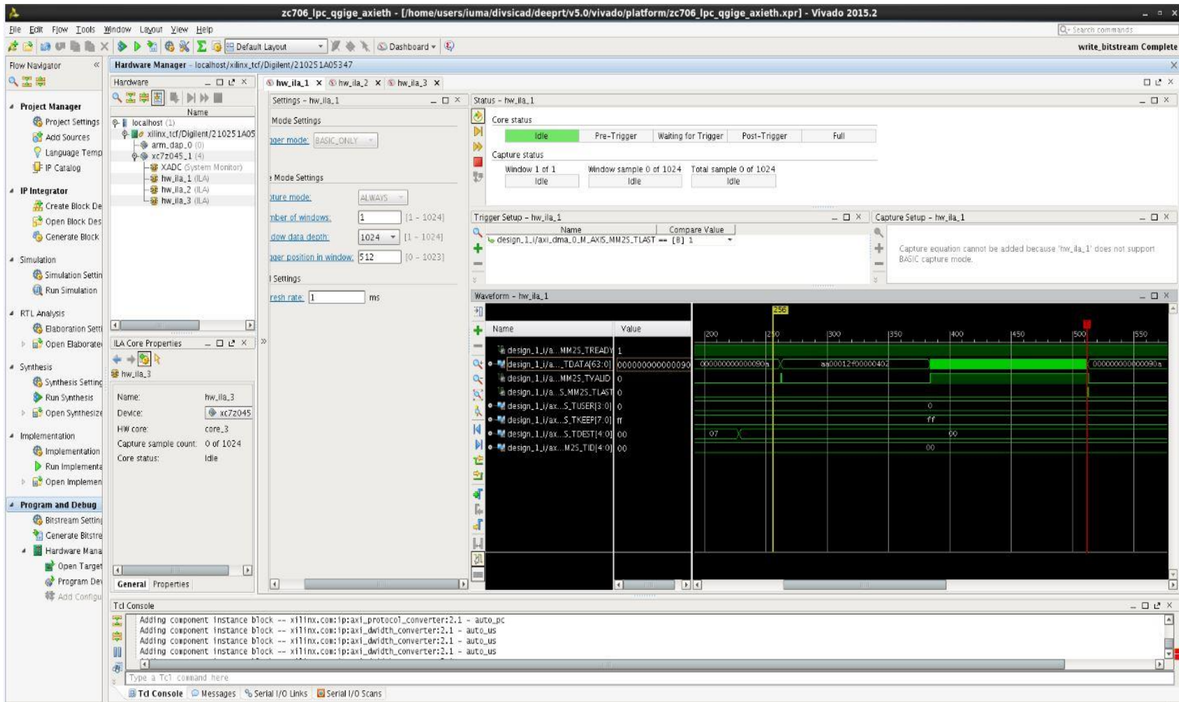


Figura 30. Ventana de trabajo del Hardware Manager

4.4.2. Generación del bitstream

Una vez que tenemos nuestra plataforma correctamente funcionando tras haber sido sintetizada, implementada y habiendo comprobado que cumple con la aplicación que se había fijado como objetivo, el siguiente paso será generar y descargar el archivo *bitstream* en la memoria interna de la FPGA con toda la configuración relevante referente a la plataforma diseñada (módulos y conexiones principalmente). La opción de generar este archivo se encuentra incluida en el *Flow Navigator* de Xilinx IDE[42], tal como se muestra en la Figura 31 [42]:

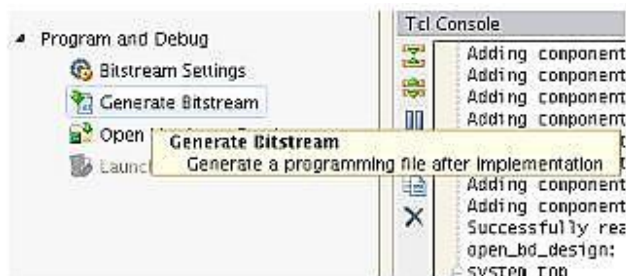


Figura 31. Opción de generar el *bitstream* en Xilinx IDE

4.4.3. Xilinx SDK

Tras haberse realizado de forma satisfactoria la generación del *bitstream*, es preciso exportar el *hardware* al entorno Xilinx SDK para el desarrollo de la aplicación *software* que se integrará en la plataforma para gobernar su funcionamiento, consiguiendo de esta forma el diseño de una aplicación a medida para el *hardware* diseñado. Para ello, todavía en Xilinx Vivado, seleccionamos la opción *File > Export > Export Hardware* y, una vez realizada esta acción, hacemos clic en la opción *Launch SDK* para lanzar el entorno de desarrollo del *software*, tal como muestra la Figura 32.

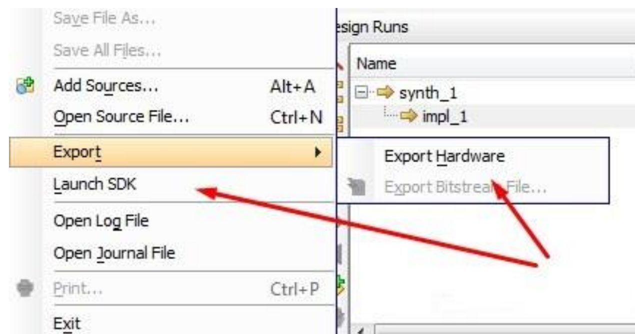


Figura 32. Exportación del *hardware* y lanzamiento posterior de Xilinx SDK

Para el desarrollo de aplicaciones software destinadas a ser implementadas en sistemas empotrados de Xilinx que incluyan *cores*, ya sean implementados de forma física en el dispositivo (PowerPC o ARM) o implementados mediante programación (*softcores*) como el MicroBlaze o el PicoBlaze, Xilinx proporciona su propio SDK, enfocado especialmente a plataformas diseñadas sobre dispositivos de la propia empresa. Este entorno incluye herramientas estándar en este tipo de programas basadas en GNU (compilador GCC, depurador GDB y un conjunto de utilidades y librerías), un depurador JTAG y un programador de memorias *flash* para realizar la descarga y comprobación del código desarrollado sobre la placa de prototipado, *drivers* para IPs de Xilinx y un editor de código para C/C++ basado en Eclipse.

Este SDK puede partir de una plataforma *hardware* previamente creada con otras herramientas de Xilinx Vivado para diseñar un *software* específico para la misma, como es nuestro caso. Para ello, es necesario proporcionarle todos los detalles asociados a los

módulos que componen dicha plataforma, así como las librerías pertinentes, creando un Board Support Package (BSP) que incluye toda esta información. Asimismo, proporciona la posibilidad de desarrollar un *bootloader* que se descargará en la plataforma física de prototipado y que permitirá arrancar la aplicación software desarrollada. El archivo que se genera se denomina Fisrt Stage Boot Loader (FSBL) [44].

4.5. Conclusiones

Durante este apartado se ha abordado el flujo de diseño de un sistema electrónico empleando las herramientas de diseño de Xilinx Vivado, explicando cada una de las etapas que lo componen, el *software* empleado para completarlas y las opciones que se pueden aplicar en cada una de ellas. Posteriormente, se ha hecho hincapié en la metodología de síntesis de alto nivel empleada, ya que resulta una novedad reciente en el desarrollo de sistemas que incluyen FPGA.

Capítulo 5. Descripción de la solución propuesta

En este capítulo se describe la implementación realizada para diseñar el sistema completo. Se trata de un sistema empotrado compuesto por una plataforma configurable (procesadores ARM y FPGA) y un *software* empotrado diseñado para cumplir con la funcionalidad de dicha aplicación.

La solución planteada debe ser lo más simple posible con el objetivo de poder integrar el bloque IP resultante en diferentes plataformas, facilitando de esta forma su reutilización para diferentes proyectos.

5.1. Descripción del diseño

Tras haber hecho un estudio de los algoritmos analizando sus pros y contras, se ha decidido realizar la implementación *hardware* empleando filtros Bloom. Esta estructura permite la búsqueda de expresiones en un tiempo de ejecución aceptable, ocupando unos recursos mínimos. La configuración del sistema es posible realizarla desde la aplicación *software* ejecutándose en el *core* ARM definiendo, por ejemplo, la introducción de una nueva dirección IP a buscar, creando la correspondiente lista negra.

Para el sistema se va a implementar la variación del filtro Bloom con contador para incluir la posibilidad de eliminar direcciones del filtro cuando se considere oportuno (por ejemplo, debido a que se trata de una dirección obsoleta o una amenaza que ya ha sido gestionada en la red), así como un parámetro que permita establecer la tasa de falsos negativos que se crea asumible para esta aplicación.

La búsqueda de la dirección IP origen se realiza analizando directamente el campo correspondiente de los paquetes Ethernet entrantes a través de la red. Tal como se observa en la Figura 33, esta información viene proporcionada en el rango de bytes comprendido entre el 26 y el 29 (en formato hexadecimal) lo que corresponde, si se hace la comparativa respecto a palabras de tipo entero de 32 bits, a la mitad menos significativa de la palabra de 32 bits número 6 y a la mitad más significativa de la palabra

entera número 7 (esta información resulta clave durante la programación del método de consulta de nuestro algoritmo) [45].

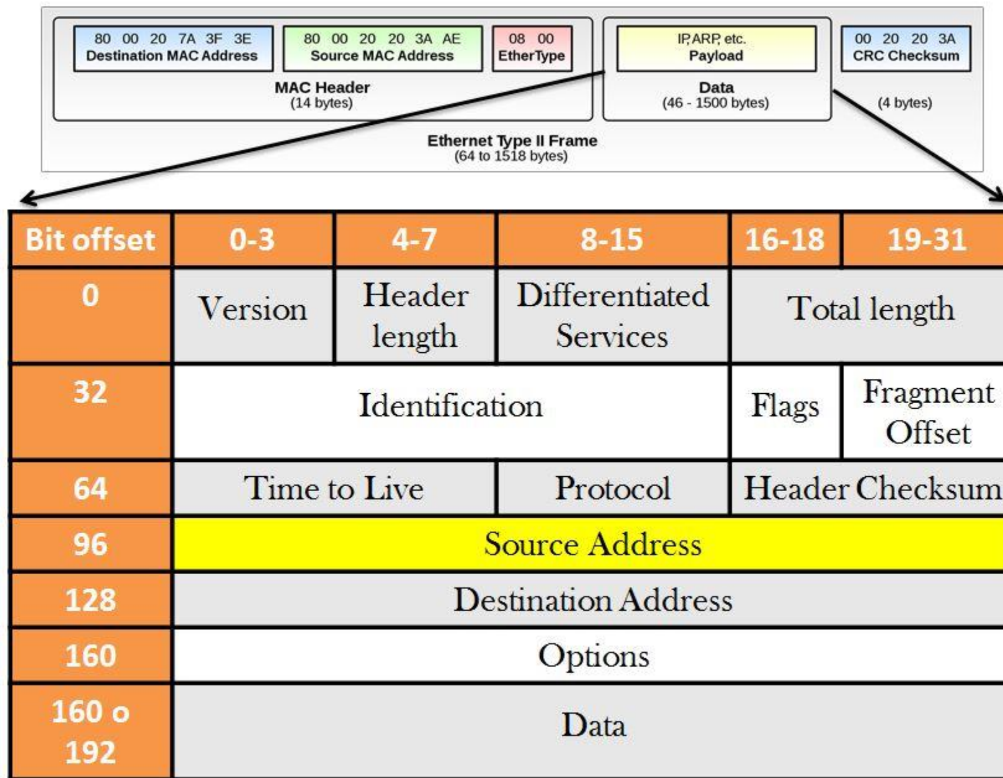


Figura 33. Localización de la dirección IP origen en un frame Ethernet

5.2. Arquitectura y estructura sobre la plataforma configurable

La arquitectura del acelerador *hardware* que se propone en este trabajo para ser implementado sobre un SoC programable usando una FPGA se muestra en la Figura 34.

Tal como se aprecia en la Figura 34, las etapas básicas de las que consta su flujo de datos son la recepción de paquetes procedentes de la red, la configuración del filtro, el análisis de la dirección IP origen y la respuesta dada al sistema para que tome las decisiones que se consideren oportunas.

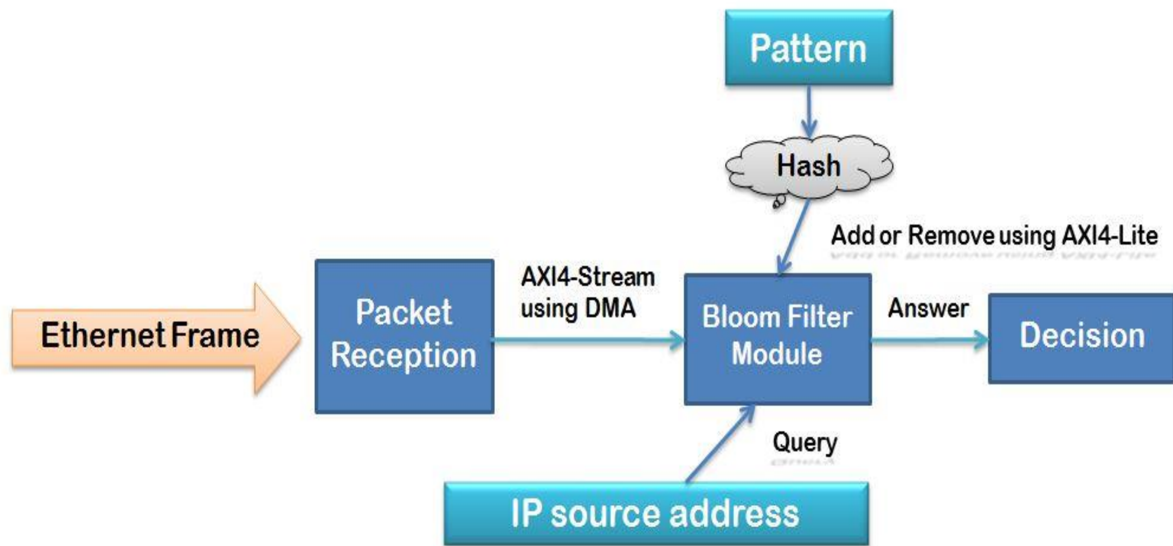


Figura 34. Arquitectura del sistema empotrado

Este diseño presenta una ventaja clave que es la robustez del sistema, al estar implementado sobre un dispositivo dedicado (en este caso, un SoC integrado en una plataforma *hardware* de prototipado). El sistema necesita ser configurado de forma previa a su instalación en su entorno de trabajo final, pudiendo ser actualizado mediante *firmware* que será descargado en la memoria del dispositivo mediante la interfaz JTAG o puede ser cargado mediante una tarjeta SD. Sin embargo, presenta el inconveniente de que los recursos (de procesamiento y almacenamiento) son reducidos en comparación con otras posibles alternativas a implementar sobre *workstations* o *boards* de mayor rendimiento (y por tanto, que acarrean un coste mayor).

Cada etapa de las indicadas en la Figura 34 incluye la correspondiente función *software* encargada de realizar una tarea concreta dentro del procedimiento de análisis global. Empleando este método de programación modular se obtienen programas más simples y de menor complejidad a la hora de detectar errores y ser depurado. Las funciones de escucha de la red, de envío de paquetes hacia el filtro Bloom (a través de un DMA) para su análisis y la toma de decisiones en función de la respuesta dada serán gobernadas mediante el *software* empotrado que se habrá diseñado exclusivamente para esta aplicación. Por otro lado, se implementará en *hardware* la funcionalidad completa

del filtro Bloom con contador que se encargará de realizar las tareas asociadas a las consultas.

El proceso de recepción y captura de los paquetes de red se lleva a cabo empleando una librería *software* proporcionada por Xilinx denominada *lwIP*, que será comentada con posterioridad en el apartado 7.3.3. [46]. Esta librería incluye las funciones necesarias para la recepción de paquetes para la mayoría de protocolos existentes en la red, lo que nos permite centrar todos los esfuerzos de desarrollo del *software* empotrado en gestionar las operaciones atribuidas al filtro en *hardware* (inserción y eliminación de direcciones IP en la lista negra, así como la realización de consultas). Una vez el paquete se encuentra almacenado en el correspondiente *buffer* a la espera de ser transmitido a través de un DMA hacia el bloque IP desarrollado, es preciso esperar a que la infraestructura de comunicación del SoC se encuentre libre para enviar la información y que el bloque IP realice una consulta sobre la dirección IP origen del paquete de red enviado, accediendo al campo de datos correspondiente. A partir de este resultado, y de si dicha dirección IP está incluida en nuestra lista negra o no, el bloque de procesamiento tomará la decisión oportuna (bloqueo o filtrado de los paquetes asociados, respectivamente). Mediante estas consultas lo que se pretende es que no accedan al resto de bloques de la plataforma los paquetes de datos procedentes de aplicaciones origen que se consideren amenazas para el correcto funcionamiento del sistema.

5.3. Arquitectura del software empotrado

En este apartado se describe el proceso de programación que se considera apropiado para cumplir con la estructura de la solución definida en el apartado anterior.

El sistema comienza con la recepción, a través del controlador Ethernet, de los paquetes entrantes procedentes de la red, los cuales se almacenan para su posterior tratamiento. Mediante un bloque DMA, se envían estos paquetes de datos al acelerador conformado por el filtro Bloom, el cual accede directamente al campo en el que se encuentra la información acerca de la dirección IP del *host* que pretende realizar

conexiones con nuestro dispositivo. Finalmente, se notificará al bloque de procesamiento del resultado de la consulta realizada por el filtro, a partir de la cual tomará las decisiones que estime oportunas.

En la Figura 35 se muestra el diagrama de flujo del algoritmo a implementar para cumplir con la estructura de la aplicación anteriormente descrita. Tal como se observa en la misma, es necesaria la realización de diferentes procesos para conformar la aplicación completa de análisis.

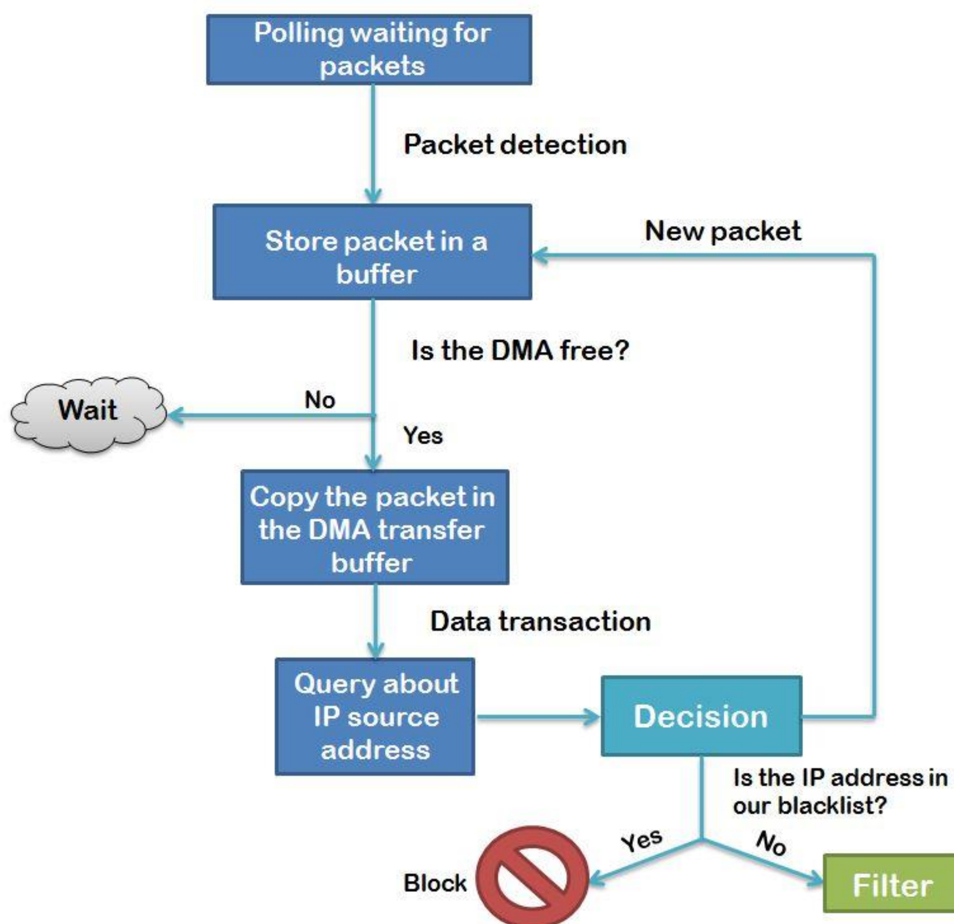


Figura 35. Flujo de diseño del programa

5.4. Diseño de la plataforma

Para el diseño del bloque IP acelerador que incluye la funcionalidad del filtro Bloom con contador, se sigue la tendencia actual de emplear una metodología de diseño basada en síntesis de alto nivel, la cual parte de una descripción algorítmica en lenguaje C/C++ para, posteriormente, obtener mediante la metodología indicada (en nuestro caso, empleando la herramienta Vivado HLS) la descripción RTL en un lenguaje de descripción del hardware (HDL) del bloque, con el objetivo de integrarlo en la plataforma y conformar el sistema final que cumpla con toda la funcionalidad especificada.

El resto de bloques que conforman la plataforma están disponibles en la biblioteca de bloques IP que facilita Xilinx, con el objetivo de facilitar el diseño mediante la reutilización de este tipo de módulos estándar.

La integración del bloque IP desarrollado con el resto de la plataforma *hardware* se aprovecha de la utilización de buses que emplean distintas interfaces del estándar AXI, permitiendo comunicaciones entre distintas partes del sistema a diferente velocidad, en función de la información a transmitir.

5.5. Conclusiones

En este capítulo se ha explicado la solución que se ha decidido implementar para cumplir con la aplicación deseada. Tal como se ha expuesto, se implementará un NGFW en *hardware* basado en filtros Bloom con contador por tratarse de un algoritmo bastante eficiente, tal como se mencionó en el apartado 2.3.2. Dicha solución se implementará sobre una plataforma Xilinx Zynq y empleando distintas interfaces de comunicación AMBA AXI en función de las características de la información a transmitir. Adicionalmente, se ha explicado la arquitectura del diseño tanto en *hardware* como en *software* y la estructura de un *frame* Ethernet, con el objetivo de definir el campo concreto al que se debe acceder para la detección de la dirección IP origen.

Capítulo 6. Capítulo 6. Diseño del bloque IP

6.1. Introducción

Tras hacer hincapié en el flujo de diseño a tener en cuenta a la hora de desarrollar el sistema utilizando una metodología de alto nivel y describir la tecnología de referencia, en este capítulo pasamos a explicar el diseño del bloque IP.

6.2. Diseño hardware del filtro

Como se ha hecho mención con anterioridad, la misión del bloque IP a desarrollar será la detección de la dirección IP origen de los paquetes recibidos a través de la red mediante filtros Bloom con contador, con el fin de poder filtrar aquellas aplicaciones con las que se decida trabajar en el sistema y desestimar las solicitudes del resto, bien por considerarse que puedan resultar una amenaza para el correcto funcionamiento de nuestra plataforma o simplemente porque se consideran no relevantes en cuanto al análisis de su contenido, optimizando de esta forma la ejecución del sistema en cuanto a cómputo y accesos de memoria realizados.

El diseño del bloque IP, tal como ya ha sido mencionado, se realiza siguiendo la metodología de diseño basada en síntesis de alto nivel, para lo que se emplea la herramienta Xilinx Vivado HLS.

Este bloque se ha diseñado siguiendo un modo incremental, realizando versiones básicas del algoritmo que permitan realizar realimentaciones rápidas y periódicas en el tiempo de desarrollo para poder revisar mediante casos de test la implementación realizada y añadir funcionalidad de forma iterativa. El modelo en alto nivel se realizará empleando los lenguajes de programación C y C++, a partir del cual se obtendrá la correspondiente descripción RTL aplicando la síntesis de alto nivel.

6.2.1. Implementación

El algoritmo constará esencialmente de dos funcionalidades bien diferenciadas. En primer lugar se dispone de la funcionalidad asociada al filtro Bloom, como son la inserción y eliminación de elementos en la tabla de referencias, así como la consulta de si una determinada dirección IP se encuentra dentro del conjunto (*pattern matching*). En segundo lugar, encontramos los métodos asociados al manejo de los contadores vinculados a cada posición del vector del filtro, estando cada una de sus funciones asociadas a un método del filtro (incremento, decremento o comprobación de una determinada posición del vector).

Los métodos de inserción y de eliminación de un elemento del filtro siguen una estructura de programación bastante similar. En primer lugar, le pasan por parámetros al método encargado de ejecutar la función *hash* el elemento sobre el que se va a actuar para, posteriormente, generar tantos índices a partir del mismo como funciones *hash* sea necesario ejecutar en función de la capacidad y de la tasa de error asumible que se definen al inicializar el filtro. El resultado de la función *hash* es almacenado en una variable global para que sea accesible por todos los métodos del filtro. Finalmente, se incrementa o decrementa (en función de la operación aplicada) el contador en las posiciones del vector que han sido calculadas para el elemento en cuestión, evaluando posibles casos de error por desbordamiento del contador (al ser de 4 bits, no permite mapear más de 15 elementos en una misma posición del vector) o por haberle pasado un índice que se encuentra fuera del tamaño del vector.

```
ap_uint<1> counting_bloom_add(ap_uint<8> pattern[WORD_SIZE]) {

	ap_uint<10> index;
	ap_uint<4> i;
	short offset;

	hash = 0;
	hash_func(pattern, matrix);

	for (i = 0; i < bloom.nfuncs; i++) {
		offset = i * bloom.nfuncs;
		index = hash + offset;
		//printf("El indice generado es: %u \n", index.VAL);
		bitvector_increment(index);
	}
```

```
        bloom.elements++;
        return 0;}
ap_int<1> bitvector_increment(ap_uint<10> index) {

        ap_uint<8> access = index / 2;
        if (access >= sizeof(bloom.array)) {
        printf("Error: indice a incrementar fuera de la dimension del vector\n");
                return -1;
        }
        ap_uint<5> n = bloom.array[access];

        if (n == 0x0f) {
                printf("Error, desbordamiento del contador de 4 bits\n");
                return -1;
        }

        n = n + 1;

        bloom.array[access] = n;
        return 0;
}
```

El método de consulta, por su parte, difiere ligeramente de los dos anteriormente citados. En su caso, la palabra sobre la cual se va a ejecutar la función *hash* se obtiene accediendo al campo concreto del paquete de red entrante en el que sabemos que se encuentra la dirección IP origen a través de la función *range()*; este segmento del paquete de red corresponde a la posición en la que se ubica la dirección IP origen, la cual es fija y previamente conocida. Finalmente, tras aplicar la función *hash* tantas veces como sea necesario, se comprueba que en las posiciones resultantes los contadores asociados reflejan que en dicha posición se encuentra mapeado un elemento; si todos los contadores poseen un valor mayor a 0, se devuelve una respuesta afirmativa al sistema y, por el contrario, en caso de que algún índice tenga un valor de contador asociado igual a 0, se devolverá una respuesta negativa.

```
ap_uint<1> counting_bloom_query(ap_uint<32> * payload) {

        ap_uint<10> index;
        ap_uint<4> i;
        ap_int<2> answer;
        short offset;
        ap_uint<8> key[WORD_SIZE];

        key[3] = payload[6].range(15, 8);
        key[2] = payload[6].range(7, 0);
        key[1] = payload[7].range(31, 24);
        key[0] = payload[7].range(23, 16);

        hash = 0;
        hash_func(key, matrix);
```

```
	for (i = 0; i < bloom.nfuncs; i++) {
		offset = i * bloom.nfuncs;
		index = hash + offset;
		//printf("El indice generado es: %u \n", index.VAL);
		if (!(bitvector_check(index))) {
			answer = 0;
		} else {
			answer = 1;
		}
	}
	return answer;
}
```

```
ap_int<1> bitvector_check(ap_uint<10> index) {

	ap_uint<8> access = index / 2;
	if (access >= sizeof(bloom.array)) {
	printf("Error: indice a comprobar fuera de la dimension del vector\n");
		return -1;
	}
	if (bloom.array[access] != 0) {
		return 0;
	} else {
		return 1;
	}
}
```

En una primera versión, se optó por emplear una función *hash* de la familia MurmurHash (en concreto su versión 3, la más reciente), una función *hash* no criptográfica que proporciona un resultado de 32 o 128 bits, rápida y eficiente para la obtención y codificación de índices aleatorios, la cual se basa en el empleo únicamente de operaciones de rotación y multiplicación; sin embargo, esta función resultó ser demasiado lenta en su ejecución y consumía numerosos recursos lógicos, lo que llevó a descartar esta opción.

En su lugar, se empleó una función de la clase de funciones *hash* universales H3, idónea para una implementación *hardware,* como la que nos incumbe. Debido a que se basa únicamente en operaciones AND bit a bit (y su posterior acumulación mediante operaciones OR exclusiva o XOR) entre el patrón objeto de la operación y una matriz previamente definida como variable global de su misma dimensión, se trata de una alternativa bastante eficiente en cuanto a tiempo de ejecución y consumo de recursos se refiere.

```
void hash_func(ap_uint<8> pattern[WORD_SIZE], ap_uint<8> matrix[MATRIX_SIZE]) {

       ap_uint<3> i;

       for (i = 0; i < WORD_SIZE; i++) {
              hash ^= (pattern[i] & matrix[i]);
       }
       printf("El valor del hash es: %u \n", hash.VAL);
}
```

6.2.2. Test, depurado y verificación

Con el fin de comprobar el correcto funcionamiento del algoritmo implementado, se hacía necesario el diseño de un *testbench* que permitiera verificar las distintas situaciones que podrían darse al ejecutar el programa con el fin de evitar arrastrar errores, que serán más difíciles de detectar y paliar en etapas posteriores del flujo de diseño.

En este caso, el *testbench* diseñado incluye operaciones de lectura desde un archivo de texto previamente editado de una serie de direcciones IP en formato decimal que se incluyen como elementos al filtro, algunas de los cuales posteriormente se eliminan antes de realizar, mediante otro fichero de texto que simula la dimensión y formato de paquetes de red reales (y en los cuales se sitúa en la ubicación correcta la dirección a buscar), una consulta sobre si la cadena localizada en la posición que debería corresponder a la dirección IP origen se encuentra dentro de las añadidas con anterioridad en nuestro filtro.

Para la interacción con los archivos de texto se han hecho necesarias funciones estándar de C para la gestión de ficheros. Además, se ha empleado la función *strtoul()*, encargada de transformar una cadena de caracteres ASCII dada a un entero sin signo en el formato deseado, utilizada para transformar la dirección IP origen extraída del paquete en formato hexadecimal a su forma decimal para su comparación con las direcciones insertadas previamente en el filtro.

Finalmente, indicar que para la lectura correcta del fichero en el que se definen las direcciones IP a insertar o eliminar del filtro, es necesaria una función que determina

hasta dónde leer una palabra concreta, bien porque se encontraba un símbolo de retorno de línea o de fin de fichero. Esta función es la que se define como *jump_line()* en el *testbench* mostrado.

```
//Inicializacion del filtro a partir de los parametros especificados
status_t = cbf_top(NULL, NULL, 0);
if (status_t) {
	printf("ERROR: No se pudo crear el filtro.\n");
	return 1;
}
printf("\n Filtro Bloom creado satisfactoriamente.\n");

//Apertura del fichero contenedor de los patrones
fp = fopen(PATRONES_FILE, "r");

if (!fp) {
	printf("ERROR: No se pudo abrir el archivo.\n");
	return 1;
}
//Insercion de los elementos del fichero en el filtro
printf("Insercion de los patrones en el filtro.\n");
for (i = 0; fgets(pattern, KEY_SIZE, fp) && (i < CAPACITY); i++) {
	jump_line(pattern);
	pattern32 = strtoul(pattern, NULL, 10);
	status_t = cbf_top((ap_uint<8> *) &pattern32, NULL, 1);
	elements_added++;
	printf("%s se ha insertado en el filtro.\n", pattern);
}
//Eliminacion de patrones especificos del filtro
printf("Eliminacion de patrones incluidos en el filtro.\n");
fseek(fp, 0, SEEK_SET);
for (i = 0; fgets(pattern, KEY_SIZE, fp) && (i < CAPACITY); i++) {
	jump_line(pattern);
	pattern32 = strtoul(pattern, NULL, 10);
	if (i % 2 != 0) {
		status_t = cbf_top((ap_uint<8> *) &pattern32, NULL, 2);
		elements_removed++;
		printf("%s eliminado del filtro.\n", pattern);
	} else {
	}
}
fclose(fp);

//Consulta de si una cadena de un payload dado se encuentra en el filtro
printf("Busqueda del patron en el payload. \n");
ft = fopen(PACKET_FILE, "r");
for (i = 0; fgets(word, 9, ft) && (i < PACKET_SIZE / 8);
		i++) {
	packet[i] = strtoul(word, NULL, 16);
}
status_t = cbf_top(NULL, (ap_uint<32> *) packet, 3);
if (status_t == 0) {
	printf("se ha encontrado un elemento del conjunto\n");
	elements_included++;
} else {
	printf("no hay coincidencias.\n");
}

fclose(ft);
return 0;}
```

6.3. Modelado de interfaces orientado a hardware

Una vez que hemos desarrollado nuestro algoritmo en lenguaje de alto nivel, el primer paso para adaptarlo a *hardware* consistirá en definir la función *top* que defina las interfaces de entrada/salida del bloque IP (Figura 36). En nuestro caso, la función *top* contará con tres parámetros:

- *Pattern*. Define el elemento o patrón que se desea insertar o eliminar del filtro.
- *Payload*. Representa el paquete de datos que se le pasará al filtro para realizar una consulta sobre su dirección IP origen y verificar si se encuentra almacenada en nuestro filtro o no.
- *Mode*. Permite decidir qué operación del filtro se realizará en cada momento.

Figura 36. Diagrama de E/S del bloque IP

Una segunda tarea indispensable para reducir el posterior consumo de recursos de la FPGA será la modificación de los tipos de datos definidos para el almacenamiento de los datos en variables utilizando la librería *ap_int* de Vivado HLS. Esta librería presenta precisión en bits que ajustamos a nuestras necesidades, no teniendo que utilizar los tipos estándar de mínimo 8 bits de precisión y reduciendo con ello el número de recursos consumidos del dispositivo lógico final sobre el que se implementará el sistema completo.

```
ap_int<2> cbf_top(ap_uint<8> pattern[WORD_SIZE], ap_uint<32> * payload,
            ap_uint<3> mode) {

      ap_int<2> status;

      if (mode == 0) {
            counting_bloom_init(CAPACITY, ERROR_RATE, 0);
            status = 0;
      } else if (mode == 1) {
```

```
                status = counting_bloom_add(pattern);
        } else if (mode == 2) {
                status = counting_bloom_remove(pattern);
        } else if (mode == 3) {
                status = counting_bloom_query(payload);
        } else {
                status = -1;
        }

        return status;
}
```

Finalmente, y de forma previa a la síntesis del diseño, se definen las interfaces del sistema. En nuestro caso, hemos decidido definir tanto *pattern* (patrón de referencia) como *mode* (modo de operación) como interfaces AXI4-Lite esclava, ya que de esta forma se nos permitirá interactuar con estos parámetros y gobernar su valor en cada momento desde el SDK de Vivado. Para ello, se definen ambos parámetros con la directiva de INTERFACE *s_axilite*. Asimismo, definimos con el mismo tipo de interfaz la variable *status*, que será la respuesta proporcionada por el bloque IP acerca del resultado de las operaciones realizadas.

```
#pragma HLS INTERFACE s_axilite port=pattern bundle=axi_lite
#pragma HLS INTERFACE s_axilite port=mode bundle=axi_lite
#pragma HLS INTERFACE s_axilite port=return bundle=axi_lite
```

Esta interfaz AXI está destinada a accesos de memoria con un bajo *throughput* y sólo permite una transferencia de datos por transacción realizada, pudiendo ser el tamaño de las palabras enviadas entre 8 y 1024 *bits*. Además, permite un reloj diferente para cada par maestro-esclavo, e incluso la inserción de etapas de *pipeline* entre registros para tener un control prácticamente total en el ámbito temporal. Este protocolo incluye señales independientes para lectura y escritura, tanto de dirección como de datos, además de señales adicionales que marcan el estado de los dispositivos situados a ambos extremos del proceso de comunicación (VALID, READY).

Por el contrario, se ha definido *payload* (interfaz de flujo de datos) como una interfaz AXI4-Stream mediante la directiva de INTERFACE *axis*, ya que el objetivo es que los paquetes que se envíen al filtro para realizar la consulta sean suministrados a través de un flujo directo y constante a través de la red y esta interfaz permite la transferencia

de datos en ráfagas sin limitaciones por transferencia [47][48]. De esta forma, se consigue un flujo de datos unidireccional a altas tasas de transferencia mediante *handshaking*, que consiste en establecer de forma dinámica los parámetros del bus (tasa de transferencia, codificación, paridad, etc.) antes de que comience la comunicación por el canal. El proceso de *handshaking* se lleva a cabo a través de las señales TVALID por el lado del dispositivo maestro y TREADY desde el lado del módulo esclavo, activas a nivel alto [49]. Finalmente, se ha definido su valor de profundidad a 400, ya que es aproximadamente el número máximo de palabras de 32 bits que incluye un paquete de red (1500 *bytes*).

```
#pragma HLS INTERFACE axis depth=400 port=payload
```

Además de los *bytes* de datos, el protocolo AXI4-Stream define otros dos tipos de *bytes*, como son los *bytes* de posición, que indican la posición de un *byte* de datos determinado dentro de un *stream* transmitido, y los *bytes* nulos, los cuales no contienen información y se emplean de relleno para completar el número de *bits* de un paquete de información transmitido, debiendo ser eliminados por el dispositivo esclavo una vez recibido dicho paquete[48].

A continuación, se enumeran en la Tabla 3 las señales de dicha interfaz, así como una breve descripción de cada una de ellas, donde *n* representa el ancho de banda del bus en *bytes*, *i* el ancho de la señal TID (8 bits como máximo recomendado), *d* el ancho de la señal TDEST (4 bits como recomendación máxima) y *u* el ancho de la señal TUSER, normalmente un múltiplo entero del ancho de banda del bus.

Tabla 3. Listado de señales del protocolo AXI-Stream

Señal	Fuente	Descripción
ACLK	Generador de reloj	Señal global de reloj; las señales son muestreadas en el flanco de subida de la misma
ARESETn	Generador de *reset*	Señal global de *reset*, activa a nivel bajo
TVALID	Maestro	Indica que el maestro está iniciando una transferencia válida
TREADY	Esclavo	Indica que el esclavo puede aceptar una transferencia en el ciclo de reloj actual
TDATA [(8n-	Maestro	*Payload* a transmitir

Señal	Fuente	Descripción
1):0]		
TSTRB [(n-1):0]	Maestro	Indica qué byte del *payload* es de datos y cuál de posición
TKEEP [(n-1):0]	Maestro	Indica qué parte de cada byte recibido se procesa como datos o como bit nulo
TLAST	Maestro	Enviada para indicar el final de un paquete de datos
TID [(i-1):0]	Maestro	Identificador del *stream* de datos
TDEST [(d-1):0]	Maestro	Proporciona información de *routing* al flujo de datos
TUSER [(u-1):0]	Maestro	Define información adicional que puede ser transmitida junto con el flujo de datos

6.4. Síntesis, análisis de resultados y optimización

Una vez completado el proceso de diseño del bloque IP y habiendo comprobado mediante el *testbench* incluido de que funciona de forma totalmente correcta, es turno de realizar la síntesis del código generado para comprobar que cumple con las restricciones temporales especificadas al comienzo de su desarrollo y que no consume demasiados recursos de los disponibles en la FPGA, ya que podría suponer un problema a la hora de su integración en la plataforma completa que conforma el sistema.

De forma previa a la generación de dicha descripción *hardware*, se hace necesario el análisis de qué tipo de directivas se pueden aplicar a nuestro diseño con el objetivo de optimizar sus características temporales y los recursos lógicos consumidos. La primera opción que surge es el desenrollado de bucles con numerosas iteraciones, siempre que no se realice dentro del mismo operaciones de lectura y escritura sobre una misma variable y que el número de iteraciones sea invariante en la vida del programa.

La otra estrategia de optimización la conforma la implementación de *pipelining* para mejorar los tiempos de ejecución al aplicarse distintas tareas en paralelo en un mismo ciclo de reloj, la cual se puede aplicar en nuestro diseño en el bucle de la función *hash* aplicada. A continuación se exponen los resultados de la síntesis de alto nivel para nuestro diseño sin aplicar dicha directiva e incluyéndola posteriormente.

6.4.1. Análisis del diseño sobre Zynq sin directivas de optimización

En este primer caso, se aplicará la síntesis de alto nivel al diseño completo una vez finalizado y sin aplicarle ningún tipo de directiva. La restricción temporal de funcionamiento que se le ha aplicado al diseño es un ciclo de reloj de 3,3 ns.

Tal como se aprecia en la Figura 37, el tiempo de ciclo del reloj de funcionamiento del diseño es inferior a la restricción inicial establecida, pudiendo funcionar nuestro diseño hasta a 365 MHz, 65 MHz por encima de los 300 MHz establecidos como requisito mínimo de funcionamiento.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.30	2.74	0.41

Figura 37. Resultados temporales de la síntesis de alto nivel del diseño

Por su parte, el consumo de recursos lógicos de la FPGA por parte de nuestro diseño es el que se muestra en la Figura 38 siendo, tal como se puede observar, ínfimo en comparación con los recursos totales de que dispone el sistema.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	44
FIFO	-	-	-	-
Instance	2	-	311	322
Memory	1	-	0	0
Multiplexer	-	-	-	45
Register	-	-	70	-
Total	3	0	381	411
Available	280	220	106400	53200
Utilization (%)	1	0	~0	~0

Figura 38. Consumo de recursos lógicos del diseño IP realizado

Finalmente, en la Figura 39 y la Figura 40 se muestran las planificaciones de las operaciones de inserción de elementos en el filtro (la planificación para la eliminación de direcciones del mismo es idéntica) y de consulta sobre un paquete dado. Tal como se desprende de su análisis, se estima que cada una de las operaciones se puede realizar en 7 ciclos de reloj, estando localizadas las zonas críticas de ejecución en los bucles contenidos, que son los que se optimizan posteriormente aplicando las directivas de *pipeline*. Por tanto, el tiempo de ejecución total de cada operación, para un tiempo de ciclo de 3,3 ns, será de 23,1 ns.

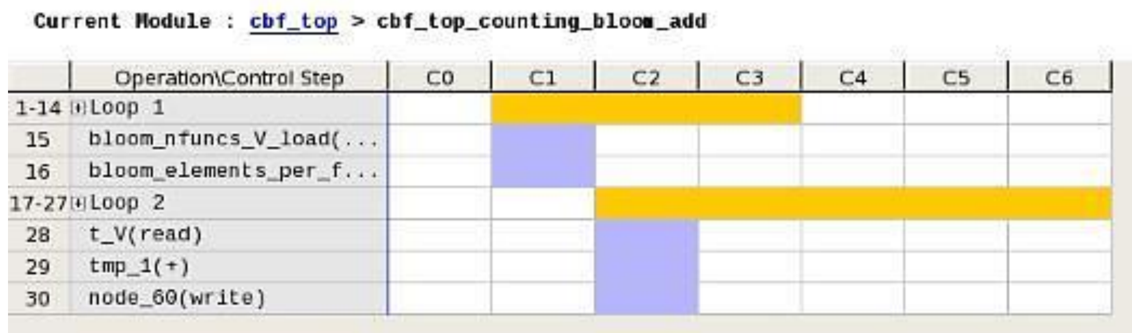


Figura 39. Planificación de las operaciones para la inserción de direcciones

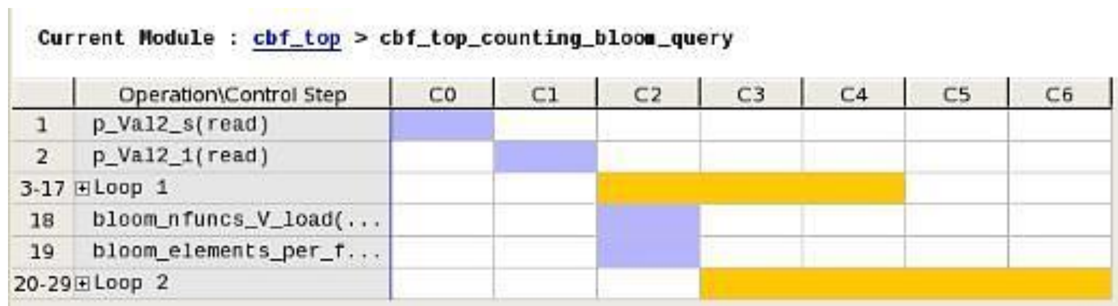


Figura 40. Planificación de las operaciones para la consulta sobre un *payload*

6.4.2. Análisis del diseño sobre Zynq aplicando *pipeline*

Para este segundo caso de análisis, aplicaremos la directiva de *pipeline* al bucle de generación del *hash*, ya que resulta ser el más crítico y puede estar sujeto a optimización, ya que no hay dependencia de datos entre operaciones de lectura y escritura.

Tal como se observa en la Figura 41, dicha directiva da como resultado un considerable aumento del ciclo de funcionamiento del diseño, estando en este caso por encima (si le sumamos el error posible en la estimación) de la restricción temporal de 3,3 ns fijada inicialmente. Por tanto, no se puede dar como satisfactorio este resultado, pues de esta forma la frecuencia de funcionamiento está por debajo de los 300 MHz estipulados como mínima frecuencia de funcionamiento para nuestro diseño.

Performance Estimates

⊟ **Timing (ns)**

⊟ **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	3.30	3.25	0.41

Figura 41. Resultados temporales aplicando la directiva de *pipeline*

Además de influenciar sobre el rendimiento temporal del bloque IP, la aplicación de dicha directiva provoca también un aumento de los recursos lógicos consumidos (Figura 42), aunque en este caso se trata de un incremento muy sustancial y prácticamente no tiene incidencia sobre el consumo total final. Finalmente, al tratarse de un bucle con muy pocas iteraciones, la optimización aplicada no tiene ningún reflejo sobre la planificación de las operaciones a realizar, las cuales se realizan en el mismo número de ciclos que si no aplicáramos la directiva.

Utilization Estimates

⊟ **Summary**

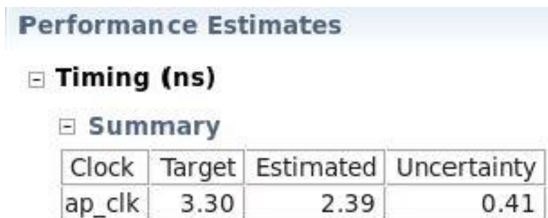
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	8
FIFO	-	-	-	-
Instance	2	-	507	464
Memory	1	-	0	0
Multiplexer	-	-	-	38
Register	-	-	27	-
Total	3	0	534	510
Available	280	220	106400	53200
Utilization (%)	1	0	~0	~0

Figura 42. Consumo de recursos aplicando la directiva de pipeline

6.4.3. Análisis del diseño sobre Virtex-7

Adicionalmente, se ha decidido analizar el rendimiento y el consumo de recursos del diseño IP sobre la familia de FPGAs Virtex-7 de Xilinx, la cual compone la gama más alta del catálogo de dispositivos lógicos de dicha empresa. Para ello, se ha creado una nueva solución en Vivado HLS destinada a implementar el bloque diseñado sobre un dispositivo final perteneciente a dicha familia y se le han añadido exactamente los mismos ficheros fuente que para el diseño sobre Zynq.

Al tratarse del dispositivo más eficiente de la gama en cuanto a rendimiento temporal se refiere, los resultados en cuanto a tiempo de ejecución del algoritmo son sustancialmente mejores que en Zynq (del orden de 0,5 ns aproximadamente en la estimación), tal como se aprecia en la Figura 43.


Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.30	2.39	0.41

Figura 43. Resultados temporales sobre Virtex-7

En relación con el consumo de recursos lógicos de la FPGA, se aprecia un ligero descenso en la utilización de *flip-flops* y LUTs (Figura 44), siendo el porcentaje de utilización de todos los elementos lógicos también más bajos debido a que el total de recursos disponibles es bastante mayor que en la familia Zynq.

Para finalizar, indicar que la planificación de las operaciones no varía, ya que el programa actúa de la misma forma de manera independiente al dispositivo de la familia de FPGAs de Xilinx sobre el que se implemente.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	-	-	-	-
Instance	2	-	467	429
Memory	1	-	0	0
Multiplexer	-	-	-	35
Register	-	-	20	-
Total	3	0	487	466
Available	2060	2800	607200	303600
Utilization (%)	~0	0	~0	~0

Figura 44. Consumo de recursos en Virtex-7

6.5. Verificación RTL y exportación del IP

Para finalizar con el proceso de modelado del bloque IP resulta imprescindible, una vez generada la descripción RTL correspondiente, la comprobación de que dicha descripción en lenguaje *hardware* funciona de forma idéntica a como lo hace su homóloga en lenguaje C/C++. Con el fin de verificar esta situación, Vivado HLS proporciona la herramienta de cosimulación que crea, a partir del *testbench* diseñado en alto nivel, un banco de pruebas equivalente en formato RTL para comprobar dicha descripción *hardware*, ejecutándolos paralelamente para comprobar que proporcionan la misma respuesta.

En nuestro caso, se ha elegido Verilog como lenguaje *hardware* a emplear para la generación de la descripción RTL, obteniendo tras realizarse dicha función de cosimulación de forma correcta los datos de latencia temporal de nuestro diseño, tal como se observa en la Figura 45:

Cosimulation Report for 'cbf_top'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	46	88	100	38	78	92

Figura 45. Resultados de la cosimulación

Se puede apreciar que el número medio de ciclos de reloj que emplea nuestro diseño para realizar las funciones especificadas en el *testbench* creado para verificar su funcionamiento es de 88, lo cual da como resultado, si se toma el ciclo de reloj como el establecido como objetivo de 3,3 ns, una latencia total de ejecución de 290,4 ns (o lo que es lo mismo, 0,2904 us).

Como paso final, se hace necesario empaquetar y exportar nuestro diseño como un bloque IP que pueda ser manipulado desde la herramienta IP Integrator de Vivado IDE con el objetivo, como es el caso de este trabajo, de incluir dicho módulo en una plataforma que conforme el sistema final y que estará compuesta por más bloques además del desarrollado en alto nivel. Para ello, empleamos la herramienta de exportación que incluye Vivado HLS, en la que se nos abre un cuadro de diálogo como el que se muestra en la Figura 46, pudiendo definir los parámetros de identificación del bloque y en el que se nos ofrece la posibilidad de evaluar nuevamente la descripción RTL generada para obtener resultados finales de rendimiento temporal del diseño, así como de consumo de recursos lógicos.

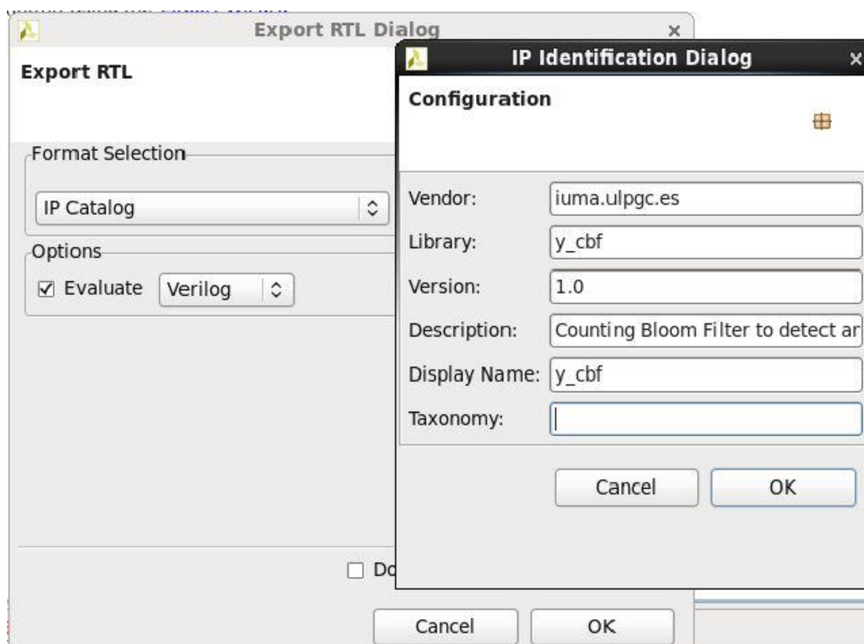


Figura 46. Cuadro de diálogo para la exportación del IP

Tal como se puede observar en la Figura 47 y la Figura 48, los resultados finales obtenidos tras evaluar la descripción RTL muestran que nuestro diseño no puede trabajar por debajo del periodo de reloj definido inicialmente como máximo para el algoritmo. Esto refleja que las estimaciones de la síntesis en alto nivel realizadas por la herramienta Vivado HLS no son demasiado precisas (diferencia porcentual del 61.2 %) haciendo necesario, una vez se han obtenido estos resultados, volver a sintetizar con el tiempo real de ejecución que va a tener nuestro diseño para recopilar información más fiable y semejante a lo que ocurrirá una vez nuestro bloque IP sea integrado en la plataforma completa. La razón de esta discrepancia en los resultados en ambas etapas del proceso de síntesis de alto nivel, puede ser debida a que durante la parte de exportación la descripción *hardware* ya incluye las interfaces de comunicación inferidas y AXI4-Lite trabaja con un bajo *throughput* (entre 150 y 200 MHz); por tanto, se hace necesario llegar a un compromiso en cuanto a frecuencia de funcionamiento se refiere, pues no podemos renunciar al uso de esta interfaz, que simplifica de forma considerable el proceso de inicialización y configuración del filtro.

Resource Usage

	Verilog
SLICE	103
LUT	240
FF	298
DSP	0
BRAM	3
SRL	0

Figura 47. Consumo de recursos calculados durante la evaluación RTL

Final Timing

	Verilog
CP required	3.300
CP achieved	4.417

Timing not met

Figura 48. Informe temporal del diseño tras evaluar la descripción RTL

Por tanto, fijaremos el ciclo de reloj de nuestro sistema a un valor de 4,5 ns lo que supone, al no variar la planificación de las operaciones respecto al primer caso, que cada operación tomará un tiempo de 27 ns en ser ejecutada completamente. Por otro lado, tras ejecutar posteriormente la cosimulación, obtenemos el mismo número de ciclos de reloj de media para ejecutar el *testbench* (88 ciclos), pero al tomarse en este caso un tiempo de 4,5 ns, obtenemos una latencia total de ejecución del algoritmo de 396 ns.

En este caso, los resultados de la evaluación RTL previa a la exportación sí resultarán favorables, obteniendo los valores que se muestran en la Figura 49 y la Figura 50.

Resource Usage

	VHDL
SLICE	127
LUT	273
FF	374
DSP	0
BRAM	3
SRL	0

Figura 49. Consumo de recursos para la evaluación del RTL con 4.5 ns

Final Timing

	VHDL
CP required	4.500
CP achieved	4.211

Timing met

Figura 50. Información temporal para la evaluación del RTL con 4.5 ns

Finalmente, bastará con añadir desde el Vivado IDE (en el catálogo de IPs de Xilinx) el archivo .zip generado a partir de la exportación de nuestro bloque para poder interactuar con él desde la herramienta Vivado Integrator.

6.6. Conclusiones

Este capítulo incluye la primera parte de diseño del proyecto. Después de haber introducido la solución que se tiene por objetivo implementar para cumplir con la aplicación especificada, en este apartado se describe con detalle el proceso de diseño del bloque IP que integra la funcionalidad completa de un filtro Bloom con contador, destacando aquellas partes de la implementación más relevantes, los casos de verificación asumidos y la definición de las interfaces *hardware* que rigen el proceso de comunicación con el resto de bloques que conforman la plataforma.

Posteriormente, se proporciona los valores obtenidos de aquellos parámetros del diseño (tiempo de ciclo y consumo de recursos) que se consideran clave, planteando distintos casos tanto para nuestra solución final (con y sin directivas de optimización) como para otra alternativa empleando un dispositivo de mayores prestaciones. Estos resultados se obtienen de las distintas fases de que consta la etapa de diseño de síntesis de alto nivel empleando la herramienta Vivado HLS, como son la síntesis propiamente dicha, la planificación de las operaciones, los resultados de cosimulación y los valores obtenidos durante la exportación del IP.

Hay que indicar que la inclusión de las interfaces estandarizadas del diseño se produce durante la fase de exportación del IP al flujo de diseño de síntesis lógica. Este proceso introduce modificaciones en el comportamiento temporal pudiendo aparecer rutas críticas no contempladas durante la síntesis de alto nivel. Ello produce discrepancias entre la información de frecuencia de funcionamiento proporcionada por Vivado HLS entre los valores obtenidos durante la etapa de síntesis y la exportación del IP.

Capítulo 7. Integración en la plataforma

7.1. Introducción

Una vez explicado el proceso de diseño del bloque IP que implementa la funcionalidad de un filtro Bloom con contador, es turno en este capítulo de abordar su integración con el resto de módulos que componen el sistema definitivo.

7.2. Primera iteración: diseño de plataforma básica

7.2.1. Arquitectura de la plataforma

La estructura de la plataforma básica no es excesivamente complicada si atendemos al número de bloques principales que la integran, así como a las comunicaciones entre dichos módulos. Como bloques principales se cuenta con el bloque de procesamiento de la plataforma Zynq, el cual se podría considerar como el núcleo de la plataforma, junto con el controlador DDR de la memoria RAM *off-chip* y el bloque DMA para comunicarse con el módulo diseñado, que componen los dos elementos de memoria integrados en el diseño, además del bloque IP diseñado, que es el que se encarga de analizar la dirección IP fuente del paquete que se le aplique como entrada a la plataforma para tomar, a partir de su valor, las decisiones que se consideren oportunas. En el apartado siguiente se explican los módulos usados en la plataforma.

Para todas y cada una de las comunicaciones que tendrán lugar entre los módulos que conforman la plataforma se utilizan buses e interfaces AXI. Este estándar facilita el diseño de bloques IP al margen de la plataforma que se use finalmente en la implementación, ya que es totalmente independiente de la tecnología de diseño. Para

esta plataforma en concreto, se dispone de una interfaz AXI4 disponible mediante el IP AXI Interconnect que se emplea para establecer las comunicaciones del PS con los periféricos del sistema y del DMA con el controlador DDR y viceversa; para la comunicación directa entre el PS y el DMA para indicarle la dirección en la memoria RAM donde se ubica el paquete de datos a analizar, así como para configurar e inicializar el bloque IP diseñado y el propio DMA se emplea una interfaz AXI4-Lite, mientras que para la comunicación entre estos dos últimos módulos se utilizará una interfaz AXI4-Stream [12]. En la Figura 51 se puede observar el diagrama de bloques de la arquitectura de dicha plataforma.

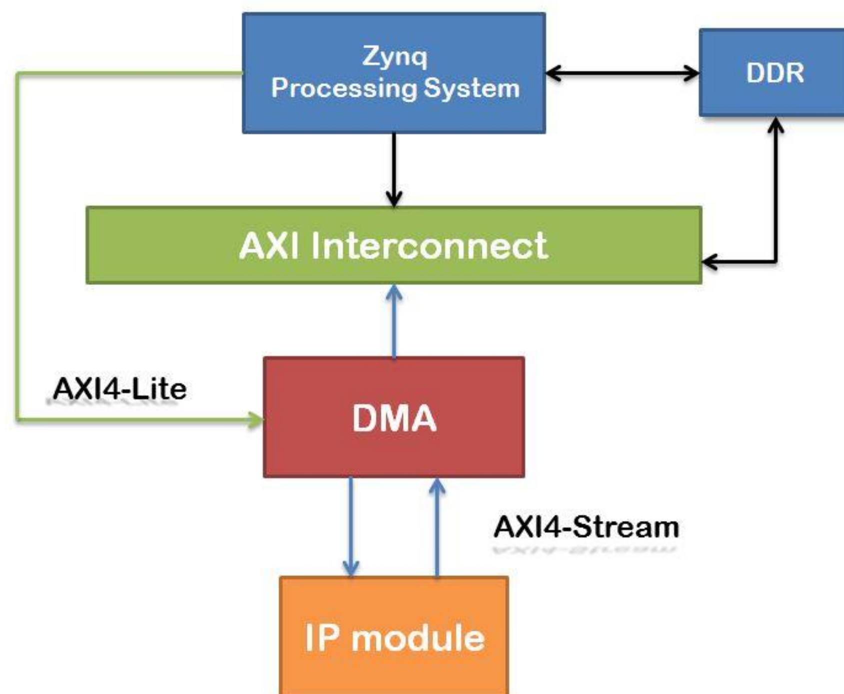


Figura 51. Diagrama de bloques de la arquitectura de la plataforma básica de prueba

7.2.2. Bloques IP

La plataforma en su conjunto está compuesta por un conjunto de 7 bloques IP, de los cuales 6 de ellos están integrados en el catálogo de bloques IP que proporciona Xilinx,

por lo que solo se hace necesario el estudio de su funcionamiento y de sus parámetros para su utilización. El séptimo bloque en cuestión es el que ha sido desarrollado siguiendo una metodología de síntesis de alto nivel y cuya misión básica es la detección de la dirección IP origen de los paquetes de datos recibidos y la comprobación de si se encuentran entre las incluidas en el registro de direcciones IP del filtro. La descripción RTL generada se encapsula como un bloque IP mediante la aplicación IP Packager, pudiendo integrarse como un bloque más en la plataforma empleando la herramienta Vivado IP Integrator [12]. En la Figura 52 se muestra el diagrama completo de esta plataforma.

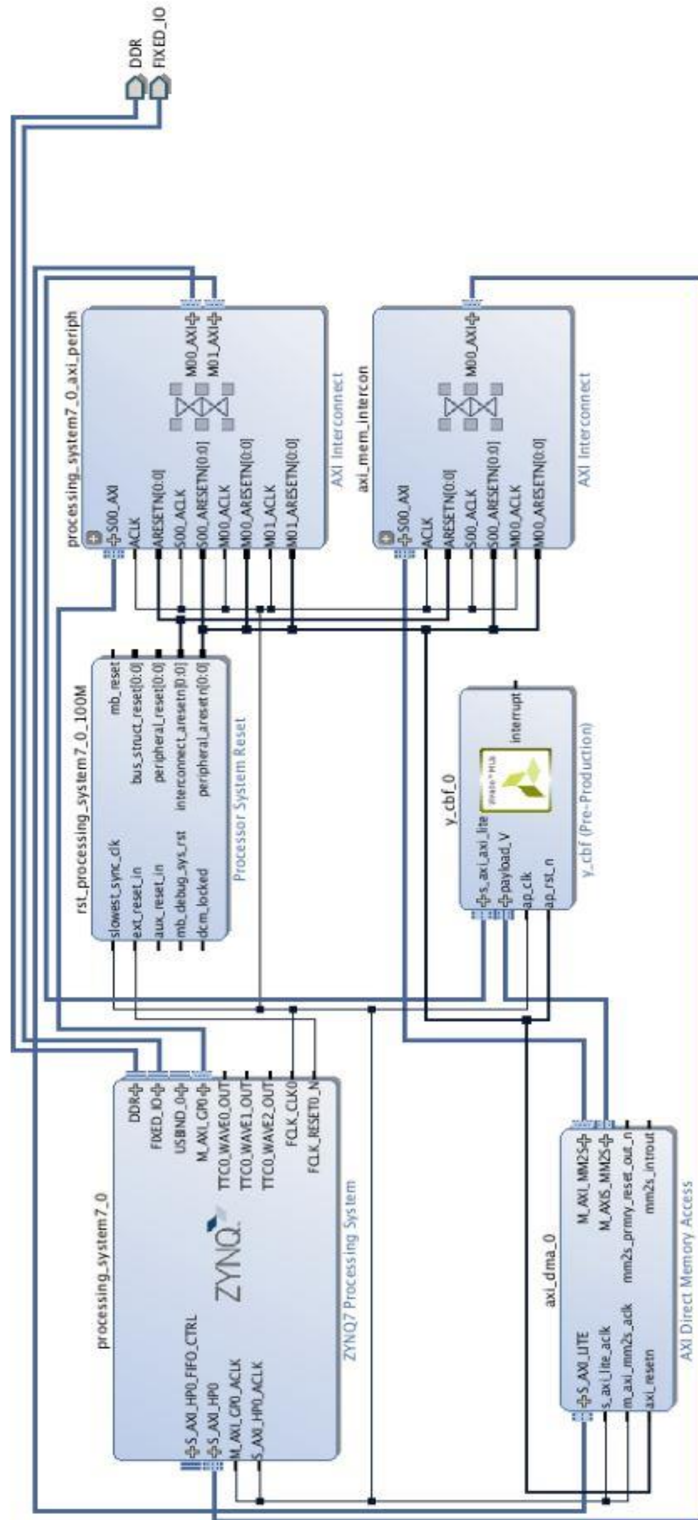


Figura 52. Diagrama de bloques de la plataforma

A continuación, se pasan a comentar con detalle cada uno de ellos, haciendo hincapié en aquellas características o parámetros especialmente relevantes para la aplicación específica que nos incumbe:

- *processing_system7_0.* Bloque central de la plataforma que comprende la instanciación del bloque de procesamiento de un SoC de la familia Zynq-7000 para su interacción y comunicación con la lógica programable (FPGA) y con elementos externos de la placa de prototipado. La comunicación entre el PS y el bloque lógico se lleva a cabo entre distintas interfaces del protocolo de buses AXI en función de los elementos a interconectar y su propósito[50].

- *rst_processing_system7_0_50M*. Bloque IP que se encarga de gestionar las señales de *reset* del resto de bloques integrados en la plataforma.

- *processing_system7_0_axi_periph y axi_mem_intercon.* Bloques IP encargados de configurar y establecer las conexiones de buses AXI entre los distintos bloques IP de la plataforma durante la etapa de diseño de la misma. Además, permiten la conexión de distintos pares de dispositivos maestro-esclavo que trabajen con distintos anchos de banda, frecuencia de reloj e interfaz de AXI (excepto AXI4-Stream, que requiere de un módulo adaptador específico) [51].

- *axi_dma_0.* Este bloque DMA, crea una vía directa de acceso a memoria para la realización de transacciones con la misma sin pasar por la CPU del sistema. A través de este bloque, el PS envía los datos al bloque IP diseñado. Presenta el inconveniente de que el sistema de procesamiento no puede ejercer un control directo sobre el bloque IP al tener que usar el DMA; dicho de otra forma, el bloque *cbf_top* es "invisible" para el PS, el cual solo ve dentro de su entorno de interacción el acceso al DMA. Una singularidad de este bloque radica en que se considera lectura el intercambio de datos hacia el bloque *cbf_top*, siendo la escritura el procedimiento en el sentido contrario (datos desde nuestro bloque IP hacia el DMA) [52].

- *cbf_top.* Bloque implementado en lenguaje C/C++ siguiendo una metodología de síntesis de alto nivel y que constituye la parte principal de desarrollo de esta plataforma, encargándose de la inserción y eliminación de direcciones IP en sus registros de almacenamiento para su posterior comparación con la dirección IP origen de los paquetes de red entrantes y la realización de las operaciones que se consideren oportunas a partir de la respuesta generada (filtrado, bloqueo, descarte de paquetes). Consiste en un bloque compuesto por, además de la señal de reloj y de *reset*, dos puertos adicionales vinculados a distintas interfaces AXI: uno destinado a las comunicaciones mediante AXI4-Lite con el diseñador a través del entorno Vivado SDK para la inicialización del bloque y gestión de los patrones a tratar (inserción y eliminación), así como para devolver una respuesta con el resultado de cada operación realizada al PS para la toma de decisiones. Un segundo puerto está dedicado a la comunicación mediante una interfaz AXI4-Stream entre el DMA y el bloque IP para la transmisión de paquetes sobre los cuales realizar consultas, los cuales en un primer momento estarán ubicados en la memoria RAM del PS y serán accesibles por este módulo a través del DMA (Figura 53).

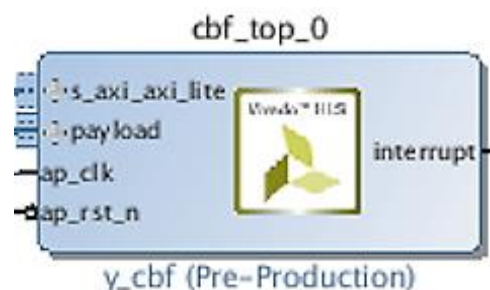


Figura 53. Bloque cbf_top_0 desarrollado mediante metodología HLS

7.2.3. Métodos de comunicación. Interfaces y buses

Como se ha explicado en el apartado anterior en el que se definen todos y cada uno de los bloques IP incluidos en la plataforma, la conexión no se realiza punto a punto entre los distintos módulos que la integran, sino que se precisa del bloque AXI Interconnect que permite las comunicaciones entre el PS con los distintos bloques que

conforman la plataforma y desempeña funciones de arbitraje cuando surgen discrepancias sobre qué módulo debe tomar el control del bus en cada momento.

El PS de la plataforma Zynq realizará la comunicación con el DMA empleando interfaces del tipo AXI Memory-Mapped, conocida también como AXI4 simplemente, mediante la cual a cada bloque conectado al bus se le asigna un rango de direcciones de memoria que lo hace fácilmente ubicable a la hora de realizar una transacción. Para pasarle al bloque IP sus parámetros de configuración e inicialización, así como para proporcionarle al DMA la información básica para comunicarse con la memoria RAM *off-chip* con el fin de obtener la dirección de memoria donde se encuentran los paquetes de datos sobre los que se van a realizar las consultas (PS como maestro, DMA como esclavo), se emplea una interfaz AXI4-Lite(Figura 54)[48].

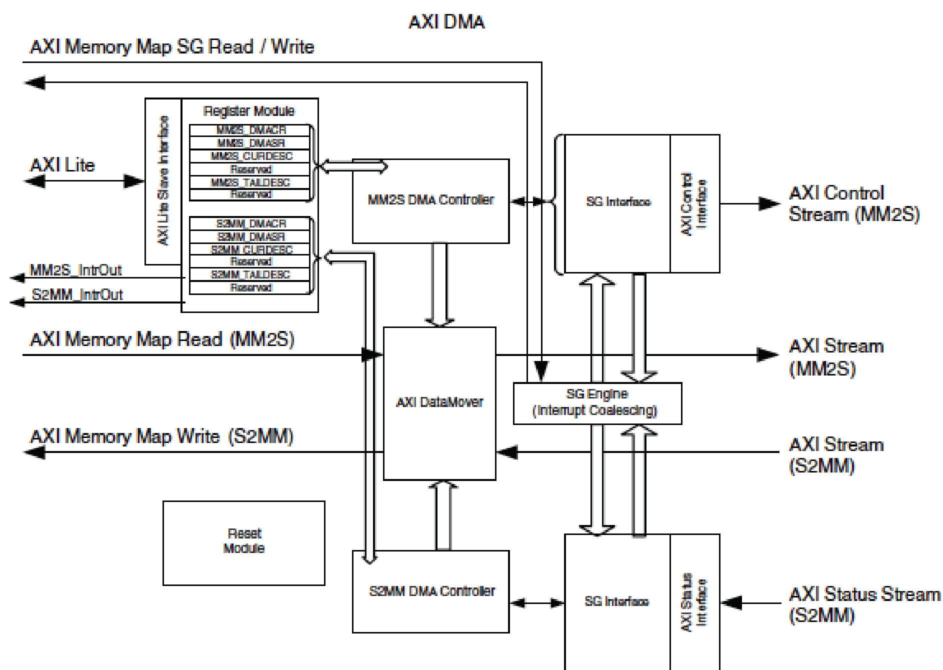


Figura 54. Diagrama de bloques del AXI DMA

Finalmente, como se ha indicado con anterioridad, la comunicación entre el DMA y el bloque *cbf_top* se lleva a cabo a través de una interfaz AXI4-Stream, lo que supone establecer de forma independiente las líneas de escritura y lectura. Sin embargo, esta

interfaz presenta algunas ventajas, como son la agilización de las operaciones de lectura y escritura y el menor consumo de recursos de la FPGA.

7.2.4. Diseño del software empotrado

Llegados al punto de haber diseñado y verificado el bloque IP de forma independiente, y tras haberlo integrado en la plataforma que conforma una primera aproximación al sistema completo final, se hace necesaria la programación y la implementación del software que gobernará su funcionamiento.

7.2.4.1. Programación de la aplicación software

Como punto de partida, al haber exportado el *hardware* desde Vivado, se dispone de una serie de *drivers* llamados BSPs, que funcionan como controladores de aquellos elementos del dispositivo final de prototipado que son necesarios emplear en nuestra aplicación concreta, así como funciones específicas para manejar nuestro bloque IP (inicialización, manejo de elementos, consultas), facilitando de esta forma la integración *hardware/software* del diseño[44][53].

La función encargada de dirigir el funcionamiento de nuestro bloque está organizada de forma que realice una u otra operación en función del valor de *mode*, variable cuyo valor dictaminará el diseñador mediante *software* y que será pasada al bloque mediante una interfaz AXI4-Lite usando la función *Set_mode()*. Con el primer modo se realiza la etapa de configuración e inicialización del filtro, mediante que con los dos siguientes se añade o elimina direcciones IP del mismo empleando la función *Write_Pattern_bytes()*.

Finalmente, el modo que más interés despierta es el 3, el cual realiza consultas sobre los paquetes albergados en memoria, determinando si su dirección IP origen concuerda con alguna de las almacenadas previamente en el filtro. Tras realizar una copia de dicho paquete de datos en el *buffer* de transmisión del DMA, se realiza una transmisión simple hacia el filtro (este sentido de comunicación es el único existente en

nuestra aplicación entre ambos módulos); adicionalmente, disponemos de un bucle que evita sobrescribir el *buffer* de transmisión antes de que los datos sean enviados, esperando hasta que el DMA y el canal de transmisión estén libres y asegurando de esta forma el envío de los paquetes. A partir de la respuesta generada por el bloque, la cual se indicará que está disponible para su lectura cuando la señal *IsDone* esté a nivel alto, se devolverá una respuesta afirmativa o negativa al bloque de procesamiento para que posteriormente tome las decisiones que considere oportunas.

Debido a que nuestra aplicación es *bare-metal* (es decir, no incluye un sistema operativo), los bloques IP se encontrarán mapeados en memoria, estando cada uno ubicado en una dirección determinada y fija para dicho bloque.

```
int packetInspection(u32 mode, unsigned char ip_address[4], u32* payload) {

	int Index = 0, status;

	switch (mode) {

	//Inicializacion del filtro
	case 0:
		XCbf_top_Set_mode_V(&block, mode);
		XCbf_top_Start(&block);
		status = XCbf_top_Get_return(&block);
		break;

	//Insercion de elementos en el filtro
	case 1:
		XCbf_top_Set_mode_V(&block, mode);
	status = XCbf_top_Write_pattern_V_Bytes(&block, 0,(int)ip_address,
IP_SIZE);
		break;

	//Eliminacion de elementos del filtro
	case 2:
		XCbf_top_Set_mode_V(&block, mode);
	 status = XCbf_top_Write_pattern_V_Bytes(&block, 0,(int)ip_address,
IP_SIZE);
		break;

	//Consulta al filtro sobre paquetes almacenados en memoria
	case 3:

	// Copia del payload en el buffer del DMA previa a la transmision
		for (Index = 0; Index < PACKET_LIMIT; Index++) {
			dma_buf[Index] = payload[Index];
		}

		// Transmision del DMA hacia el filtro
		status = XAxiDma_SimpleTransfer(&axi_dma,  &dma_buf[0],
				DMA_XFER_LENGTH, XAXIDMA_DMA_TO_DEVICE);


		if (status != XST_SUCCESS) {
```

```
				xil_printf("ERROR! Failed to kick off MM2S transfer!\n\r");
				return XST_FAILURE;
			}

			// Espera en caso de que el DMA se encuentre ocupado
			while ((XAxiDma_Busy(&axi_dma, XAXIDMA_DMA_TO_DEVICE))) {
				/* Wait */
			}


			XCbf_top_Set_mode_V(&block, mode);

			xil_printf("DMA transfer complete!\n\r");

			// Analisis de la respuesta del filtro
			while (XCbf_top_IsDone(&block) != 1) {
				/* Wait */
			}

			status = XCbf_top_Get_return(&block);
			break;

		default:
			break;
		}
		return status;
}
```

Por otra parte, contamos con la función *bloom_application(),* mediante la cual se gestionan las operaciones a realizar con el filtro para verificar su correcto funcionamiento una vez se ha realizado la integración *hardware/software* del diseño. Inicialmente, se configuran los paquetes albergados en memoria con los cuales se llevarán a cabo las pruebas vinculadas a consultas, para a continuación inicializar el filtro, insertar y eliminar direcciones IP previamente creadas e inicializadas y concluir realizando una serie de consultas para validar el sistema, cuya respuesta se analizará mediante el valor de la variable *status*.

```
int bloom_application() {

	int status = 0;

	configPayload0(payload);

	/* ... */

	status = packetInspection(0, NULL, NULL);

	if (status != 0) {
		xil_printf("Test failed. Filter couldn't be created\r\n");
		return XST_FAILURE;
	}
	// Creacion y declaracion de las direcciones IP a introducir
	unsigned char ip_address0[4] = { 253, 2, 20, 172 };
```

```
        unsigned char ip_address1[4] = { 192, 168, 0, 1 };
        /* ... */

        // Interaccion con el filtro ejecutando distintas operaciones
        status = packetInspection(1, ip_address0, NULL);
        status = packetInspection(1, ip_address1, NULL);
        if (status != IP_SIZE) {
                xil_printf("Test failed. Pattern couldn't be added\r\n");
                return XST_FAILURE;
        }

         status = packetInspection(2, ip_address0, NULL);
         if (status != IP_SIZE) {
         xil_printf("Test failed. Pattern couldn't be deleted\r\n");
         return XST_FAILURE;
         }

        status = packetInspection(3, NULL, payload);
        /* ... */

        if (status != 0) {
                xil_printf("Test failed. Pattern isn't in the filter\r\n");
                return XST_FAILURE;
        }
        return XST_SUCCESS;
}
```

La función *main()* se encarga, además de servir como punto de entrada del programa, de inicializar el filtro y ejecutar las funciones de arranque y configuración del DMA, así como de lanzar la función *bloom_application()* ya citada. Finalmente, valora el resultado del test en función del valor de la variable *status*.

```
 int main() {

        int Status = 0;

        // Inicializacion del modulo del filtro
        XCbf_top_Initialize(&block, XPAR_CBF_TOP_0_DEVICE_ID);
        XCbf_top_Start(&block);

        xil_printf("\r\n--- Entering main() --- \r\n");

        //Inicializacion del DMA
        Status = dma_init(cfg_ptr);
        if (Status != XST_SUCCESS) {
                xil_printf("ERROR! DMA initialization failed!\n\r");
                return XST_FAILURE;
        }

        // Ejecucion de la funcion de transferencia entre el DMA y el filtro
        Status = bloom_application();

        // Analisis del resultado del test
        if (Status != XST_SUCCESS) {

                xil_printf("Test Failed\r\n");
                return XST_FAILURE;
        } else {
```

```
            xil_printf("Test Passed\r\n");
            return XST_SUCCESS;
        }

        xil_printf("--- Exiting main() --- \r\n");

        return 0;

}
```

7.2.4.2. Software de gestión del DMA

Empleando un DMA como bloque de gestión del bloque IP se consigue abstraer al mismo del resto del sistema, simplificando de esta forma el proceso de inicialización y configuración. De esta forma, se considera a este elemento de memoria como un punto clave en el correcto funcionamiento del sistema ya que, aunque nuestro bloque funcione perfectamente, si el rendimiento del DMA no es el requerido puede provocar problemas en la comunicación entre el bloque IP y el bloque de procesamiento.

El DMA requiere, como en el caso de nuestro bloque, de un proceso de inicialización y de otro de configuración inicial anterior al proceso de comunicación. Estas etapas se realizan mediante lo que se conoce como descriptores de *buffer*. En nuestro caso, Vivado proporciona la librería *xparameters.h* en la que se encuentran definidos todos los parámetros relacionados con el DMA, pudiendo ser modificados los valores de alguno de ellos por el diseñador, como puede ser el tamaño de los *buffer* de transmisión y recepción o el tamaño de las ráfagas de comunicación[52].

La función que gestiona el funcionamiento del DMA está formada por una serie de funciones básicas proporcionadas por el BSP que se encargan de configurarlo e inicializarlo. Adicionalmente, para nuestra aplicación en concreto decidimos desactivar las interrupciones, ya que en nuestro sistema el DMA estará realizando un *polling* esperando a tener datos disponibles para realizar una transmisión.

```
int dma_init(XAxiDma_Config* cfg_ptr) {

	int status = 0;

	xil_printf("Initializing DMA...\n\r");
	// Busqueda de la configuracion hardware del DMA
	cfg_ptr = XAxiDma_LookupConfig(XPAR_AXIDMA_0_DEVICE_ID);
	if (!cfg_ptr) {
		xil_printf("No config found for %d\r\n", XPAR_AXIDMA_0_DEVICE_ID);
		return XST_FAILURE;
	}

	// Inicializacion del driver
	status = XAxiDma_CfgInitialize(&axi_dma, cfg_ptr);
	if (status != XST_SUCCESS) {
		xil_printf("Initialization failed %d\r\n", status);
		return XST_FAILURE;
	}

	// Test para Scatter Gather
	if (XAxiDma_HasSg(&axi_dma)) {
		xil_printf("Device configured as SG mode \r\n");
		return XST_FAILURE;
	}

	// Deshabilitacion de interrupciones
	XAxiDma_IntrDisable(&axi_dma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);

	xil_printf("DMA initialization complete!\n\r");

	return XST_SUCCESS;
}
```

7.3. Segunda iteración: plataforma final

En esta segunda y última iteración no se realizarán cambios en la plataforma en cuanto a arquitectura y estructura de bloques se refiere, ya que sólo se incluirán núcleos de depuración para comprobar el estado de determinadas señales de comunicación de la plataforma y se llevarán a cabo únicamente modificaciones en la funcionalidad del bloque IP diseñado y en el *software* empotrado que gobernará el funcionamiento del sistema completo.

7.3.1. Modificación de la plataforma hardware

Tal como se ha explicado, al conjunto de bloques que componen la plataforma se le han añadido unos adicionales denominados ILA para verificar el correcto funcionamiento del sistema. Estos bloques incluidos en el diseño en la etapa final de

validación se comportan como núcleos de depuración y no son más que analizadores lógicos digitales que permiten monitorizar señales internas del sistema, siendo conectados a aquellas líneas de comunicación de la plataforma que se consideran más críticas para el correcto funcionamiento de todo el conjunto [7]. Para nuestro sistema en concreto, se ha decidido monitorizar las interfaces AXI4-Lite y AXI4-Stream que sirven como entrada a nuestro bloque para su configuración y para la obtención de los paquetes de datos, respectivamente, con el fin de verificar que las comunicaciones en *hardware* se realizan de forma correcta (Figura 55). Además, mediante la opción *Mark Debug* que proporciona Vivado, se puede depurar el estado de líneas internas de aquellos bloques IP que se estime oportuno con el mismo fin de verificar el correcto funcionamiento de las señales críticas de la plataforma [42].

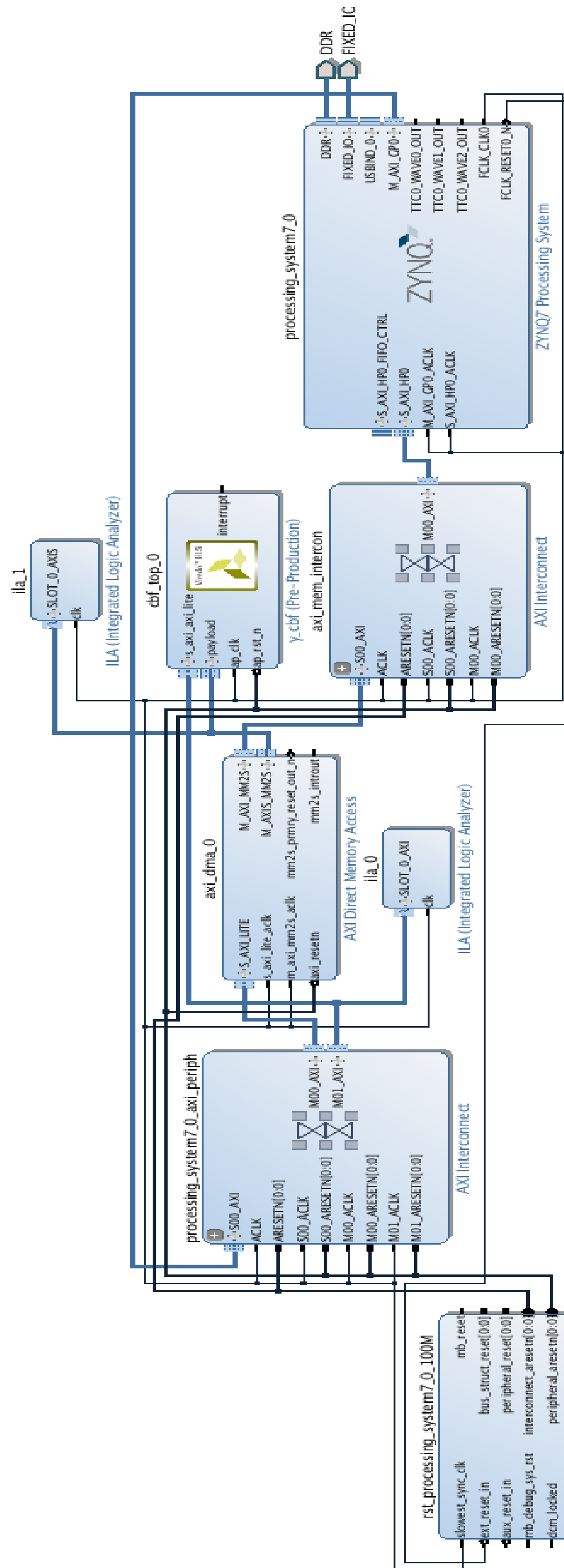


Figura 55. Plataforma completa incluyendo ILAs de depuración

La modificación más importante que se ha realizado en *hardware* en esta segunda iteración ha sido la inclusión de un método de consulta constante mientras a través de la interfaz AXI4-Stream se proporcionen paquetes de datos que analizar; en la primera iteración realizada, la consulta se realizaba de forma específica para un determinado paquete cuando era solicitada por el diseñador a través del *software* empotrado que se incluyó en la plataforma.

La nueva implementación de consulta se realizará sustituyendo la anterior, cambiando el tercer modo posible de ejecución de la función *top* y desechando la solicitud de consultas simples, ya que no tiene mucho sentido más allá de para la realización de pruebas de verificación (una vez implementado el diseño final, debe actuar de forma autónoma, no mediante la interacción continua con el diseñador). Para ello, se ha hecho necesario volver a la primera etapa del ciclo de diseño, con el objetivo de modificar el algoritmo en lenguaje de alto nivel, lo cual resulta bastante menos tedioso que trabajar con la descripción RTL generada por Vivado HLS, ya que el diseñador no tiene ningún control sobre cómo se ha generado dicho código en lenguaje *hardware*.

El primer paso realizado consiste en cambiar el tipo de la señal *payload* de la función *top* al tipo específico de Xilinx para señales de flujo continuo de datos *ap_axis*[39], el cual nos permite tener acceso a cada una de las señales de la interfaz AXI4-Stream. De esta forma, podemos establecer que mientras la señal *last* de nuestra variable sea distinta de 1, lo que indicaría que el paquete de datos entrante a través de dicha interfaz ha sido totalmente procesado, nuestro filtro se mantenga realizando una consulta sobre el mismo.

El procesado constante de paquetes a través de la red se conseguirá encerrando el proceso de consulta individual de un paquete en un bucle *while*, cuya condición de salida estará marcada por la variable *mode*, permaneciendo el sistema en modo de consulta mientras el valor de la misma siga siendo 3. Aprovechando el bucle que es necesario implementar para iterar la señal *last* y conocer su valor en cada momento del proceso de análisis, incluimos en este caso durante el diseño del nuevo modo la extracción del campo

de la dirección IP origen, simplificando de esta forma la funcionalidad y la llamada al método de consulta, al cual únicamente será necesario pasarle dicha dirección IP y no el paquete de datos completo.

Otra decisión tomada ha sido registrar al comenzar el proceso de consulta el valor de las señales *data* y *last* de la variable *stream* declarada, ya que de lo contrario no se garantizaría un acceso secuencial a los datos de la interfaz, quedándose bloqueado el programa a la espera de alcanzar la condición de extracción de la IP. Asimismo, se ha incluido la orden *ap_wait(),* que permite asegurar durante el proceso de síntesis que el código seleccionado se realizará en un único ciclo de reloj.

Finalmente, indicar que en la función *top* se ha decidido cambiar la estructura condicional *if-else* por una estructura *switch*, la cual no establece prioridades entre los distintos modos de ejecución y solo requiere de la ejecución de una comparación.

```
 ap_uint<1> counting_bloom_query(ap_uint<8> ip_address[WORD_SIZE]) {

	ap_uint<10> index;
	ap_uint<4> i;
	ap_int<2> answer;
	short offset;

	hash = 0;
	hash_func(ip_address, matrix);

	for (i = 0; i < bloom.nfuncs; i++) {
		offset = i * bloom.nfuncs;
		index = hash + offset;
		//printf("El indice generado es: %u \n", index.VAL);
		if (!(bitvector_check(index))) {
			answer = 0;
		} else {
			answer = 1;
		}
	}
	return answer;
}
```

```
 ap_int<2> cbf_top(ap_uint<8> pattern[WORD_SIZE], ap_axis<32, 1, 1, 1> * payload,
ap_uint<3> mode){

	ap_uint<6> i;
	ap_int<2> status;
	ap_uint<8> ip_address[WORD_SIZE];
	ap_uint<1> tmp_last =0;
	ap_uint<32> tmp =0;
```

```
        switch (mode) {

        /***/

        case 3:

        i = 0;

        while (mode == 3){

                do {
                 tmp =  payload[i].data;
                 tmp_last =  payload[i].last;

                 if (i == 6) {
                 ip_address[3] = tmp.range(15, 8);
                 ip_address[2] = tmp.range(7, 0);

                 } else if (i == 7) {
                 ip_address[1] = tmp.range(31, 24);
                 ip_address[0] = tmp.range(23, 16);
                 } else {
                 } ap_wait();
                 i++;
                 } while (tmp_last == 0); ap_wait();

                 status = counting_bloom_query(ip_address);
        }
                break;

        default:
                status = -1;
                break;
        }
        return status;
}
```

Para verificar la nueva funcionalidad implementada, será necesario realizar modificaciones en el *testbench* del que se disponía en la primera iteración (modificación del tipo de la señal que se usará para simular el paquete de datos entrante), especialmente en la parte que se encarga de la realización de consultas, en la cual se debe acceder a las señales *data* y *last* del tipo *ap_axis* con el objetivo de extraer los datos incluidos en el paquete y determinar cuándo termina el flujo de información entrante, respectivamente.

7.3.2. Síntesis e implementación de la plataforma

Tras realizar las modificaciones anteriormente mencionadas de forma satisfactoria, obtenemos en primera instancia que nuestro sistema es capaz de trabajar con un reloj de 150 MHz para la lógica programable, obteniendo un *slack* positivo que nos

indica en qué proporción es posible disminuir el periodo de reloj de nuestro sistema para conseguir que funcione a una frecuencia más alta (Figura 56).

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.590 ns	Worst Hold Slack (WHS):	0.046 ns	Worst Pulse Width Slack (WPWS):	2.250 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4757	Total Number of Endpoints:	4757	Total Number of Endpoints:	2021

All user specified timing constraints are met.

Figura 56. *Slack* para una frecuencia de reloj para el PL de 150 MHz

A partir de los datos anteriores, se decide establecer el reloj del PL a una frecuencia de 180 MHz, la cual es algo superior finalmente (187,512 MHz), debido a que es generada por un PLL que no consigue fijar de forma exacta la frecuencia al valor solicitado. Para los procesadores se empleó el reloj por defecto de 666,66 MHz.

Además, hay que indicar que durante el proceso de implementación de la plataforma se decidió aplicar una estrategia de optimización de *retiming* de entre las proporcionadas por Vivado [40], con el fin de poder alcanzar una frecuencia de funcionamiento lo mayor posible. Hay que tener en cuenta que las características del diseño pueden ser difícilmente mejorables al estar condicionada la frecuencia de funcionamiento del sistema por la del DMA (entre 150 y 200 MHz de frecuencia máxima debido al empleo de la interfaz AXI4-Lite para inicialización y configuración).

En la Figura 57 se muestran los resultados de *slack* resultantes estableciendo 180 MHz como la frecuencia de reloj del PL obteniendo, como para el primer caso de 150 MHz, un *slack* positivo, aunque en este caso se puede apreciar como el margen de mejora de la frecuencia de funcionamiento es sustancialmente menor, por lo cual se decide establecer dicho parámetro como final para nuestro diseño.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.083 ns	Worst Hold Slack (WHS):	0.049 ns	Worst Pulse Width Slack (WPWS):	1.416 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4757	Total Number of Endpoints:	4757	Total Number of Endpoints:	2021

All user specified timing constraints are met.

Figura 57. *Slack* para una frecuencia de reloj para el PL de 180 MHz

Una vez determinada la frecuencia máxima de funcionamiento de nuestro sistema, es turno de analizar el consumo de recursos que supondrá su implementación sobre la FPGA. En la Tabla 4 se muestran los resultados de área del diseño, a partir de la cual se puede deducir que el consumo de recursos lógicos por parte de nuestro sistema es reducido, más aún si nos centramos en los recursos empleados para poder implementar el bloque IP basado en filtros Bloom diseñado (*cbf_top*). En la Figura 58 se muestran los mismos resultados de consumo de recursos de la plataforma de una forma más gráfica y resumida.

Tabla 4. Consumo de recursos del bloque IP y la plataforma

Recurso	Total en la FPGA	Usado por el bloque IP	Usado por la plataforma
Slices	13.300	124 (0,93%)	783 (5,89%)
Slice LUTs	53.200	271 (0,51%)	1.947 (3,66%)
LUTs usadas como lógica	53.200	271 (0,51%)	1.869 (3,51%)
LUTs usadas como flip-flops	53.200	145 (0,27%)	1.234 (2,32%)
LUTs usadas como memoria distribuida	17.400	0 (0%)	78 (0,45%)
Registros usados como flip-flops	106.400	362 (0,34%)	2.619 (2,46%)
BRAM	140	1,5 (1,07%)	4 (2,86%)

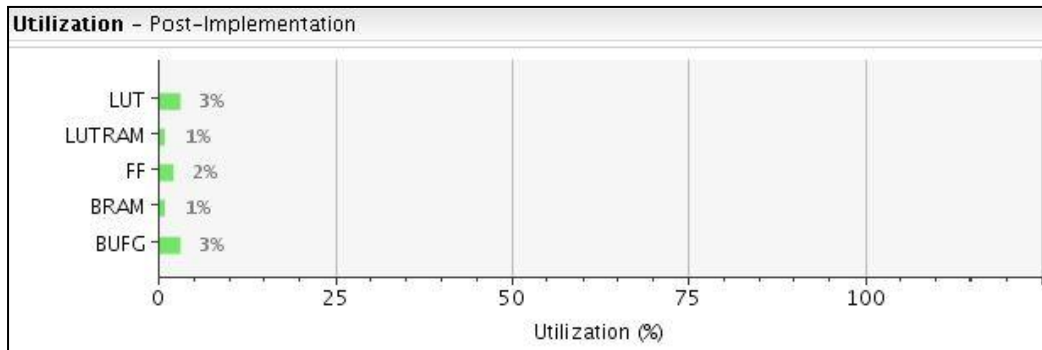


Figura 58. Recursos utilizados por la plataforma

Para hacernos una idea más visual del consumo de recursos que supone la implementación del sistema completo desarrollado sobre la FPGA, también resulta útil un análisis del *layout* resultante. En la Figura 59 se muestra el total de recursos empleados por la plataforma, siendo únicamente los consumidos por el bloque IP diseñado aquellos resaltados en color azul celeste. Tal como se desprende de dicha imagen, el sistema demanda una cantidad de recursos prácticamente ínfima en comparación con el total para su implementación.

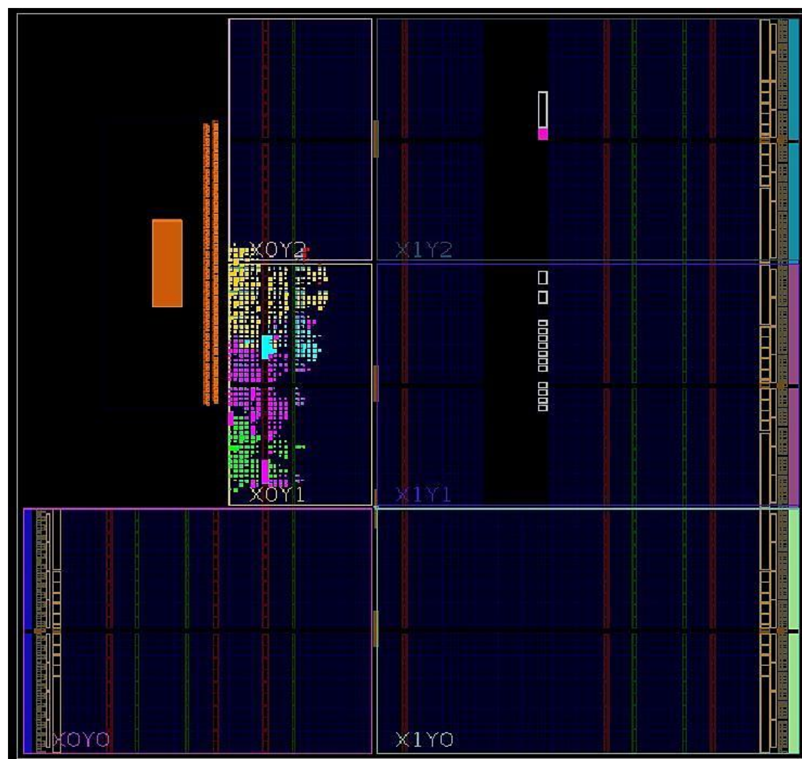


Figura 59. Layout de la plataforma

Un último análisis que resulta crucial llevar a cabo para comprender las características funcionales de nuestro sistema radica en el análisis de la potencia consumida por nuestro bloque IP, la cual se puede deducir restándole a la potencia total del sistema (en este caso, 1.716 W) la consumida por el bloque de procesamiento. Tras hacer dicho cálculo a partir de los datos proporcionados por la Figura 60, obtenemos un consumo de potencia por parte del bloque diseñado de 180 mW.

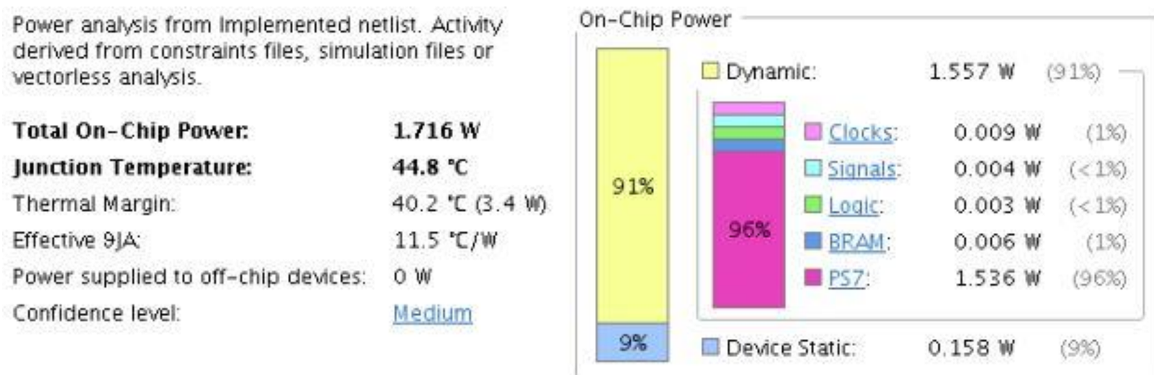


Figura 60. Consumo de potencia de la plataforma

7.3.3. Modificación del software empotrado

Respecto al *software* diseñado para la primera iteración, habrá que realizar una serie de cambios con el objetivo de que en este caso los paquetes a analizar sean proporcionados al sistema directamente a través de la red. Para ello, será necesario incluir la especificación de un controlador Ethernet que permita, mediante un generador de tráfico creado específicamente para esta aplicación, verificar el correcto funcionamiento del sistema.

Para ello, se hace uso del TEMAC que integra el dispositivo XC7Z020 de la familia Zynq-7000 empleado, el cual redirige los paquetes Ethernet recibidos a través de la red hacia el puerto HP (*High Performance*), con el fin de que sean transmitidos finalmente al bloque lógico del sistema desde las memorias del bloque de procesamiento. Este módulo cuenta con dos interfaces AXI4 esclavas, una destinada a funciones de control y una segunda que sirve como ruta de datos desde la FIFO de almacenamiento de los paquetes hasta la FPGA. En nuestro caso, los paquetes de datos solo serán dirigidos hacia el PL, ya

que el bloque de procesamiento no va a realizar ningún tipo de acción sobre el flujo de información recibida [54].

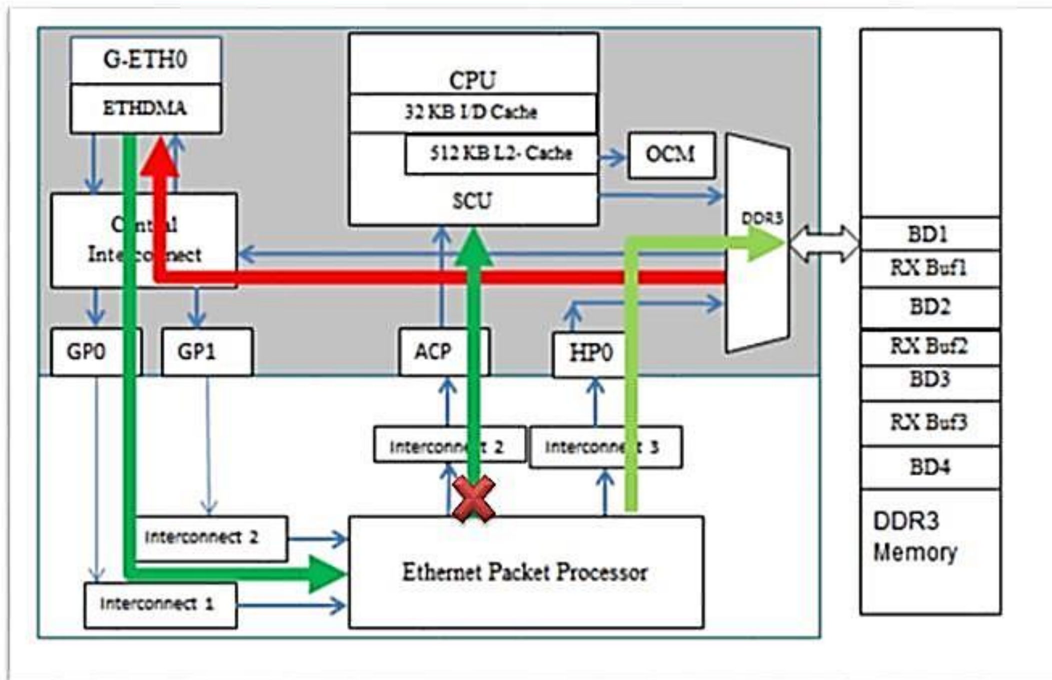


Figura 61. Esquema de bifurcación de paquetes Ethernet sin análisis en el PS

Para integrar la funcionalidad del controlador Ethernet con el manejo de un flujo de datos TCP/IP como el que gobierna el tráfico habitual en la red, será necesario el empleo de la librería *Light-Weight IP*, que permite implementar un *stack* del protocolo TCP/IP para sistemas electrónicos empotrados en ausencia de sistema operativo. Adicionalmente, proporciona soporte para los protocolos usados en Internet, incluidos IP, TCP, UDP, DHCP, ARP o ICMP. El tipo de aplicación a implementar será una RAW API, la cual está basada en *callbacks*, que no son más que llamadas de funciones como si de un parámetro de otra función se tratara, pudiendo acceder de esta forma las aplicaciones al *stack* TCP y viceversa[46].

En este caso, al no disponer de sistema operativo, se ha decidido diseñar distintos módulos que incluyan las distintas funcionalidades de la aplicación final con el objeto de simplificar la programación y la depuración del diseño. Entre estos módulos se incluyen la configuración e inicialización del DMA y de nuestro bloque IP, así como la transferencia

de información entre ambos y las operaciones vinculadas al funcionamiento del filtro. La implementación del *stack* TCP/IP tiene el fin de desarrollar una aplicación que realice un eco del mensaje transmitido, en la cual se configuran todas las funciones vinculadas a la configuración de los parámetros del equipo y las escuchas y confirmaciones de las conexiones entrantes. Y un tercer bloque que incluye el punto de entrada al programa y la llamada a las funciones de los dos módulos *software* anteriormente mencionados.

Como funcionalidad adicional, se ha aprovechado la librería destinada al protocolo DNS que proporciona lwIP para pasarle al filtro las direcciones IP a introducir en nuestra lista negra mediante el dominio web de las mismas, no siendo necesario de esta forma conocer su IP numérica sino simplemente su URL, lo cual resulta más sencillo para el usuario del sistema. Para cumplir con esta funcionalidad, es preciso:

1. Utilizar la función *dns_init ()* para iniciar la comunicación con el servidor DNS.
2. Cambiar la dirección IP del DNS al cual está conectado nuestro *host* mediante la función *dns_setserver()*; y
3. finalmente utilizar la función *dns_gethostbyname()*, a la cual le pasamos el nombre del dominio web que se desea introducir en nuestro filtro y genera de forma automática su dirección IP numérica, almacenándola en una variable de tipo *ip_addr* previamente creada.

Entrando en la estructura del *software* desarrollado (Figura 62), a partir de la breve descripción aportada anteriormente se puede deducir que consta de dos partes esenciales:

1. Escucha de la red a la espera de paquetes y transmisión del mismo al DMA.
2. Paso del paquete al filtro, realización de consulta y toma de decisiones a partir del resultado de la misma.

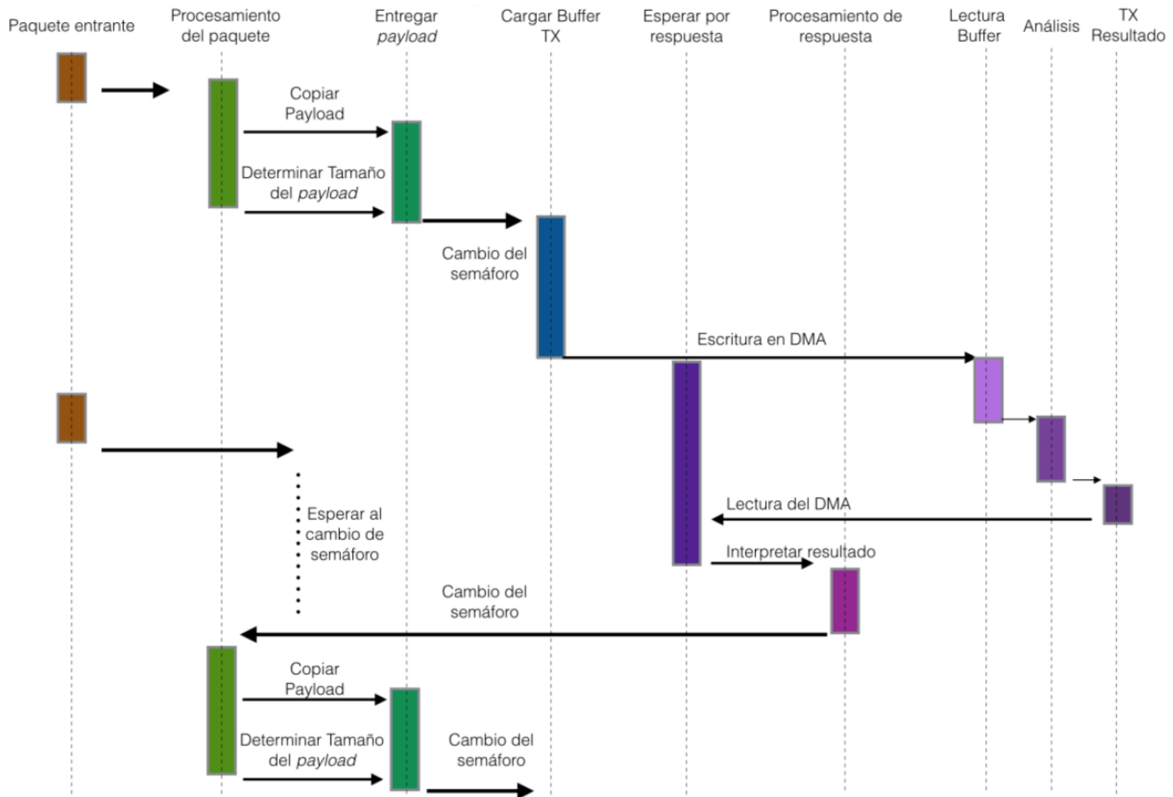


Figura 62. Interacción entre el hardware y el software del sistema

Para gestionar el acceso al DMA para la transmisión de paquetes hacia el filtro, se ha definido una variable compartida que será implementada a modo de semáforo, de forma que cuando esté a 1 se bloqueará el acceso al DMA hasta que la misma esté a 0, lo cual indica que no hay ningún paquete pendiente de ser transmitido y se permite el acceso a datos nuevos. Mediante esta variable compartida, se logra sincronizar las distintas tareas de que consta el *software* empotrado y el intercambio de información entre ellas.

La tarea de recepción de paquetes es la encargada de recibir la información y almacenarla en un *buffer* de forma previa a la transmisión hacia el bloque con el fin de evitar pérdidas de información, definir el tamaño del paquete y actualizar el valor del semáforo. Si se reciben paquetes de datos mientras el semáforo se encuentra cerrado, estos serán almacenados en otros *buffers* creados para dicho fin y así evitar que dicha información se pierda. La variable que indica el tamaño del paquete sirve para adaptar el

tiempo del proceso de copia para paquetes más pequeños y evitar el consumo de tiempo y cómputo innecesario en partes vacías del *buffer*. Finalmente, se emplea la función *pbuf_header()* para poder iterar el paquete de datos desde su comienzo, ya que la librería lwIP por defecto comienza a procesar desde el *payload* de la capa de transporte.

```
err_t recv_callback(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
	/* do not read the packet if we are not in ESTABLISHED state */
	if (!p) {
		tcp_close(tpcb);
		tcp_recv(tpcb, NULL);
		return ERR_OK;
	}
	/* indicate that the packet has been received */
	tcp_recved(tpcb, p->len);

	/* echo back the payload */
	/* in this case, we assume that the payload is < TCP_SND_BUF */
	if (tcp_sndbuf(tpcb) > p->len) {

		//MAC Ethernet + IP header + TCP header
		pbuf_header(p, PBUF_LINK_HLEN + PBUF_IP_HLEN + PBUF_TRANSPORT_HLEN);
		payload = p->payload;
		size = p->len;
		sem = 1;
		count++;

		err = tcp_write(tpcb, p->payload, p->len, 1);
	} else
		xil_printf("no space in tcp_sndbuf\n\r");

	/* free the received pbuf */
	pbuf_free(p);

	return ERR_OK;
}
```

Por otra parte, la función que se encarga de comunicar el DMA con el bloque IP se encuentra realizando un *polling* de forma continua del valor del semáforo. Cuando éste se encuentra activo, lo que indica que hay un paquete de datos listo para ser transmitido, se copia el valor del mismo al *buffer* de transmisión del DMA.

Una vez realizada la transmisión, se lee el resultado de la consulta. Esta tarea se realiza empleando, al igual que en la primera iteración, la función *Get_Return()* de nuestro bloque, que es proporcionada por el BSP y permite leer directamente el valor almacenado en la posición concreta del mapa de memoria donde se encuentra la respuesta. Finalmente, el bloque de procesamiento toma la decisión oportuna sobre la

dirección IP de cada paquete analizado en función del resultado de dicha consulta y del registro de direcciones almacenado en su lista negra.

En la Figura 63 y la Figura 64 se muestra el resultado de dos consultas distintas, tanto en *hardware* (analizando el diagrama de señales proporcionado por el Hardware Manager) como en *software* (mediante la impresión de mensajes de estado en un *minicom*), forzando en una de ellas una respuesta negativa al tener el paquete entrante una dirección IP origen que no se encuentra almacenada en el filtro, y en la otra consultando sobre una dirección IP que sí se encuentra en nuestra lista negra. El resultado de la consulta, en ambos casos, se refleja en la señal AXI_RDATA.

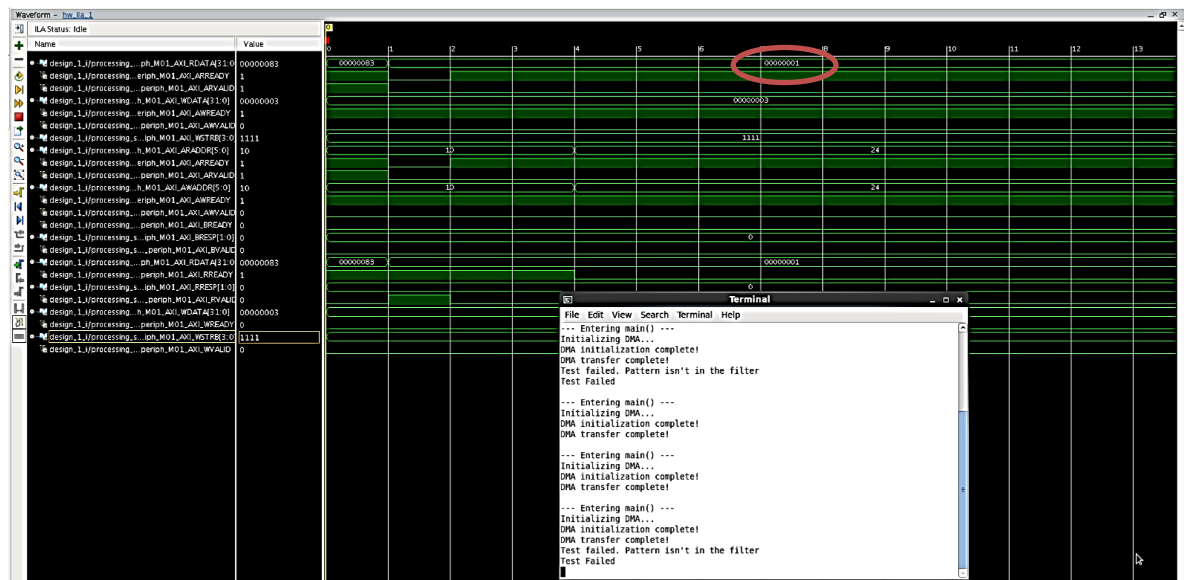


Figura 63. Consulta con respuesta negativa

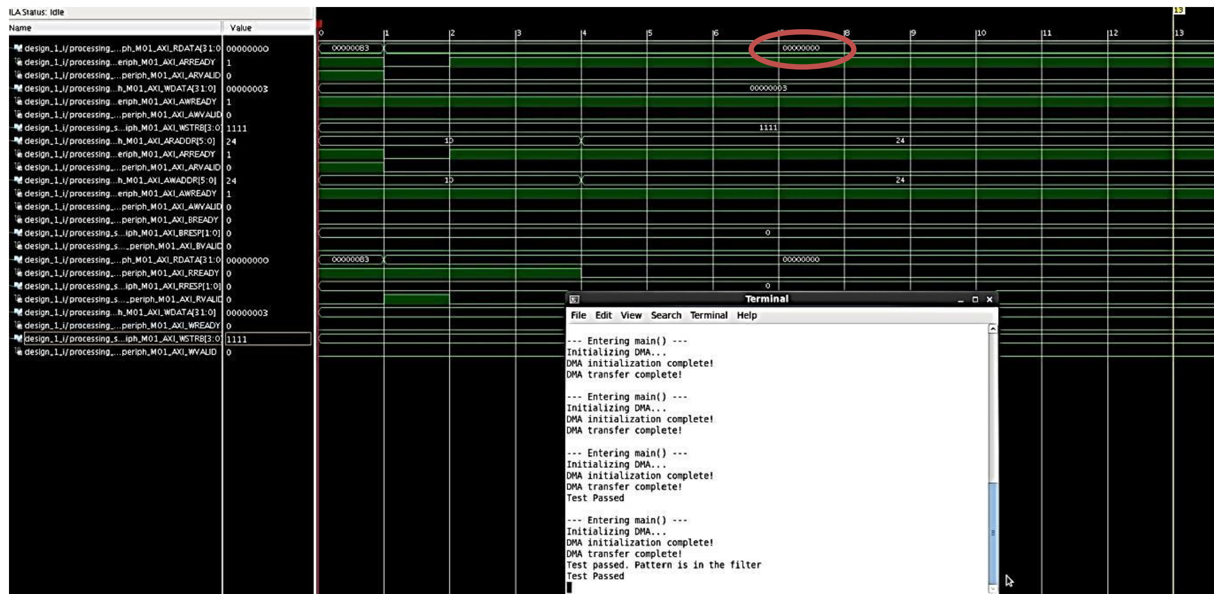


Figura 64. Consulta con respuesta positiva

7.4. Conclusiones

Tras haber descrito minuciosamente el proceso de diseño del bloque IP que integra la funcionalidad completa de un filtro Bloom con contador, en este apartado se ha abordado su integración con el resto de módulos necesarios para formar el sistema completo requerido para cumplir con la funcionalidad pensada, proporcionando detalles de la arquitectura definida y de su forma de trabajo. Tal como se ha descrito, se ha seguido una metodología de desarrollo basado en iteraciones, incrementando las prestaciones del diseño en cada una de ellas y siendo de esta forma más directa la detección de posibles errores, así como la depuración del diseño.

En relación a la metodología de diseño y en base a la experiencia adquirida con la herramienta Vivado HLS y una descripción algorítmica en C/C++, se ha comprobado la dificultad para tener control detallado sobre los protocolos de comunicación (AXI) en este entorno, al utilizar niveles de abstracción muy altos, sin control sobre el protocolo a nivel de ciclo de bus. La utilización de un lenguaje algorítmico sin información temporal supone una cierta desventaja en diseños con una cierta complejidad, dejando a procedimientos automáticos la generación de interfaces. Es preciso por tanto la estandarización de las

comunicaciones con el exterior, limitando los grados de libertad del diseñador. La utilización de SystemC y de la información temporal facilita el desarrollo del diseño del sistema ya que permite al diseñador definir y controlar al detalle la arquitectura de comunicación del sistema.

Capítulo 8. Fase de validación

En este capítulo se procede a recopilar los resultados relevantes que se extraen de la verificación completa del diseño, vinculados esencialmente al tiempo de ejecución tanto del bloque IP como de la plataforma completa, así como al consumo de recursos del sistema implementado sobre la FPGA.

8.1. Prestaciones del diseño a obtener

Los cálculos temporales concretos que se llevarán a cabo sobre el funcionamiento del bloque IP serán determinados tanto en *hardware* como en *software* para poder llevar a cabo una comparación entre ambos valores. Los valores concretos a determinar en el ámbito temporal serán:

1. Latencia del filtro en *hardware* para distintos tamaños de paquetes.
2. Latencia de la plataforma completa en *software*, desde la entrada de un paquete por la interfaz Ethernet hasta que el filtro devuelve una respuesta acerca de una consulta sobre dicho paquete.

Dichos valores serán leídos desde *hardware* utilizando la información temporal proporcionada por las herramientas de depurado Hardware Manager de Vivado, conectando dicho entorno con la placa de prototipado a través de su puerto JTAG [42]. Por otro lado, la latencia en *software* será calculada utilizando librerías temporales disponibles en el entorno de Xilinx, que permiten realizar el cálculo de tiempos de ejecución para bloques de código concretos.

Una vez hallados los parámetros anteriores, se aprovecharán para determinar las prestaciones del sistema, es decir, la capacidad de información que nuestra plataforma puede mover por unidad temporal.

Centrándonos finalmente en el consumo de recursos del bloque lógico por parte de nuestra plataforma durante la fase de implementación, resulta relevante la medición del factor de utilización de las *slices*, ya que determina cómo de buena ha sido la implementación realizada mediante el flujo de diseño del sistema electrónico. Dicho factor de utilización será medido en relación con el número de LUTs y de *flip-flops* de que se dispone en cada *slice*, cuya cantidad es de 4 y 8, respectivamente [29]. Dichos valores serán extraídos del informe de consumo de recursos proporcionado por Vivado una vez finalizada la etapa de implementación del diseño sobre la FPGA.

8.2. Configuración del banco de pruebas

A continuación, se comentará la forma de proceder para verificar que el sistema empotrado desarrollado ejerce las funciones necesarias para cumplir con la aplicación para la que fue diseñado. Para ello, se describe el conexionado y los equipos necesarios para realizar dicha validación, así como el proceso de generación del tráfico a analizar y el proceso de recogida de datos para su posterior estudio.

8.2.1. Diagrama de conexión

El sistema de prueba está compuesto por un ordenador portátil que se encargará de mandar paquetes de red, mediante el uso de un generador de tráfico, a la placa de prototipado ZedBoard a través de una conexión Ethernet punto a punto. En la Figura 65 se muestra el esquema de conexión utilizado.

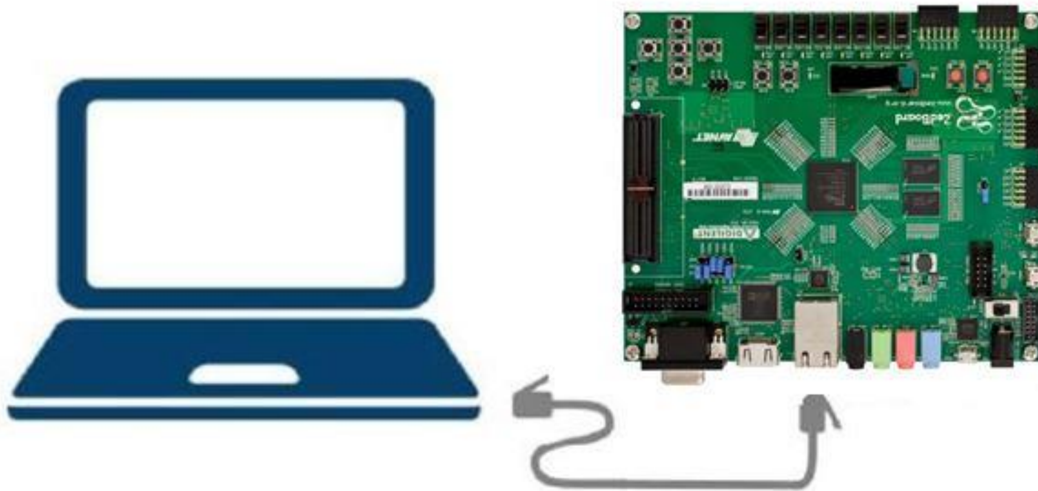


Figura 65. Esquema de conexión para validación del sistema empotrado

8.2.2. Metodología de verificación

Para verificar el correcto funcionamiento del sistema, se emplea un conjunto de paquetes previamente obtenidos de la red, pudiendo predecir de esta forma el comportamiento que ha de presentar el sistema al saber a ciencia cierta el valor de la dirección IP origen incluida. Se utilizarán paquetes de distinto tamaño para comprobar si este parámetro condiciona el tiempo de ejecución del sistema o si se comporta de forma lineal independientemente de dicha longitud.

La lectura de la latencia del sistema en *hardware* se realizará analizando las formas de onda de las señales proporcionadas por el Signal Viewer de Vivado HLS y el Hardware Manager de Vivado, y en *software* mediante la utilización de librerías específicas, tal como se ha mencionado con anterioridad.

8.3. Descripción de los equipos empleados

El *host* encargado de albergar los paquetes a partir de los cuales se realizarán las pruebas pertinentes al diseño se trata de un ordenador portátil con las siguientes características principales:

- Marca Acer y sistema operativo Ubuntu 10.04
- Procesador Intel Core 2 Duo a una frecuencia de 2.53 GHz
- 4 GB de memoria RAM
- Interfaz de red Gigabit Ethernet

Como se ha indicado anteriormente, la placa de prototipado empleada será una ZedBoard, la cual contará con la siguiente configuración durante el proceso de validación:

- Uso de un núcleo del ARM Cortex-A9
- Reloj del bloque de procesamiento a una frecuencia de 667 MHz
- Reloj de la parte lógica a una frecuencia de 190 MHz
- Interfaz Ethernet Gigabit

En la Figura 66 podemos observar el montaje y conexión del PC con la placa de prototipado, así como una captura de la forma de onda proporcionada por el ILA en un determinado instante del periodo de verificación para comprobar que el sistema empotrado se encuentra funcionando de forma correcta.

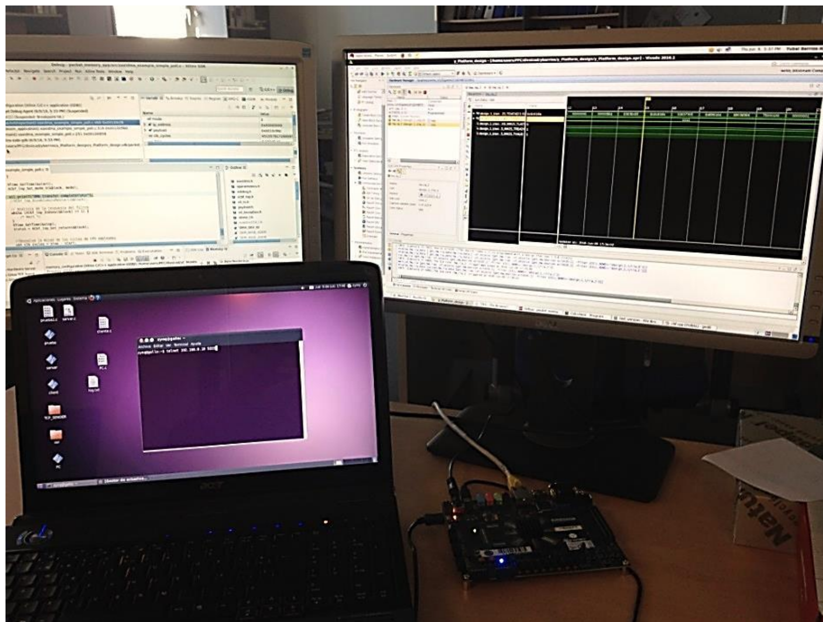


Figura 66. Instante del proceso de verificación del software con ILA en Vivado

8.4. Resultados obtenidos

El cálculo de la latencia del filtro, es decir, el tiempo que transcurre desde que se recibe un paquete de datos hasta que se devuelve una respuesta sobre el resultado de la consulta realizada, será realizado tanto en *hardware* como en *software*.

El cálculo en *hardware* se realizará, tal como se ha indicado anteriormente, analizando el diagrama de señales de la descripción RTL generada en Vivado HLS, situando un marcador temporal cuando comienza a pasarse un paquete de red al filtro a través de la interfaz AXI4-Stream y un segundo tras el momento en el que el mismo devuelve una respuesta sobre la consulta aplicada a dicho paquete a través de la interfaz AXI4-Lite.

La latencia del bloque será la diferencia temporal entre ambos marcadores, la cual es en nuestro caso de **216,758 ns**, tal como se aprecia en la Figura 67. En la misma figura podemos observar el correcto funcionamiento de la interfaz AXI4-Stream, pudiendo apreciarse como la señal TVALID se encuentra a nivel alto durante todo el periodo en el que se encuentra un paquete de datos disponible en la interfaz, mientras que la señal TREADY sólo se pone a nivel alto cuando se le solicita una consulta al filtro (modo 3), momento en el que se le pasa al filtro la IP extraída del paquete a través de la señal *ip_address* y a la que se le aplicará una consulta. Por último, resaltar que la señal TLAST se pone a nivel alto un breve intervalo temporal en el momento en el que un paquete de datos se termina de procesar.

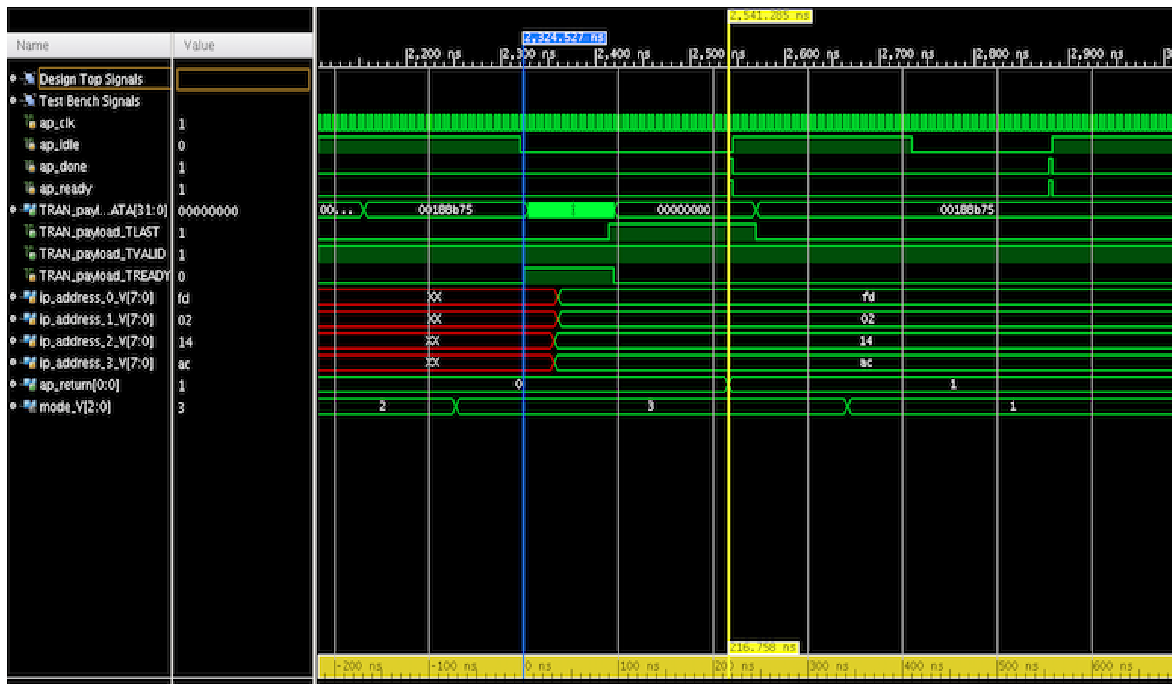


Figura 67. Cálculo de la latencia en hardware

Por otro lado en dicha figura también podemos observar un ejemplo del funcionamiento de la interfaz AXI4-Lite; en el momento en el que se activan los modos 1 o 2, correspondientes a la inserción y eliminación de patrones, se proporciona a través de la señal *pattern* la dirección IP a incluir o desechar de nuestra lista negra y, tal como se puede desprender del valor de la señal *return* (valor 0 durante la ejecución de dichas operaciones), las gestiones realizadas con el filtro se han realizado de forma totalmente satisfactoria.

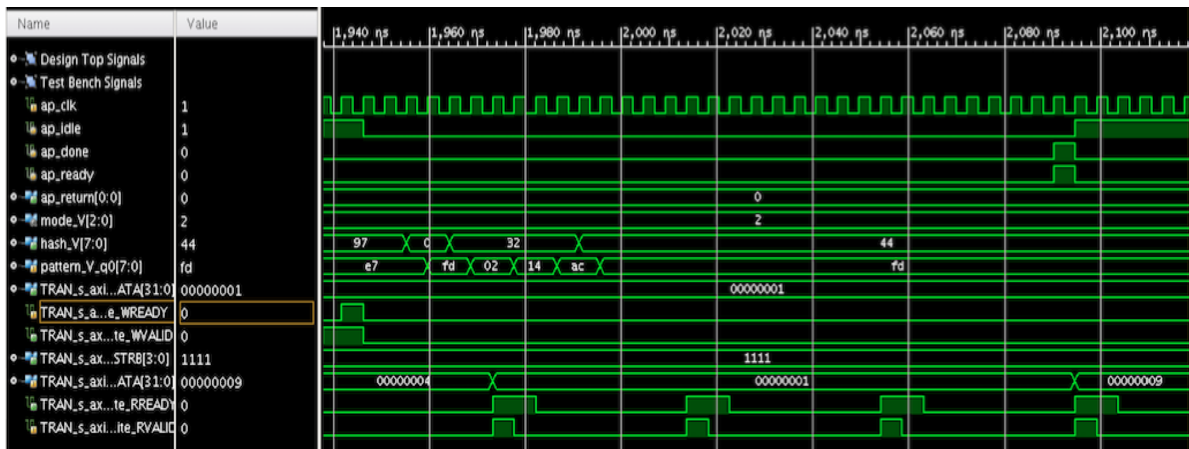


Figura 68. Verificación del funcionamiento de la interfaz AXI4-Lite

La medición de la latencia en *software* (ciclos de CPU consumidos para la realización de una consulta por parte del filtro) se realizará utilizando la librería *xtime_l.h,* propiedad de Xilinx y perteneciente al lenguaje de programación de alto nivel C++ en el entorno de diseño Vivado SDK, mediante la cual se pueden aplicar marcas temporales en el comienzo y el fin de un bloque de código determinado, calculando de esta forma cuántos ciclos de reloj tarda en ejecutarse el código incluido entre las mismas (hay que tener en cuenta que esta librería devuelve la mitad del número de ciclos empleados, teniendo que multiplicar por 2 posteriormente este valor)[55].

En nuestro caso, se aplicará a la función de consulta desde el instante en el que entra un paquete en el bloque IP hasta el momento en el que se devuelve una respuesta. Teniendo en cuenta que la frecuencia de funcionamiento del microprocesador empotrado empleado para el proceso de depuración y validación es de 666.66 MHz, el valor de latencia obtenido en *software* resulta ser de:

$$Latencia_{sw} = N_{ciclos} \cdot t_c = 325 \cdot \frac{1}{333 \cdot 10^6} = \mathbf{0.975\ \mu s} \qquad (3)$$

A partir de los resultados de latencia obtenidos, podemos concluir que la solución implementada en *hardware* es más eficiente en términos de tiempo de ejecución que su equivalente en *software.* Concretamente, estamos hablando de un incremento de las prestaciones casi **4 veces superior**. Esta diferencia radica principalmente en el hecho de

que el cálculo del *hash* se realiza en *hardware* de una forma prácticamente instantánea, mientras que en *software* requiere de un considerable tiempo de cómputo.

A continuación, pasamos a determinar el *throughput* del bloque IP, el cual podemos calcular conociendo su latencia en procesar un número de bits determinado. Teniendo en cuenta que nuestro bloque tarda 216.758 ns en procesar un paquete de datos de 168 bytes (1344 bits), obtenemos un *throughput* resultante de:

$$Throughput = \frac{1.344}{216{,}758 \cdot 10^{-9}} = \mathbf{6{,}2\ Gbits/s} \quad (4)$$

Si se quisiera aumentar este parámetro, podría plantearse la implementación de varios filtros Bloom en paralelo, analizando cada uno de ellos la dirección IP de un número determinado de paquetes de red de forma concurrente. Este número de bloques trabajando paralelamente únicamente estará limitado por las prestaciones en cuanto a recursos disponibles se refiere de la parte lógica del dispositivo que se emplee para la implementación del sistema.

Finalmente, el factor de utilización de las *slices* del diseño se pueden obtener de los datos desprendidos del informe proporcionado por Vivado reflejados en la Tabla 4, se obtiene que:

$$FU_{LUTs} = \frac{N_{LUTs}}{N_{slices}} = \frac{1947}{783} = \mathbf{2{,}49} \quad (5) \qquad FU_{FFs} = \frac{N_{FFs}}{N_{slices}} = \frac{2619}{783} = \mathbf{3{,}35} \quad (6)$$

Tal como se puede observar, el factor de utilización de las LUTs se encuentra algo por encima de la mitad del valor ideal (valor igual a 4), lo cual se debe a que buena parte de la lógica gestionada por la herramienta de implementación requiere de un tratamiento combinacional. Por otro lado, la lógica secuencial está reflejada por el uso de *flip-flops*, cuyo factor de utilización en nuestro caso se encuentra algo lejos del valor ideal de 8, lo cual puede ser provocado a que nuestro diseño demanda de distintas señales de control para su funcionamiento, permitiendo cada *slice* únicamente una señal de control para todos los *flip-flops* contenidos en la misma. Igualmente no se ha empleado una política de compactación agresiva durante la implementación del diseño.

8.5. Conclusiones

En este octavo apartado del documento se han comentado las etapas llevadas a cabo para validar nuestro diseño. Inicialmente, se indican los parámetros del sistema que resulta clave medir en relación con la aplicación para la que está destinado y cómo se van a obtener. A continuación, se describen los equipos empleados y el conexionado entre ellos.

Finalmente, se exponen los resultados obtenidos de latencia, tanto en *hardware* como en *software*, que resultan de la evaluación final de nuestro sistema, así como el *throughput* que puede alcanzar el diseño. Como aspectos adicionales, se incluyen transiciones empleando las interfaces AXI4-Stream y AXI4-Lite para comprobar su correcto funcionamiento, así como el cálculo del factor de utilización de las *slices*, parámetro que resulta relevante para comprobar el grado de aprovechamiento de las mismas por parte de Vivado durante la etapa de implementación.

Capítulo 9. Conclusiones y trabajos futuros

En este documento de resumen del proyecto realizado se ha descrito con minucioso detalle el proceso de diseño e implementación tanto del bloque IP como del sistema completo, así como las herramientas y la tecnología empleadas para completar el flujo de diseño completo de un sistema electrónico de esta magnitud. Una vez obtenidos los resultados derivados de dicha implementación, es turno de establecer una serie de conclusiones acerca de si se cumple con la funcionalidad de la aplicación definida al comienzo del trabajo.

9.1. Conclusiones del proyecto

Tras un análisis de los resultados derivados de la fase de validación del sistema y generalizando la solución (en la fase de validación no se pueden abordar todos los casos posibles, solo pruebas con un número específico de paquetes), podemos llegar a la conclusión de que la implementación de un NGFW como un sistema empotrado que cuente con un acelerador *hardware* basado en filtros Bloom con contador supone una solución bastante eficiente para el tratamiento de los paquetes de red entrantes en función de su dirección IP origen, especialmente si se compara con un *software* diseñado para el mismo fin, alcanzando tiempos de ejecución sustancialmente más rápidos en el primer caso al presentar una latencia más baja en lo que al procesamiento de información se refiere.

Además, estamos ante una solución que consume hasta dos órdenes de magnitud menos de potencia cuando se encuentre funcionando de forma autónoma que un PC destinado para la misma misión, incluyendo también la posibilidad de reconfigurar

dinámicamente la aplicación como en el caso de la implementación *software*, ya que bastará con interrumpir el proceso de consulta cambiando el modo de ejecución para poder incluir o eliminar direcciones IP de la lista negra de nuestro filtro. También hay que resaltar que esta solución *hardware* requiere un menor coste de implementación y mantenimiento que una solución implementada en *software* con la misma finalidad.

Finalmente, es necesario mencionar que este bloque presenta la ventaja de que puede ser reutilizado tanto en alto nivel (C++) como una vez implementado como bloque IP (RTL, en Verilog o VHDL), pudiendo integrarse en distintas plataformas y emplearse para soluciones diversas. Este bloque permite la integración de búsquedas rápidas en una plataforma que precisa de un preprocesado de paquetes a la mayor velocidad posible y que no precisa de interfaz de flujo de datos desde el dominio *software* para comunicarse con el exterior, pero presenta la flexibilidad necesaria para facilitar su configuración en tiempo de funcionamiento.

9.2. Trabajos futuros

Al tratarse de una solución *hardware* bastante flexible, podemos plantearnos el empleo de los filtros Bloom para examinar más campos de la cabecera de un paquete de red Ethernet más allá de la dirección IP origen (parámetro que ha sido elegido para el desarrollo de este trabajo), que se consideren únicos y sirven para identificar con certeza su procedencia o aplicación. Incluso es posible acceder a subcampos concretos dentro de los mismos (por ejemplo, en lugar de analizar la dirección IP entera, únicamente las dos primeras cifras para detectar la red de procedencia).

Esto se consigue disponiendo tantos filtros Bloom en paralelo como campos del *frame* se deseen analizar. Este funcionamiento en paralelo nos puede servir también para establecer diferentes tablas de búsqueda (por ejemplo, una lista blanca y una lista negra), las cuales tomen distintas decisiones sobre los mismos paquetes de red entrantes.

Como añadido, también se puede optar por extender este diseño al protocolo IPv6, cuya estructura de la cabecera es algo distinto a IPv4, protocolo de red empleado para el desarrollo de este trabajo.

Además, si el número de elementos a insertar en el filtro aumenta de forma considerable, habría que plantearse la implementación de una tabla de patrones en memoria rápida interna a la FPGA, de forma que si varios elementos se mapean en exactamente las mismas posiciones del vector del filtro no bastaría con saber si el elemento se encuentra o no en el conjunto. Mediante una simple comparación con dicha tabla podremos saber qué elemento concreto de dichas posiciones se desea procesar. A pesar de que esta opción supondría una latencia mayor que la versión implementada (debido a los numerosos accesos a memoria que habría que realizar) y que el consumo de recursos lógicos también incrementaría, obtendríamos una aplicación con tasa de falsos positivos inexistente.

Finalmente, como mejora en la experiencia de uso, se puede optar también por mejorar la interfaz de usuario para la interacción con el filtro, mediante un pequeño servidor web que permita acceder al control del bloque desde cualquier lugar siempre que se disponga de conexión a Internet.

Referencias

[1] Cisco Systems Inc., “Cisco Visual Networking Index : Global Mobile Data Traffic Forecast Update , 2010 – 2015,” *Growth Lakel.*, vol. 2016, no. 4, pp. 2010–2015, 2016.

[2] M. Ahmed, A. Naser Mahmood, and J. Hu, “A survey of network anomaly detection techniques,” *J. Netw. Comput. Appl.*, vol. 60, pp. 19–31, 2016.

[3] PWC, “Turnaround and transformation in cybersecurity - Key findings from The Global State of Information Security Survey 2016,” p. 45, 2016.

[4] Symantec, “Internet Security Threat Report 2016,” vol. 19, no. April, 2016.

[5] E. Pais, “Entrevista a John Lyons, experto en ciberseguridad.” 2016.

[6] A. Becher, D. Ziener, K. Meyer-Wegener, and J. Teich, “A co-design approach for accelerated SQL query processing via FPGA-based data filtering,” *2015 Int. Conf. F. Program. Technol.*, pp. 192–195, 2015.

[7] B. R. Raghunath and S. N. Mahadeo, “Network Intrusion Detection System (NIDS),” *2008 First Int. Conf. Emerg. Trends Eng. Technol.*, 2008.

[8] C. Fuchs, “The Privacy & Security Research Paper Series - Implications of Deep Packet Inspection (DPI) Internet Surveillance for Society,” p. 127, 2012.

[9] G. Ras, G. Note, J. Pescatore, and G. Young, “Defining the Next-Generation Firewall,” no. October, pp. 1–3, 2009.

[10] O. Entry, “What Makes a NextGenerationFireWall(NGFW) & Why Use It_ – Oversite Sentry.” 2015.

[11] F. Eberli, “Next Generation FPGAs and SOCs - How Embedded Systems Can Profit,” in *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*,

2013, pp. 610–613.

[12] B. Vega Del Pino, “TELECOMUNICACIÓN Y ELECTRÓNICA TRABAJO FIN DE GRADO Análisis de Paquetes de Datos TCP / IP en Plataforma Configurable Zynq,” 2015.

[13] Y. Zhou, “Hardware acceleration for power efficient deep packet inspection,” no. August, 2012.

[14] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” *Commun. ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[15] V. R. Knuth, D.E., Morris, J.H., & Pratt, J. H. Morris, Jr., and V. R. Pratt, “Fast Pattern Matching in Strings,” *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, 1977.

[16] J. Botwicz, P. Buciak, and P. Sapiecha, “Building dependable intrusion prevention systems,” *Proc. Int. Conf. Dependability Comput. Syst. DepCoS-RELCOMEX 2006*, pp. 135–142, 2007.

[17] V. Dimopoulos, J. Papaefstathiou, and D. Pnevmatikatost, “A memory-efficient reconfigurable Aho-Corasick FSM implementation for intrusion detection systems,” *Proc. - 2007 Int. Conf. Embed. Comput. Syst. Archit. Model. Simulation, IC-SAMOS 2007*, pp. 186–193, 2007.

[18] O. Erdem, “Tree-based string pattern matching on FPGAs,” *Comput. Electr. Eng.*, vol. 49, pp. 117–133, 2016.

[19] S. Arudchutha, T. Nishanthy, and R. G. Ragel, “String matching with multicore CPUs: Performing better with the Aho-Corasick algorithm,” *2013 IEEE 8th Int. Conf. Ind. Inf. Syst. ICIIS 2013 - Conf. Proc.*, pp. 231–236, 2013.

[20] R. Sidhu and V. Prasanna, “Fast regular expression matching using FPGAs,” *Field-Programmable Cust. Comput. Mach. 2001. FCCM ’01. 9th Annu. IEEE Symp.*, pp. 227–238, 2001.

[21] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, “Theory and Practice of Bloom Filters for Distributed Systems,” *IEEE Commun. Surv. Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[22] S. Dharmapurikar, M. Attig, and J. Lockwood, “Design and Implementation of a String Matching System for Network Intrusion Detection using FPGA-based Bloom Filters,” *Proc. 12 th Annu. IEEE Symp. FieldProgrammable Cust. Comput. Mach.*, vol. 1, no. 3, pp. 186–189, 2004.

[23] V. Akhlaghi, A. Rahimi, and R. K. Gupta, “Resistive Bloom Filters: From Approximate Membership to Approximate Computing with Bounded Errors,” *Des. Autom. Test Eur.*, pp. 1–4, 2016.

[24] H. Yu, R. Cong, L. Chen, and Z. Lei, “Blocking pornographic, illegal websites by internet host domain using FPGA and bloom filter,” *Proc. - 2010 2nd IEEE Int. Conf. Netw. Infrastruct. Digit. Content, IC-NIDC 2010*, pp. 619–623, 2010.

[25] Xilinx Inc., “Zynq-7000 All Programmable SoC Overview Zynq-7000 All Programmable SoC First Generation Architecture Processing System (PS) I / O Peripherals and Interfaces Zynq-7000 All Programmable SoC Overview Programmable I / O Blocks,” vol. 190, pp. 1–21, 2013.

[26] Xilinx Inc., “A Generation Ahead for Smarter Systems: 9 Reasons why the Xilinx Zynq-7000 all Programmable SoC Platfom is the Smartest Solution,” 2014.

[27] Xilinx Inc., “ug474 - 7 Series FPGAs Configurable Logic Block,” vol. 474, pp. 1–72, 2014.

[28] Xilinx Inc., “XILINX 7 Series FPGAs Overview - Product Specification,” vol. 180, pp. 1–12, 2015.

[29] Xilinx Inc, “Zynq-7000 All Programmable SoC Technical Reference Manual,” vol. 585, pp. 1–1836, 2014.

[30] Xilinx Inc., “7 Series FPGAs,” vol. 472, pp. 1–112, 2015.

[31] Avnet, “Zedboard (Zynq™ Evaluation and Development) Hardware User’s Guide,” no. January, 2014.

[32] Xilinx Inc., “Vivado Design Suite User Guide Getting Started,” vol. 901, pp. 1–120, 2013.

[33] Xilinx, “Vivado Design Suite User Guide: Design Flows Overview (v2014.1),” vol. 892, pp. 1–69, 2014.

[34] Xilinx Inc., “UltraFast Design Methodology Guide for the Vivado Design Suite (v2014.1),” vol. 949, pp. 1–322, 2014.

[35] Xilinx Inc., “Vivado Design Suite Tcl Command Reference Guide,” vol. 835, pp. 1–977, 2013.

[36] Xilinx Inc., “Vivado Design Suite User Guide for IP,” vol. 901, pp. 1–120, 2015.

[37] Xilinx Inc., “Vivado Design Suite IP subsystems,” vol. 901, pp. 1–120, 2013.

[38] Xilinx Inc., “new UG903 Using Constraints,” vol. 901, pp. 1–120, 2013.

[39] Xilinx Inc., “High-Level Synthesis,” vol. 902, pp. 1–672, 2016.

[40] Xilinx Inc., “Vivado Design Suite Implementation User Guide,” vol. 901, pp. 1–120, 2013.

[41] Xilinx Inc., “Vivado Design Suite Synthesis User Guide,” vol. 901, pp. 1–120, 2013.

[42] Xilinx Inc., “Vivado Design Suite User Guide Programming and Debugging,” vol. 901, pp. 1–120, 2015.

[43] Xilinx Inc., “LogiCORE IP Serial,” pp. 1–189, 2013.

[44] Xilinx Inc., “Zynq-7000 All Programmable SoC Software Developers Guide,” vol. 821, 2015.

[45] J. P. Talledo, “Design and Implementation of an Ethernet Frame Analyzer for High Speed Networks,” no. Conielecomp, pp. 1–6, 2005.

[46] Xilinx Inc., “lwIP 1.4.0 Library (v2.1),” pp. 1–14, 2014.

[47] ARM Ltd., “AMBA AXI and ACE Protocol Specification,” pp. 1–306, 2011.

[48] Xilinx Inc., “AXI User Guide for Xilinx applications,” vol. 612, pp. 1–186, 2012.

[49] ARM Ltd., “AMBA 4 AXI4-Stream Protocol,” 2010.

[50] Xilinx Inc., “LogiCORE IP Processing System 7,” 2015.

[51] Xilinx Inc., “LogiCORE IP AXI Interconnect,” no. November, 2015.

[52] Xilinx Inc., “Axi DMA IP Vivado Design Suite,” no. November, pp. 1–98, 2015.

[53] B. R. Fredericks and W. River, “What is a Board Support Package?,” 2012.

[54] Xilinx Inc., “LogiCORE IP Tri-Mode Ethernet MAC,” pp. 1–210, 2013.

[55] Xilinx Inc., “SDK System Performance,” vol. 1145, pp. 1–79, 2016.

Presupuesto

Durante el transcurso de este trabajo, ha sido necesaria la utilización de diversos recursos, tanto humanos como materiales para el diseño del bloque IP y su integración en la plataforma, así como para la validación completa del sistema. En los siguientes puntos se recogen todos los costes asociados al desarrollo del proyecto.

1. Recursos Hardware

En la Tabla 5 se enumeran los distintos recursos *hardware* empleados, así como los costes asociados a los mismos, los cuales han sido calculados de forma aproximada en función de su ciclo de amortización y el periodo temporal empleado.

Tabla 5. Costes de recursos hardware

Recurso	Coste total	Tiempo de empleo	Coste estimado
ZedBoard	424,30€	52 horas	424,30€
Workstation de diseño y programación	1.800,00€	276 horas	299,28€
Laptop como generador de tráfico	700,00€	104 horas	43,86€
Cables y equipo auxiliar	40,00€	52 horas	40,00€
Total			**807,44€**

2. Recursos Software

De la misma forma, las licencias de los recursos *software* utilizados han sido empleadas para varios proyectos, no únicamente para este; los costes derivados son los que se muestran en la Tabla 6:

Tabla 6. Coste de recursos software

Recurso	Tipo de licencia	Coste de licencia	Mantenimiento anual
Xilinx Vivado Suite	Universitaria	Donación	214,00€
Entorno Eclipse para programación en C	Pública	-	-
Microsoft Office 2013	Empresa	269,00€	-
Total			**483,00€**

3. Recursos Humanos

En este apartado, se ha calculado el coste por hora en función del montante anual que supone tener contratado a un ingeniero de telecomunicaciones desempeñando tareas de investigador en proyectos[1].

Tabla 7. Coste de recursos humanos

Tarea	Coste/hora	Horas	Días	Coste total
WP1. Estudio de los algoritmos de búsqueda de patrones regulares	16,65€/hora	4	16	1.065,60€

[1] Según la resolución del BOULPGC del 4 de noviembre de 2010

WP2. Modelado, verificación y síntesis de alto nivel de los algoritmos de búsqueda de patrones regulares	16,65€/hora	4	16	1.065,60€
WP3. Integración del bloque acelerador en la plataforma de referencia	16,65€/hora	4	21	1.398,60€
WP4. Validación	16,65€/hora	4	18	1.198,80€
WP5. Memoria y presentación	16,65€/hora	4	10	666,00€
Total				**5.394,60€**

4. Material fungible

En este apartado se incluyen los gastos derivados del empleo de las impresoras (tóner y papel) y demás material de papelería, así como de los discos para copias de seguridad. El coste total estimado es de 150 €.

5. Coste de edición del proyecto

El coste de edición de la memoria del proyecto, incluyendo tanto el número de folios empleados como los gastos derivados de su impresión y encuadernación, ascienden a un total de 80 €.

6. Coste total del proyecto

Finalmente, la suma total de los costes anteriormente mencionados da como resultado el coste total del proyecto, el cual se recoge en la Tabla 8:

Tabla 8. Coste total del proyecto

Recursos	Coste
1. Recursos hardware	807,44€
2. Recursos software	483,00€
3. Recursos humanos	5.394,60€
4. Material fungible	150,00€
5. Coste de edición del proyecto	80,00€
Subtotal	6.915,04€
IGIC (7%)	484,05€
Coste total del proyecto	**7.399,09€**

D. Yubal Barrios Alfaro declara que el presupuesto del presente proyecto asciende a siete mil trescientos noventa y nueve euros y nueve céntimos (7.399,09 Euros).

Las Palmas de Gran Canaria, a 13 de junio del 2016

Fdo.: Yubal Barrios Alfaro

Pliego de condiciones

Tanto el procedimiento llevado a cabo como los resultados generados por la realización de este proyecto son válidos para los recursos que se citan a continuación:

1. Recursos hardware

- ZedBoard Zynq-7000 ARM/FPGA SoC Development Board

2. Recursos software

Workstation de diseño y programación:

- SO: Red Hat Versión 6

Herramientas de diseño y síntesis:

- Vivado HLS 2016.1
- Vivado Design Suite 2016.1

3. Recursos de edición del proyecto

- Microsoft Word Office 2013 sobre Windows 10/Mac OS X El Capitán