UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

# Máster en
# Tecnologías de Telecomunicación


## Trabajo Fin de Máster

## Desarrollo de una plataforma virtual de un acelerador hardware FPGA para DPI

Autor: Sonia Raquel León Martín
Tutor(es): Pedro Pérez Carballo
Antonio Núñez Ordóñez
Fecha: Julio de 2018

IUMA
t +34 928 451 086
f +34 928 451 083
iuma@iuma.ulpgc.es
www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria
Versión: 5. – 26 julio 2018

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

# Máster en
# Tecnologías de Telecomunicación


# Trabajo Fin de Máster

## Desarrollo de una plataforma virtual de un acelerador hardware FPGA para DPI

## HOJA DE FIRMAS

| | | | |
|---|---|---|---|
| **Alumna:** | Sonia Raquel León Martín | Fdo.: | LEON MARTIN SONIA RAQUEL - 42221387B<br>Firmado digitalmente por LEON MARTIN SONIA RAQUEL - 42221387B<br>DN: C=ES, SERIALNUMBER=IDCES-42221387B, G=SONIA RAQUEL, SN=LEON MARTIN, CN=LEON MARTIN SONIA RAQUEL - 42221387B<br>Razón: Soy el autor de este documento<br>Ubicación:<br>Fecha: 2018-07-26 12:55:44<br>Foxit Reader Versión: 9.1.0 |
| **Tutor:** | Pedro Pérez Carballo | Fdo.: | PEREZ CARBALLO PEDRO FRANCISCO - 42163642L<br>Firmado digitalmente por PEREZ CARBALLO PEDRO FRANCISCO - 42163642L<br>Fecha: 2018.07.26 13:16:01 +01'00' |
| **Tutor:** | Antonio Núñez Ordóñez | Fdo.: | NUÑEZ ORDOÑEZ ANTONIO - 50278395G<br>Firmado digitalmente por NUÑEZ ORDOÑEZ ANTONIO - 50278395G<br>Fecha: 2018.07.26 19:15:03 +01'00' |
| **Fecha:** | **Julio de 2018** | | |

IUMA
t +34 928 451 086
f +34 928 451 083
iuma@iuma.ulpgc.es
www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Versión: 5. – 26 julio 2018

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

# Máster en
# Tecnologías de Telecomunicación


## Trabajo Fin de Máster

## Desarrollo de una plataforma virtual de un acelerador hardware FPGA para DPI

## HOJA DE EVALUACIÓN

**Calificación:** .....

**Presidente** Fdo.:

**Secretario** Fdo.:

**Vocal** Fdo.:

**Fecha: Julio de 2018**

IUMA
t +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira
f +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

# Agradecimientos

A Pedro Pérez Carballo, por su tiempo, por su dedicación, por confiarme este proyecto, por cada charla que me ha hecho crecer de manera personal y profesionalmente.

A mis padres y mi hermana, por confiar siempre en mí, por su apoyo constante, por escucharme aunque no entendieran, por sus palabras de ánimo en los momentos más duros y por no permitir que me rindiera.

A los que estuvieron aquí, a los que no, también.

# Resumen

En este trabajo se realiza la arquitectura en TLM-2.0 de un sistema de captura y filtrado de paquetes TCP/IP, que incluye un bloque destinado a eliminar las cabeceras de dichos paquetes Ethernet, con objeto de procesar únicamente su carga útil o *payload*. Posteriormente, se crea una plataforma virtual del sistema basado en el dispositivo *System on Chip* FPGA de la serie Xilinx Zynq-7000. El trabajo realizado consiste en estudiar el uso de lenguajes en alto nivel SystemC TLM-2.0, el uso y ventajas de las plataformas virtuales, la funcionalidad de la plataforma de referencia y, por último, la creación de su prototipo virtual con objeto de validar y analizar el funcionamiento del sistema de Inspección Profunda de Paquetes (DPI). El sistema DPI debe recibir tramas Ethernet, extraer su cabecera y determinar si se realiza su filtrado o no. En caso positivo, el paquete será enviado al motor de búsqueda para el análisis de la carga útil del paquete o reenviado por la interfaz de red Ethernet.

Una vez estudiado la metodología de diseño basado en plataformas virtuales, se pasa a estudiar el modelado a nivel de transacciones. En concreto, se profundiza en el estándar SystemC TLM-2.0, el cual se basa la herramienta de creación de plataformas virtuales usada en este trabajo. Esta herramienta, Vista de Mentor Graphics, ofrece un entorno para el desarrollo, integración, validación y optimización de los diseños complejos de los sistemas embebidos. Permite ejecutar *software* en los modelos de los procesadores que proporciona, además de los modelos funcionales del *hardware*.

Con Vista se realiza la arquitectura del DPI en TLM-2.0 y a través de *Virtual Prototype Kits*, se dispone del prototipo virtual del dispositivo Xilinx Zynq-7000. Se realiza la

integración de la plataforma y se desarrolla de la aplicación empotrada que se ejecuta en el procesador ARM Cortex A9 disponible en el SoC. Posteriormente, se genera la plataforma virtual, que es un ejecutable de la simulación derivado de la plataforma TLM creada en Vista pudiendo ser usada para realizar la simulación, depurado y análisis *hardware/software*.

# Abstract

This work presents the design on TLM-2.0 of a TCP / IP packet capture and filtering system, including a block destined to eliminate the headers of Ethernet packets, in order to process only its payload. With this architecture, its develops a virtual platform based on a Xilinx Zynq-7000 Series System on Chip FPGA device. The work carried out consists of studying the use of SystemC TLM-2.0 high-level language, the use and benefits of virtual platform, functionality of the reference platform and its main blocks, and, finally, creating its virtual prototype in order to validate and analyze the operation of the Deep Packet Inspection (DPI). The DPI system must receive Ethernet frames, extract its header, and determine whether it is filtered or not. If so, the packet will be sent to the search engine for payload analysis or forwarded to the network by the Ethernet network interface.

Once the design methodology based on virtual platform and the transaction-level modelling are studied. In particular, the work put emphasis on SystemC TLM-2.0 standard, because of the virtual platform are based on this standard. Vista Mentor Graphics is the tools used to create virtual platform which offers an environment to develop, integrate, validate and optimize complex designs of embedded systems.

In order to perform the DPI integration, with Vista create the TLM-2.0 DPI architecture, and with Virtual Prototype Kits provide configurable virtual prototype for Xilinx Zynq-7000. Finally, integration HW/SW is realized and its develop of the embedded application that runs on the ARM Cortex A9 processor available in the SoC. In this point, virtual platform is created which is a stand-alone simulation executable derived from the

TLM platform created in Vista and can be used to hardware/software simulation, debugging and analysis.

# Contenido

# Índice de figuras

# Índice de tablas

# Índice de códigos

# Acrónimos

| | |
|---|---|
| ACP | Accelerator Coherency Port |
| ADC | Analog-to-Digital Conversion |
| AMBA | Advanced High-Performance Bus |
| AMP | Asymmetrical Multiprocessing |
| APU | Application Processor Unit |
| AT | Approximately Timed |
| AXI | Advanced eXtensible Interface |
| BRAM | Block RAM |
| BSP | Board Support Package |
| CABA | Cycle Accurate Bus Accurate |
| CAN | Controller Area Network |
| CAS | Cycle Accurate Simulator |
| CLB | Configurable Logic Block |
| DDR | Double Data Rate |
| DFA | Deterministic Finite Automaton |
| DMA | Direct Memory Access |

| | |
|---|---|
| DMI | Direct Memory Interface |
| DPI | Deep Packet Inspection |
| DSP | Digital Signal Processor |
| EDIF | Electronic Design Interchange Format |
| EMIO | Extended MIO Interfaces |
| FF | Flip-Flop |
| FIFO | First In-Firs Out |
| FPGA | Field Programmable Gate Array |
| GIC | General Interrupt Controller |
| GMII | Gigabit Media-Independent Interface |
| GPIO | General Purpose Input/Output |
| GUI | Graphical User Interface |
| HDF | Hierarchical Data Format |
| HDL | *Hardware* Description Level |
| HLS | High-Level Synthesis |
| I2C | Inter-Integrated Circuit |
| IDS | Intrusion Detection System |
| IEEE | Institute of Electrical and Electronics Engineers |
| ILA | Integrated Logic Analyzer |
| IoT | Internet of Things |
| IP | Intellectual Property |
| IP | Internet Protocol |
| IPS | Intrusion Prevention System |

| | |
|---|---|
| ISS | Instruction Set Simulator |
| IUMA | Instituto Universitario de Microelectrónica Aplicada |
| JTAG | Joint Test Action Group Interface |
| LT | Loosely Timed |
| LUT | Look-Up Tables |
| lwIP | Lightweight IP |
| MAC | Media Access Control |
| MII | Media-Independent Interface |
| MIO | Multiplexed Input/Output |
| MM | Memory Mapped |
| NFA | Non-Deterministic Finite Automaton |
| NMS | Network Monitoring System |
| OCM | On-Chip Memory |
| PCI | Peripheral Component Interconnect |
| PHY | Physical Layer |
| PL | Programmable Logic |
| PPI | Private Peripheral Interrupt |
| PS | Programmable System |
| PV | Programmers' View |
| QoS | Quality of Service |
| RAM | Random Access Memory |
| RGMII | Reduced Gigabit Media-Independent Interface |
| ROM | Read-Only Memory |

| | |
|---|---|
| RTL | Register Transfer Level |
| SCU | Snoop Control Unit |
| SDIO | Secure Digital Input Output |
| SDK | *Software* Development Kit |
| SGI | *Software* Generated Interrupt |
| SGMII | Serial Gigabit Media-Independent Interface |
| SLCR | System-Level Control Registers |
| SMP | Symmetrical Multiprocessing |
| SoC | System on Chip |
| SPI | Serial Peripheral Interface |
| SPI | Shared Peripheral Interrupt |
| SRAM | Static Random-Access Memory |
| T | Timing |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| TEMAC | Tri-mode Ethernet MAC |
| TFTP | Trivial File Transfer Protocol |
| TLM | Transaction-Level Modeling |
| TTM | Time To Market |
| UART | Universal Asynchronous Receiver-Transmitter |
| USB | Universal Serial Bus |
| VCD | Value Change Dump |
| VHDL | VHSIC *Hardware* Description Language |

# Capítulo 1. Introducción

En este capítulo se exponen las razones por las que surge la necesidad de desarrollar este proyecto, incluyendo sus objetivos y la estructura del documento.

## 1.1 Introducción

La complejidad de los circuitos integrados se ha ido incrementado exponencialmente, tal y como se formuló en la Ley de Moore [1], integrando millones de componentes con geometrías cada vez más reducidas. Debido a esto, los últimos *System on Chip* (SoC) alcanzan niveles de integración equivalentes a millones de puertas lógicas.

El método de diseño dominante basado en el nivel de transferencia de registros (RTL), presenta problemas de escalabilidad a la complejidad de los diseños actuales. Por ello, la implementación de algoritmos complejos en sistemas electrónicos se ha convertido en una ardua tarea en la que se requiere planificar las operaciones, definir las interfaces y decidir la arquitectura de comunicación. Todo esto se resume en que usar métodos de diseño que utilicen niveles de abstracción superior, ya sea a nivel algorítmico o de sistema supone una ventaja competitiva para poner el producto en el mercado (TTM) [2].

En este nuevo nivel de abstracción, superior a RTL, los detalles temporales y estructurales se abstraen y la comunicación entre los módulos se denomina transacción, siendo por lo tanto este nivel de abstracción el de modelado a nivel de transacciones (TLM). Este modelado se basa en lenguajes de alto nivel para la descripción de sistemas, como

C/C++/SystemC. Los componentes se modelan como módulos con un conjunto de procesos que calculan y representan su comportamiento. Estos módulos intercambian información a través de un canal abstracto modelado en alto nivel. Se implementan interfaces en los canales para encapsular los protocolos de comunicación, por los cuales los procesos pueden acceder para establecer comunicación. La principal característica por tanto es separar las comunicaciones del procesamiento, con lo que cada parte del sistema se puede evolucionar por separado (Figura 1).

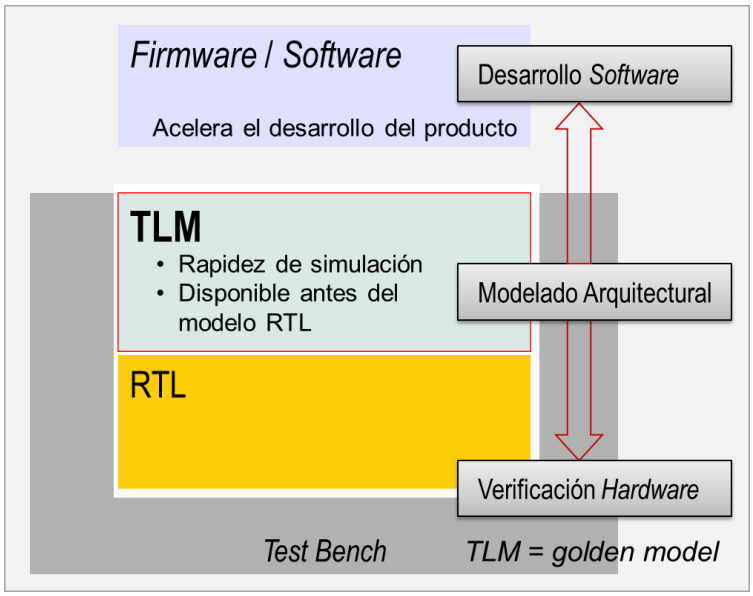



**Figura 1: Utilización de modelos TLM (adaptada de** [3]**)**

## 1.2 Diseño a nivel de sistema

Con el objetivo de hacer frente a la complejidad de los SoC que existen en la actualidad, en los que se incluyen múltiples núcleos de procesamiento, surge el diseño a nivel de sistema. Se basa en tres pilares fundamentales:

1. **Abstracción**: de igual manera que los diseños se abstrajeron del nivel de células estándar a lenguajes de descripción hardware con el RTL, se propone trabajar con niveles de abstracción superior al RTL hasta llegar al nivel de sistema.
2. **Separación**: separar los diversos aspectos del diseño, más allá de *hardware* y *software* para permitir exploraciones efectivas de la arquitectura.
3. **Refinamiento**: para reducir la complejidad de los diseños, se permite el refinamiento progresivo del sistema desde una descripción abstracta hasta la

implementación. Este proceso se consigue de manera automática a través de herramientas de síntesis de alto nivel que transforman una descripción algorítmica (C/C++/SystemC) a una implementación en RTL.

## 1.3 Plataformas virtuales

Una plataforma virtual o un prototipo virtual es un modelo *software* escrito en SystemC a nivel de transacciones que representa un modelo funcional multinivel y multidominio de un sistema embebido que consta tanto de parte *software* como de parte *hardware*, es decir, incluyendo procesador, buses, memorias, periféricos y la funcionalidad de la parte *hardware*, siendo su comportamiento funcional equivalente al del dispositivo real.

Con el modelado a nivel de transacciones se consigue aumentar la velocidad de simulación para ejecutar modelos de sistemas complejos. El aumento del nivel de abstracción permite crear modelos con menos esfuerzo, realizar cambios más rápidos al contener menos detalles y crear nuevas soluciones en menor tiempo. Todo ello facilita la realización de un estudio más amplio del espacio de diseño.

Las plataformas virtuales aportan diversas ventajas. En primer lugar, facilita la paralelización del diseño *hardware* y *software* y reduciendo el esfuerzo de desarrollo *software*, así como el proceso de verificación. Otra ventaja que supone es la capacidad de realizar análisis a la arquitectura al completo de manera más rápida y con menor esfuerzo, detectando y corrigiendo los cuellos de botellas de la arquitectura, tanto para los elementos de procesamiento como para las comunicaciones del sistema, o analizando el consumo total de potencia de cara a su utilización en entornos con requisitos energéticos exigentes, por ejemplo, sistemas en movilidad. Además, el uso de plataformas virtuales supone la reducción de costes derivados de la utilización de sistemas de prototipado y desarrollo [4].

## 1.4 Antecedentes

El IUMA ha desarrollado diferentes plataformas y prototipos basados en FPGA para aplicaciones DPI para el análisis del tráfico en la red. La realización de estas plataformas

permite gestionar de forma consistente los problemas asociados al crecimiento actual del volumen de datos en Internet. Por una parte, se puede identificar los problemas asociados a la calidad del servicio (QoS), tratando de dar prioridad a aquellos servicios que precisan una comunicación con requisitos de tiempo real. Otro de los problemas identificados hace referencia a la seguridad de la red, de la información perteneciente a los usuarios y de su disponibilidad [5].

Para solventar los problemas de seguridad se pueden seguir diferentes estrategias de análisis del tráfico, disponiendo de sistemas equipados con cortafuegos o *firewalls* que analizan parámetros orientados a la conexión (nivel de red), tratando de evitar que los equipos conectados detrás del cortafuego reciban conexiones desde equipos no autorizados. Para ello se analizan las cabeceras del paquete TCP/IP y se actúa (dejar pasar o desechar) sobre el paquete en función de un conjunto de reglas pre-configuradas [6]. Existen diferentes soluciones para problemas concretos tales como detección de intrusiones (IDS), prevención de intrusiones (IPS), monitorización de seguridad en la red (NSM) o procesamiento offline de los datos capturados para realizar un análisis forense de lo sucedido en la red [7][8].

Otras soluciones también analizan el contenido de la carga útil, o *payload,* del paquete de datos para identificar patrones que pudieran ser indicio de un contenido malicioso del mismo o con objeto de realizar una clasificación del tráfico para determinar la calidad del servicio. Estas técnicas se conocen como Inspección Profunda de Paquetes o DPI (*Deep Packet Inspection*) [9].

Los sistemas DPI están conformados generalmente por un bloque de captura de paquetes, encargado de analizar la cabecera del paquete recibido y, dependiendo de su análisis, se envía o no a un bloque de análisis del *payload* en busca de un patrón determinado. Un DPI puede poseer múltiples bloques de análisis, realizando el despacho del *payload* a un bloque u otro, dependiendo del resultado generado al analizar la cabecera.

El análisis del *payload* a través de los sistemas DPI está basado en buscar en su contenido patrones predefinidos. Normalmente denominado como máquina de patrones, es uno de los bloques más importantes en estos sistemas. Es por ello, que diseñar un

algoritmo eficiente en cuanto a tiempo y consumo se convierte en una prioridad dado que la mayor parte del tiempo de ejecución del sistema es consumida por estos algoritmos [10]. Para realizar esta búsqueda se implementan arquitecturas basadas en sistemas de autómatas deterministas (DFA) o no deterministas (NFA) para búsqueda de expresiones regulares [11][12] o con algoritmos de búsqueda de patrones fijos, como es el caso de algoritmo de Boyer-Moore [13] o de Wu-Manber [14]. Igualmente se integran otras técnicas avanzadas del tipo *Machine Learning* [15].

Todas estas máquinas de patrones no presentan las prestaciones necesarias en las implementaciones *software*, creando cuellos de botella en el flujo de datos del sistema, por lo que es natural plantear la realización de un acelerador *hardware* para los sistemas DPI aprovechando las características de los dispositivos FPGA como aceleradores *hardware*. Esta tecnología permite realizar procesos en paralelo y además cuenta con la capacidad de reconfiguración, consiguiendo mejor rendimiento, reduciendo la latencia y potencia del sistema manteniendo los costes del sistema reducidos [11].

Las aplicaciones creadas en el grupo de trabajo se basan en un acelerador *hardware* basado en Zynq que contiene un bloque que implementa el algoritmo de Boyer-Moore para búsqueda de patrones. La plataforma presentada en [16] captura los paquetes provenientes de la red a través de un bloque Ethernet TEMAC y se envían al microprocesador de la plataforma, responsable de almacenar los paquetes en memoria y propagarlos a un bloque dedicado. Este último realiza la inspección del *payload* utilizando una implementación hardware del algoritmo de búsqueda de patrones de Boyer-Moore al que se le han añadido funciones lógicas y temporales y creando un sistema paralelo de búsquedas muy robusto. Sin embargo, este sistema presenta algunos inconvenientes derivado de la implementación *software* del *stack* TCP/IP que generan latencias elevadas que limitan el tráfico para altas tasas de transferencia (> 2.5 Gbps).

Con objeto de incrementar las prestaciones del sistema se ha desarrollado un bloque específico para capturar, filtrar y clasificar paquetes TCP/IP [17]. En [18] se integra el bloque de captura en la plataforma diseñada en el IUMA, consiguiendo así incrementar y optimizar sus prestaciones para soportar tráfico multi-Gigabit Ethernet (Figura 2).

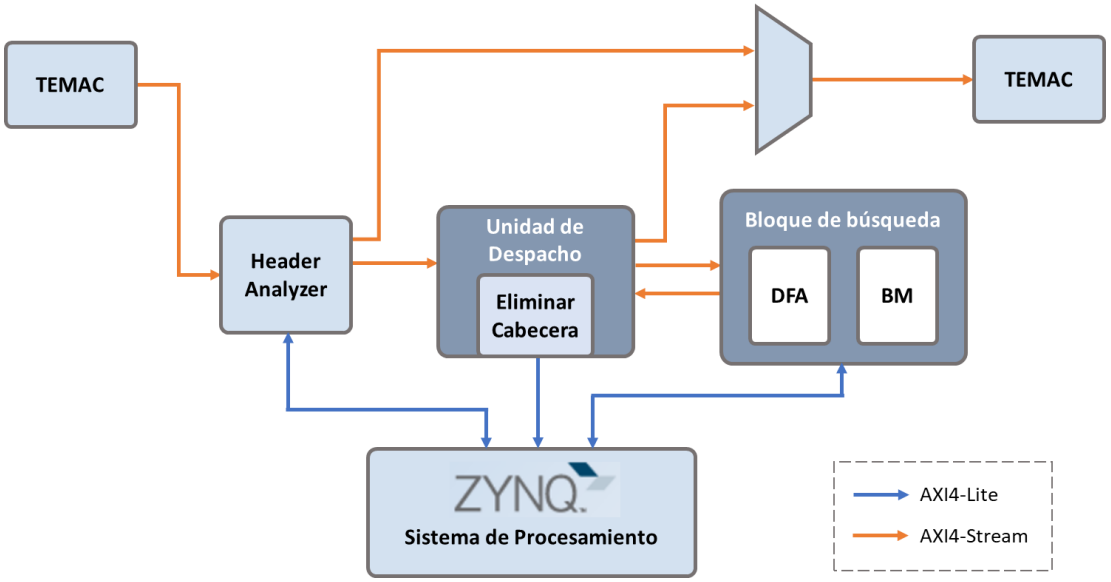



**Figura 2: Diagrama de bloques de la arquitectura DPI de** [18]

## 1.5 Planteamiento del problema

Debido a la complejidad de la arquitectura actual del DPI se propone el desarrollo de una plataforma virtual para facilitar la experimentación de diferentes arquitecturas y evaluar su rendimiento, anchos de banda de los buses, consiguiendo identificar los cuellos de botella que pueda presentar la arquitectura en etapas tempranas y tomar las pertinentes decisiones de diseño, facilitando así la exploración del espacio de diseño en alto nivel.

Las plataformas virtuales se han convertido en una herramienta aliada para el análisis de diseños complejos, permitiendo comprobar la funcionalidad de los diseños antes de ser implementados. Además, permite que el desarrollo del *software,* depurado y análisis comience antes y se realice al mismo tiempo que se desarrolla el *hardware*, pudiendo paralelizar las diversas tareas del proyecto. Todas estas ventajas se perciben en un aumento de la productividad en el desarrollo *software* reduciendo su esfuerzo, una disminución del tiempo del proyecto y por tanto en el TTM (Figura 3) [19].

Por tanto, este Trabajo Fin de Máster pondrá énfasis en la creación de una plataforma virtual que emule el comportamiento de la plataforma real basada en SoC FPGA Zynq. Esta plataforma facilita la validación de la plataforma incorporando nuevas funcionalidades sobre la plataforma básica, definida en SystemC a nivel TLM. Y por otra

parte permita el análisis y la cuantificación de parámetros claves de prestaciones, detectando cuellos de botella y optimizando el diseño.

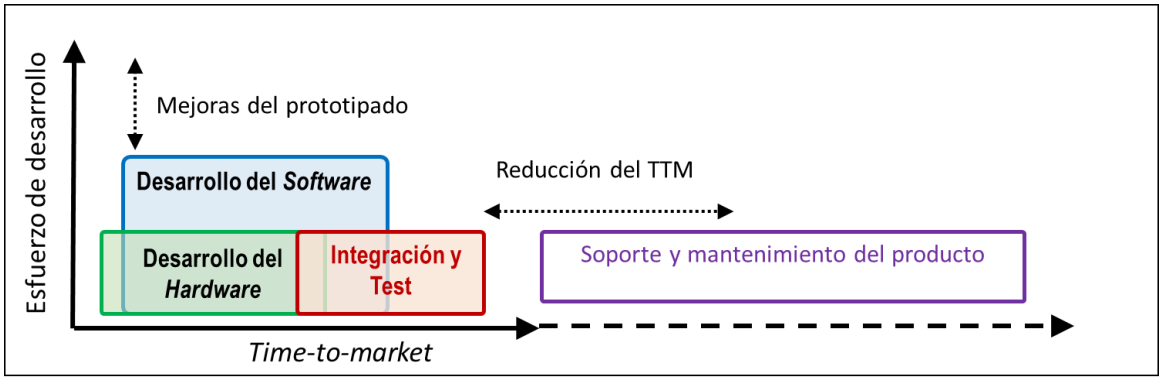



**Figura 3: Reducción del esfuerzo de diseño debido a las plataformas virtuales en el flujo de diseño. Adaptado de** [20]

Es común en las arquitecturas de las plataformas virtuales que la emulación del comportamiento del procesador se realice a través del QEMU y el subsistema modelado en SystemC corresponde con la parte *hardware* de un SoC. Al elevar el nivel de abstracción y usar el modelado a través de transacciones se consigue realizar la comunicación entre ambas partes a través de un puente definido en TLM-2.0. (Figura 4).

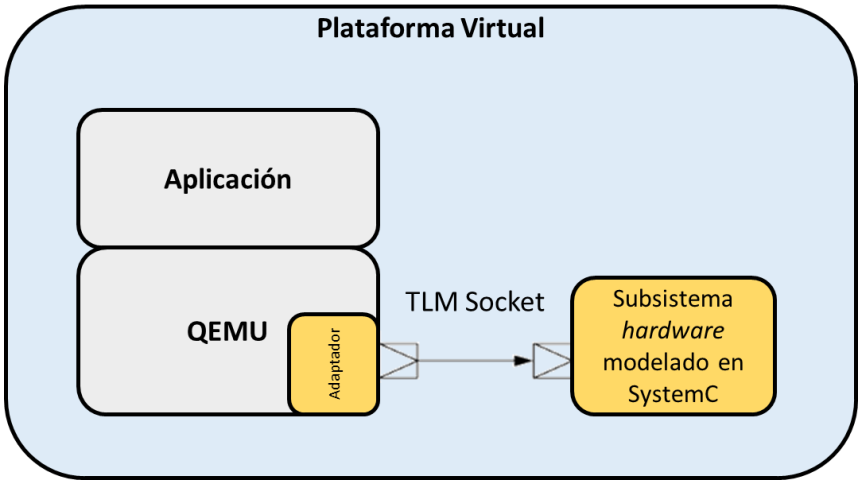



**Figura 4: Diagrama de bloques de plataforma virtual basada en SystemC. Adaptado de** [21]

## 1.6 Objetivos

El objetivo principal de este Trabajo Fin de Máster es la creación de una plataforma virtual basada SystemC TLM-2.0 para la verificación y optimización de la arquitectura de un sistema acelerador *hardware* para la inspección profunda de paquetes (DPI). Con el uso de

esta plataforma virtual se pretende realizar un análisis de la arquitectura actual con el que se puedan detectar posibles puntos críticos en la arquitectura, para optimizar la arquitectura, estudiando los límites de prestaciones especialmente en cuanto al tráfico de datos, con la ventaja de que este proceso se realiza antes de ser implementado en el dispositivo físico.

Los objetivos específicos que se proponen se pueden resumir en los siguientes puntos:

O1. Estudiar la metodología de diseño basada en plataformas virtuales.
O2. Estudiar el modelado a nivel de transacciones (TLM-2.0).
O3. Diseñar el modelo TLM-2.0 de la arquitectura del sistema.
O4. Realizar la simulación funcional en alto nivel.
O5. Obtener resultados y realizar comparaciones con la plataforma implementada.
O6. Documentar el proyecto mediante una memoria y exponerla.

## 1.7 Estructura del documento

Este documento se divide en nueve capítulos donde se describe el trabajo realizado durante el proceso del proyecto. En el primer capítulo se presentan los antecedentes del proyecto, se define el diseño a nivel de sistemas y las plataformas virtuales, se plantea los problemas del proyecto y se exponen los objetivos. En el segundo capítulo se describe el lenguaje de alto nivel SystemC y TLM-2.0 que son usados en este proyecto y son la base para modelar las plataformas virtuales. A continuación, en el tercer capítulo, se define el concepto de plataforma virtual y se analiza en el estado del arte las diversas plataformas virtuales que se han estudiado. En el cuarto capítulo se explica el entorno de Vista, la herramienta elegida para realizar el desarrollo de la plataforma virtual.

En el quinto capítulo se presenta la arquitectura del dispositivo que se emula en la plataforma virtual, Zynq-7000. Mientras que en el sexto capítulo se describe la arquitectura del sistema DPI y se procede al modelado en TLM-2.0 de los bloques. Conociendo el modelado del dispositivo y teniendo la arquitectura DPI modelada en TLM-2.0, en el capítulo 7 se crea la plataforma virtual que recoge los pasos de ensamblaje de la

arquitectura, desarrollo del *software*, comprobación de la funcionalidad de la plataforma y por último la creación de la plataforma virtual.

Con la plataforma virtual creada se puede realizar la exploración y los análisis de la arquitectura, esto se recoge en el capítulo ocho. Por último, el capítulo final es el de las conclusiones del proyecto exponiendo los resultados a los que se han llegado y presentando las líneas futuras que pueden mejorar el proyecto. Finalmente, se incluye la bibliografía en la que se apoya el documento.

# Capítulo 2. SystemC y TLM

Debido a la alta complejidad de los sistemas actuales han surgido lenguajes con un alto nivel de abstracción como SystemC y posteriormente la incorporación de la librería de *Transaction Level Modelling* (TLM).

## 2.1 SystemC

Los lenguajes de descripción *hardware* han tenido que migrar a un nivel de abstracción más alto para satisfacer las necesidades de complejidad de diseño de los sistemas electrónicos donde aparecen nuevos métodos de modelado y verificación. Lenguajes como SystemC o SystemVerilog surgen para cubrir esta necesidad de describir los sistemas electrónicos a un nivel de sistema.

SystemC es un lenguaje estándar IEEE 1666-2011 [22] orientado al diseño *hardware* que pretende unificar el modelo arquitectural con la implementación. Además, facilita el modelado y la verificación a un alto nivel, el refinamiento de módulos iterativos desde un nivel superior a uno inferior separando las especificaciones de comunicación desde su implementación y define un modelo de computación que puede ser personalizado (tiempos discretos, eventos, activación de procesos, etc.). SystemC aporta modularidad y noción temporal a la visión funcional aportada por C/C++.

SystemC un conjunto de librerías y macros implementadas en C++ que permiten diseñar y verificar sistemas *hardware/software* mixtos para diferentes niveles de

abstracción. Al estar SystemC basado en C++ permite que el proceso de aprendizaje de este lenguaje sea rápido, en el que se introducen algunas semánticas, pero manteniendo la sintaxis de C++.

### 2.1.1 Elementos de SystemC

Con el objetivo de crear modelos de sistemas, SystemC añade al lenguaje de C++ otras construcciones que están presentes en lenguajes de descripción *hardware* como VHDL o Verilog. A continuación, se da una breve introducción a los componentes de diseño de SystemC más importantes (Figura 5).

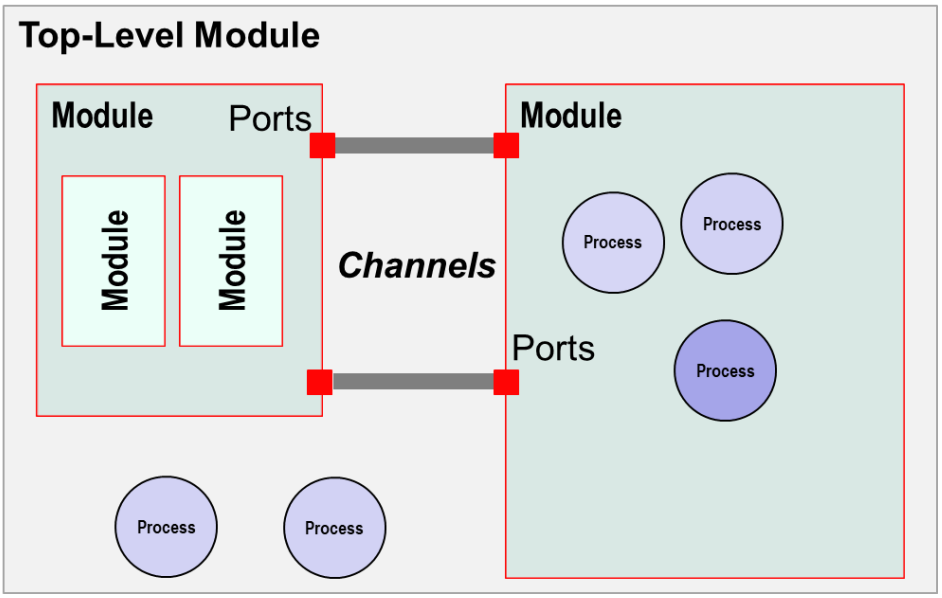



**Figura 5: Jerarquía en los modelos SystemC (adaptada de** [23]**)**

#### 2.1.1.1 Módulos

Son bloques de construcción básicos para la partición del diseño. Permite dividir un diseño complejo en bloques más pequeños y manejables que contienen representación de datos y algoritmos de otros módulos. Un módulo es una clase C++ que modela la estructura del diseño, encerrando en él la funcionalidad del sistema. Se puede usar cualquier código en C o C++. Un módulo puede contener puertos, procesos, canales y otros módulos.

#### 2.1.1.2 Procesos

Define una parte o todo el comportamiento del modelo de simulación. Debe estar contenido en un módulo. SystemC tiene tres tipos de procesos dependiendo de las diferentes necesidades de simulación: SC_METHOD, SC_THREAD y SC_CTHREAD.

El proceso SC_METHOD se ejecuta cuando se produce un evento de su lista de sensibilidad y su código siempre se ejecuta sin posibilidad de ser suspendido por lo que no se puede llamar a la función *wait()* directa o indirectamente.

El proceso SC_THREAD se ejecuta ante un evento, por lo tanto, debe contar con una lista de sensibilidad. Puede suspender su ejecución mediante la llamada de la función *wait(),* ya que el hilo recuerda el punto donde fue suspendido así como sus variables locales.

El proceso SC_CTHREAD es una variación de SC_THREAD diferenciándose de este en que es sensible a eventos de reloj, proporcionando código completamente síncrono, facilitando su simplificación. También proporciona un nuevo comportamiento de la función *wait()*, reanudando el código en el siguiente ciclo de reloj. Además, introduce la posibilidad de realizar el *reset* de señales.

#### 2.1.1.3 Eventos

Los eventos son objetos representados por la clase *sc_event* que determinan si la ejecución de un proceso debería activarse o reanudarse y cuándo. Un evento representa una condición que puede ocurrir durante la simulación. Además, controla la activación de los procesos que sean sensibles a dicho evento. Por lo tanto, se puede considerar un evento como un método de comunicación entre dos procesos, que típicamente incluye datos asociados.

#### 2.1.1.4 Interfaces

Son un conjunto de operaciones donde se especifica, no se implementa, la cabecera de cada operación (nombre, parámetros, qué devuelve). No especifica cómo se implementan o se definen las operaciones, facilitando la posibilidad de realizar diferentes implementaciones para una misma interfaz. Todas las interfaces derivan de la clase base abstracta *sc_interface* que proporciona una función virtual *register_port()* que se invoca

cuando un puerto se conecta a un canal a través de una interfaz. Los puertos y los canales se desarrollan en base a las interfaces para permitir la conectividad de estos con interfaces compatibles.

#### 2.1.1.5 Puertos

Proporcionan a los módulos la capacidad de conectarse y comunicarse con otros módulos. Se representan como objetos, haciendo más comprensible el código y permitiendo simplificar la sintaxis. Ello aporta que no es necesario que los puertos estén ligados a un módulo en la etapa de iniciación y se tenga que pasar como parámetro en el constructor. SystemC proporciona también puertos especializados: *sc_in, sc_inout* y *sc_out*. Los puertos *sc_inout* y *sc_out* pueden ser usados para inicializar las señales evitando que tomen un valor aleatorio en el momento de la simulación.

#### 2.1.1.6 Canales

Un canal permite la comunicación entre módulos y entre los procesos que están dentro de un módulo. Las restricciones en la conectividad y su acceso dependen exclusivamente del canal. SystemC tiene dos tipos de canales: primitivos y jerárquicos.

### 2.1.2 Simulación en SystemC

SystemC da soporte a un modelo de simulación basado en eventos. Aporta una implementación de referencia para el simulador, disponible en Accellera, conjuntamente con la librería de objetos que da soporte al lenguaje [24]. De esta forma se pueden realizar estimaciones temporales, de rendimiento del sistema y verificar que las condiciones impuestas al diseño se cumplen. Además, con la simulación se puede realizar el depurado del diseño y obtener la forma de onda de las señales en varios formatos.

El modelo de ejecución del motor de simulación presenta diferentes fases, desde la inicialización de los constructores de las clases principales (módulos, puerto, canales, etc.) y las fases principales de ejecución, controladas por *sc_start()*, donde se evalúan los procesos que son sensibles a los eventos modificados y se actualizan los canales que lo requieren. El proceso global se muestra en la Figura 6.

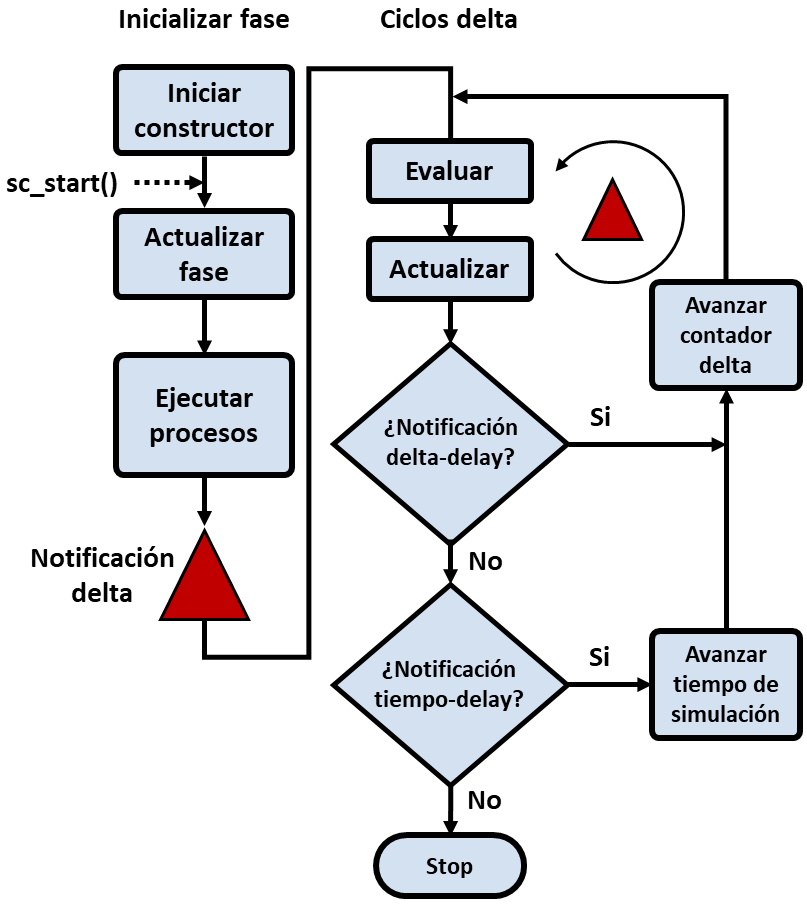



**Figura 6: Modelo de ejecución del simulador SystemC** [25]

## 2.2 Modelos TLM

La complejidad de los sistemas hace que la creación de sus modelos de referencia, incluso en altos niveles de abstracción, sean complejos, complicando el análisis del sistema en su globalidad.

El modelado a nivel de transacciones (TLM) aparece en 2007 con el fin de convertirse en un estándar de SystemC, cuyo reto es cambiar el estilo de modelado de las comunicaciones entre distintos bloques [26]. Esto se consigue separando las comunicaciones de la funcionalidad de los módulos. Las comunicaciones se modelan como canales en los que se pretenden abstraer las interfaces de comunicación, haciendo que no dependan del modelo del bus y creando modelos que puedan ser fácilmente interconectados entre sí (Figura 7). Las comunicaciones se realizan a través de las interfaces y los datos se envían a través de transacciones, la cual es una secuencia que incorpora datos de control y los datos a transferir entre dos o más componentes de un sistema.

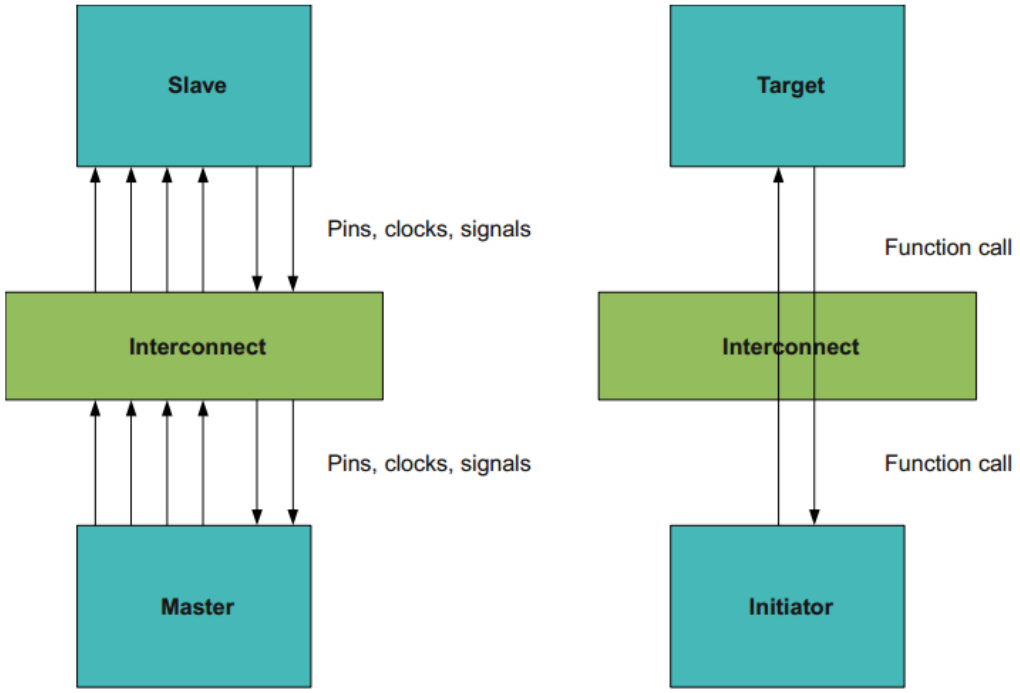



**Figura 7: Diferencias entre el modelado RTL y TLM** [27]

Al no depender los módulos del modelo del bus, TLM aporta ventajas significativas, incluyendo las siguientes:

a. Los diseños se vuelven más sencillos de modelar.
b. Se consigue mejorar la velocidad de las simulaciones al omitir detalles que no se ejecutan en tiempo de simulación.
c. Permite validar el software antes de tener una implementación hardware.
d. Reduce los tiempos a mercado del producto.

El uso de TLM permite realizar el desarrollo de las especificaciones de un sistema en alto nivel, pudiendo realizar análisis arquitecturales de manera rápida y eficiente. También permite desarrollar aplicaciones *software*, facilitando la creación de modelos virtuales de las plataformas para desarrollar, probar y analizar el código. Con todo esto, se consigue facilitar el proceso de integración *hardware*/*software*.

Es preciso indicar que la utilización de modelos TLM es un concepto que se plantea en los centros de diseño (*in-house*) ante la necesidad de probar los conceptos antes de su desarrollo. Sin embargo, estos diseños hechos a medida presentan problemas de eficiencia al centrarse en aspectos concretos del modelo, dificultades para su reutilización entre

proyectos y no están pensados para ser utilizados por terceros. Por tanto, la estandarización de los modelos TLM (conjuntamente con SystemC) facilita la creación de componentes reutilizables entre proyectos y define las fronteras entre los diferentes estilos de modelado ya estén orientados al desarrollo *hardware* o destinados a crear plataformas virtuales para el desarrollo *software*.

### 2.2.1 TLM-2.0

El objetivo del estándar TLM-2.0 es resolver los problemas asociados a la integración de distintos módulos o bloques permitiendo la reutilización del diseño en nivel elevados de abstracción. Aunque los diversos estándares de buses definen cómo deben interconectarse los componentes no se consigue resolver los tipos de comunicación o la estructura de datos a transmitir para que otros componentes que no estén conectados a través de un bus estándar puedan interactuar. Para conseguir este objetivo define una interfaz sólida que modele comunicaciones rápidas, consiguiendo así que los modelos puedan interactuar sin necesidad de cambiar su descripción.

Es por esto, por lo que el uso de TLM está recomendado en las siguientes etapas del proceso de un diseño:

1. Desarrollo *hardware*
    a. Análisis de rendimiento
    b. Análisis arquitectural
    c. Refinado e implementación
    d. Verificación funcional
2. Desarrollo *software*
    a. Desarrollo de la aplicación
    b. Análisis de rendimiento
    c. Análisis arquitectural
3. Integración *hardware/software*

### 2.2.2 Estilos de modelado de TLM-2.0

Los modelos TLM se pueden definir como modelos donde la comunicación entre los bloques se abstrae del nivel de pines que aportaba el estilo RTL usando ahora llamadas a función. En TLM-2.0 existen diversos estilos de modelado donde su principal diferencia es la precisión de la temporización de la comunicación [28]. La elección de un estilo u otro se basa en la acción que se va a realizar, que engloba tanto el *software* como el *hardware.* En la Figura 8 se muestra un resumen de los estilos que se emplean en cada caso.

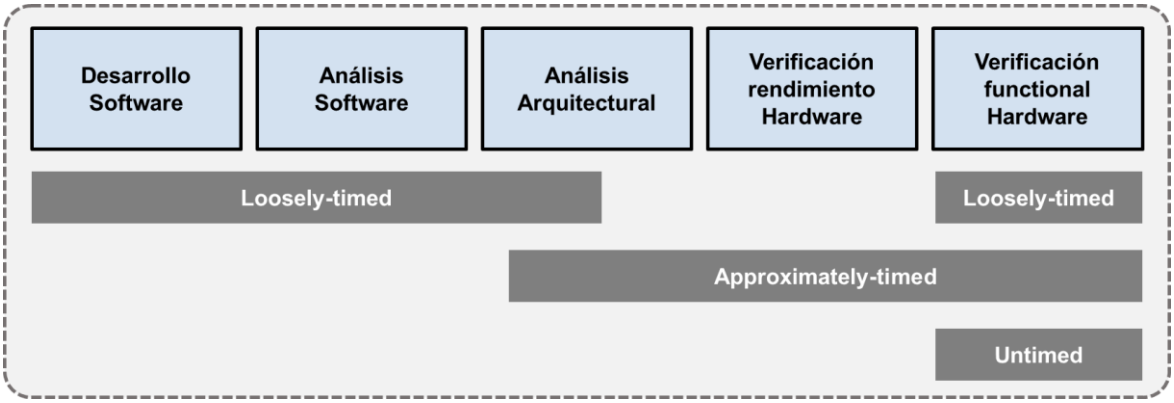



**Figura 8: Uso de los estilos de modelados según su uso (adaptado de** [3]**)**

#### 2.2.2.1 Untimed

Se representa un modelo funcional puro la cual realiza la comunicación a través de interfaces bloqueantes donde no hay mención explícita de tiempo o ciclos en las respuestas. Al no poseer detalles temporales la sincronización de las operaciones se realiza a través de eventos, *mutex,* etc. A este nivel no existen detalles propios de los dispositivos y la simulación del modelo puede completarse en tiempo cero.

#### 2.2.2.2 Loosely-timed

Con este estilo de modelado se hace uso de interfaces bloqueantes y se obtiene dos puntos de tiempo: cuando la transacción comienza y cuando acaba. Estos dos puntos temporales pueden ocurrir en el mismo tiempo de simulación o en distintos si el *Target* realiza una llama al *wait().*

Este estilo es apropiado para el desarrollo de *software* usando modelos de plataformas virtuales dado que soporta modelado de *timers* e interrupciones, suficiente para añadir un sistema operativo al dispositivo.

También soporta *temporal decoupling*, donde se permite que algunos procesos se ejecuten en su tiempo local sin avanzar el tiempo de simulación.

#### 2.2.2.3 Approximately-timed

Este estilo presenta más detalles de implementación a nivel de sistema y por tanto mayor tiempo de simulación. Hace uso de interfaces no bloqueantes en las que se tiene detalles temporales de todas las fases de una transacción. Por ello, es apropiado para realizar exploración de arquitecturas y análisis de rendimiento.

Generalmente, el modelado *approximately-timed* no puede soportar *temporal decoupling* debido a la precisión temporal. En su lugar, se ejecuta por separado cada proceso con el planificador de SystemC y se anota el retardo que genera. Cuando se crea el modelo *approximately-timed* se define este retardo que representa el tiempo de transferencia para la escritura o lectura y la latencia del *target.*

### 2.2.3 Componentes de TLM

TLM se basa en tres componentes básicos que define cómo los datos se transmiten, cómo se reciben y cómo es la estructura de estos datos (Figura 9). Estos componentes son:

1. ***Socket Initiator***. Es elemento encargado de comenzar el envío de los datos a otro módulo. Se basa en interfaces bloqueantes y no bloqueantes.
2. ***Socket Receiver.*** Elemento encargado de recibir los datos que provee otro módulo.
3. ***Generic Payload.*** Es una estructura que define como deben ser los datos para transacciones entre modelos TLM-2.0. Esta estructura cuenta con información sobre la acción que se está realizando, dirección de memoria, datos, etc.

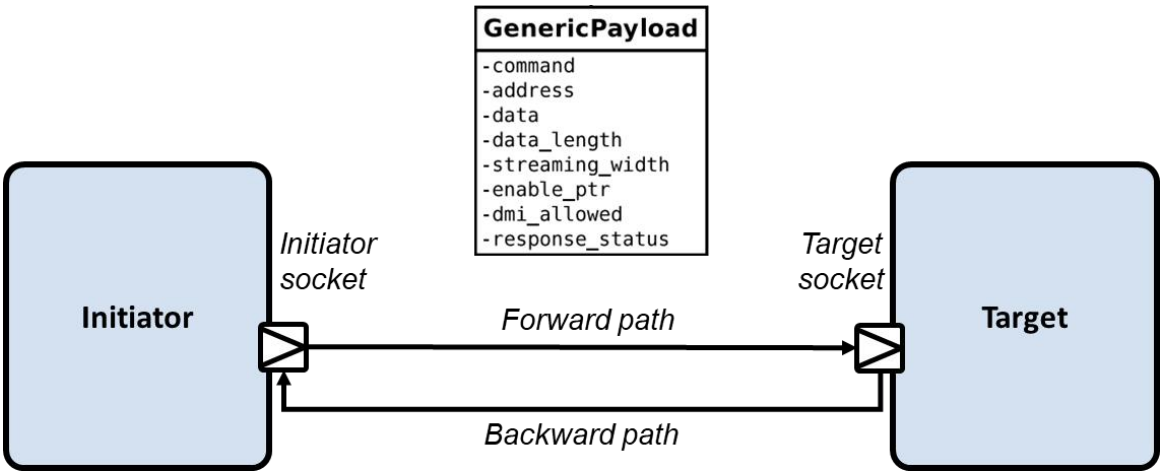



**Figura 9: Componentes básicos TLM (adaptado de** [3]**)**

## 2.2.4 Interfaces de transporte

TLM-2.0 define dos interfaces de transporte, bloqueantes y no bloqueantes, una interfaz directa a memoria (DMI) y una interfaz de depurado. Las interfaces de transporte son el mecanismo usado para realizar la transacción entre *initiator, target* e interconectores.

Las interfaces de transporte y el *payload* genérico están diseñados para ser usados en conjunto, permitiendo modelar rápidamente y de manera abstracta buses mapeados en memoria.

### 2.2.4.1 Interfaces bloqueantes

Las interfaces bloqueantes están destinadas a ser implementadas bajo el estilo de modelado *loosely-timed*. Estas interfaces realizan la comunicación entre dos módulos a través de la llamada a la función *b_transport()* la cual la realiza el *Initiator* y se implementa en el *Target*. El uso de esta interfaz tiene implícito dos puntos temporales, uno cuando se realiza la llamada a la función y otro cuando se devuelve a la función. La transacción termina con el retorno a la función y el *Target* puede insertar llamadas a *wait()* cuando responde a la transacción (Figura 10). El uso de este tipo de interfaces es apropiado cuando se desea que la transacción de un *initiator* se complete en una única llamada a la función de transporte.

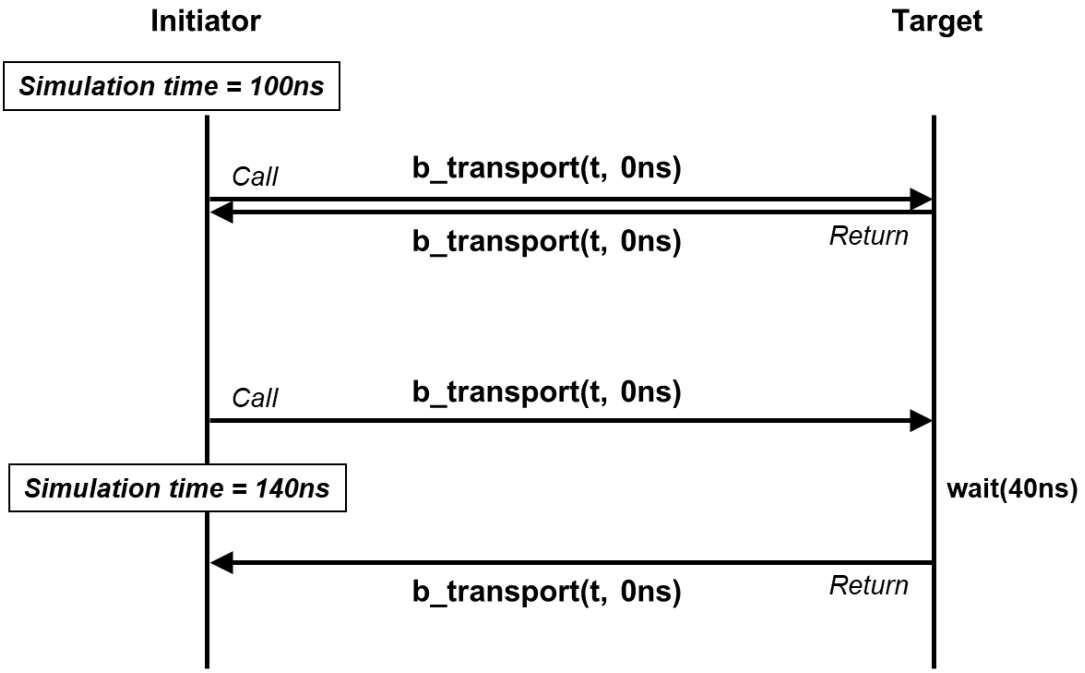



**Figura 10: Interfaz de transporte bloqueante** [26]

#### 2.2.4.2 Interfaces no bloqueantes

Las interfaces no bloqueantes están destinadas al estilo de modelado *approximately-timed.* La comunicación entre dos módulos en estas interfaces se realiza de manera asíncrona mediante la llamada del método de transporte. La comunicación se divide en múltiples fases (generalmente 4) donde en cada fase se tiene un punto temporal (Figura 11). Por lo tanto, cada llamada y retorno de esta en una interfaz no bloqueante puede corresponder a una fase.

Para restringir el número de puntos temporales a dos, se puede usar las interfaces no bloqueantes con el estilo de modelado *loosely-timed*, pero no suele ser recomendado ya que una particularidad de estas interfaces es en la utilización en modelos que aplican *pipeline* en sus transacciones.

A diferencia de las interfaces bloqueantes, estas interfaces cuentan con dos métodos de transporte: *nb_transport_fw()* y *nb_transport_bw().* La primera hace uso del *forward path* y la segunda del *backward path.* Ambos métodos no pueden realizar llamadas a la función *wait()*, directa o indirectamente. Como respuesta a los métodos de transporte se obtiene tres valores distintos:

- **TLM_ACCEPTED**: indica que el canal de retorno no está siendo usado y no se está modificando el estado de la transacción.
- **TLM_UPDATED**: indica que el canal de retorno está siendo usado y el estado de la transacción tiene que avanzar.
- **TLM_COMPLETED:** indica que el canal de retorno está siendo usado y la transacción se ha completado.

Cabe destacar que para completar una transacción no es obligatorio tener una respuesta de TLM_COMPLETED, la transacción se completa cuando se recibe la fase final (END_RESP).

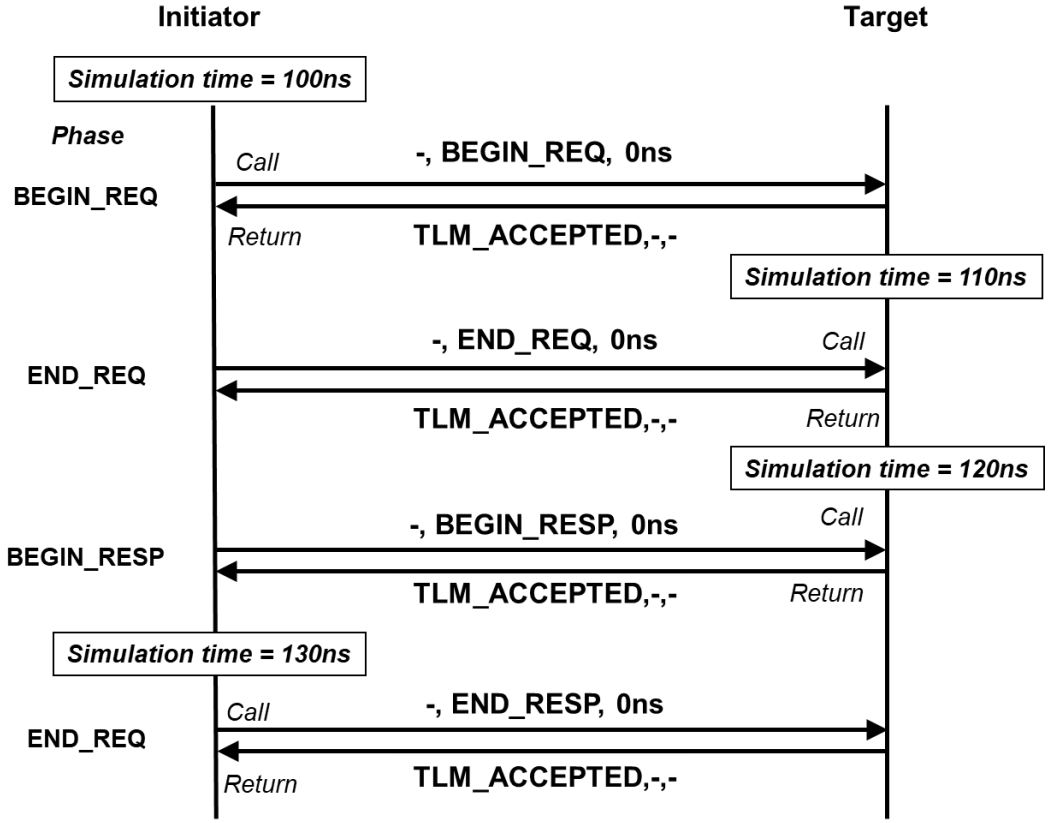



**Figura 11: Interfaz de transporte no bloqueante** [26]

### 2.2.4.3 *Direct Memory Interface*

TLM-2.0 proporciona una interfaz especializada denominada *Direct Memory Interface* (DMI) que provee un acceso directo a un área de memoria de un *Target* (Figura 12). Está orientado a modelos que acceden de manera constante a memorias durante el tiempo de simulación. Con esto se consigue acelerar el acceso a la memoria debido a que

se tiene acceso directo sin tener que pasar por una transacción gracias al uso de punteros. El *Target* puede invalidar un puntero en cualquier momento notificando al *Initiator* de que no puede usar el puntero para acceder a esa región de memoria del *Target.*

Existen dos tipos de interfaces: una para realizar la llamada desde el *initiator* al *target* a través del *forward path,* se usa para pedir acceso de lectura o escritura a una dirección; y la otra para hacer la llamada desde el *target* al *initiator* desde el *backward path,* usada por el *target* para invalidar el puntero. El tipo de transacción usado en esta interfaz es el *payload* genérico donde los atributos de comando y dirección son los únicos que se usan.

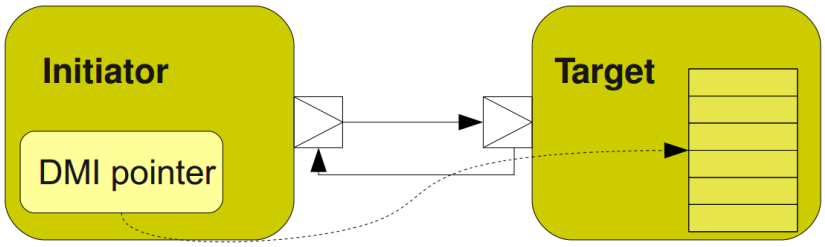



**Figura 12:** ***Direct Memory Interface*** [27]

#### 2.2.4.4 Interfaz de depurado

La interfaz de depurado es una interfaz no intrusiva debido a que no añade retardos, esperas, notificaciones de eventos u otros efectos asociados a una transacción normal. El tipo de transacción usado es el *payload* genérico donde sólo se usa los atributos de comando, dirección, longitud del dato y puntero del dato.

Esta interfaz permite que un depurador de *software* se conecte a un ISS para revisar una dirección de memoria desde el punto de vista de la CPU. También puede permitir a un *initiator* ver el contenido de la memoria del sistema durante la simulación.

### 2.2.5 *Temporal decoupling*

Con el objetivo de acelerar la simulación, TLM-2.0 ofrece el uso de *temporal decoupling*, mecanismo que permite que algunas partes del modelo se ejecuten en su tiempo de simulación sin avanzar el tiempo de la simulación hasta que alcance un punto donde sea necesario sincronizarse con el resto del sistema, en ese momento el control de

la simulación se devuelve a la simulación del sistema. Con esta técnica un módulo se encuentra en un tiempo avanzado respecto a otro módulo. Si se requiere comunicarse con otro módulo que se encuentra en otro tiempo con el objetivo de leer o escribir un dato se puede tomar dos soluciones:

1. Suponer el valor de esa variable y seguir con la simulación avanzada, donde posteriormente se corregirá el valor la variable.
2. Esperar a que el resto de los módulos lleguen a su tiempo consiguiendo sincronizarse con el resto de la simulación.

Estas suposiciones son válidas en el contexto de simulación de plataformas virtuales donde la pila del *software* no depende de los detalles a bajo nivel del *hardware*. El uso de *temporal decoupling* puede dar lugar a simulaciones rápidas en algunos sistemas debido a que se reduce la carga del planificador del simulador. *Temporal decoupling* es característico del estilo de modelado *loosely-timed* (Figura 13).

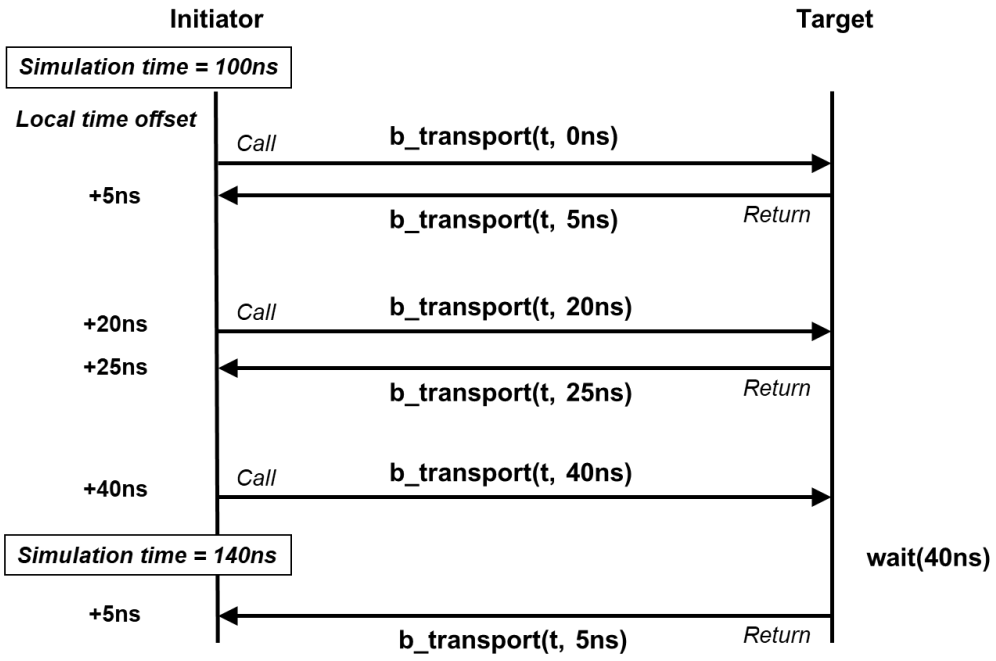



**Figura 13: Uso de t*emporal decoupling* con interfaz bloqueante** [26]

## 2.3 Conclusiones

En este capítulo se ha explicado los lenguajes a nivel de sistema que elevan el nivel de abstracción a la hora de crear un modelo. SystemC otorga mayor control a la hora de

planificar las tareas, mientras que TLM-2.0 permite separar la funcionalidad del modelo de sus interfaces.

Gracias a este tipo de lenguaje, y en particular a la creación del estándar de TLM-2.0, se han podido diseñar plataformas virtuales de manera más sencilla y rápida, permitiendo así mejorar la verificación *hardware/software.*

# Capítulo 3. Plataformas Virtuales

En este capítulo se define el concepto de plataforma virtual y su utilización como herramienta para desarrollar de manera rápida el *software* y para realizar análisis de diversas soluciones a un diseño en etapas tempranas del diseño.

## 3.1 Definición de una plataforma virtual

Se puede definir un prototipo o plataforma virtual como un modelo *software* ejecutable de un sistema *hardware* que se ejecuta en un ordenador anfitrión (*host*) teniendo en cuenta que el modelo debe ser compatible a nivel binario con el sistema real (juego de instrucciones, registros, mapa de memoria) y debe ejecutar el *stack software*, incluyendo desde código de arranque (*boot*) hasta el sistema operativo y la aplicación, incluyendo al menos las utilidades de depurado que puedan existir en el sistema físico. Este tipo de plataformas están realizadas en niveles de abstracción elevados que permite centrarse en los elementos esenciales y permitir que las simulaciones se realicen con mayor velocidad [29].

Su uso se ha extendido en muchos grupos de diseño para poder mantener la planificación de sus proyectos. Las fases de depurado y test son las etapas que más tiempo toman, un 23% del tiempo total del diseño y que se incrementa a la par que crece la complejidad de los SoC debido a que se vuelve imposible verificar el espacio de diseño completo [30].

Para cumplir con todas las funcionalidades de un sistema, las plataformas virtuales incluyen:

1. Modelo del procesador usado, normalmente un *Instruction Set Simulator* (ISS).
2. Modelos de memorias del sistema con RAM, FLASH, etc.
3. Modelo de interfaces de entrada/salida como UART, USB, Ethernet, etc.,
4. Modelo de buses.

Las ventajas que supone el uso de las plataformas virtuales son múltiples. Ya se ha comentado anteriormente que se consigue desarrollar el *software* de manera paralela al *hardware* reduciendo así los tiempos de ejecución de un proyecto. Debido a que estas plataformas cuentan con una representación completamente funcional de un sistema, este *software* empotrado puede ser incorporado en el sistema real sin necesidad de realizar modificaciones en su funcionalidad de una plataforma a otra (Figura 14).

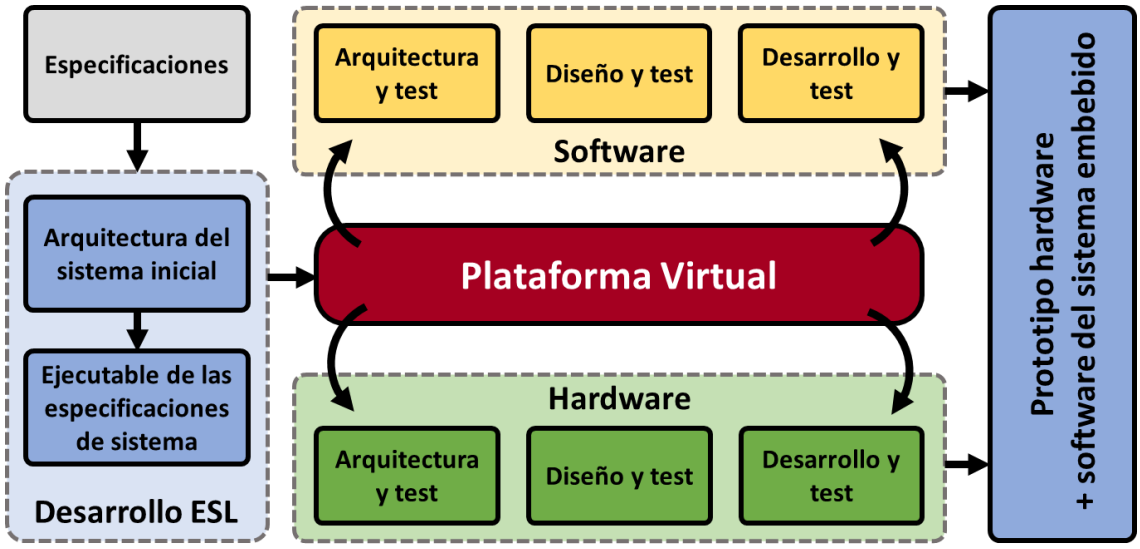



**Figura 14: Aproximación del uso de una plataforma virtual (adaptado de** [31]**)**

Se puede realizar el depurado de la plataforma al completo (*hardware/software*) reduciendo el tiempo del depurado debido a que no es necesario realizar la programación del dispositivo, que puede tomar varios minutos dependiendo del tamaño del diseño global, en el mejor de los casos. En otros casos, la plataforma *hardware* no estará disponible hasta la completa fabricación del ASIC que personaliza la plataforma. Además, se consigue un depurado no intrusivo, es decir, sin la necesidad de incorporar analizadores lógicos al diseño, consiguiendo un control total del depurado con conocimiento de las señales y

registros internos, sin limitación del número de señales o registros a realizar su seguimiento.

Otra ventaja que cuenta las plataformas virtuales en el depurado es la capacidad de observar el *software,* bloques *hardware*, subsistema y del sistema en general, algo que no se puede realizar en un *hardware* real. Se puede tener control sobre todos los relojes, pudiendo parar el tiempo de la simulación y modificar el contenido de las memorias, registros o interrupciones y continuar.

## 3.2 Creación de plataformas virtuales

El desarrollo de una plataforma se debe realizar en las primeras etapas del proyecto, en paralelo con el desarrollo del diseño *hardware* para permitir avanzar lo antes posible en el desarrollo *software*. Es necesario que las especificaciones del proyecto estén definidas para poder desarrollar la plataforma acorde sus necesidades.

Las plataformas virtuales están creadas en lenguajes de alto nivel, tales como SystemC. La estandarización de SystemC TLM-2.0 de Accellera [26] supuso un impulso en la creación de estas plataformas, promoviendo la uniformidad e interoperabilidad de los IP modelados y permitiendo la creación de modelos adecuados para cada caso gracias a los diversos estilos que posee TLM. El estilo de modelo más usado es el de *Loosely Timed* que modela la funcionalidad del *hardware* con una información temporal suficiente para permitir la sincronización del *software*.

### 3.2.1 *Instruction Set Simulator*

Un *Instruction Set Simulator* (ISS) es un modelo que simula la microarquitectura de un procesador a nivel del juego de instrucciones (ISA). Los ISS tradicionales proporcionan precisión a nivel de juego de instrucciones, mientras que los simuladores a nivel de ciclo están relacionados con la implementación de la arquitectura (CAS – *Cycle Accurate Simulator*). Se puede igualmente distinguir aquellos que son precisos a nivel de señales y de ciclos (nivel RTL) y por tanto se puede hablar de simuladores CABA *(Cycle Accurate Bus Accurate).* Es igualmente de interés extender los ISS con comportamiento preciso a nivel

de ciclo como un paso adicional hacia la creación de plataformas virtuales para el desarrollo de SoC [32], [33].

Con el aumento del nivel de abstracción que aporta TLM-2.0 se pueden crear modelos ISS donde la velocidad de la simulación se incrementa y permite explorar espacios de diseños extensos. Además, el acceso a la memoria por parte de ISS se puede optimizar con el uso de *Direct Memory Interface* que introduce TLM-2.0 [34].

### 3.2.2 *System-on-Chip* FPGA

Un SoC FPGA incluye en un único dispositivo un núcleo de procesamiento y una lógica configurable. Ambas partes se comunican a través de un bus de comunicaciones. Una plataforma virtual que simule el comportamiento de este tipo de dispositivos debe contar con una librería de transacciones que soporte los protocolos de los principales buses de comunicación.

## 3.3 Estado del arte

Las principales compañías de semiconductores han visto las ventajas que supone el uso de plataformas virtuales en sus proyectos, ya que, aparte de las ventajas que ya se han comentado, permiten desarrollar y validar el *software* en un entorno de *hardware-in-the-loop.* Es por eso por lo que empresas como Intel FPGA (antes Altera) soportan la plataforma virtual de Synopsys para mejorar el diseño de sistemas basados en SoC FPGA. Xilinx ha trabajado de manera conjunta con Cadence para crear Virtual System Platform (VSP) en donde se pueden crear plataformas virtuales basados en la serie Zynq-7000 [35].

Algunos ejemplos de plataformas virtuales son Wind River Simics [36], Open Virtual Platform de Imperas [37], Vista Virtual Prototyping de Mentor Graphics [38], Virtualizer de Synopsys [20], Bochs [39] o QEMU [40]. En 2007 surge QEMU-SystemC [41] como herramienta para realizar emulaciones *hardware/software* en el desarrollo de SoC. En 2009, se realiza una mejora [21] añadiendo el uso de TLM-2.0 para modelar las interfaces. Se han realizado diversas plataformas virtuales basadas en QEMU-SystemC [42], [43], [44].

En este proyecto se ha realizado un análisis previo de las plataformas virtuales basadas en QEMU para implementar la arquitectura DPI sobre una herramienta de código

libre. Estas herramientas son: QEMU, QEMU-SC y QBOX. Como alternativa se ha optado por el uso de Vista Virtual Prototyping, de Mentor Graphics. A continuación, se explica en detalle las diferentes herramientas que se han estudiado durante la realización de este trabajo.

### 3.3.1 QEMU

QEMU (Quick Emulator) es un emulador de código libre basado en traducción dinámica de instrucciones que puede emular de Sistemas Operativos como GNU/Linux y procesadores como Intel, ARM, MIPS, SPARC, etc.

El traductor dinámico realiza una conversión de las instrucciones de la CPU objeto (*target*) en el juego de instrucciones del *host*. El código binario resultante se almacena en una memoria cache de traducción de instrucciones de tal forma que puede ser reutilizado. La principal ventaja es que las instrucciones del sistema *target* se buscan y se decodifican una sola vez.

El proceso de traducción de instrucciones es como sigue. En primer lugar, se dividen las instrucciones de la CPU *target* en operaciones más simples conocidas como micro-operaciones. Cada micro-operación se implementa en C y se compila con GCC en un fichero objeto. La generación de estas micro-operaciones se hace de forma estratégica, aprovechando las ventajas de la asignación estática de registros del compilador de la CPU *host*, de tal forma que sea posible emular todas las combinaciones de instrucciones y operandos de la CPU *target*. Durante el tiempo de compilación y a partir del fichero objeto que contiene las micro-operaciones se crea el generador dinámico de código mediante la utilidad *dyngen*. Este generador de código dinámico, que se invoca durante el tiempo de ejecución, genera el ejecutable del *host* que concatena las diferentes micro-operaciones. *Dyngen* facilita la reubicación del código para habilitar las referencias a los datos estáticos y funciones en las micro-operaciones [40].

La comunicación entre el procesador y los dispositivos emulados se realiza a través de llamadas a regiones de memoria del bus de sistema. QEMU está basado en diversos subsistemas [40]:

- Emulador de CPU (ARM, SPARC, etc.)

- Emulador de dispositivos (VGA, puertos series, tarjetas de red, etc.)
- Dispositivos genéricos
- Descripción de máquinas (PC, PowerMac, etc.)
- Depurador
- Interfaz de usuario

### 3.3.2 QEMU-SC

Desde que se empezó a usar QEMU como una plataforma virtual de código abierto y junto a la poca utilización de lenguajes de modelado en alto nivel para la creación de las plataformas virtuales, surge la idea de añadir soporte SystemC a QEMU, con esto nace el proyecto QEMU-SC [41].

En la primera versión de este proyecto se realiza la integración a nivel RTL (Figura 15 (a)), donde la comunicación entre QEMU y acelerador *hardware* modelado en SystemC se realiza a través de los buses de comunicación de manera convencional. Al estar basado en este nivel RTL presenta problemas en la velocidad de la simulación.

Posteriormente, y con el objetivo de mejor la velocidad de simulación, se pasa a utilizar TLM como nivel de abstracción elegido, permitiendo que las comunicaciones entre el QEMU y los módulos en SystemC se realizaran a través de transacciones. Para ello, se dota al QEMU un módulo con características TLM modelado en SystemC. Después se crea un puente que sirve de enlace entre este módulo y el dispositivo a conectar. Este puente, implementado en TLM-2.0, hace la función de *Initiator* de las transacciones y el dispositivo SystemC es el *Target* (Figura 15 (b)).

En esta solución, para la sincronización del sistema se utiliza dos nociones de tiempo distintas, una para el QEMU y otra para SystemC. Esto genera problemas de sincronización. La solución que se plantea es hacer un modelo híbrido de tal forma que para cada acceso desde el procesador emulado el tiempo de simulación de QEMU se detiene y la simulación en SystemC comienza. Una vez terminada esta simulación, QEMU se reanuda y la simulación continua.

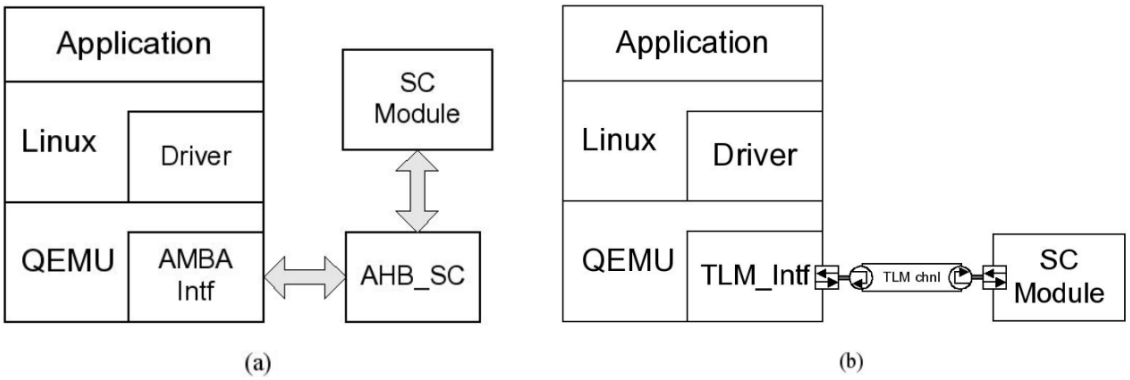



**Figura 15: QEMU-SystemC en RTL (a) y en TLM (b)** [41]

Con esto se consigue que QEMU-SC cumpla con los objetivos de una plataforma virtual que permite simular un sistema entero en etapas tempranas de su desarrollo. En cambio, tiene una limitación en el número de dispositivos que puede soportar, por lo que es una solución viable para sistemas pequeños.

### 3.3.3 QBox

Basado en la solución presentada anteriormente, se propone otra manera de usar QEMU. La técnica aplicada en QBox (QEMU in a Box) [45] es incluir QEMU como un módulo SystemC a través de un *wrapper* (Figura 16). Con esto se consigue eliminar el puente que une ambas partes del sistema ya que ahora cuentan con las mismas interfaces de comunicación. Este *wrapper* se ha implementado con TLM-2.0 usando la técnica de *Temporal Decoupling*. La idea de *temporal decoupling* es que en un sistema paralelo los hilos de procesamiento utilizan su propio esquema temporal y únicamente se sincronizan cuando necesitan comunicarse entre ellos. En esta herramienta, es el simulador de SystemC el que actúa como maestro y el QEMU como esclavo, por lo que el primero tiene control sobre la ejecución del QEMU.

Es necesario tener en cuenta las nociones temporales usadas tanto en el simulador de SytemC y como en QEMU para asegurar el tiempo de sincronización debido a que trabajan en diferentes dominios. Cuando la simulación en SystemC comienza, el *wrapper* invoca al bucle principal del QEMU que es el que cuenta con las funciones de control de la simulación. Con el uso del *Temporal Decoupling* se puede usar las funciones de sincronización de *Quantum Keeper* de TLM-2.0, el cual permite manejar los tiempos de simulación y tomar el control de esta cuando el tiempo del QEMU es mayor al tiempo global

permitido por la simulación en SystemC. Cuando ocurre esto, la simulación del QEMU entra a espera dejando que el simulador de SystemC continúe. Cuando el *wrapper* recupera el control de la simulación reanuda la ejecución del QEMU.

QBox se adapta a los requerimientos de las plataformas virtuales. Permite instanciar componentes TLM-2.0 para cubrir la creciente demanda de integrar múltiples procesadores en una plataforma. Cuenta con un *wrapper* denominado TLM2C que permite convertir modelos basados en C a modelos en SystemC con el estándar TLM-2.0.

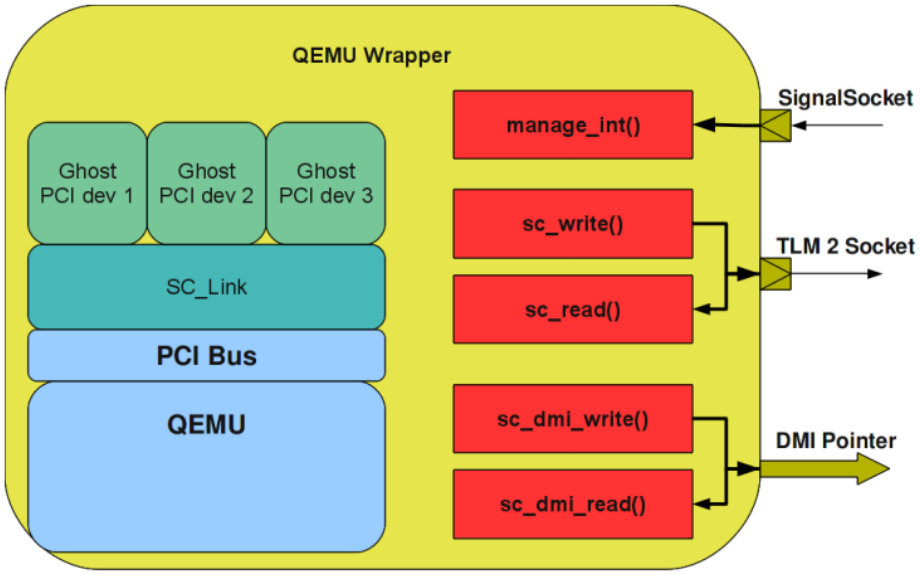



**Figura 16: *Wrapper* QEMU para QBox** [45]

### 3.3.4 Vista *Virtual Prototyping*

Mentor Vista ofrece un entorno para el desarrollo, integración, validación y optimización de los diseños complejos de los sistemas embebidos. Permite ejecutar *software* en los modelos de los procesadores que proporciona, además de los modelos funcionales del *hardware*. La plataforma está basada en TLM lo que permite realizar simulaciones rápidas y precisas [38] (Figura 17).

La ejecución del *hardware* se puede realizar en dos modos. Por una parte, está el modo funcional, que soporta la integración, validación y depurado del *software*. Por otra parte, el modo orientado a prestaciones permite analizar y optimizar el *software* para mejorar prestaciones y reducir el consumo de potencia. Estos modos se basan en la

utilización de dos tipos de modelado de TLM-2.0: *loosely timed* (LT) y *approximately timed* (AT).

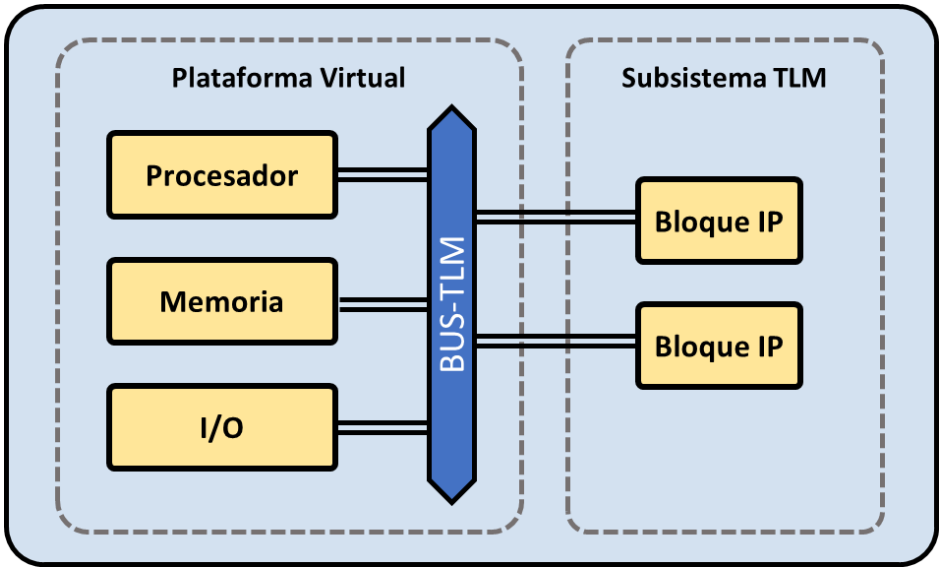



**Figura 17: Diagrama de bloques de plataforma virtual de Vista. Adaptado de** [46]

Una ventaja que cuenta esta herramienta es la incorporación de *Virtual Prototype Kits*, donde proporciona prototipos virtuales configurables de alguno de los dispositivos más conocidos: Altera Arria-V, Xilinx Zynq, ARM Versatile Express y Freescale Semiconductor's i.MX 6.

## 3.4 Conclusiones

Se ha realizado un estudio de diferentes alternativas para el desarrollo de plataformas virtuales que den soporte a SoC reconfigurables, en este caso a la plataforma Zynq de Xilinx.

Por una parte, se ha estudiado la utilización de QEMU como ISS conectado con SystemC, tanto a nivel RTL como TLM-2.0, usando diferentes estrategias temporales para la comunicación entre ambos dominios de simulación. En esta aproximación se utiliza SystemC para el modelado de los bloques IP y de la arquitectura global del SoC, aprovechando la capacidad de modelado multinivel de SystemC. En la literatura se pueden encontrar diferentes soluciones como las indicadas, siendo todas ellas dependientes de las versiones integradas de QEMU y del BSP (*Board Support Package*) generado por el entorno de Xilinx. Igualmente se estudiaron otras alternativas de Cadence y Synopsys.

El entorno de desarrollo de la plataforma virtual ha sido Mentor. Como se ha indicado, la herramienta está basada en SystemC TLM-2.0 por lo que se facilita la comunicación entre el modelo en SystemC y el modelo del procesador al estar soportados por la propia herramienta. Además, con la incorporación de *Virtual Prototype Kits* en Vista se puede crear una plataforma virtual completa. Esto se debe a que cuenta con la arquitectura completa del dispositivo Zynq de Xilinx, la cual se basa este proyecto, desde los elementos de procesamiento, pasando por memorias, buses, zonas seguras y periféricos.

# Capítulo 4. El entorno de desarrollo Mentor Vista

En este capítulo se muestra el flujo que sigue Mentor Vista para la creación de una plataforma virtual.

## 4.1 Introducción

Mentor Vista [38] es una solución basada en TLM-2.0 para la exploración arquitectural, verificación y prototipado virtual del diseño que facilita la toma de decisiones en etapas tempranas del diseño. Se trata por tanto de una plataforma nativa de *Electronic System Level* (ESL) para el diseño, verificación y análisis de arquitecturas *hardware* basadas en TLM. Vista genera una plataforma virtual para el desarrollo del *software* y su ejecución para comprobar la funcionalidad del sistema. Engloba en una única herramienta el modelado en TLM, la creación de la plataforma de manera gráfica, depurado y análisis. Facilita la creación de modelos a nivel de sistema, simular de manera rápida este modelo para validar y optimizar el diseño, crear arquitecturas *hardware*, interfaces de SoC y ejecutables del sistema para exhibir el comportamiento del sistema.

Los principales elementos y características que cuenta Vista se describen a continuación [47]:

1. Librería de modelos genéricos en TLM-2.0 que incluye modelos de procesadores y periféricos con un modelado completo de su funcionalidad respetando las políticas temporales y de potencia reales.

2. Utilización de diagrama de bloques sobre los modelos SystemC/TLM-2.0 para la creación de diseños jerárquicos.
3. Simulación de modelos, con visualización de las transacciones TLM-2.0 que permite la depuración de los modelos, pudiendo ver en detalle las transacciones, incluyendo el *payload*, fases, estados y tiempos.
4. Herramienta de análisis temporal y de potencia en el que se puede realizar estadísticas de las transacciones, rendimiento o latencia.
5. Generación de un prototipo virtual (objeto ejecutable) de la plataforma h*ardware* en TLM-2.0 para desarrollar el *software* y la integración de este.

Las diversas etapas que se siguen para la creación del prototipo virtual se describen en los siguientes puntos de este capítulo y son los siguientes (Figura 18):

1. Modelado
2. Ensamblaje
3. Compilar y construir el proyecto
4. Análisis
5. Creación del prototipo virtual

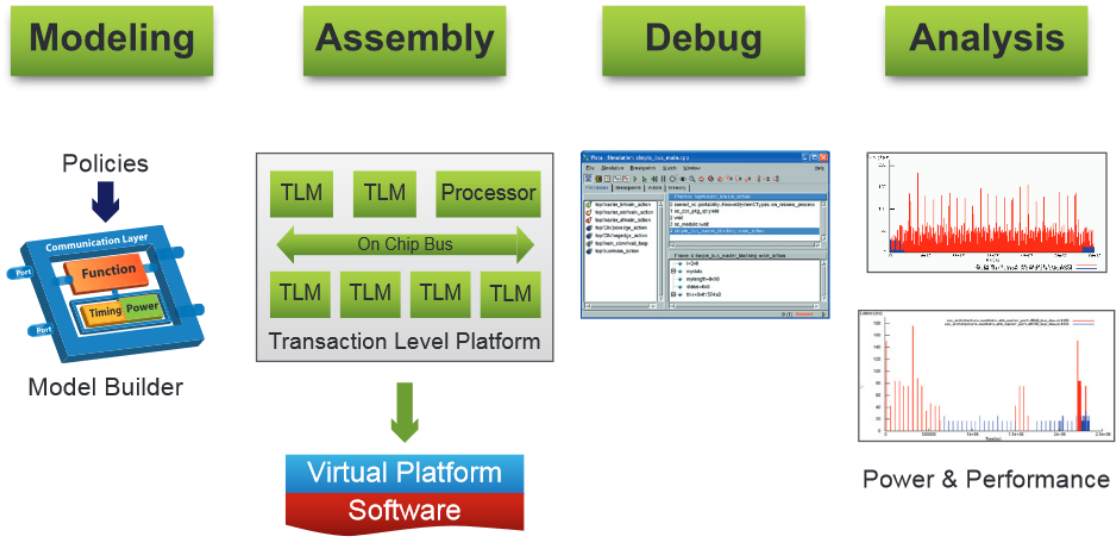



**Figura 18: Principales etapas en Vista** [48]

## 4.2 Modelado

El modelado en Vista difiere de los modelados tradicionales en SystemC, donde en un único modelo se combina los atributos temporales con los del comportamiento a la vez que se describen las interfaces que cuenta el modelo. En Vista el modelo se divide en capas, separando las interfaces, el funcionamiento y el tiempo (Figura 19).

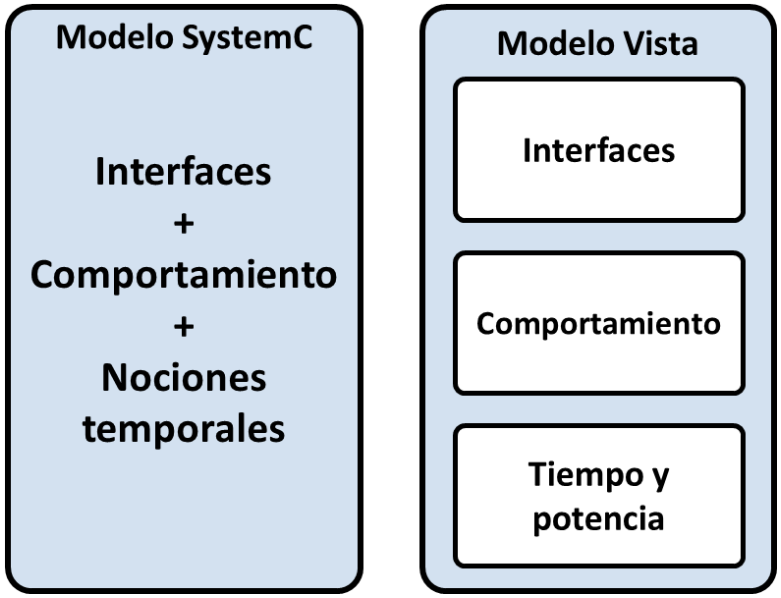



**Figura 19: Diferencias de modelado entre SystemC y Vista (**[49]**)**

Mediante la interfaz gráfica de Vista se pueden crear modelos TLM desde el inicio o desde un modelo en SystemC. Estos modelos creados se componen de dos capas completamente separadas permitiendo así separar la funcionalidad de la implementación [49]:

- La capa "PV" (*Programmers' view*) contiene el comportamiento funcional.
- La capa "T" (*Timing*) contiene el comportamiento temporal y de potencia.

Con esta separación se permite realizar la validación del *software* y el prototipado virtual desactivando la capa "T" para realizar simulaciones HW/SW funcionales. Activando de nuevo esta capa se puede realizar los análisis temporales y de potencia. Cuando las dos capas se encuentran activas se denomina modelo PVT (Figura 20).

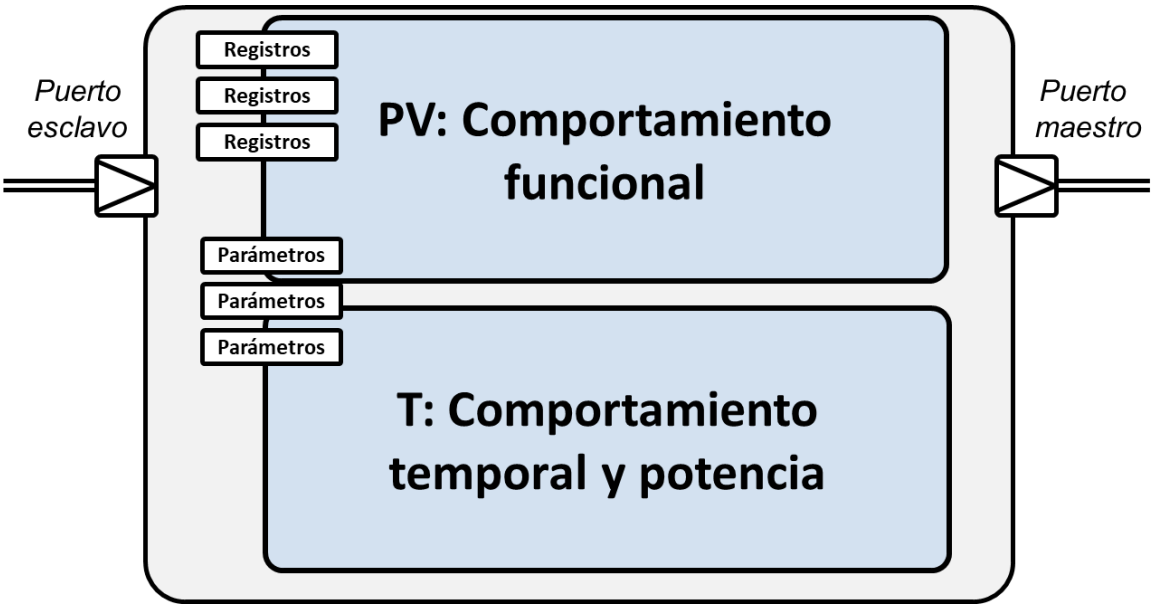



**Figura 20: Modelo PVT**

### 4.2.1 Modelado PV

El modelado del comportamiento describe cómo reacciona el diseño ante la llegada de una transacción a través de un puerto esclavo. Este comportamiento se define en un algoritmo pudiendo procesar los datos recibidos o realizar un control de ellos [50].

Estos modelos suelen estar basados en métodos y no en hilos, por lo que el uso de *wait()* deben evitarse en esta capa. Además, hace uso de interfaces bloqueantes que permite mejores rendimientos en la simulación al mismo tiempo que el código se vuelve más eficiente y simplificado. En las interfaces bloqueantes, el *Initiator* se bloquea hasta que recibe el retorno de la llamada notificando que la transacción se ha completado o informando de un error si llega a ocurrir (Figura 21).

Es necesario definir el protocolo de comunicación que se desea implementar en el modelo, ya que es quien define cómo los datos se comparten entre un maestro y un esclavo y cómo se controlan.

Al generar un modelo en Vista se crea siguiendo el estándar TLM-2.0, el cual incluye los *sockets* del *Initiator* y el *Target*, las funciones de lectura y escritura de las transacciones en el maestro y las funciones de *callback* de lectura y escritura en el esclavo.

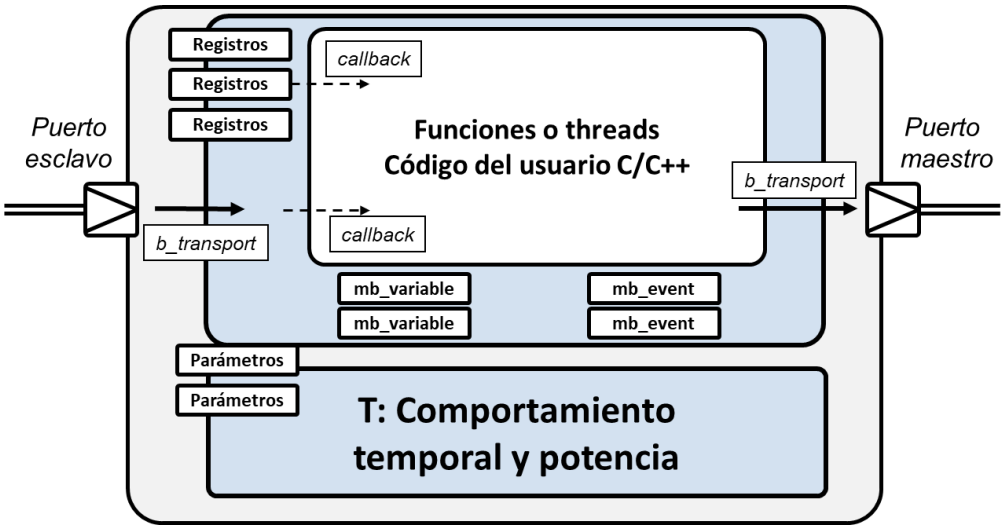



**Figura 21: Modelo PV**

### 4.2.2 Modelado T

Este estilo de modelado permite añadir políticas temporales y de consumo de potencia para definir los comportamientos de los modelos. En comparación al modelado PV, este modelado usa interfaces no bloqueantes. El modelado temporal también permite sincronizar la capa PV y la T.

Existen diversas políticas temporales que se pueden definir únicamente a los puertos, como *Delay* y *Split*, o a las operaciones internas del modelo, como *Sequential* y *Pipeline.* En estas políticas temporales se puede definir también cuanta potencia se consume durante el proceso de la política [50] (Figura 22).

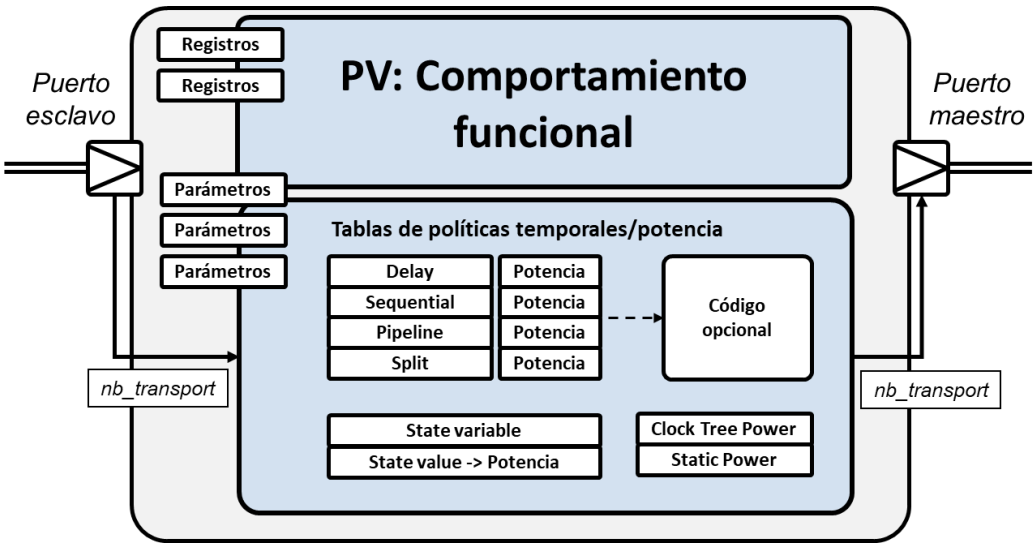



**Figura 22: Modelo T**

#### 4.2.2.1 Política de *Delay*

La política de *delay* se aplica únicamente a los puertos esclavos y representa el tiempo de respuesta que el bloque tarda en procesar los datos de una transacción y la potencia necesaria para procesar la transacción en el puerto. La política de *delay* tiene tres parámetros que en conjunto definen el tiempo de respuesta:

- **Sincronización**. Al estar activo este parámetro se especifica que la respuesta se envía cuando la función *callback* del esclavo termina.
- **Latencia**. Tiempo que se asigna sólo a la primera palabra si es una transferencia a ráfagas.
- **Estados de espera**. Tiempo de espera definido en ciclos de reloj por cada palabra recibida.

El *delay* se calcula a través de la ecuación (1) donde *palabras* se refiere al número de palabras que tiene la transacción, *estados* es el número de estados de espera y el reloj del protocolo depende del definido en el protocolo usado.

$$Delay = latencia + palabras \cdot (estados\ +\ reloj\ del\ protocolo) \qquad (1)$$

En la Figura 23 se muestra el esquema de la política de *delay* sin la sincronización, donde se observa el tiempo que transcurre desde la petición del puerto X hasta la respuesta del puerto Y. El tiempo entre palabra está definido por el número de estados de espera y el tiempo de reloj del protocolo que se representa como "*k*".

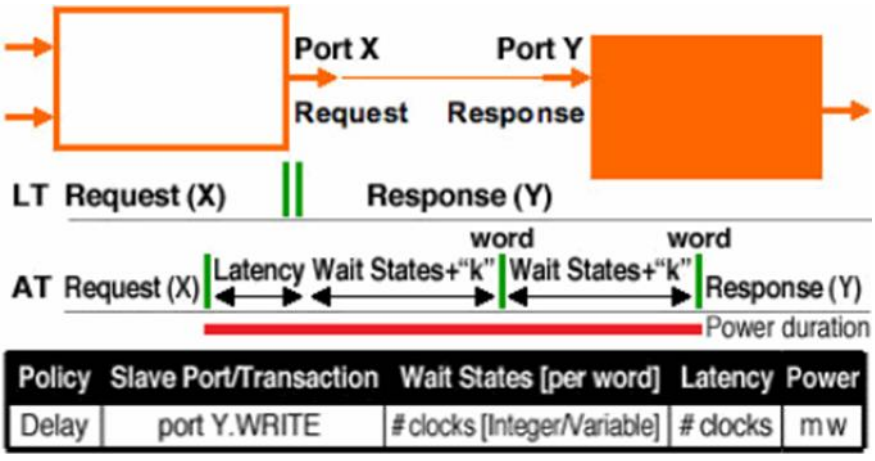



**Figura 23: Esquema de la política de *delay***

En la Figura 24 se muestra el esquema de la política de *delay* con sincronización. El tiempo que transcurre desde que se envía la petición desde el puerto X hasta que se recibe la respuesta del puerto Y es el definido por la ecuación (1) más el tiempo de la función *callback.*

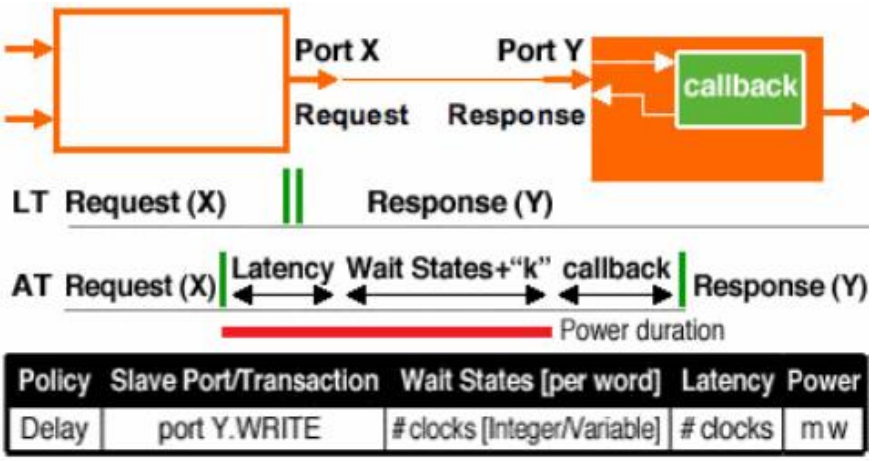



**Figura 24: Esquema de la política de *delay* con sincronización**

En esta política se puede añadir la potencia que necesita para comunicar el dato, incluyendo la potencia consumida por el adaptador del bus y los registros, si existieran. La duración del consumo de potencia está definida por el *delay* definido en la política. Cuando se activa la sincronización, el tiempo del *callback* no se contempla en la duración del consumo de potencia.

#### 4.2.2.2 Política de *Split*

Esta política se aplica en los puertos maestros para dividir las transacciones. Esto depende del bus y del protocolo implementado. La transacción del modelo PV se divide y se emite en ráfagas. La latencia del modelo será el tiempo mínimo que existe entre cada una de las ráfagas.

- **Tamaño de la ráfaga.** Es el tamaño del buffer por el cual el dato se transmite de un puerto a otro. La transacción se divide en ráfagas del mismo tamaño, pudiendo la última ser de menos tamaño.
- **Latencia.** Tiempo mínimo entre cada una de las ráfagas.

En la Figura 25 se muestra el esquema de esta política donde se observa lo comentado anteriormente, el paquete es dividido y es transmitido cada cierto tiempo

definido por la latencia. En esta política, la potencia se consume únicamente en cada transacción.

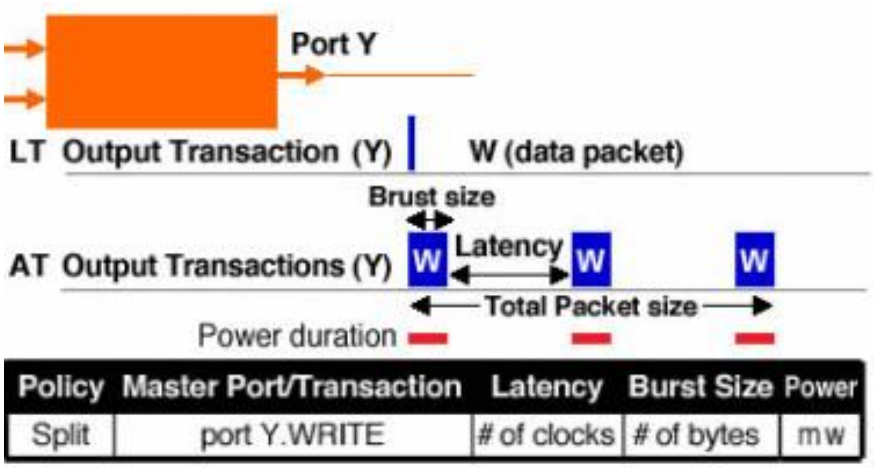



**Figura 25: Esquema de la política de *Split***

#### 4.2.2.3 Política de *Sequential*

Esta política permite definir la latencia que hay entre el efecto de un evento en el modelo PV y su causa. Un evento se entiende como la lectura o escritura sobre un puerto maestro o esclavo y escritura o lectura sobre variables o registros. La latencia es el número de ciclos internos en el modelo, desde el final de la causa y el comienzo del efecto. Los parámetros que se pueden ajustar en esta política son:

- **Sincronización.** Si está activo especifica que el efecto debe cumplir las políticas temporales antes de comenzar.
- **Causa.** Especifica la causa por la que activa el evento.
- **Latencia.** La latencia se computa como el número de ciclos internos de reloj en el modelo.

En la Figura 26 se muestra el esquema de esta política donde la lectura del puerto Y es causada por la escritura del puerto X. Esta lectura se realiza tiempo después de la causa y está definida por la latencia.

La potencia que se consume es desde que el evento de la causa termina hasta que el efecto comience, es decir, la latencia asignada en el parámetro de esta política.

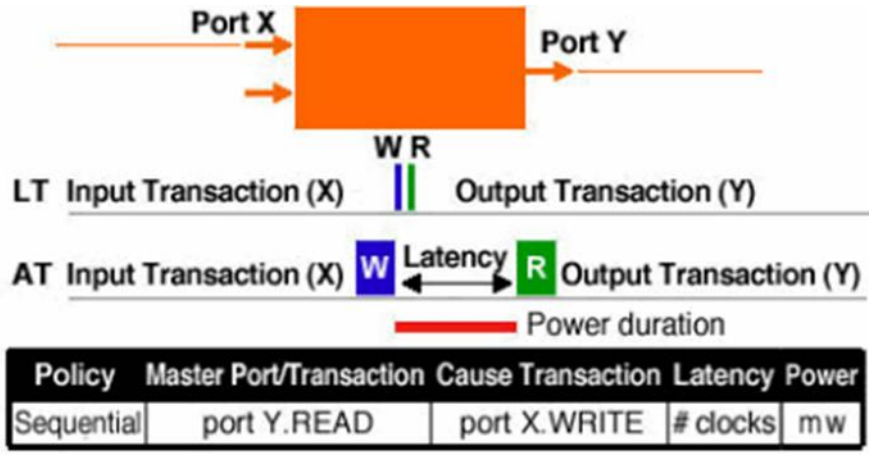



**Figura 26: Esquema de la política *Sequential***

#### 4.2.2.4 Política de *Pipeline*

Con esta política se consigue reflejar el *pipeline* del *hardware,* donde la latencia se mide desde el comienzo de la causa de la transacción. Esta política se define en la transacción entre dos puertos. Los parámetros que se pueden asignar son:

- **Sincronización.** Si está activo especifica que el efecto debe cumplir las políticas temporales antes de comenzar.
- **Causa.** El dato que se realizará el *pipeline.*
- **Latencia.** Número de ciclos internos de reloj que toma el modelo en computar el efecto.
- **Tamaño del buffer.** Número de bytes del dato que el buffer mantiene entre la entrada y la salida de un maestro. Si el valor asignado es cero significa que no existe buffer.

Cuando no existe el buffer las transacciones se pueden superponer. Esto ocurre si la latencia en la política es menor que la latencia de la causa, haciendo que el efecto comience antes que la causa termine. Esto se puede evitar si se aplica una política de tipo *Sequential* o si el protocolo soporta múltiples transacciones pendientes. Además, para evitar resultados distintos entre la simulación en LT y AT es recomendable añadir la política de *delay* con el parámetro de sincronización activo.

Si la causa es la lectura de un puerto maestro y el efecto es la escritura sobre otro puerto maestro y existe un buffer entre la entrada y salida de este, la escritura ocurrirá

cuando la lectura llene el buffer. Dependiendo del tamaño del buffer seleccionado los datos pueden ser divididos, por lo que no será necesario aplicar políticas de *Split.*

La política de *pipeline* se puede aplicar sobre dos puertos maestros distintos (Figura 27) o sobre el mismo puerto (Figura 28).

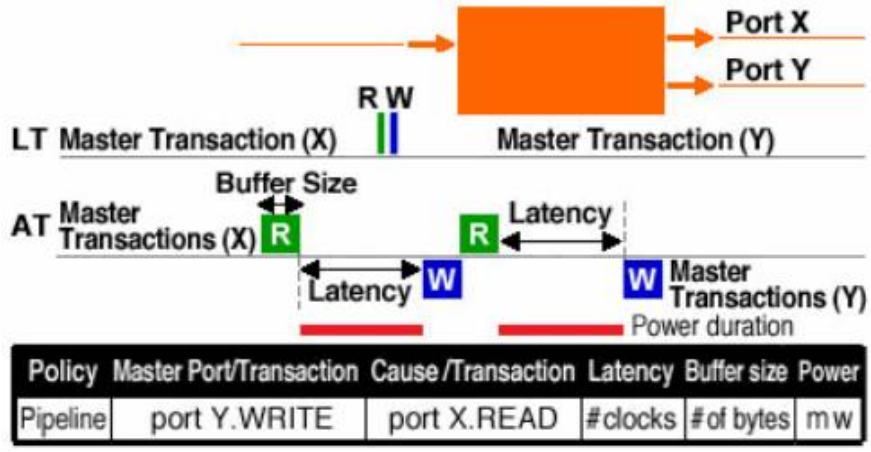



**Figura 27: Esquema de la política de *pipeline* para dos maestros**

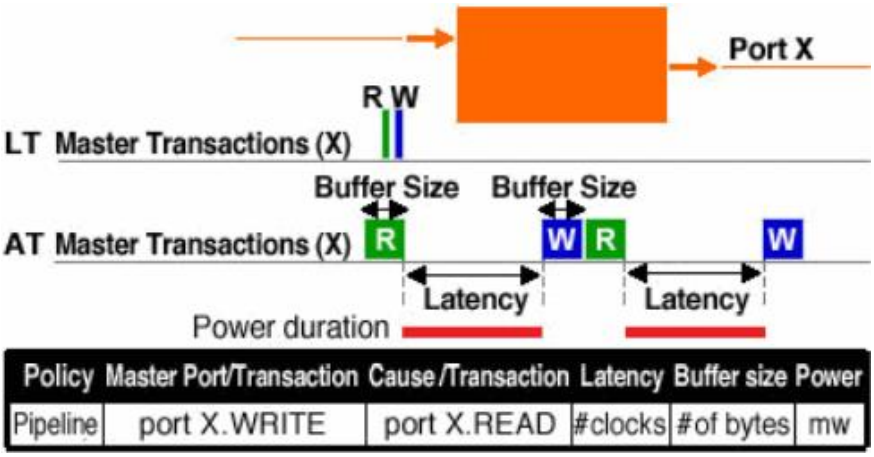



**Figura 28: Esquema de la política de *pipeline* para un maestro**

La potencia consumida en esta política es desde que el final de la transacción de la causa hasta el comienza del efecto.

### 4.2.3 Modelos genéricos

A parte de la posibilidad de creación de modelos propios, Vista proporciona una librería que contiene modelos genéricos que el usuario puede usar, configurar y modificar en sus proyectos. En la Tabla 1 se muestra un resumen de los modelos disponibles en la librería genérica, con los puertos y registros definidos por defecto.

**Tabla 1: Resumen de los modelos genéricos** [51]

| MODELO | NOMBRE PUERTO | TIPO PUERTO | REGISTROS |
|---|---|---|---|
| **BUS** | Bus_slave | Esclavo | |
| | Bus_master | Maestro | |
| **CPU** | CPU_slave | Esclavo | |
| | CPU_master | Maestro | |
| **CACHE** | Host | Esclavo | Address<br>Size<br>Activate |
| | Memory | Maestro | |
| **DMAC** | Host (múltiple) | Esclavo | Source<br>Destination<br>Size<br>Trigger<br>Acknowledge |
| | DMA_master (*single bus master)* | Maestro | |
| | IRQ (*single signal*) | Maestro | |
| **INTC (INTERRUPT CONTROLLER)** | Host | Esclavo | IRQStatus<br>RowStatus<br>IntEnable<br>IntClear<br>IntAddress |
| | IRQ | Maestro | |
| | Int_source1 | Esclavo | |
| | Int_source2 | Esclavo | |
| **INTERCONNECT** | Slave_1 (múltiple) | Esclavo | |
| | Master_1 (múltiple) | Maestro | |
| **MEMORY** | Slave | Esclavo | |
| **SNOOP** | Memory | Esclavo | Address<br>Size<br>Activate |
| | Slave_1 (múltiple) | Esclavo | |
| **TIMER** | Host | Maestro | Clear_IRQ<br>Clock_count<br>Int_count<br>Restart |
| | IRQ | Esclavo | |

Las modificaciones que se pueden realizar sobre los modelos son:

1. Añadir puertos
2. Añadir parámetros
3. Cambiar valores por defecto de los parámetros
4. Modificar direcciones y anchos de los registros
5. Añadir o cambiar políticas

Después de realizar las modificaciones, se genera el modelo a través de la plantilla del modelo base con los cambios que se han realizado.

### 4.2.4 Procesadores genéricos

Los modelos de los procesadores en Vista son modelos de *Fast* ISS (*Instruction-Set Simulator*) basados en QEMU para las familias de procesadores de ARM y PPC. Los modelos de estas ISS se pueden integrar en cualquier plataforma SystemC/TLM-2.0 creada en Vista, permitiendo realizar la validación y el análisis de la plataforma, así como el depurado del *software* y el análisis de la plataforma virtual.

La infraestructura del modelo de QEMU en el que se basa los procesadores se muestra en Figura 29, donde los bloques proporcionados por QEMU se muestran en naranja y los bloques proporcionado por Vista se muestran en verde.

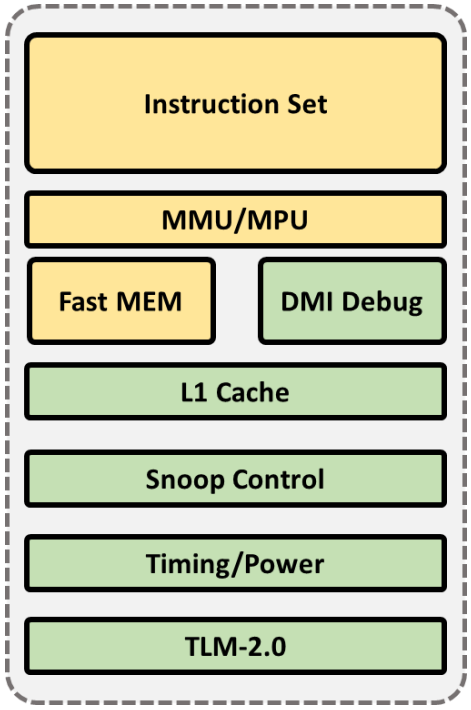



**Figura 29: Infraestructura del modelo de QEMU (adaptado de** [51]**)**

Gracias a la utilización de QEMU, los procesadores cuentan con diversas prestaciones, algunas de ellas se nombran a continuación:

- Especificar un reloj para la CPU
- Reiniciar la CPU
- Realizar la simulación en modo LT o AT
- Definir la latencia o energía consumida por cada instrucción
- Seleccionar quién controla la cache (ISS o modelo *hardware)*

- Definir parámetros de la configuración de la memoria cache
- Cargar una imagen ELF a la memoria y posteriormente el procesador carga el ejecutable de la imagen
- Definir el puerto para el depurado *software*
- Seleccionar el nivel de los mensajes informativos del depurado

## 4.3 Integración

La interfaz gráfica de Vista cuenta con un editor de diagrama de bloques que permite instanciar los diversos modelos que se han generado en la etapa de modelado y unirlos a través de una representación gráfica. En este editor se muestran los componentes en TLM-2.0 con los *sockets* del *Initiator* y el *Target,* también se muestran canales primitivos si el modelo lo tuviera [52].

Para realizar el montaje de la arquitectura es necesario crear un esquemático que esté asociado a una librería que contenga los modelos para ser instanciados. Una vez creado el esquemático se pueden incluir los modelos necesarios al diagrama. Para realizar la conexión de los diversos módulos, Vista ofrece tres tipos de conexiones: *binders, signals* y *channels*.

1. ***Binders****:* sirven para conectar los sockets de modelos TLM.
2. ***Signals***: usados para conectar señales SystemC.
3. ***Channels***: usados para canales tipo *sc_fifo* o *sc_buffer.*

Desde el editor se puede crear jerarquías del diseño. Esto se consigue instanciando un diagrama de bloques y uniéndolo a otro diagrama de bloques. Así se permite crear diseños usando la técnica de *Bottom-up.*

Todas las acciones que se realiza sobre el editor del diagrama de bloques se almacenan en un fichero generado por la herramienta, que contiene todos los módulos instanciados y sus conexiones. El usuario puede editar el diagrama de bloques desde este fichero. La validación del diagrama se realiza en el momento en el que se guarda el diagrama de bloques, si hubiera algún error de conexión entre los módulos se muestra un mensaje.

## 4.4 Compilar y construir el proyecto

Para realizar la verificación del diseño y su depurado es necesario crear un ejecutable del diseño y de su *testbench.* Para conseguir esto, la herramienta realiza tres etapas: compilar el proyecto, construir el proyecto y elaborar el diseño [53].

1. **Compilar el proyecto**: por defecto la herramienta usa el compilador de GCC. Al realizar la compilación se analiza el código fuente y se generan ficheros tipo objeto de cada código en C++ y se incluyen los ficheros de la cabecera al diseño.
2. **Construir el proyecto**: al construir el proyecto se enlazan todos los objetos y se crea un ejecutable. Se puede realizar una reconstrucción en donde se eliminan todos los objetos del proyecto y se vuelven a generar para posteriormente enlazarlos. Durante este proceso se identifica el *sc_main* del proyecto y se mueve al directorio de simulación.
3. **Elaborar el diseño**: la elaboración del diseño permite realizar la simulación y el depurado de este ya que en la etapa anterior se ha identificado el *sc_main* del proyecto.

Una vez que estos pasos se han realizado, el diseño puede ser simulado para verificar que el comportamiento corresponde con las especificaciones originales. Para invocar la simulación es necesario tener un *top* del diseño (*sc_main*), un ejecutable y un fichero de parámetros. Este fichero contiene parámetros para el procesamiento interno de los modelos.

Desde la simulación se puede realizar una monitorización de lo que ocurre en ese proceso [54]. Con la vista en forma de onda se muestra el comportamiento de los puertos, canales u objetos en el tiempo. También se pueden monitorizar los cambios de valores de objetos o los eventos, previamente seleccionados y llevados a la carpeta de *Watch.* Otro tipo de monitorización es la relacionada con los procesos, donde se informa si están activos, inactivos, suspendidos, terminados o en espera.

## 4.5 Análisis

Una vez el diseño se ha construido, se puede realizar la simulación. Previamente los parámetros internos de los modelos deben estar definidos (ya sea manualmente desde la creación del modelo o por medio de un fichero de parámetros), como el rango de direcciones, ubicación del fichero ELF, latencias, etc.

Tras la ejecución de la simulación se pueden realizar análisis de potencia y temporales gracias a la generación de una base de datos que contiene toda la información de las transacciones y la actividad de consumo de potencia de la simulación [55]. Con esto se puede:

- Visualizar transacciones y variables de diversas instancias
- Analizar el rendimiento y latencia de los distintos módulos del sistema
- Analizar la distribución de potencia
- Comparar los resultados de varias simulaciones
- Analizar datos durante una simulación

Desde la interfaz gráfica se puede realizar este análisis a través de gráficas (Figura 30). La estructura del diseño se muestra en el panel izquierdo donde se puede seleccionar qué elementos se quieren llevar al gráfico. Se pueden combinar distintos tipos de análisis en un mismo gráfico. En las pestañas superiores del panel izquierdo se puede seleccionar cuatro formas distintas de análisis.

1. **Resumen**: el resultado de consumo de potencia de cada objeto y la tasa de transferencia de cada transacción.
2. **Potencia**: el consumo de potencia del diseño mostrando la potencia total y la dinámica.
3. ***Sockets**:* muestra las tasas de transferencias de cada *socket* medidas en transacciones, bytes y latencia.
4. **Atributos**: muestra atributos que el usuario ha especificado que sean analizados.

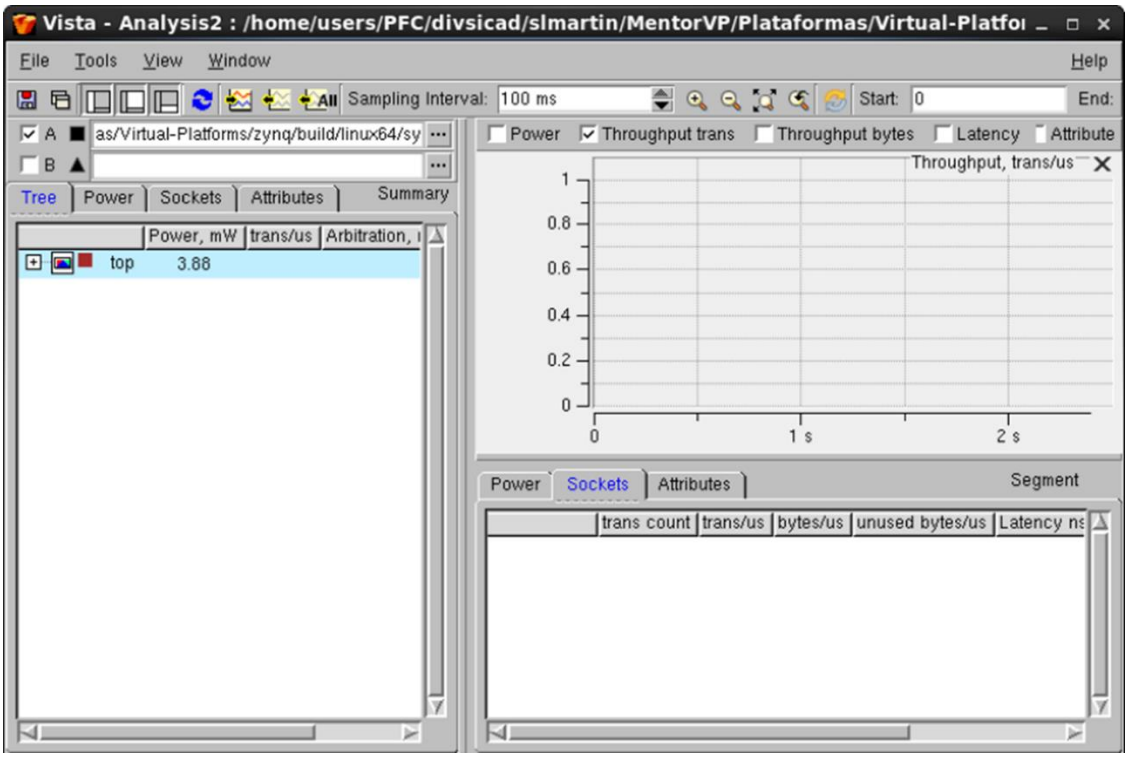



**Figura 30: Ventana de análisis**

Se puede tener acceso a lo que está ocurriendo durante la simulación con la ventana del control de simulación, que permite tener información sobre los procesos, puntos de ruptura, memorias, etc. (Figura 31).

Desde esta ventana, se muestra en el panel izquierdo los procesos de los bloques que contiene el diseño. En la carpeta *MB Dynamic Processes* se encuentran los procesos dinámicos que cuenta el *Model Builder*. Estos procesos son creados cuando se ejecuta la simulación en el modo AT y sirven para ejecutar los *callback* de los módulos. Se puede observar que los procesos se distinguen por un código de colores (Tabla 2) y se diferencian entre *methods* (M), *thread* (T) y *Cthread* (C).

**Tabla 2: Código de colores de los procesos**

| COLOR | METHODS | THREADS / CTHREADS |
|---|---|---|
| **AZUL** | Inactivo | Sin comenzar |
| **VERDE** | Activo | |
| **AMARILLO** | Se ejecutará o tiene previsto ejecutarse. | |
| **ROJO** | N/A | Suspendido |
| **GRIS** | N/A | Terminado |

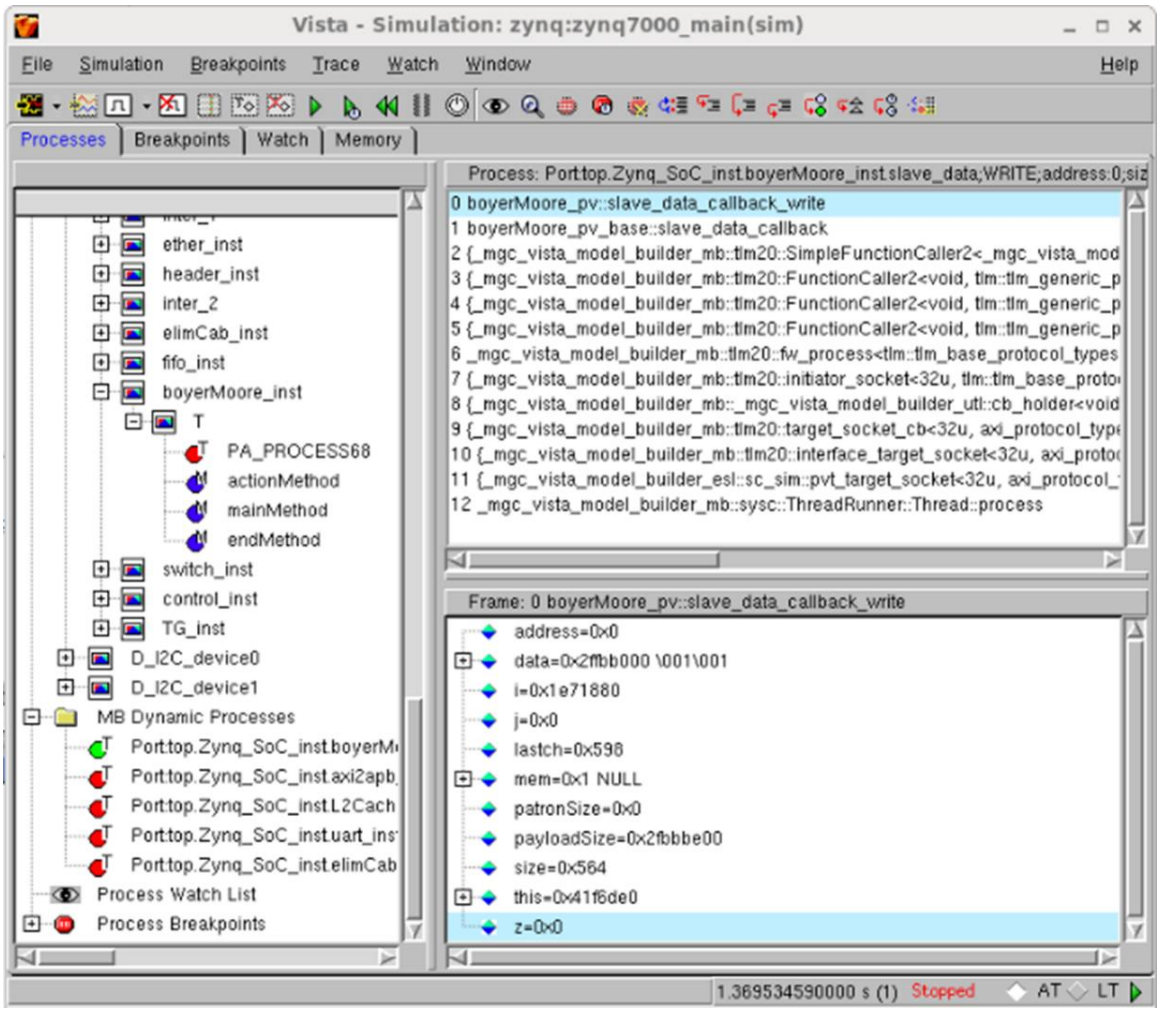



**Figura 31: Ventana de control de simulación**

En el panel inferior se muestran las variables declaradas en la función donde se encuentre la simulación. Desde las pestañas que se encuentran en la parte superior se puede acceder al listado los puntos de rupturas que se han dispuestos, a listado de las funciones que se quiere realizar un seguimiento y por último a la memoria del diseño.

Por último, se puede realizar el depurado del *software* ya que el modelo ISS de QEMU soporta el puerto GDB. Para realizar este depurado primero se debe especificar el valor de este puerto en los parámetros del modelo o con el fichero externo de parámetros (`gdbstub_port = 1234`). Posteriormente, se invoca la simulación la cual se queda a la espera debido que QEMU busca una conexión desde el puerto indicado (Figura 32).

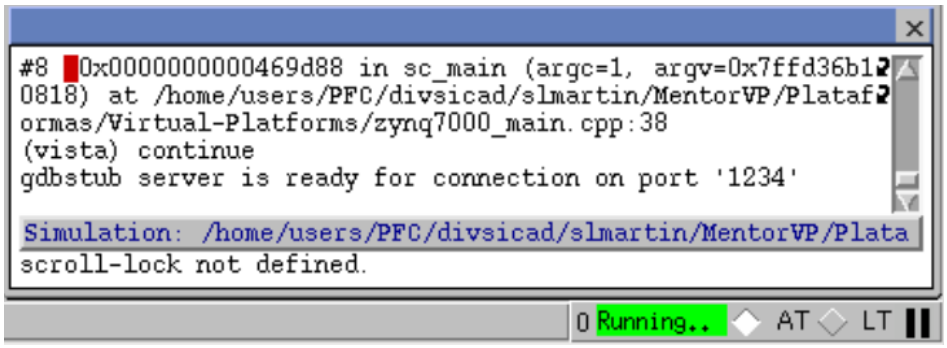



**Figura 32: Simulación en espera de la conexión del puerto 1234**

Desde un terminal se ejecuta el depurado de ARM añadiendo el fichero de la imagen ELF. Al realizar esto, desde la consola del simulador se muestra un mensaje indicando que el servidor está preparado para la conexión (Figura 33).

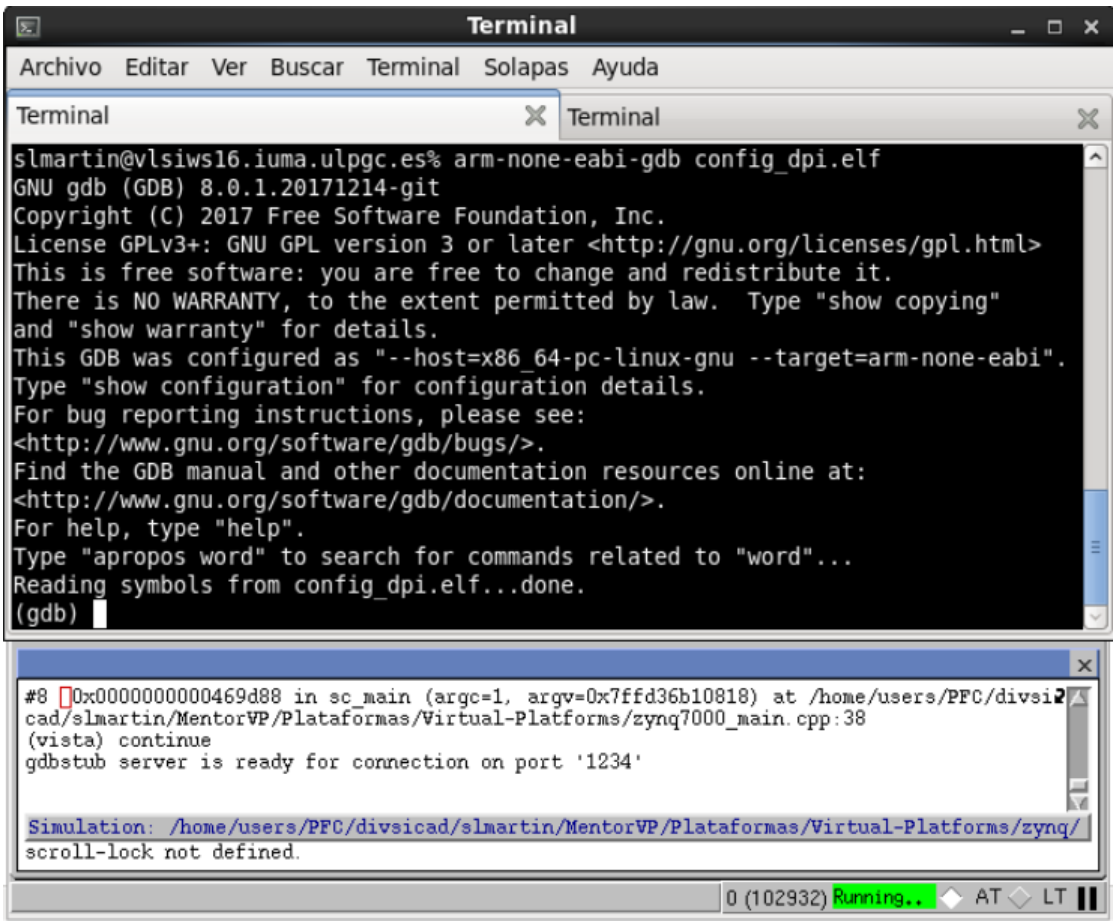



**Figura 33: Ejecutar la herramienta de depuración**

De nuevo en el terminal GDB, se define el puerto remoto que debe coincidir con el indicado en el servidor a través del comando `target remote: 1234`. Desde la consola de Vista se debe mostrar un mensaje indicando que la conexión se ha realizado. El terminal

GDB muestra que se ha cargado la tabla de vectores que contiene los valores de reinicio de la pila de punteros, direcciones de inicio, excepciones, etc. (Figura 34)

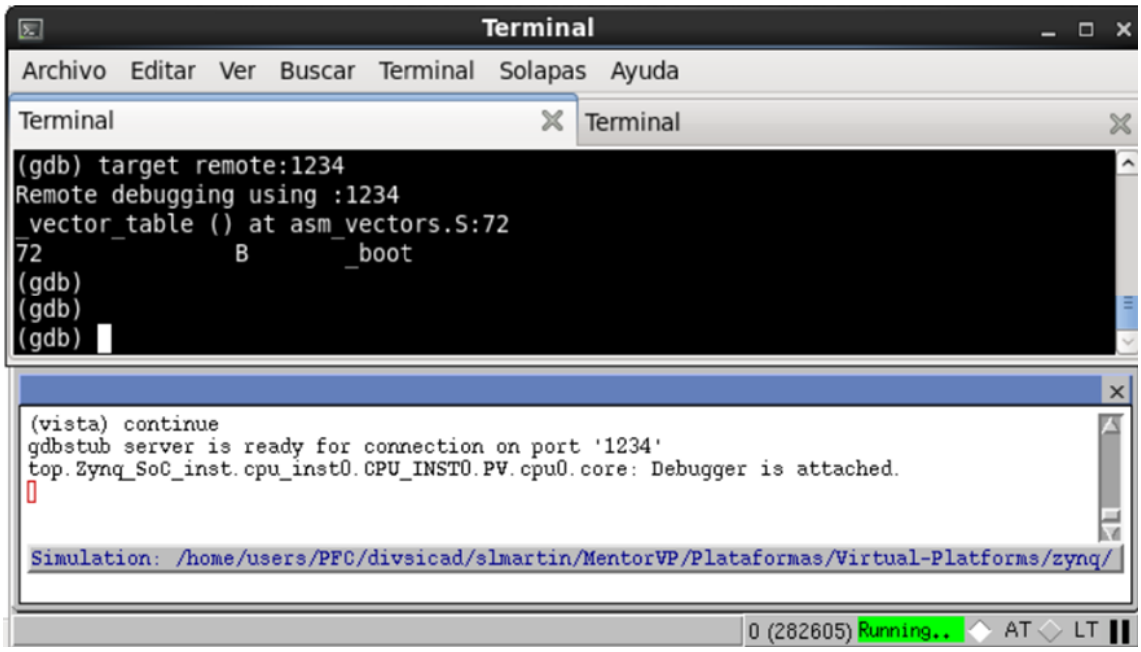



**Figura 34: Conexión al depurador realizada**

Haciendo uso de los comandos del depurador se puede avanzar en el *software* y desde Vista realizar el depurado del *hardware.* Con el terminal GDB se puede ver en qué punto de la aplicación se encuentra la simulación (Figura 35).

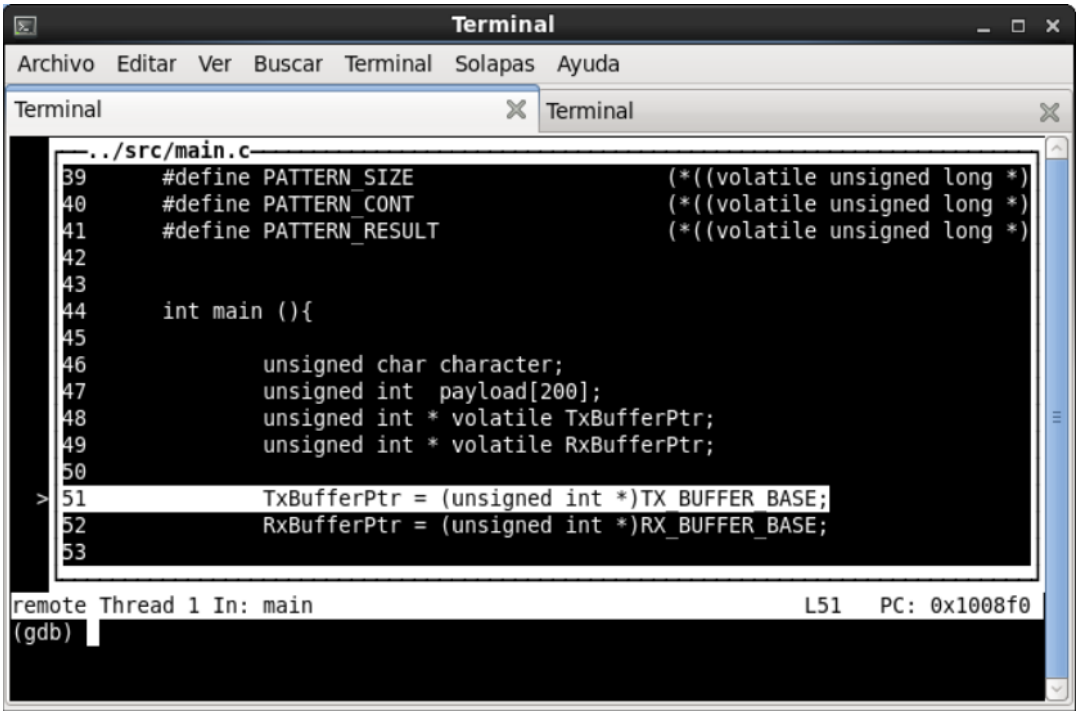



**Figura 35: Depurado del *software***

## 4.6 Creación del prototipo virtual

El prototipo virtual o plataforma virtual es un ejecutable el cual puede usarse para invocar la simulación de un diseño que se ha creado en Vista [56]. Por lo tanto, una plataforma virtual consta de:

1. El ejecutable de la simulación.
2. Librerías compartidas que sean requeridas para la simulación.
3. Archivos de tiempo de ejecución de Vista necesarios para ejecutar la simulación.
4. Archivos opcionales que se especifican en la creación de la plataforma.

Desde el entorno gráfico de Vista se puede crear la plataforma virtual del diseño a partir del fichero *sc_main*. Posteriormente es necesario configurar diversos parámetros, como el nombre que se le va a dar, la ruta de los ficheros necesarios, imagen a cargar en el procesador, tipo de licencia que se dispone para la ejecución de la plataforma.

## 4.7 Conclusiones

En este capítulo se muestra los conceptos básicos de la herramienta y los diversos pasos que se deben seguir para la creación de una plataforma virtual. Esto servirá como base para la creación de la plataforma virtual de la arquitectura DPI.

# Capítulo 5. Arquitectura Xilinx Zynq 7000

En este capítulo se explica la arquitectura de la Xilinx Zynq 7000, en concreto, la parte del sistema de procesamiento que es la que se emulará para conseguir la plataforma virtual. Además, se explica las interfaces AXI, necesarias para la comunicación entre el sistema.

## 5.1 SoC Programable Xilinx Zynq 7000

Los sistemas de la familia Zynq 7000 están compuestos por un *System on Chip* en el que en un único circuito integrado incorpora una FPGA y dos núcleos microprocesadores ARM Cortex-A9 con sus correspondientes extensiones NEON y otros bloques de control. Ambas partes están conectadas a través de interfaces AMBA AXI4, permitiendo la comunicación *hardware*/*software* con coprocesadores integrados en un mismo espacio de memoria. Ello permite manejar los elementos del sistema.

La arquitectura de los dispositivos Zynq está organizada en dos grandes bloques: Sistema de Procesamiento (PS) y Lógica Programable (PL). Ambos bloques pueden usarse de manera conjunta o de forma independiente, pudiendo realizar diseños complejos en un paradigma de integración *hardware*/*software*. Estas plataformas cuentan, por tanto, con la flexibilidad que ofrecen los núcleos procesadores y la escalabilidad y prestaciones que ofrece el paralelismo de la lógica programable. Una ventaja adicional y que facilita su adopción en términos de TTM, es la ventaja en comparación con los sistemas basados en ASIC, ya que reduce el tiempo de desarrollo y el coste.

La arquitectura de este dispositivo se muestra en la Figura 36, donde se diferencian los dos bloques que conforman la arquitectura y se interconectan con las interfaces AMBA AXI. La parte de procesamiento cuenta con dos núcleos ARM Cortex A9, además de distintos periféricos.

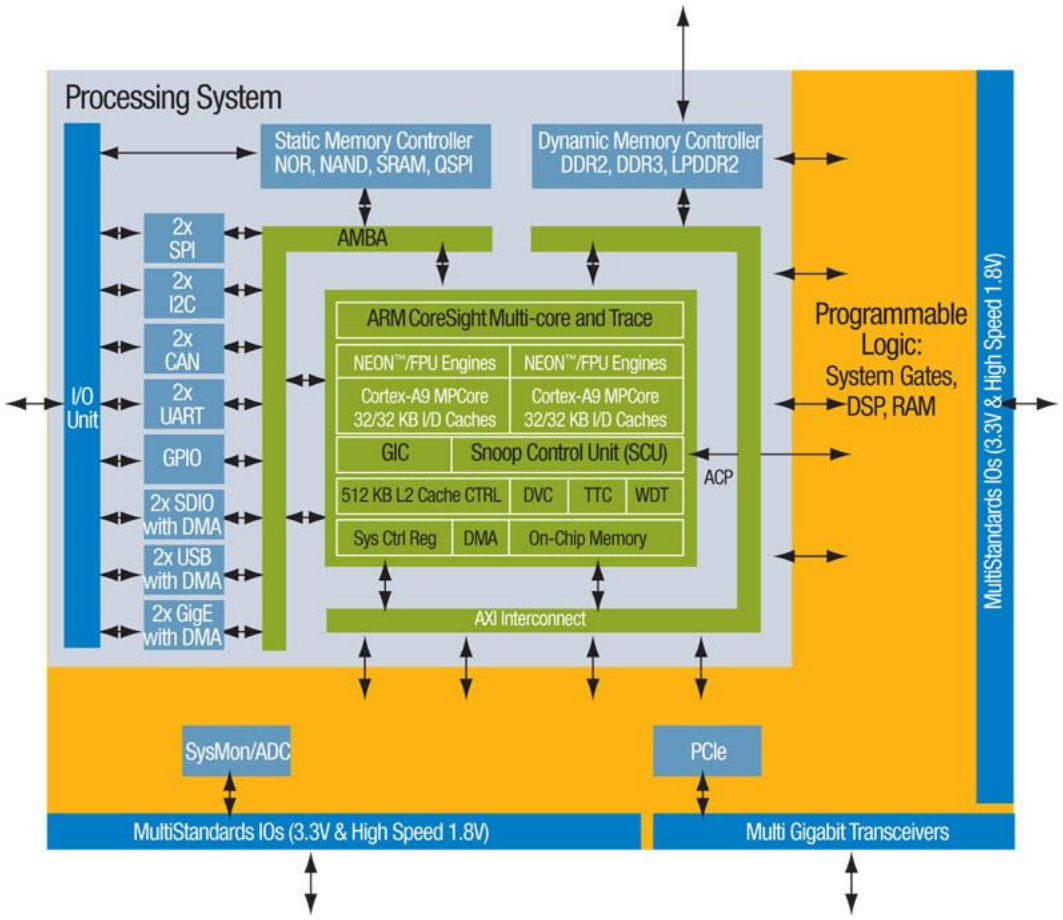



**Figura 36: Arquitectura de Zynq 7000** [57]

### 5.1.1 Sistema de Procesamiento (PS)

En los dispositivos Zynq se distingue la parte de procesamiento de la lógica programable debido a que posee una solución *hard processor*, contando con dos núcleos de procesamiento dedicados

Los núcleos ARM Cortex-A9 pueden operar a una frecuencia de hasta 1 GHz. Una particularidad de estos dispositivos es que da la posibilidad de configurar el sistema para ejecutar un único procesador, elegir cuál de los dos seleccionar o también proporcionar multiprocesamiento simétrico (SMP) o asimétrico (AMP). Cada uno de los procesadores

puede funcionar con sistema operativo (soluciones Linux para ARM principalmente) o sin sistema operativo en configuración *baremetal* (Figura 37).

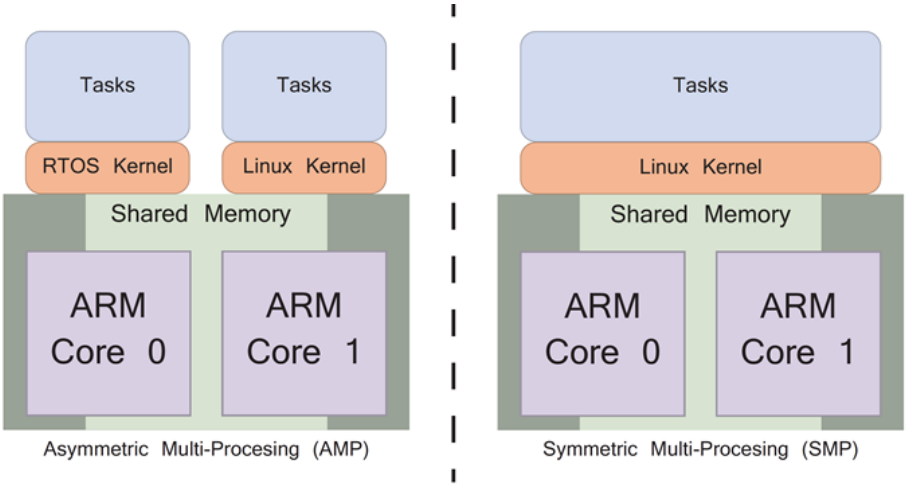



**Figura 37: Configuraciones multiproceso en el dispositivo Zynq: configuración asimétrica y simétrica**

La elección de cada configuración depende del tipo de aplicación en la que está utilizándose el dispositivo ya sea para aplicaciones orientadas a tiempo real, donde se prefiere una solución AMP principalmente *baremetal,* frente a soluciones orientadas al procesamiento de altas prestaciones donde se emplea una solución SMP (Figura 38).

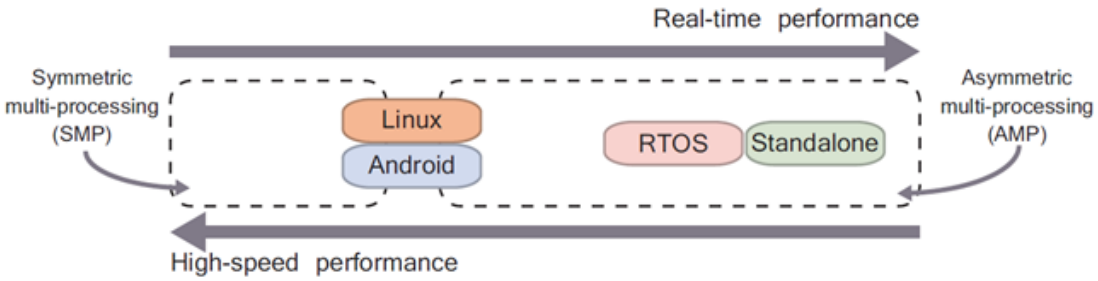



**Figura 38: Determinación del sistema operativo en función de la aplicación**

A través de la herramienta Xilinx *Software Development Kit* (SDK) se puede desarrollar el *software* para los procesadores ARM, ya que el compilador soporta el desarrollo de aplicaciones empotradas para diferentes procesadores integrados en el ecosistema de Xilinx. Además, es capaz de generar el lincador a medida del sistema. El flujo de desarrollo de la aplicación con SDK se muestra en la Figura 39. En la se muestra la interfaz de usuario del entorno SDK.

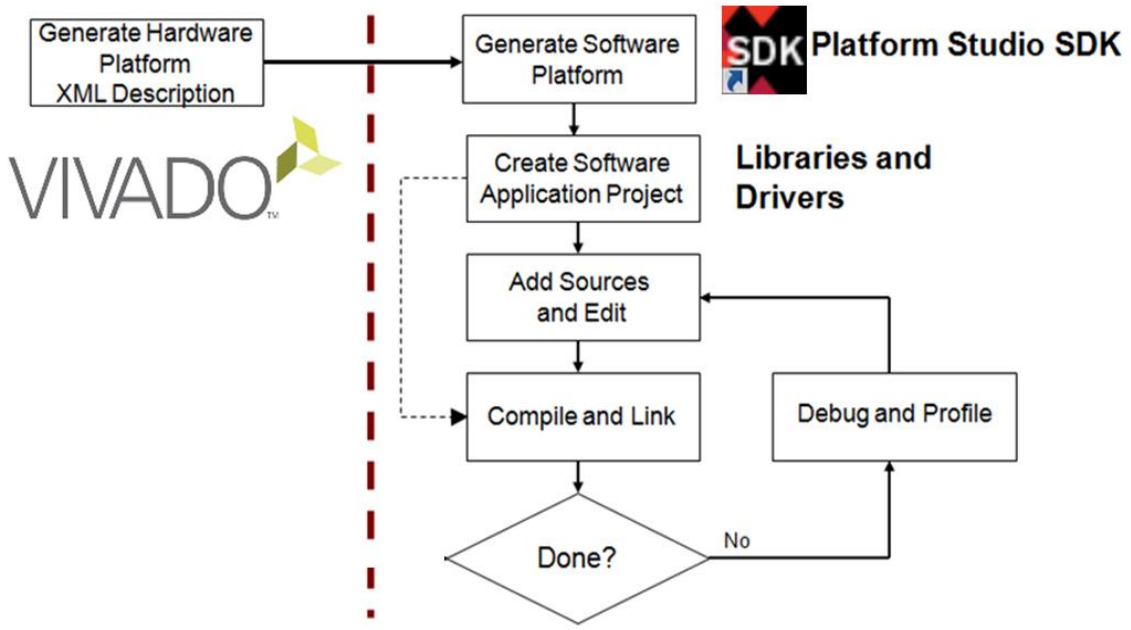



**Figura 39: Flujo de desarrollo del software con SDK.**

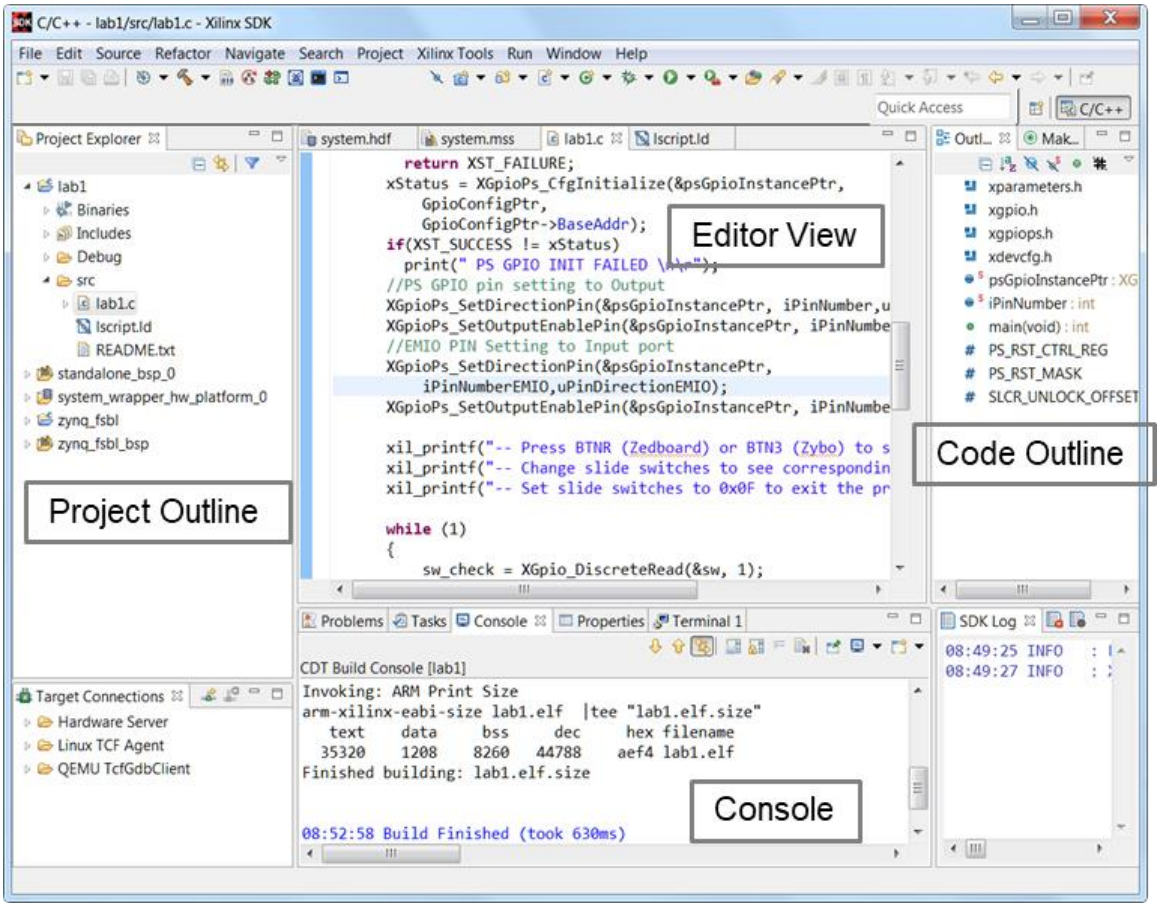



**Figura 40. Interfaz de usuario de SDK**

EL PS se organiza en diferentes bloques, tal como se indica a continuación:

- Unidad de Procesamiento de Aplicación (APU)
    - ARM Cortex-A9 MPCore, que implementa la arquitectura ARM v7. Incluye el coprocesador NEON, memorias Cache L1 y L2 y *timers.*
    - Elementos de gestión del sistema:
        - Registros de Control de Nivel de Sistema (SLCRs) que permiten controlar el comportamiento completo del PS.
        - La Unidad de Control de *Snoop* (SCU) permite la transparencia de datos y la coherencia entre los procesadores.
        - El Puerto de Coherencia del Acelerador (ACP) que, a través de una arquitectura basada en maestro/esclavo, realiza la conexión entre los bloques PL y PS, donde el maestro es la parte lógica y el esclavo el sistema de procesamiento.
        - El Controlador de Interrupciones Generales (GIC) que se encarga de manejar las interrupciones de tres tipos distintos: interrupciones de periféricos privados (PPI), interrupciones generadas por *software* (SGI) e interrupciones de periféricos compartidos (SPI).
        - Temporizadores y *watchdog.*
        - Controlador DMA compartido para PS y PL.
        - Memoria SRAM integrada de 256 KB con paridad y doble puerto.
- Interfaces de memoria: DDR Controller, Quad-SPI controller, Static Memory Controller (SMC).
- Periféricos de E/S: El PS es el encargado de manejar las interfaces con los componentes externos. Esta comunicación entre el PS y las interfaces externas se consigue a través de entradas y salidas multiplexadas (MIO), que se componen de 54 pines que permite una conexión flexible debido a que es el usuario quien define el mapeado entre el periférico y el pin. También cuenta con entradas y salidas de propósito general (GPIO) como son botones, *switchs* y LEDs que pueden ser usadas de diferentes maneras. Los periféricos que ofrece el dispositivo son los siguientes:

- 2x SPI (*Serial Peripheral Interface Bus*)
- 2x I2C (*Inter-Integrated Circuit*)
- 2x Bus CAN (*Controller Area Network*)
- 2x UART (*Universal Asynchronous Receiver-Transmitter*)
- GPIO (*General Purpose I/O*): 54 canales MIO, 64 canales EMIO
- 2x SDIO (*Secure Digital Input Output*)
- 2x USB
- 2x Gigabit Ethernet

### 5.1.2 Lógica programable (PL)

La parte de lógica programable está formada por una estructura FPGA, que se compone de distintos bloques lógicos configurables (CLB) unidos entre sí por una matriz de conexiones programables, multiplexores y bloques de entrada/salida. Un CLB está compuesto por dos *slices* que alberga cada una 4 LUTs, 8 Flip-Flops y sumadores para funciones aritméticas. Las LUTs son recursos flexibles que permiten implementar funciones lógicas, memorias ROM, RAM, registros de desplazamiento y, de manera combinada, memorias de mayor tamaño.

Aparte de estos bloques, existen dos componentes de propósito especial: *Block* RAM (BRAM) para uso extensivo de memoria y bloques DSP para aritmética de alta velocidad. Las BRAM pueden implementar memorias RAM, ROM o FIFOs, cada una de ellas pueden almacenar 36Kb de información, pudiéndose combinar para alcanzar mayores tamaños. El uso de BRAM es una alternativa eficiente al uso de RAM distribuida implementada con LUTs.

## 5.2 Arquitectura de Interconexión entre PS y PL

La parte lógica y la de procesamiento del sistema pueden comunicarse a través de una arquitectura de interconexión basadas en interfaces AMBA AXI4, así como un puerto ACP, como se muestra en la Figura 41.

Existen distintos tipos de interfaces entre el PS y el PL que se muestran continuación [58]:

- *General Purpose AXI.* Interfaces de propósito general con un tamaño de 32 bits para comunicaciones directas entre el PS y el PL. Existen cuatro interfaces de este tipo, de las cuales dos de ellas son maestras y otras dos son esclavas (considerando el PS como bloque maestro).
- *High Performance Ports.* Estas interfaces para alto rendimiento permiten que la lógica programable pueda acceder a las memorias DDR y OCM que se encuentra dentro del PS a través de esta interfaz de alto rendimiento, ya que dispone de cuatro interfaces designadas del PL hasta el PS configurables a 32 o 64 bits y conectadas a través de controladores FIFO con una latencia reducida.
- *Accelerator Coherency Port:* Permiten conectar el PL a la SCU (*Snoop Control Unit*) de la APU, permitiendo dar coherencia entre la caché de la APU y los elementos que componen la parte del PL, a través de un bus de 64 bits asíncrono.

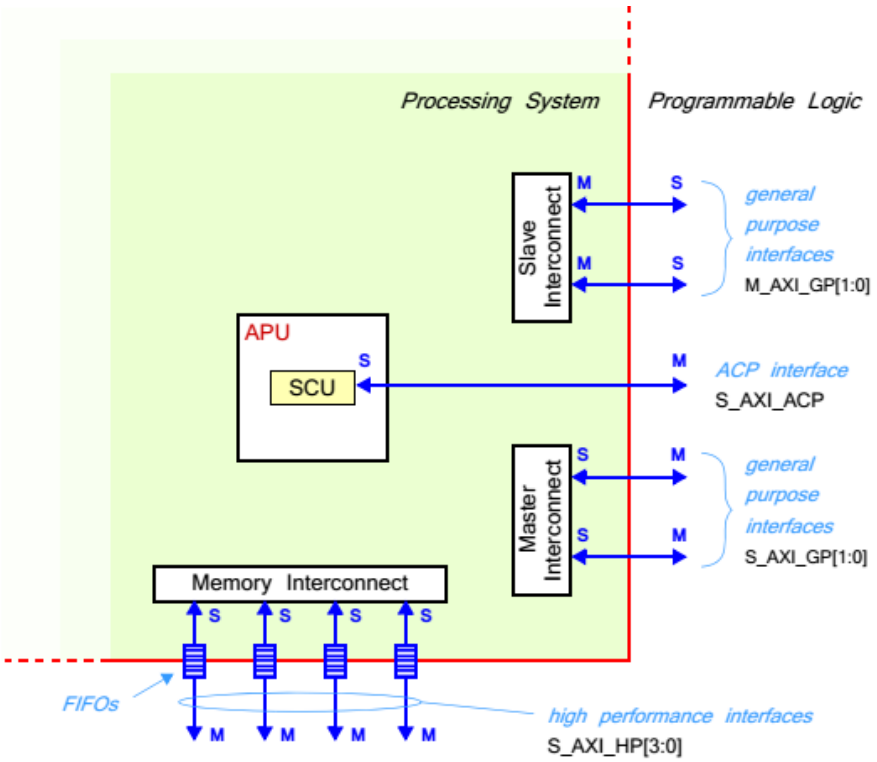



**Figura 41: Conexión entre PS y PL** [59]

## 5.3 Zynq Virtual Platform

La herramienta Vista proporciona una plataforma virtual del dispositivo Xilinx Zynq-7000 completamente funcional, incluyendo el mapa de memoria, asignación de interrupciones, incorporación de sistema operativo, etc. El diagrama de bloques de este dispositivo se muestra en la Figura 42, donde los bloques marcados están implementados en la plataforma virtual. Estos bloques están modelados en TLM-2.0 siguiendo las descripciones del manual técnico de referencia de Xilinx Zynq-7000 [58].

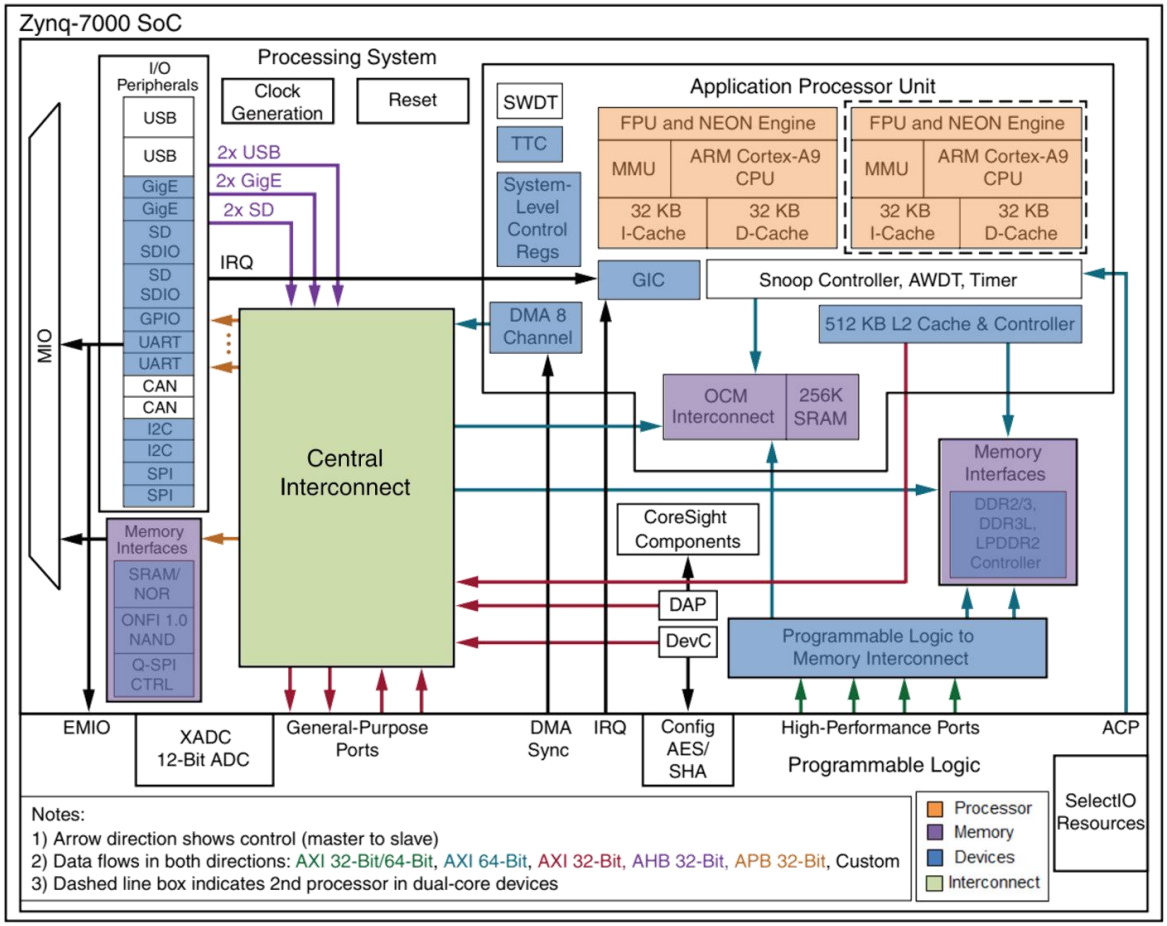



**Figura 42: Diagrama de bloques del dispositivo Zynq-7000 SoC** [60]

### 5.3.1 Modelos Zynq

Vista incluye una librería que se denomina *zynq_models* (Figura 43) donde se encuentran todos los modelos que definen el comportamiento real del dispositivo Zynq-7000. Los modelos contienen políticas temporales, que se ajustan con sus especificaciones, y políticas de consumo de potencia.

Con estos modelos, Vista proporciona el diagrama de bloques de toda la plataforma a nivel de bloques (Figura 44) y también en un único bloque que encapsula toda la plataforma (Figura 45).

En los siguientes apartados se presentan algunos de los bloques del dispositivo modelado en Vista.


**Figura 43: Librería *zynq_models***

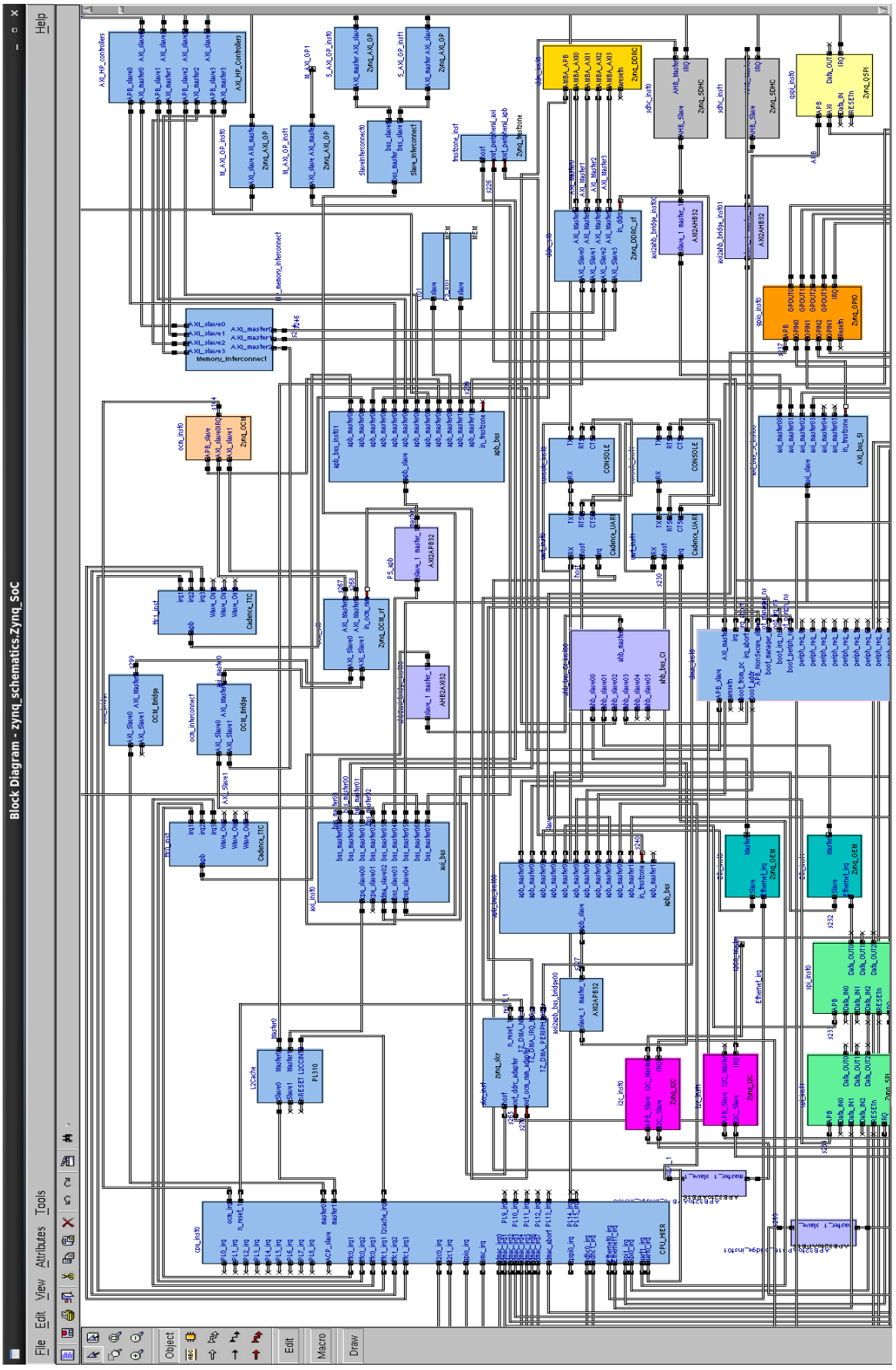



**Figura 44: Diagrama de bloques de la plataforma Zynq-7000 modelada en Vista**

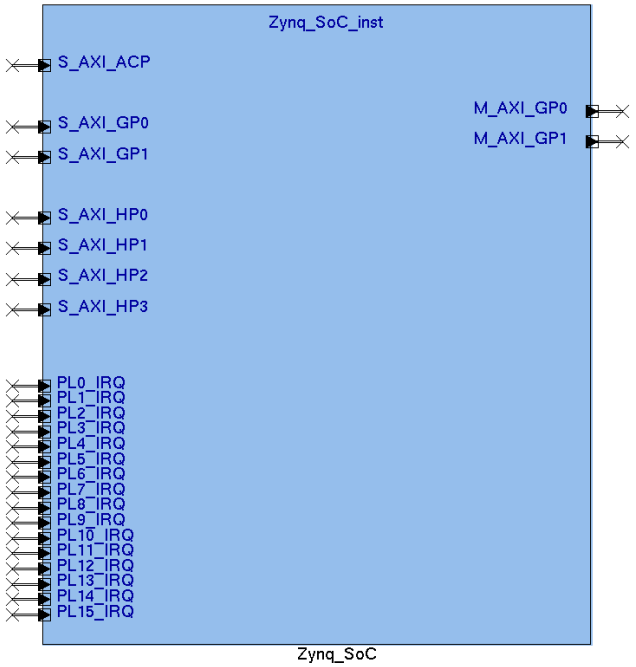



**Figura 45: Modelo del Zynq en Vista**

#### 5.3.1.1 Modelo ARM Cortex A9-MP

El modelo en Vista del Cortex A9-MP incluye en un único bloque (Figura 46) el modelado de las interfaces, parámetros específicos para controlar la CPU, registros, rangos de memorias privados, etc. Además de esto, incluye [47]:

1. V7A ARM *Instruction-Set*
2. *Thumb-2 Instruction-Set*
3. Cache de nivel 1 Datos/Instrucciones integrada
4. *V7A Memory Management Unit (MMU)*
5. *VFPv3 Floating Point Unit*
6. *PMU (Performance monitor unit)*
7. *Distributor Interrupt Controller*
8. *Snoop Control Unit*
9. *Accelerator Coherence Port*
10. *Local Timers*
11. *Global Timer*
12. *Watchdog*
13. *Timer*
14. NEON

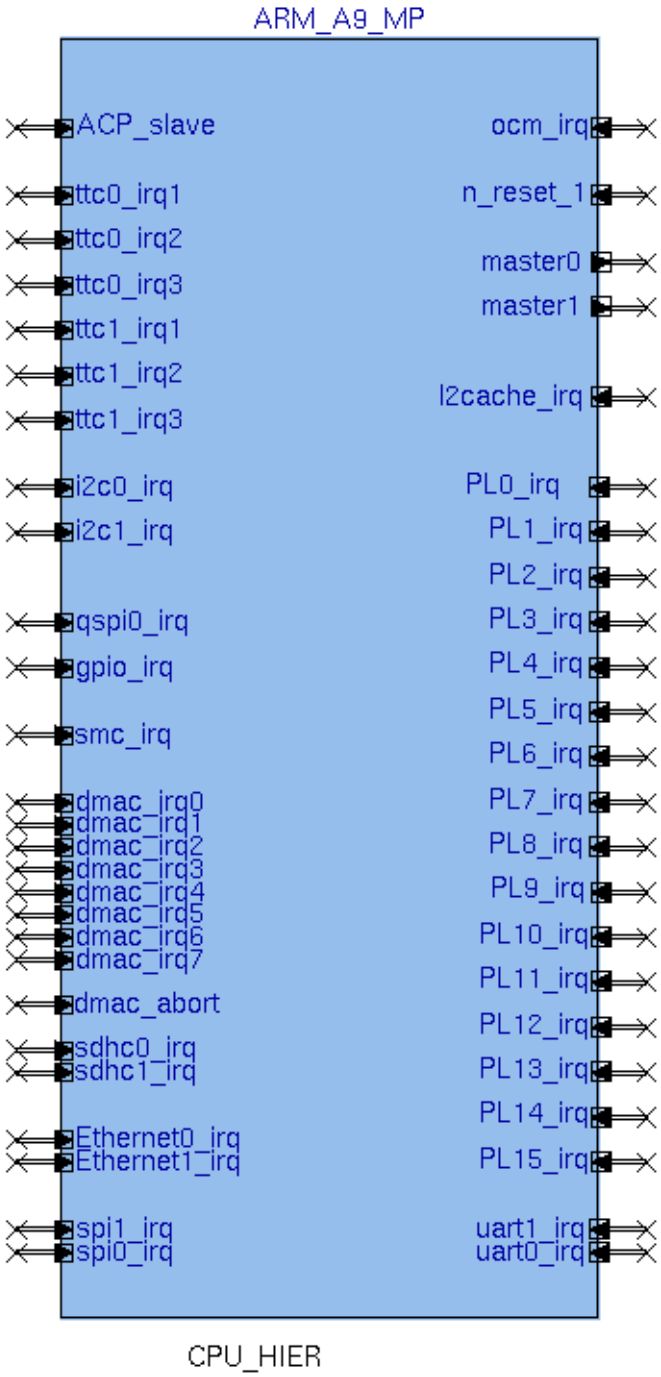



**Figura 46: Modelado del Cortex A9-MP en Vista**

### 5.3.1.2 Modelo AXI *High Performance*

El modelo *Zynq_AXI_HP* define cada una de las interfaces AXI HP que contiene el dispositivo (Figura 47). Haciendo uso de las jerarquías, se dispone del modelo *AXI_HP_Controllers* (Figura 48), que es el controlador de interfaces AXI *High Performance* que proporciona a la lógica programable comunicarse con la memoria DDR y OCM del PS a través de un bus con un gran ancho de banda.

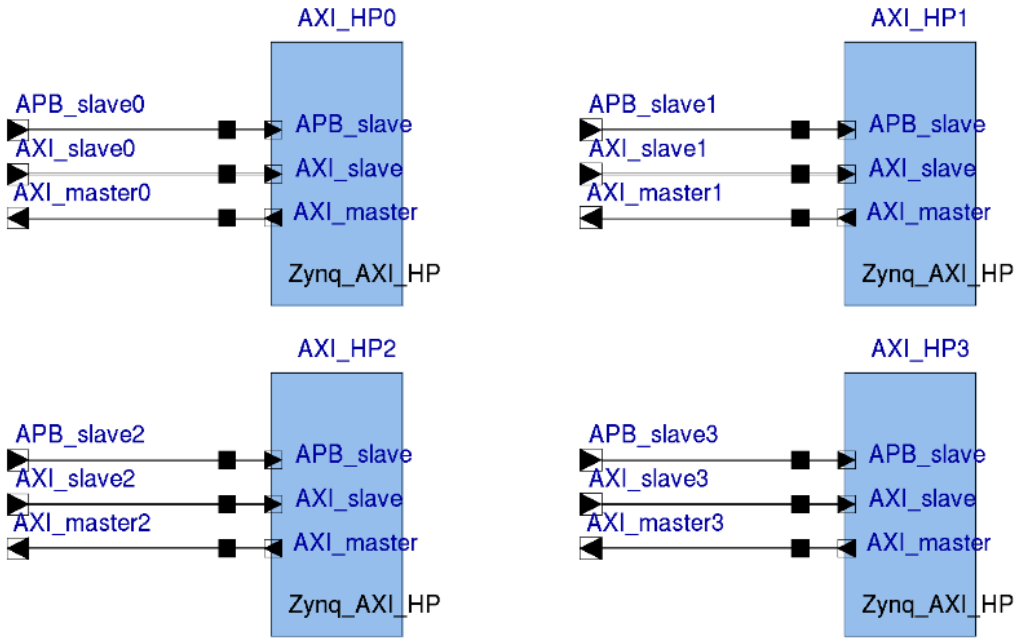



**Figura 47: Modelo de interfaces AXI HP**

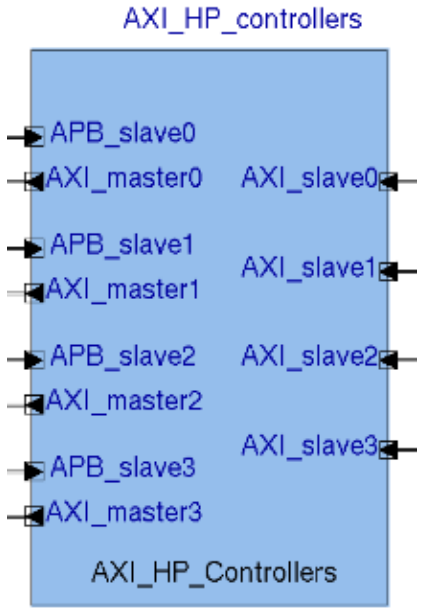



**Figura 48: Modelo del controlador AXI HP**

### 5.3.1.3 Modelo DDR *Controller*

El modelo de Zynq DDR Controller (DDRC) contiene cuatro interfaces esclavas AMBA AXI para acceder a la memoria interna del DDR y una interfaz esclava AMBA APB para programar los registros del controlador de memoria (Figura 49).

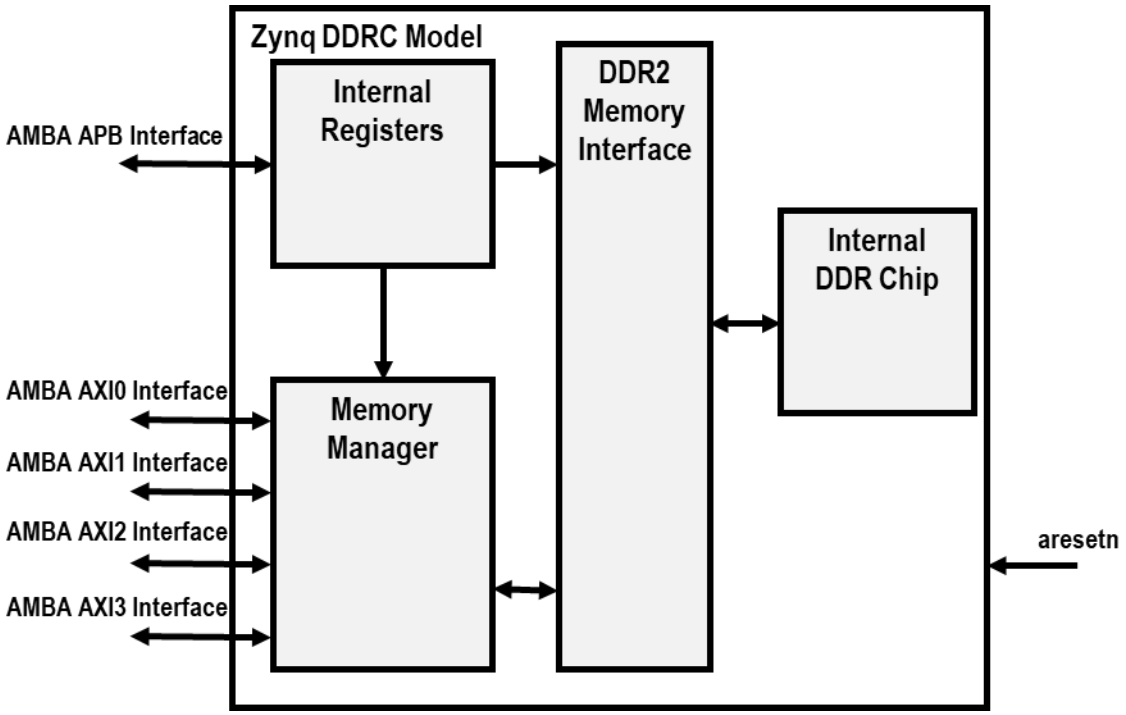



**Figura 49: Diagrama de bloques Zynq DDRC** [61]

El modelo en Vista incluye el modelado de las interfaces siguiendo el protocolo, parámetros que permiten controlar el modelo, los registros y la memoria de 1GB en un único bloque (Figura 50).

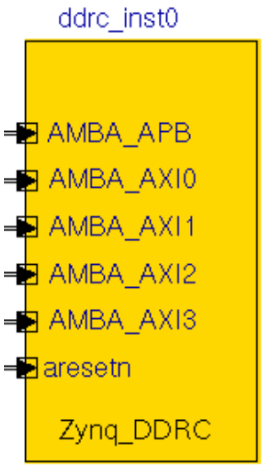



**Figura 50: Modelo en Vista del DDRC**

#### 5.3.1.4 Modelo AXI General Purpose

El modelado de las interfaces AXI GP permite comunicación directa entre el PL y el PS a través de la cache L2 haciendo uso de interconexiones y buses (Figura 51).

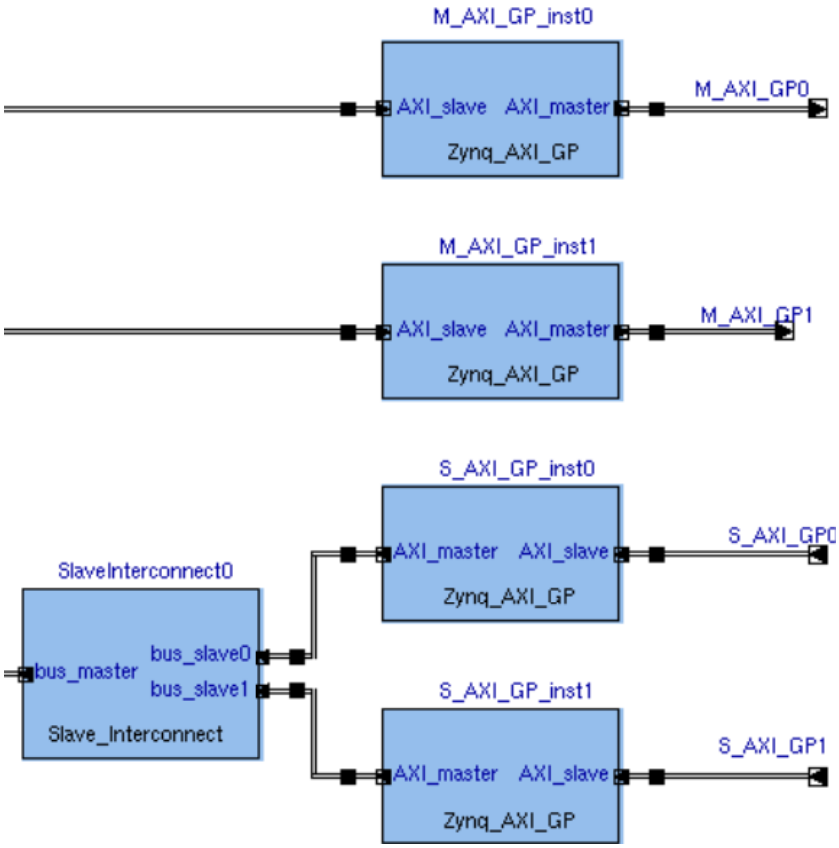



**Figura 51: Modelo de las interfaces AXI GP en Vista**

### 5.3.2 Mapa de memoria del SoC

La plataforma cuenta con un mapa de direcciones de memoria que indica las zonas reservadas para los diversos bloques. En la Tabla 3 se muestra el mapa de memoria definido en Xilinx Zynq-7000 Virtual Platform [62] que cuenta con la dirección base y el tamaño de la memoria asignada para cada recurso, siguiendo las especificaciones del manual de referencia [58].

**Tabla 3: Mapa de memoria de Zynq-7000** [62]

| MODELO | ZYNQ BASE ADDRESS | TAMAÑO |
|---|---|---|
| **DDR (MEMORY)** | 0x00000000 | 1 GB |
| **M_AXI_GP0** | 0x40000000 | 1 GB |
| **M_AXI_GP1** | 0x80000000 | 1 GB |
| **UART0** | 0xE0000000 | 4 KB |
| **UART1** | 0xE0001000 | 4 KB |
| **I2C0** | 0xE0004000 | 4 KB |

| MODELO | ZYNQ BASE ADDRESS | TAMAÑO |
|---|---|---|
| **I2C1** | 0xE0005000 | 4 KB |
| **SPI0** | 0xE0006000 | 4 KB |
| **SPI1** | 0xE0007000 | 4 KB |
| **GPIO** | 0xE000A000 | 4 KB |
| **GEM0** | 0xE000B000 | 4 KB |
| **GEM1** | 0xE000C000 | 4 KB |
| **QSPI** | 0xE000D000 | 4 KB |
| **SMC** | 0xE000E000 | 4 KB |
| **SDIO0 (SDHC0)** | 0xE0100000 | 4 KB |
| **SDIO1 (SDHC1)** | 0xE0101000 | 4 KB |
| **SMC MEMORIES** | 0xE1000000 | 96 MB |
| **SLCR** | 0xF8000000 | 2 KB |
| **TTC0** | 0xF8001000 | 4 KB |
| **TTC1** | 0xF8002000 | 4 KB |
| **DMAC WHEN SECURE** | 0xF8003000 | 4 KB |
| **DMAC WHEN NON-SECURE** | 0xF8004000 | 4 KB |
| **DDRC** | 0xF8006000 | 4 KB |
| **AXI_HP0** | 0xF8008000 | 4 KB |
| **AXI_HP0** | 0xF8009000 | 4 KB |
| **AXI_HP0** | 0xF800A000 | 4 KB |
| **AXI_HP0** | 0xF800B000 | 4 KB |
| **OCM (PROGRAMMABLE REGISTERS)** | 0xF800C000 | 4 KB |
| **ARM MP (SCU, TIMER, GIC)** | 0xF8F00000 | 8 KB |
| **L2C (PL310)** | 0xF8F02000 | 4 KB |
| **QUAD-SPI LINEAR ADDRESS FOR LINEAR MODE** | 0xFC000000 | 32 MB |
| **OCM (ADDRESSABLE MEMORY)** | 0xFFFC0000 | 256 KB |

### 5.3.3 Mapa de Interrupciones

Las interrupciones del sistema se manejan a través del *Generic Interrupt Controller* (GIC) enviando a la CPU interrupciones de los periféricos y de la lógica programable. En la Tabla 4 se muestra las interrupciones que cuenta la plataforma virtual Zynq-7000 basándose en el manual de referencia [58].

**Tabla 4: Mapa de Interrupciones Zynq-7000** [62]

| IRQ ID | MODELO | DESCRIPCIÓN |
|---|---|---|
| **34** | L2Cache | L2 Cache petición de interrupción |
| **35** | OCM | On-Chip Memory petición de interrupción |
| **42** | TTC0 | TTC 0 Timer 1 petición de interrupción |
| **43** | TTC0 | TTC 0 Timer 2 petición de interrupción |
| **44** | TTC0 | TTC 0 Timer 3 petición de interrupción |
| **45** | DMAC | DMA330 abortar petición de interrupción |
| **46-49** | DMAC [0:3] | DMA330 petición de interrupción de los canales 0:3 |
| **50** | SMC | SMC petición de interrupción |
| **51** | QSPI | Quad-SPI petición de interrupción |
| **52** | GPIO | GPIO petición de interrupción |
| **54** | GEM0 | GEM 0 petición de interrupción |
| **56** | SDIO0 | SDIO 0 (SDHC 0) petición de interrupción |
| **57** | I2C0 | I2C 0 petición de interrupción |
| **58** | SPI0 | SPI 0 petición de interrupción |
| **61-68** | PL [0:7] | Petición de interrupción desde el PL |
| **69** | TTC1 | TTC 1 Timer 1 petición de interrupción |
| **70** | TTC1 | TTC 1 Timer 2 petición de interrupción |
| **71** | TTC1 | TTC 1 Timer 3 petición de interrupción |
| **72-75** | DMAC [4:7] | DMA330 petición de interrupción de los canales 4:7 |
| **77** | GEM1 | GEM 1 petición de interrupción |
| **79** | SDIO1 | SDIO 1 (SDHC 1) petición de interrupción |
| **80** | I2C1 | I2C 1 petición de interrupción |
| **81** | SPI1 | SPI 1 petición de interrupción |
| **82** | UART1 | UART 1 petición de interrupción |
| **84-91** | PL [8:15] | Petición de interrupción del PL |

## 5.4 Conclusiones

En este capítulo se ha descrito la arquitectura del dispositivo Zynq 7000, incluyendo el sistema de procesamiento (PS) como la lógica programable (PL). Igualmente se ha descrito la arquitectura de interconexión y finalmente se presenta el modelo de plataforma virtual disponible para el SoC.

# Capítulo 6. Modelado en TLM-2.0 de la arquitectura DPI

En este capítulo se presenta la arquitectura y el flujo de datos de la plataforma DPI de referencia a partir de la cual se crea la plataforma virtual en este TFM. Una vez definidos los diversos bloques que compone el sistema se explica cómo se ha realizado la creación de los modelos en TLM.

## 6.1 Introducción

Con el uso de las plataformas virtuales se sustituye la implementación física del dispositivo por un modelo virtual donde todos los componentes de la Zynq-7000 están modelados a nivel de transacciones. Los diversos bloques IP que conforman la plataforma DPI deben elevar también su nivel de abstracción para conseguir que toda la arquitectura se encuentre modelada a nivel de sistema. Para conseguir esto se modelan los bloques a nivel TLM-2.0, permitiendo la interconexión directa entre el prototipado de la Zynq-7000 y los bloques IP al estar basado todo en el protocolo AXI4.

Una vez definido el protocolo de comunicación de los modelos TLM, el siguiente paso a seguir es definir el comportamiento de los diversos bloques IP. Anteriormente, en el Capítulo 4: El entorno de desarrollo Mentor Vista, se comentó que Vista proporciona la creación de modelos TLM a través de diseños realizados en SystemC. Tal y como están diseñados los bloques IP de la arquitectura de referencia, no es posible realizar esta acción. Esto se debe a que las interfaces de comunicación están implementadas a nivel de señales

y no en canales debido a que la herramienta de diseño utilizada no permitía la implementación del protocolo AXI4.

En este capítulo se explica el proceso de modelado en SystemC TLM-2.0 de los bloques de referencia.

## 6.2 Arquitectura de la plataforma DPI de referencia

La plataforma de referencia realiza la inspección profunda de paquetes mediante un conjunto de bloques implementados en el PL del dispositivo SoC FPGA. En el PS se implementa la aplicación *software que* configura los diversos bloques que compone la arquitectura.

Esta arquitectura se muestra en la Figura 52. Se compone de los siguientes bloques:

- Sistema de procesamiento de la Zynq.
- Bloques TEMAC que gestionan el acceso a la red
- Bloque *Header Analyzer* que se encarga de analizar la cabecera de los paquetes TCP/IP y decidir si se inspeccionan en profundidad o no.
- Bloque *Eliminar Cabecera* que separa la cabecera del *payload* del paquete.
- Bloque de memoria FIFO para almacenar los paquetes mientras se inspeccionan.
- Bloque de *Motor de Búsqueda* para realizar la inspección profunda de los paquetes.

La arquitectura de comunicación de la plataforma utiliza buses AXI4-Stream entre los bloques IP y buses AXI4-Lite para la configuración de los bloques desde el sistema de procesamiento.

El flujo de datos principal de la plataforma DPI es el siguiente. A través del bloque TEMAC1 se recoge el paquete procedente de la red y mediante el bus AXI4-*Stream* se transmite el paquete al bloque *Header Analyzer.* Este bloque es el encargado de analizar la cabecera del paquete según unas restricciones preestablecidas. Si no existe coincidencia entre la cabecera del paquete y las restricciones, se envía el paquete a la red a través de un bloque TEMAC2 de salida. En caso contrario, si existe coincidencia el paquete es enviado

a un *Motor de Búsqueda* para realizar el análisis del *payload.* Para que realice el análisis de la carga útil del paquete es necesario eliminar la cabecera del paquete. Para ello, se incorpora un bloque denominado *Eliminar Cabecera* que elimina la parte correspondiente a la cabecera del paquete. El paquete se almacena de manera completa en una memoria FIFO ya que, si después de realizar la inspección se debe devolver a la red, es necesario que cuente con su cabecera.

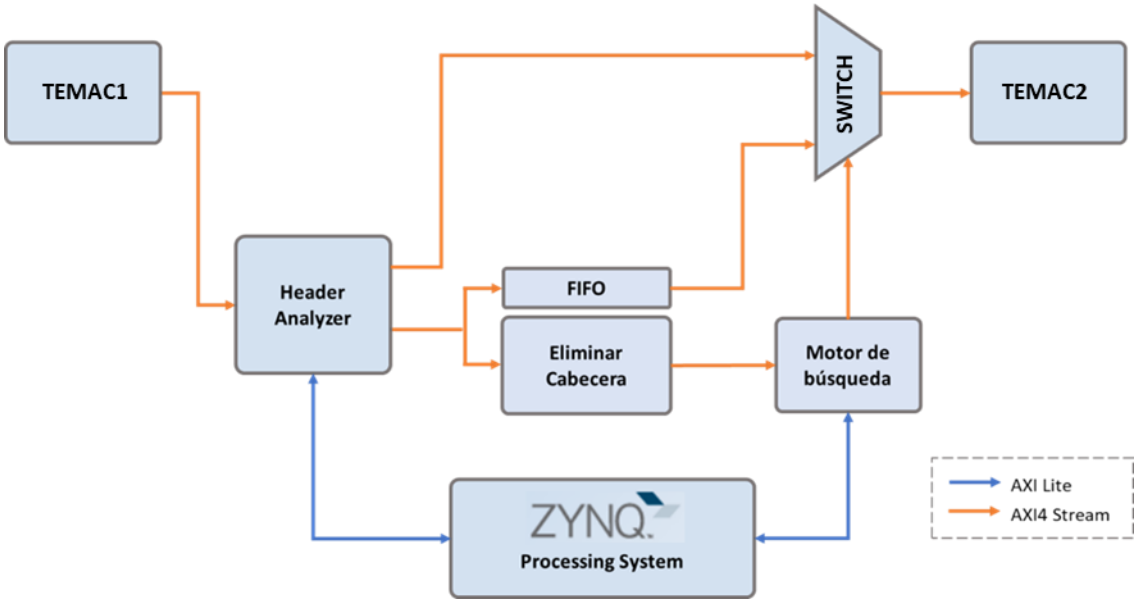



**Figura 52: Arquitectura de la plataforma DPI de referencia**

El *Motor de Búsqueda* cuenta con un número determinado de bloques de análisis y, para acelerar el proceso, los bloques implementan distintos tipos de algoritmos de búsqueda. El motor de búsqueda tras realizar la inspección profunda del paquete puede proporcionar dos tipos de respuesta: desechar el paquete, terminando aquí el flujo de este paquete; o devolver el paquete a la red.

El bloque TEMAC cuenta con una única entrada de datos, por lo que es necesario incorporar un *SWITCH* que permita pasar tanto el paquete enviado desde el *Header Analyzer* como el almacenado en la FIFO. La respuesta del *Motor de Búsqueda* la recibe este *SWITCH* y según su resultado vacía la FIFO o realiza una lectura del paquete.

Definida la plataforma de referencia y su flujo de datos, se procede a la sustituir los bloques TEMAC por una fuente que entregue paquetes a la plataforma con el fin de poder realizar la validación de ésta. Es por ello por lo que se ha optado por crear un bloque que

genere tráfico aleatorio. La plataforma queda como se muestra en el diagrama de la Figura 53.

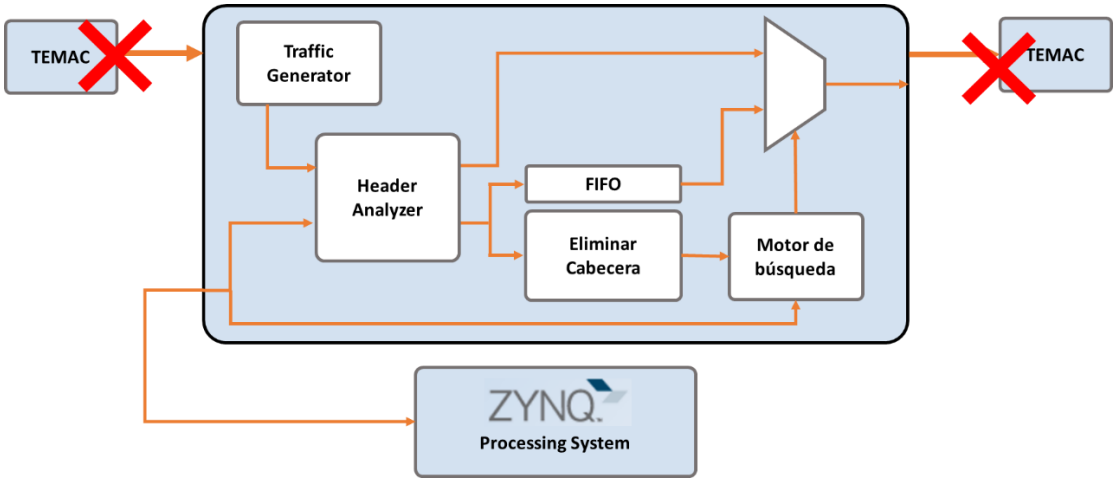



**Figura 53: Diagrama de la arquitectura DPI a implementar**

## *6.3* Modelado en TLM del bloque *Traffic Generator*

El bloque *Traffic Generator* genera tráfico aleatorio donde cada paquete creado tiene la misma longitud de un paquete tomado de la red. Los puertos con los que cuenta este bloque se definen en la Figura 54 y se detalla a continuación:

- *Slave*: Puerto esclavo de 32 bits basado en el protocolo AXI para activar el generador a través del microprocesador.
- *Master*: Puerto maestro de 32 bits basado en el protocolo AXI que realiza la escritura de los paquetes generados sobre el *Header Analyzer.*

El puerto esclavo tiene asociado un registro para la activación del bloque desde la aplicación empotrada. Se selecciona la activación del *trigger* cuando el registro sea escrito. En la Tabla 5 se muestra las características de este registro y en la Figura 55 la definición en la herramienta Vista.

**Tabla 5: Registros de configuración del *Traffic Generator***

| NOMBRE | DESCRIPCIÓN | TAMAÑO | DIRECCIÓN |
|---|---|---|---|
| **START** | Registro que activa el funcionamiento del bloque | 1 byte | 0x00000000 |

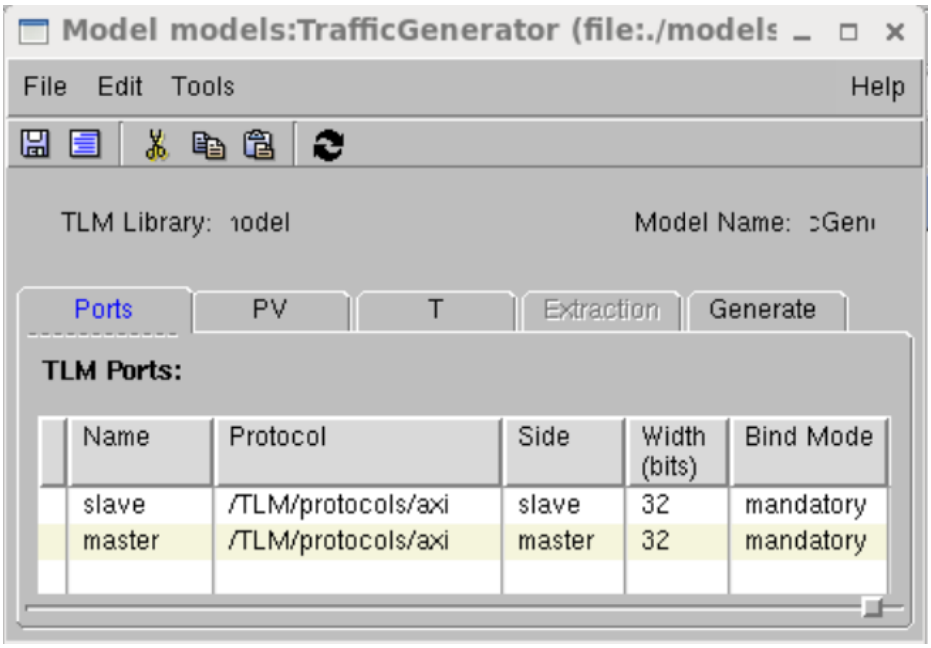



**Figura 54: Definición de puertos del *Traffic Generator***

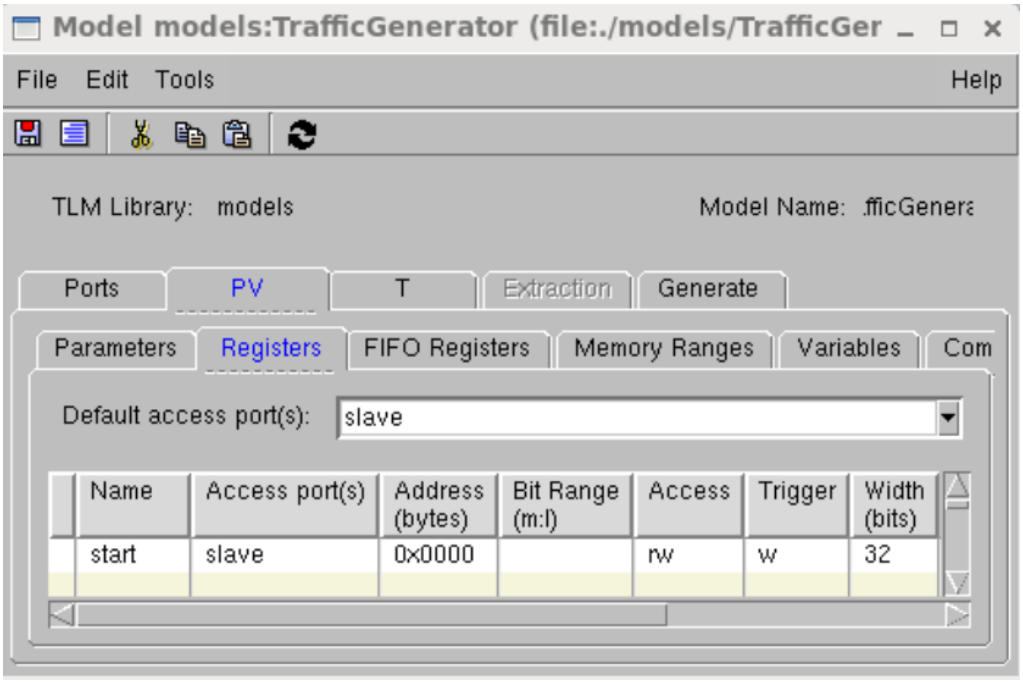



**Figura 55: Definición de registros del *Traffic Generator***

En la pestaña T se define el comportamiento temporal y de potencia del bloque. Se ha aplicado la política de *delay* para el puerto esclavo para definir el tiempo de respuesta de la transacción. La política *sequential* para definir cuánto tarda en generar una transacción de salida al ocurrir una escritura en el esclavo. En la Figura 56 se muestra la aplicación de estas políticas mediante la interfaz gráfica.

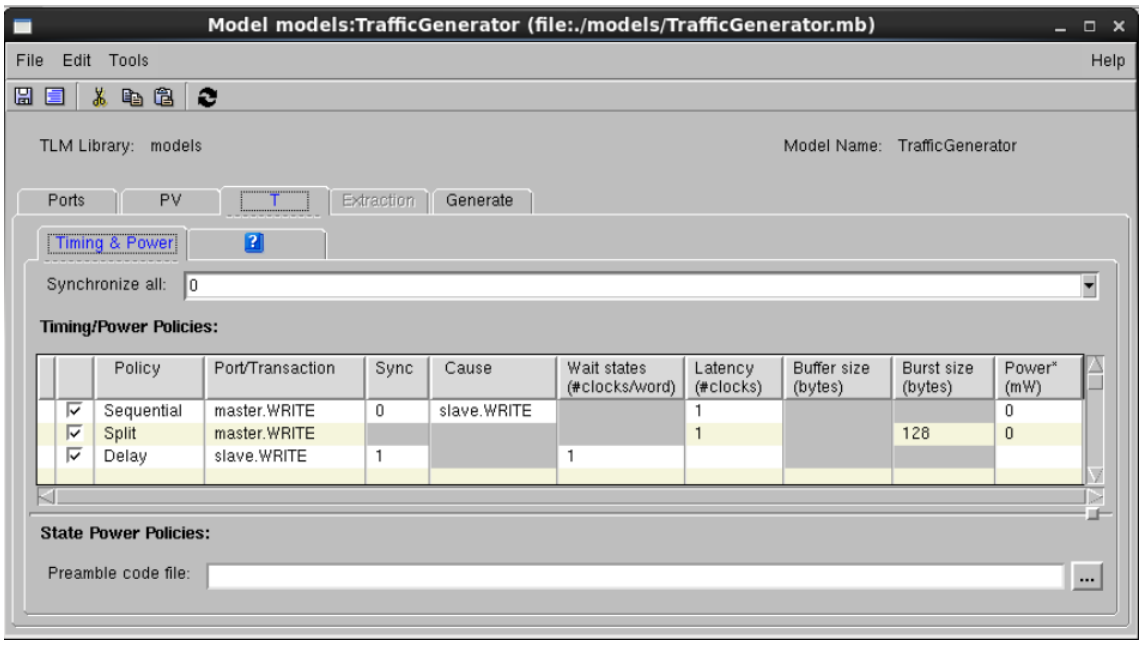



**Figura 56: Definición de políticas temporales y de potencia del *Traffic Generator***

Definidos los puertos, registros y las políticas, se puede generar el modelo en TLM-2.0 para definir su comportamiento. En la función *callback* del registro se comprueba el dato registrado y se comprueba su valor. Si el valor es 0x01 significa que la aplicación está solicitando la activación del generador de tráfico, para esto se hace una llamada a la función *generator()* que es la encargada de generar el tráfico aleatorio y transmitirlo a través del puerto maestro. Aparte del tráfico aleatorio, se ha almacenado en un array un paquete cuyos valores de cabecera de red y TCP/IP son conocidos, permitiendo saber si el analizador de cabecera realiza su función. Además, en el *payload* se han insertado diversas palabras a modo de patrón que permiten la comprobación del motor de búsqueda *Boyer-Moore.* En el Código 1 se muestra cómo se ha definido el comportamiento de este bloque.

```
1  #include "TrafficGenerator_pv.h"
2  #include <iostream>
3
4  using namespace sc_core;
5  using namespace sc_dt;
6  using namespace std;
7
8  //constructor
9  TrafficGenerator_pv::TrafficGenerator_pv(sc_module_name
10 module_name)
11   : TrafficGenerator_pv_base(module_name) {
```

```
12  }
13
14  // Write callback for start register.
15  // The newValue has been already assigned to the start register.
16  void TrafficGenerator_pv::cb_write_start(unsigned int newValue) {
17    if(newValue){
18      generator();
19    }
20  }
21  ...
22
23  void TrafficGenerator_pv::generator() {
24
25    unsigned int Size = 358;
26    unsigned int wData [Size];
27    unsigned int rData [Size];
28    for (unsigned int i = 0; i < Size; i++){
29         wData[i] = rand () % 0xffffffff;
30         rData[i] = rand () % 0xffffffff;
31    }
32    master_write (0x0, payload, Size);
33    master_write (0x0, wData, Size);
34    master_write (0x0, wData, Size);
35  }
36
37
```

**Código 1: Función *callback* del registro *Trigger* y función *generator()* del *Traffic Generator***

## 6.4 Descripción del bloque *Header Analyzer*

El bloque *Header Analyzer* de referencia se ha tomado de [17] y su objetivo es capturar los paquetes Ethernet recibidos mediante bloques controladores MAC, obteniendo un flujo constante de datos entrantes y salientes. Una vez capturados se realiza un filtrado de paquetes al analizar la cabecera Ethernet y TCP/IP. Después de realizar distintas comparaciones con los campos de la cabecera se toma una decisión. Dependiendo de este resultado el paquete puede ser devuelto a la red o enviado a un bloque de inspección profunda.

En la Figura 57 se muestran los distintos módulos que componen el bloque IP. A nivel de interfaces el bloque posee una interfaz AXI4-Lite necesaria para configuración del bloque IP, una interfaz de entrada AXI4-Stream que recibe el flujo de datos desde la red y dos interfaces de salida AXI4-Stream, para volcar los datos a la red o al analizador de paquetes.

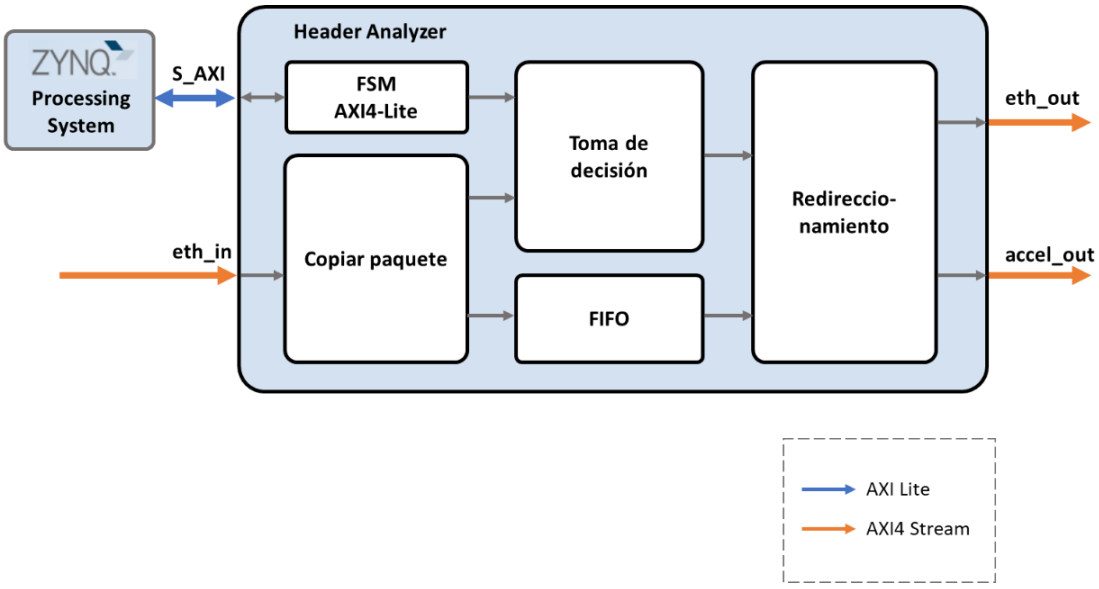



**Figura 57: Arquitectura del bloque *Header Analyzer* de referencia**

Para realizar el filtrado, el paquete se divide y se organiza en palabras de 32 bits, en lugar de manejar el paquete completo. Esto mejora las prestaciones del sistema debido a que reduce las latencias totales al no tener que esperar por el paquete entero para empezar a realizar el filtrado. La información para analizar se encuentra en las primeras trece palabras; por lo tanto, mientras se realiza el filtrado a través de condiciones específicas, como el filtrado por protocolo, dirección de destino o incluso conjunto de distintos parámetros, el resto del paquete se va copiando para después ser reenviado.

El bloque de captura cuenta con un módulo y tres procesos (Figura 58) que, en conjunto, define el funcionamiento del bloque. El módulo proporciona un proceso asociado a la máquina de estados de la interfaz AXI4-Lite que será la encargada de configurar todo el bloque y está relacionado directamente con el proceso de toma de decisión, que realiza el filtrado del paquete. Los otros dos procesos son los encargados de copiar el paquete entrante y de redireccionarlo por la interfaz correcta dependiendo del resultado del análisis.

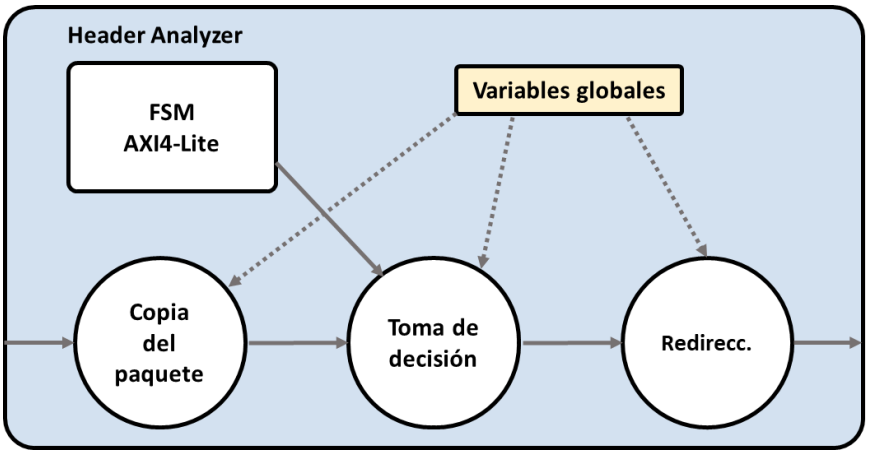



**Figura 58: Diagrama de procesos del *Header Analyzer***

### 6.4.1 Modelado en TLM del Header Analyzer

El primer paso para modelar a nivel de TLM el *Header Analyzer* es definir los parámetros necesarios para la comunicación entre bloques TLM (puertos, protocolos, tamaños). La definición de estos parámetros se muestra en la Figura 59 a través de la interfaz gráfica de Vista y son los siguientes:

- *slave_config*: Puerto esclavo de 32 bits basado en el protocolo AXI que recibe la configuración del filtrado de la cabecera.
- *slave_ether_in*: Puerto esclavo de 32 bits basado en el protocolo AXI que recibe los paquetes TCP/IP a analizar.
- *master_ether_out*: Puerto maestro de 32 bits basado en el protocolo AXI que envía los paquetes TCP/IP a la salida de red.
- *master_accel*: Puerto maestro de 32 bits basado en el protocolo AXI que envía los paquetes TCP/IP hacía los motores de búsqueda.

Para poder configurar los distintos parámetros del análisis de la cabecera de manera externa se hace uso de registros. Cada registro cuenta con una dirección pudiendo modificar o saber el valor de un registro realizando una lectura o escritura a esa dirección. Los registros de configuración de los distintos campos de la cabecera que se han implementado en el bloque se muestran en la Tabla 6.

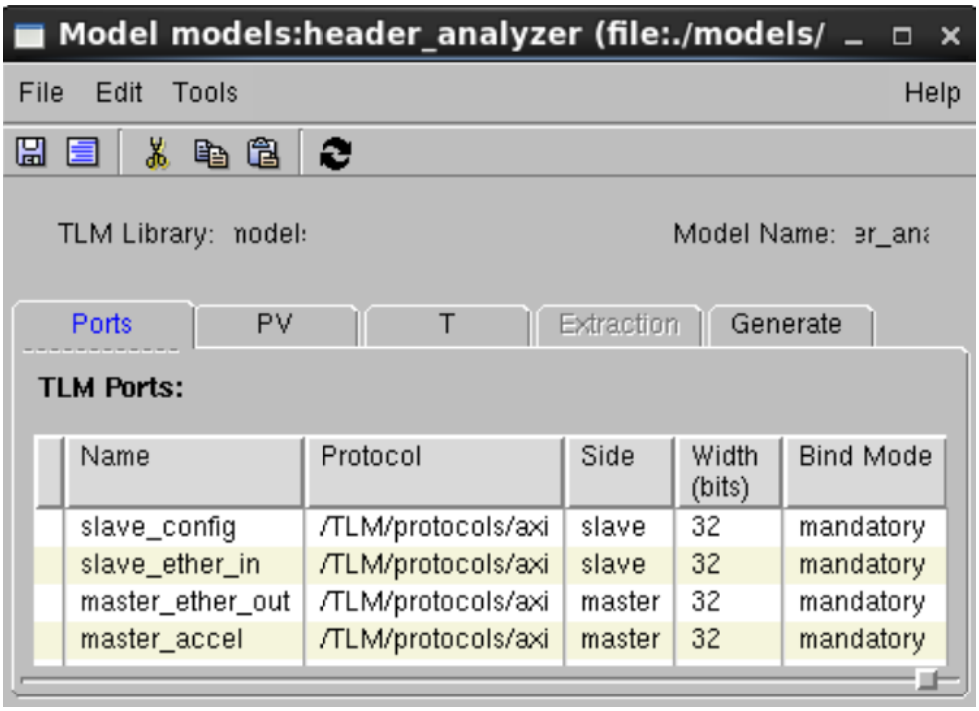



**Figura 59: Definición de puertos del *Header Analyzer***

**Tabla 6: Registros de configuración del *Header Analyzer***

| NOMBRE | DESCRIPCIÓN | TAMAÑO | DIRECCIÓN |
|---|---|---|---|
| **CHECK_ETHER** | Registro que determina si se realiza el análisis de la cabecera Ethernet | 1 byte | 0x00000090 |
| **DEST_MAC** | Registro con la dirección MAC de destino | 6 bytes | 0x00000000 |
| **ORG_DEST** | Registro con la dirección MAC origen | 6 bytes | 0x00000040 |
| **ETHERNET_TYPE** | Registro con el tipo Ethernet | 2 bytes | 0x00000070 |
| **CHECK_NETWORK** | Registro que determina si se realiza el análisis de la cabecera de red | 1 byte | 0x00001000 |
| **QOS** | Registro con el valor de la calidad de servicio | 1 byte | 0x00001010 |
| **PROTOCOL** | Registro donde se indica el tipo de protocolo se encapsula en la cabecera red | 1 byte | 0x00001018 |
| **IP_SRC** | Registro con la dirección IP origen | 4 bytes | 0x00001040 |
| **IP_DEST** | Registro con la dirección IP de destino | 4 bytes | 0x00001060 |

Desde la interfaz gráfica de Vista se puede definir los diversos parámetros de los registros: nombre, dirección, puerto esclavo asociado, tamaño en bits, accesos, etc. En la Figura 60 se muestra la configuración asociada a los registros.

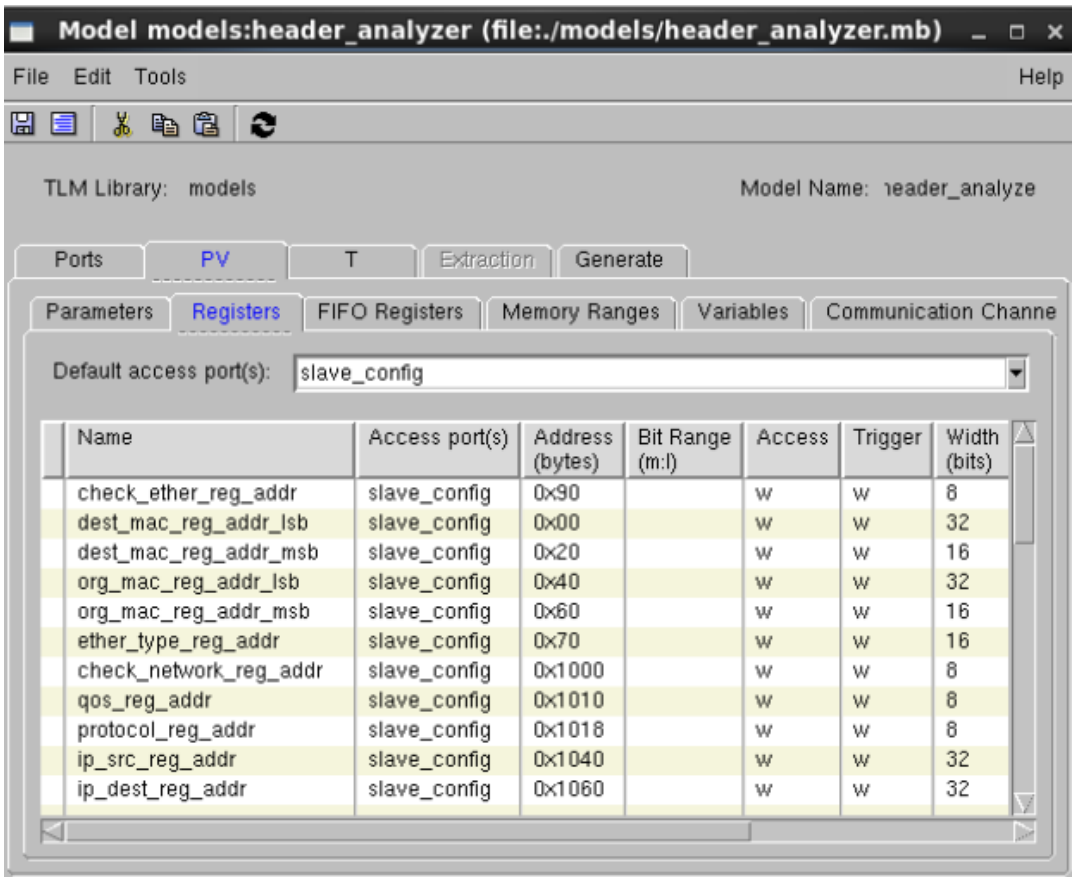



**Figura 60: Definición de registros del *Header Analyzer***

Para realizar el modelado temporal del bloque se aplica diversas políticas. La política de *delay* se aplica a los puertos esclavos donde se define el tiempo de respuesta del puerto en ciclos. Se aplica la política *sequential* en la escritura de los puertos maestros con el objetivo de definir el tiempo que el bloque tarda en realizar su función Figura 61.

Los valores definidos para la política *sequential* se ha tomado el tiempo máximo que toma el bloque *Header Analyzer* en realizar la inspección de todos los parámetros de la cabecera, que corresponde con 33 ciclos de reloj. También se ha añadido la potencia que consume la política *sequential* para definir el consumo de potencia del análisis de la cabecera. Este valor se ha tomado de los resultados obtenidos en [18] al implementar el *Header Analyzer* sobre el dispositivo Zynq ZC706.

A partir de esta información se genera el modelo del bloque basado en TLM-2.0 que cuenta con los *sockets* de los *Initiator* generados por los puertos maestros y los *sockets* de los *Target* generado por los esclavos, las llamadas a funciones de lectura y escritura, tanto para interfaces bloqueantes y no bloqueantes, *callback* de los esclavos, etc.

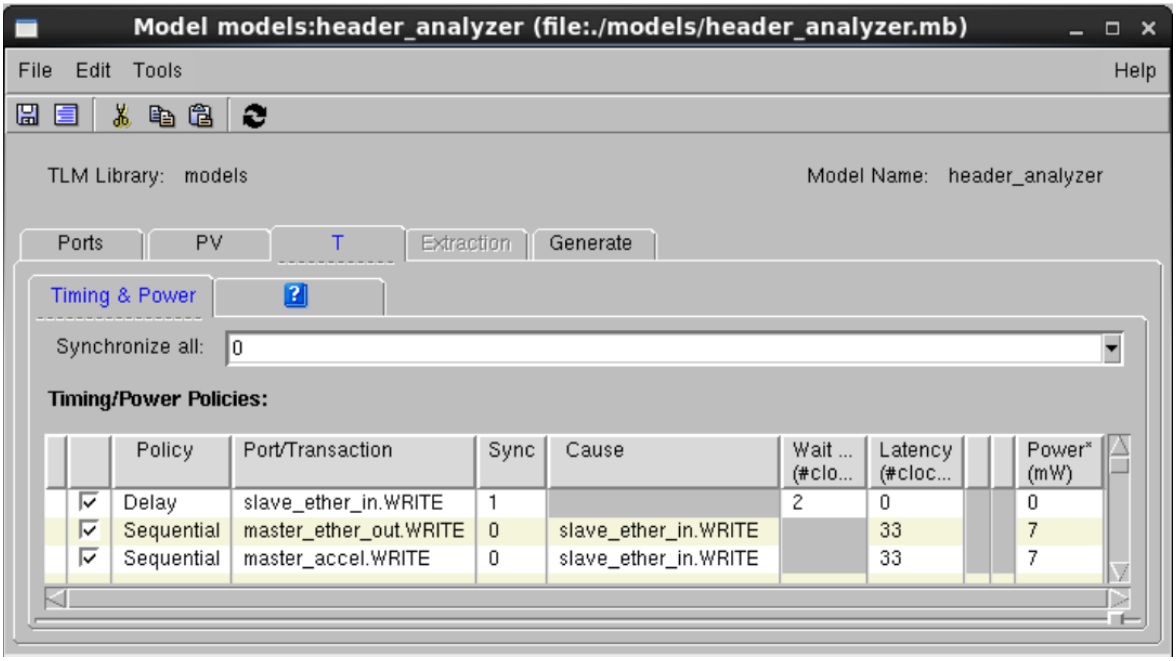



**Figura 61: Definición de políticas temporales y de potencia del *Header Analyzer***

Desde la capa PV se define el comportamiento de este bloque. Con la función *callback* del puerto esclavo que recibe los datos se toma la cabecera del paquete y se almacena en un array. Después se realiza una llamada a la función *compare()* que se encarga de analizar la cabecera. Cuando la función finaliza y retorna se analiza el resultado y, según este, se transmite los datos que están previamente almacenados en la FIFO a través de uno de los puertos maestros. Lo descrito anteriormente se muestra en detalle en el Código 2.

```
1  #include "header_analyzer_pv.h"
2  #include <iostream>
3
4  using namespace sc_core;
5  using namespace sc_dt;
6  using namespace std;
7
8  //constructor
9  header_analyzer_pv::header_analyzer_pv(sc_module_name
10 module_name)
11   : header_analyzer_pv_base(module_name) {
12 }
13
14 ...
```

```
15  // Write callback for slave_ether_in port.
16  // Returns true when successful.
17  bool header_analyzer_pv::slave_ether_in_callback_write
18  (mb_address_type address, unsigned char* data, unsigned size) {
19
20    unsigned int mem[size];
21    memcpy (mem,data,size);
22
23    for(int i=0; i<9; i++){
24      headers[i]=mem[i];
25    }
26    compare();
27    if(result==2){
28      cout << "Salida a ethernet\n" << endl;
29      master_ether_out_write(address, data, size);
30    }
31    if (result==3){
32      cout<<"Salida a acelerador\n" << endl;
33      master_accel_write(address, data, size);
34    }
35     return true;
36  }
37  ...
```

**Código 2: Función *callback* de escritura del puerto *slave_ether_in* del *Header Analyzer***

La función *compare(),* que se muestra en el Código 3, es donde se realiza el análisis y la toma de decisión de la cabecera al comparar el valor de cada uno de los registros de configuración del bloque con el valor de la cabecera.

El resultado del análisis realizado de cada campo de la cabecera se guarda en una variable interna, realizando una *or* lógica de cada resultado. Por lo tanto, si uno de los campos coincide con el almacenado en el registro el paquete debe ser enviado para su inspección al motor DPI. En caso contrario, si ninguno de los campos coincide, el paquete será reenviado a la red.

```
1  ...
2  void header_analyzer_pv::compare(){
3     bool _result;
4     if(check_ether_reg_addr == 0x1){
5  //---------------- MAC Addr Inspection ---------------------
```

```
6       _dest_mac.range(47,16) = headers[0];
7       sc_uint<32> temp =headers[1];
8       _dest_mac.range(15,0) = temp.range(31,16);
9       _org_mac.range(47,32) = temp.range(15,0);
10      _org_mac.range(31,0) = headers[2];
11  //---------------- Ethernet Type Inspection ------------------
12      _ether_type= headers[3].range(31,16);
13  //------------------Result--------------------------------
14      _result |= (_ether_type== ether_type);
15      _result |= (_org_mac== org_mac);
16      _result |= (_dest_mac== dest_mac);
17    }
18    if(check_network_reg_addr == 0x1){
19  //---------------- Qos Inspection ----------------------
20      _QoS = headers[3].range(7,0);
21  //---------------- Protocol Inspection --------------------
22      _protocol = headers[5].range(7,0);
23      _ip_src.range(31,16) = headers[6].range(15,0);
24      sc_uint<32> temp1 =headers[7];
25      _ip_src.range(15,0) = temp1.range(31,16);
26      _ip_dest.range(    31,16) = temp1.range(15,0);
27      _ip_dest.range(15,0) = headers[8].range(31,16);
28  //----------------------Result--------------------------------
29      _result |= (_QoS== QoS);
30      _result |= (_protocol== protocol);
31      _result |= (_ip_src== ip_src);
32      _result |= (_ip_dest== ip_dest);
33   }
34    if(_result) result = 3;
35    else result = 2;
36  }
37  ...
```

**Código 3: Función *compare()***

## 6.5 Descripción del bloque *Eliminar Cabecera*

Si se da el caso de que el paquete debe ser analizado por el *Motor de Búsqueda*, este paquete debe ser tratado, es decir, se debe eliminar la cabecera del paquete debido a que los analizadores solo realizan la inspección al *payload* o carga útil del paquete.

El bloque de referencia fue diseñado en [18] cuya funcionalidad cuenta con un único proceso en donde al comenzar la recepción de paquetes se activa un contador que se incrementa con cada palabra recibida. Las trece primeras palabras son desechadas y cuando el contador sea mayor que el tamaño de la cabecera se activa la interfaz de salida para enviar el paquete sin cabecera al *Motor de Búsqueda*.

### 6.5.1 Modelado en TLM del bloque Eliminar Cabecera

A nivel de modelado TLM, este bloque cuenta con dos puertos, un maestro y un esclavo, con protocolo AXI y un ancho de 32 bits para mantener la coherencia con los bloques a los que se conecta. En la Figura 62 se muestra la definición de estos puertos a través de Vista.

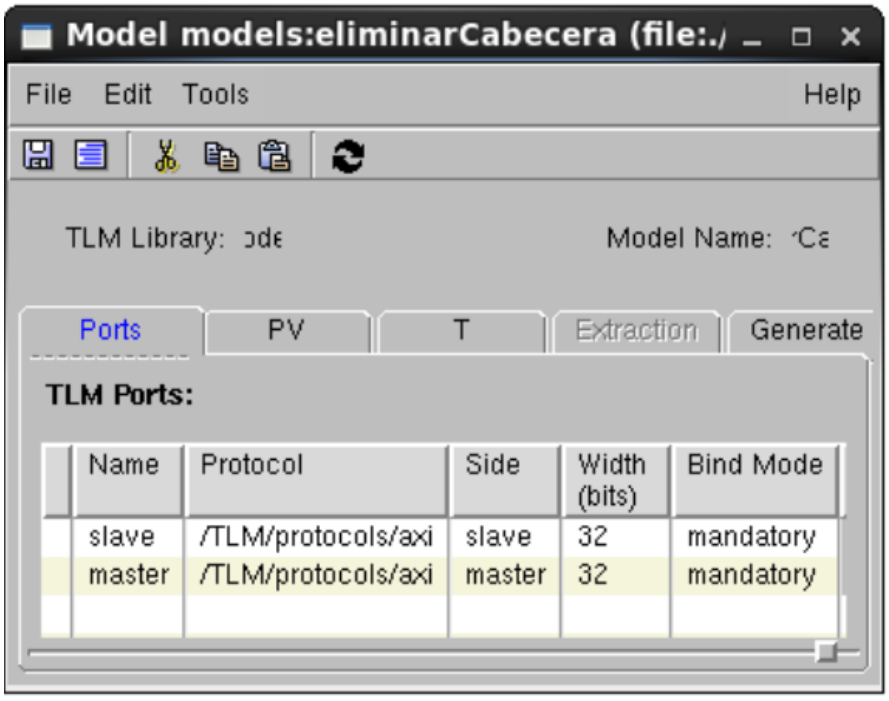



**Figura 62: Definición de puertos del bloque *Eliminar Cabecera***

Este modelo no cuenta con registros y en la capa T se definen las políticas de *delay* y *sequential* (Figura 63). En esta última se define el tiempo que tarda el bloque en procesar la transacción y se define como 14 ciclos que corresponde al tiempo empleado para eliminar la cabecera. También se define la potencia consumida en esta política que se ha definido como 1 mW y se ha tomado de la implementación física sobre la Zynq ZC706 realizada en [18].

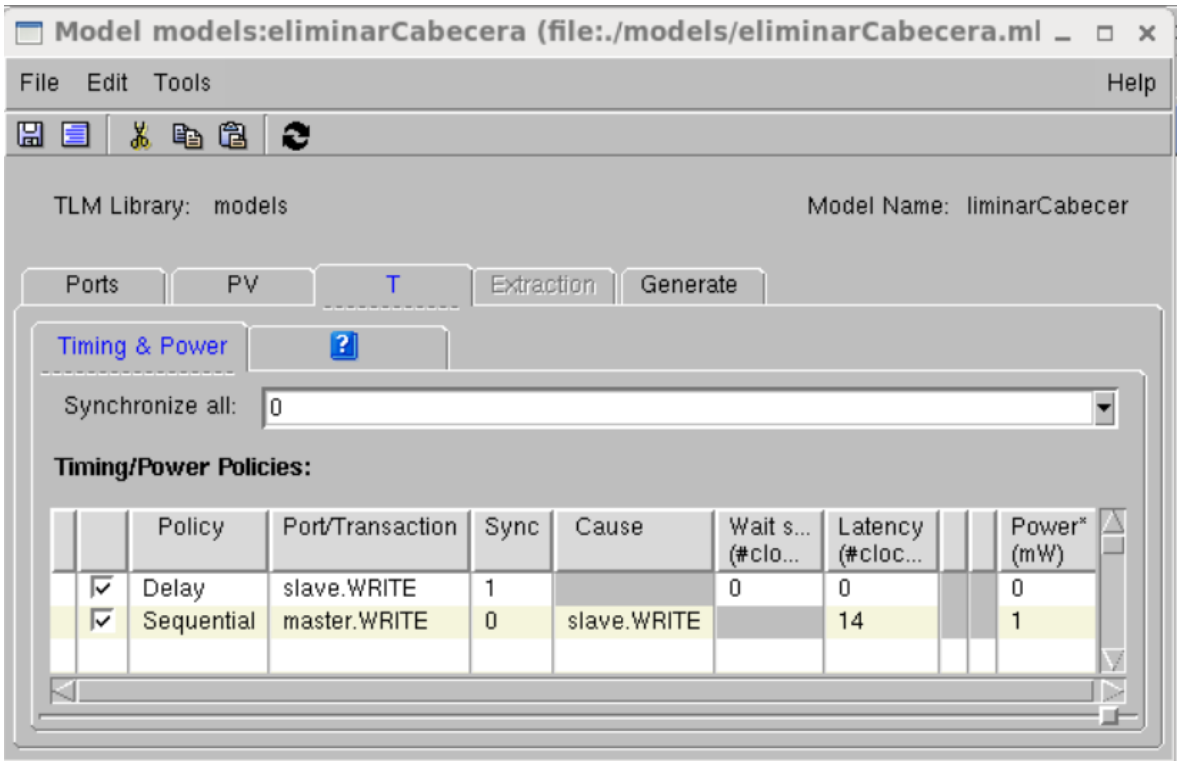



**Figura 63: Definición de políticas temporales y de potencia del *Eliminar Cabecera***

En la función *callback* del esclavo se define el comportamiento del bloque, que es el de transmitir el paquete TCP/IP hacia el motor de búsqueda sin las trece primeras palabras de 32 bits. Este proceso se muestra en el Código 4.

```
1  #include "eliminarCabecera_pv.h"
2  #include <iostream>
3
4  using namespace sc_core;
5  using namespace sc_dt;
6  using namespace std;
7
8  //constructor
9  eliminarCabecera_pv::eliminarCabecera_pv(sc_module_name
10 module_name)
11   : eliminarCabecera_pv_base(module_name) {
12 }
13
14 // Write callback for slave port.
15 // Returns true when successful.
16 bool eliminarCabecera_pv::slave_callback_write(mb_address_type
17 address, unsigned char* data, unsigned size) {
```

```
18
19    unsigned int mem[size];
20    memcpy (mem,data,size);
21    master_write(address, (mem+13),size/4);
22    return true;
23  }
24  ...
```

**Código 4: Función *callback* de escritura del puerto *slave* de *Eliminar Cabecera***

## 6.6 Descripción del bloque de análisis de patrones (*Boyer-Moore*)

Boyer-Moore es un algoritmo de búsqueda de patrones fijos dentro de una cadena de caracteres basado en dos reglas: la regla de carácter erróneo y la regla de sufijo correcto.

En la regla del carácter erróneo, cada parte de la cadena se comprueba con el patrón de derecha a izquierda. Si existe una coincidencia entre el patrón y el carácter, el patrón se desplazará por la cadena hasta la posición de coincidencia, si no coincide el patrón avanzará un número de posiciones definido por el tamaño del patrón.

La regla del sufijo correcto se aplica cuando el último carácter del patrón coincida con el último carácter de la parte de la cadena que se está inspeccionando. Se seguirá buscando coincidencias en la cadena hasta que se encuentre un carácter del patrón que no coincida con el carácter de la cadena con el que se ha alineado.

El algoritmo Boyer-Moore de la arquitectura de referencia se ha desarrollado en [16] y se encuentra implementado dentro del motor de búsqueda que se muestra en la Figura 64. El motor de búsqueda cuenta con una interfaz de entrada AXI4-Stream de 64 bits por la que recibe el *payload* y una interfaz de salida AXI4-Stream de 3 bits donde envía el resultado de la inspección. Estas interfaces son adaptadas por el bloque DIB (*Data Interface Block*) a interfaces tipo FIFO. El bloque *Ynode* se almacena el *payload* y los patrones para el análisis. Dentro de este bloque se cuenta con múltiples bloques SNB (*Search Node Block*) donde se implementan el algoritmo de Boyer-Moore. Por último, se cuenta con un bloque RTCB (*Real-Time Clock Block*) que genera un reloj en tiempo real para gestionar condiciones temporales.

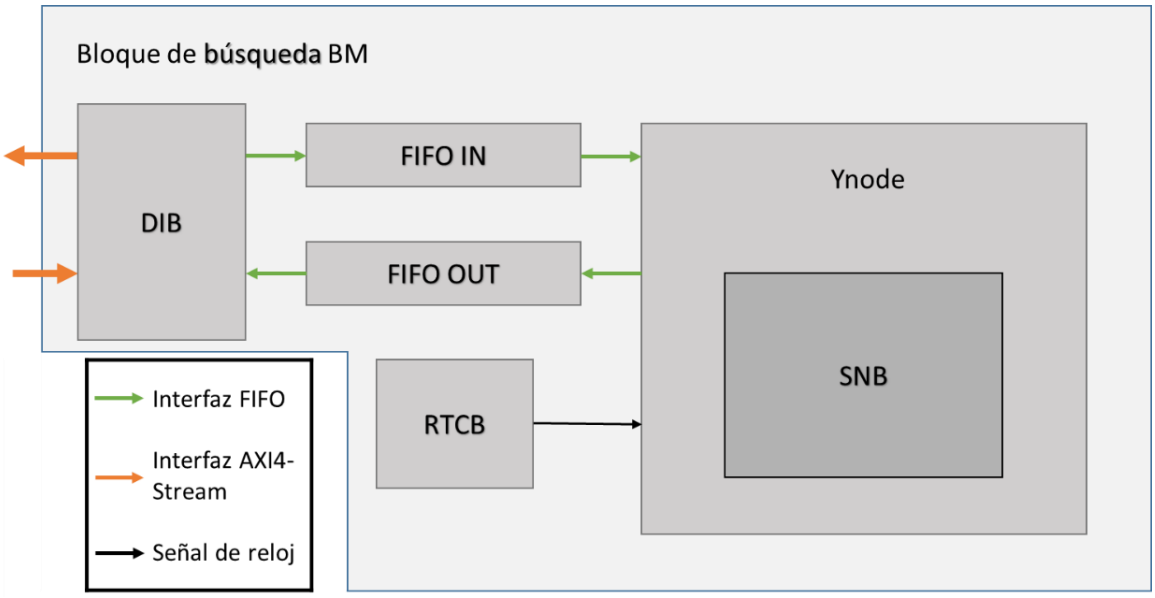



**Figura 64: Diagrama de bloques del motor de búsqueda de referencia** [63]

### 6.6.1 Modelado en TLM del bloque Boyer-Moore

Al realizar el modelado en TLM del motor de búsqueda se consigue simplificar la cantidad de bloques respecto al de referencia. El uso de del bloque DIB no es necesario ya que se hace uso del protocolo AXI para conectar el *Boyer-Moore* al resto de la plataforma. El bloque *Ynode,* y por tanto el bloque SNB, se modelan en TLM y con el objetivo de simplificar el diseño, sólo contará con un bloque SNB.

Todo esto se agrupa en un único bloque que se denomina *Boyer-Moore* y que cuenta con dos puertos esclavos y uno maestro que se detallan a continuación y se definen en la Figura 65:

- *Slave_data:* Puerto esclavo de 32 bits basado en el protocolo AXI que recibe el *payload.*
- *Slave_pattern:* Puerto esclavo de 32 bits basado en el protocolo AXI que recibe el patrón a través del microprocesador.
- *Result:* Puerto maestro de 32 bits con protocolo de señal que envía el resultado del análisis del patrón.

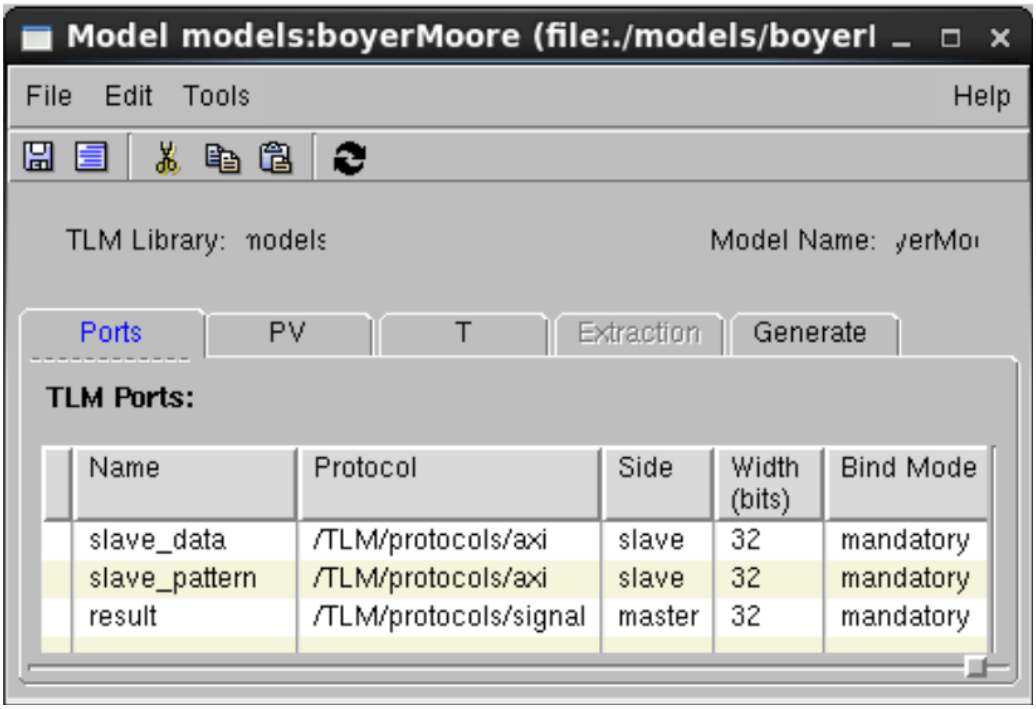



**Figura 65: Definición de puertos del bloque *Boyer-Moore***

El puerto *slave_pattern* tiene asociado una serie de registros que permite la configuración del patrón desde la aplicación de la plataforma. En la Tabla 7 se detallan los registros que cuenta el bloque y en la Figura 66.

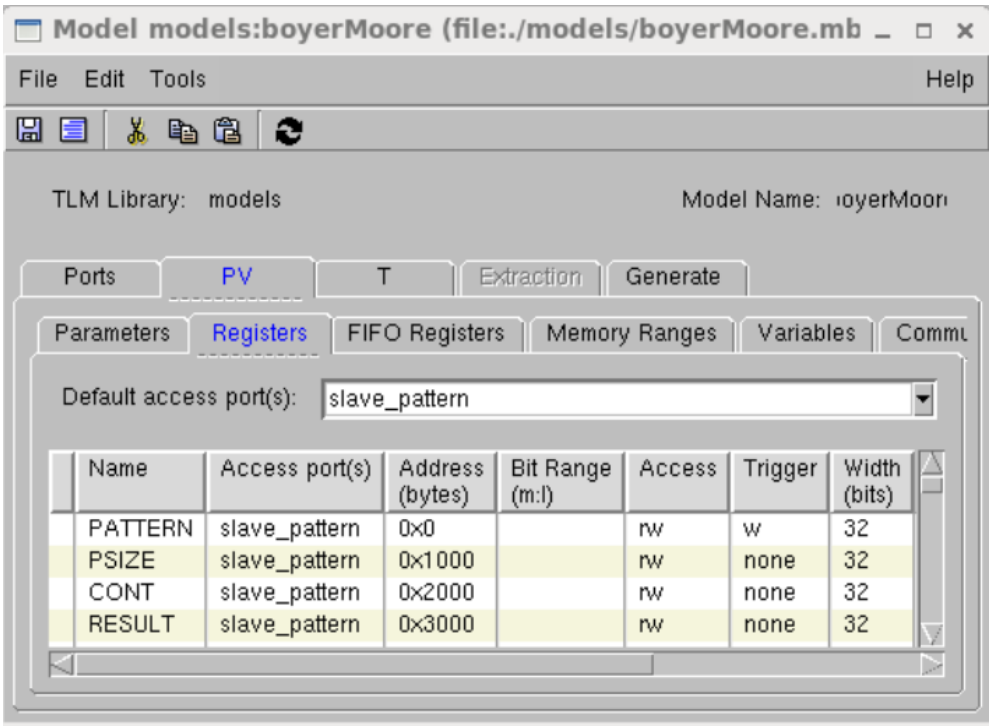



**Figura 66: Definición de los registros del *Boyer-Moore***

**Tabla 7: Descripción de los registros del bloque *Boyer-Moore***

| NOMBRE | DESCRIPCIÓN | TAMAÑO | DIRECCIÓN |
|---|---|---|---|
| **PATTERN** | Registro que recibe cada carácter del patrón | 4 bytes | 0x00000000 |
| **PSIZE** | Registro que define el tamaño del patrón | 4 bytes | 0x00001000 |
| **CONT** | Registro que indica el índice del patrón | 4 bytes | 0x0002000 |
| **RESULT** | Registro que guarda el resultado del análisis | 4 bytes | 0x0003000 |

El modelado temporal del bloque cuenta con políticas de *delay* y *sequential* donde se define el tiempo en realizar la inspección del *payload* (Figura 67). Se ha definido una latencia de 1450 ciclos, siendo este valor para el peor caso donde el paquete recibido tiene el máximo permitido y el patrón definido no se encuentra en el *payload*. Además, se define también el consumo de potencia de esta búsqueda que es de 5 mW y está tomado de la implementación realizada en [16].

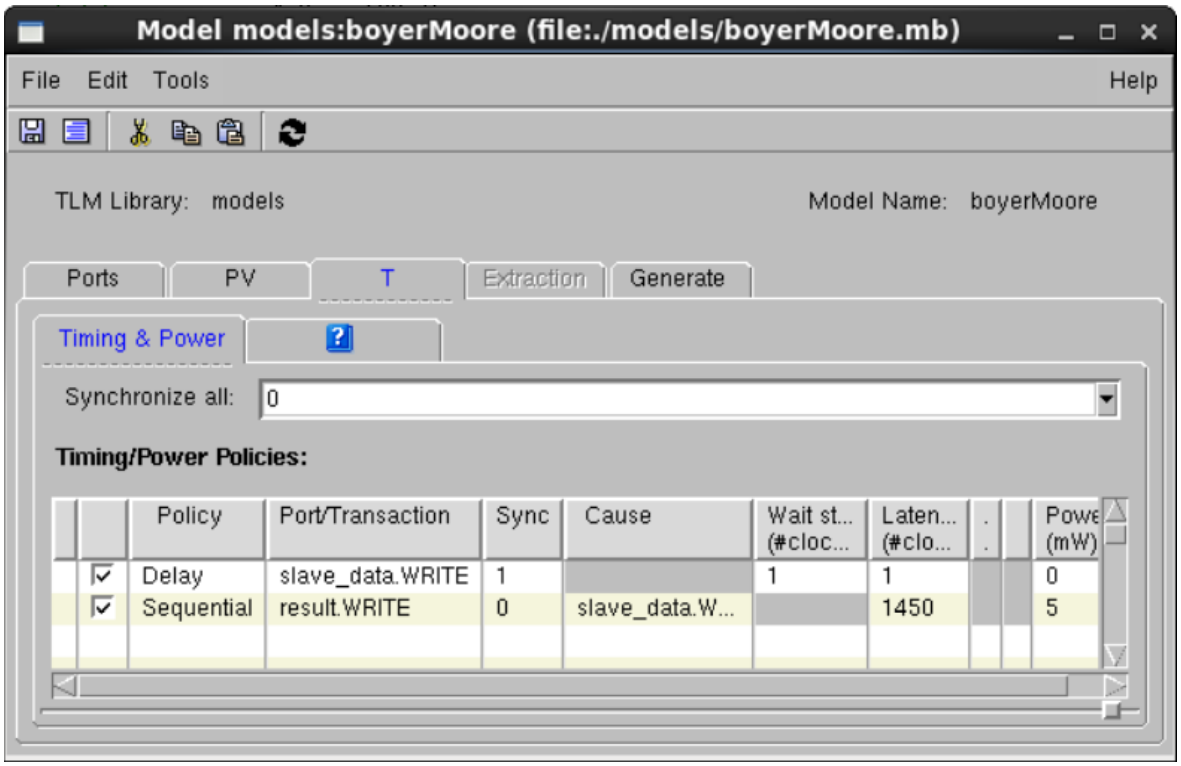



**Figura 67: Definición de políticas temporales y de potencia del *Boyer-Moore***

Establecidos los puertos, registros y las políticas, se puede generar el modelo en TLM-2.0 para definir su comportamiento. Una peculiaridad que tienen los registros es que

se pueden implementar funciones *callback* cuando estos registros son leídos o escritos si tienen la opción de *Trigger* activa. Esto ayuda a simplificar el modelado de un comportamiento. Por lo tanto, cuando se escribe un dato cualquiera en el registro de *PATTERN* significa que el usuario ha comenzado a definir el patrón y el registro inicia la llamada a la función *callback* de escritura. En esta función (Código 5) se recoge cada carácter que conforma el patrón y lo almacena en un array, con la ayuda del registro *CONT* se puede recorrer el array evitando que se sobrescriba el patrón en la misma posición.

```
1  #include "boyerMoore_pv.h"
2  #include <iostream>
3
4  using namespace sc_core;
5  using namespace sc_dt;
6  using namespace std;
7
8  //constructor
9  boyerMoore_pv::boyerMoore_pv(sc_module_name module_name)
10   : boyerMoore_pv_base(module_name) {
11   RESULT = 0;
12 }
13
14 // Write callback for PATTERN register.
15 // The newValue has been already assigned to the PATTERN
16 register.
17 void boyerMoore_pv::cb_write_PATTERN(unsigned int newValue) {
18   unsigned int cont = CONT;
19   patron[cont] = PATTERN;
20 }
21 ...
```

**Código 5: Función *callback* de escritura del registro *PATTERN* del *Boyer-Moore***

En la función *callback* de escritura del puerto *slave_data* se define el comportamiento del bloque cuando este recibe datos (Código 6). Primero se realiza un tratado del *payload* para ajustar la búsqueda del patrón al byte. Aunque las transacciones en TLM son a nivel de byte, la plataforma trabaja con anchos de palabra de 32 bits. Cuando se realiza una escritura a través de un puerto maestro, el *socket* del *initiator* se encarga de dividir el dato del *payload* genérico de la transacción. Cuando se recibe la transacción en el esclavo, el *socket* del *target* lo agrupa al mismo tamaño que fue enviado. Para realizar esta

adaptación, se toma directamente el puntero del dato de la transacción y se almacena en un array con ancho de byte. La escritura de este array no se hace de manera secuencial, debido a que se transmite primero el byte menos significativo de la palabra, por tanto, se debe hacer una reordenación de los bytes. Este proceso se muestra en las líneas 13-18 del Código 6.

```
1   ...
2   bool boyerMoore_pv::slave_data_callback_write(mb_address_type
3   address, unsigned char* data, unsigned size) {
4
5     int lastch;
6     unsigned int i, j, z, patronSize, payloadSize;
7     unsigned char mem[size];
8
9     z=0;
10    payloadSize = size;
11    patronSize = PSIZE;
12
13    for(int z=0; z<size;z+=4){
14      mem[z]=data[3+z];
15      mem[z+1]=data[2+z];
16      mem[z+2]=data[1+z];
17      mem[z+3]=data[0+z];
18    }
19
20    i = patronSize - 1;
21    j = patronSize - 1;
22
23    while (i < payloadSize){
24      if(patron[j]==mem[i]){
25        if(j==0) {
26          result.write(5);
27          RESULT = 5;
28          printf("patron encontrado\n");
29          return true;
30        }else{
31          j--;
32          i--;
33        }
34      }else{
```

```
35        lastch = find(patron, mem[i]);
36        if (lastch == -1){      // not found
37          i = i + patronSize;   // jump over
38        }else{
39          i = i + j - lastch;   // align char
40        }
41        j = patronSize - 1;     // restart from right
42      }
43    }
44    result.write(1);
45    RESULT = 1;
46    printf("patron no encontrado \n");
47    return true;
48  }
49  ...
```

**Código 6: Función *callback* de escritura del puerto *slave_data* del *Boyer-Moore***

Una vez que el *payload* se encuentra almacenado de manera correcta se comienza a realizar la búsqueda del patrón empezando por la posición definida por el tamaño del patrón. Si el último carácter del patrón coincidiera con esa posición se recorre el *payload* hacia atrás (línea 24), en caso contrario se compara cada carácter del patrón con el carácter del payload de la posición en la que se encuentre. Esto se realiza con la función *find()* que se muestra en el Código 7.

```
1   ...
2   int boyerMoore_pv::find (unsigned int P[], unsigned int ch){
3     int m, i;
4     m = PSIZE-2;
5     for (i = m; i >= 0; i--) {
6         if (ch == P[i]) {
7             return i;
8         }
9     }
10    return -1;
11  }
```

**Código 7: Función *find()* del *Boyer-Moore***

## *6.7* Modelado en TLM del bloque *Lectura FIFO*

La finalidad de este bloque es la de controlar la FIFO que almacena el paquete completo según los resultados del bloque Boyer-Moore. Cuando recibe el resultado del análisis del *payload* y existe una coincidencia con el patrón asignado, se vacía la FIFO desechando así el paquete, impidiendo que vuelva a la red. Si no existe ninguna coincidencia, se realiza una lectura a la FIFO y se transmite el paquete hacia la red. Se definen los siguientes puertos del bloque (Figura 68) que se detallan a continuación:

- *Master_read:* Puerto maestro de 32 bits basado en el protocolo AXI que realiza la lectura de la FIFO y su vaciado.
- *Master_out:* Puerto maestro de 32 bits basado en el protocolo AXI que envía el paquete hacia la interfaz de red.
- *Sresult:* Puerto esclavo de 32 bits basado en protocolo de señal que recibe el resultado del análisis desde el bloque *Boyer-Moore.*
- *Fsize:* Puerto esclavo de 32 bits basado en protocolo de señal que recibe el tamaño del paquete almacenado en la FIFO.

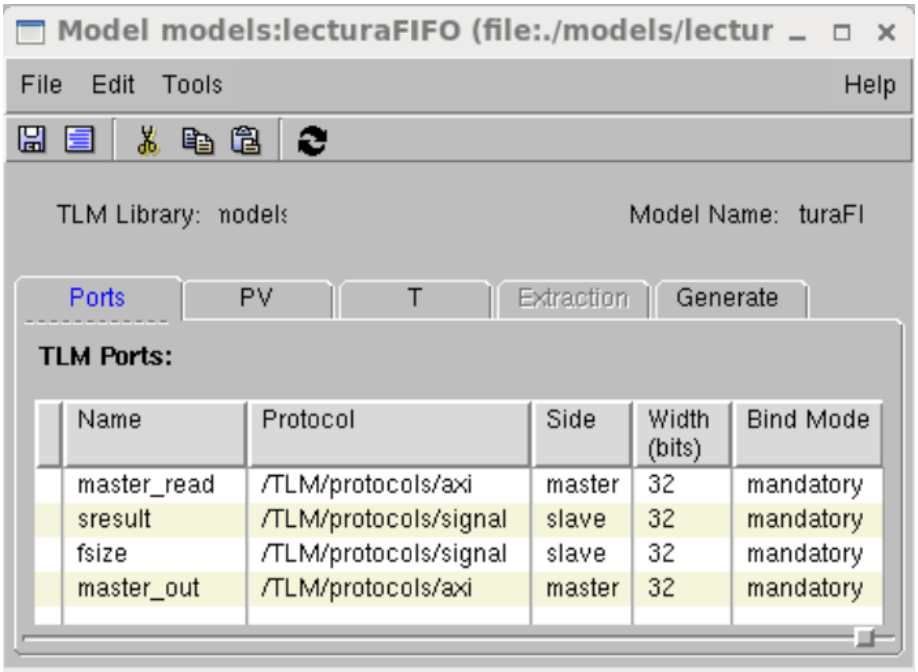



**Figura 68: Definición de puertos del bloque *Lectura FIFO***

Las políticas temporales definidas en la capa T para este modelo es la política de *sequential* donde se ha definido una latencia de 4 ciclos desde que se recibe el resultado

del *Boyer-Moore* hasta que se envía el paquete almacenado en la FIFO. Para este bloque se ha estimado un consumo de potencia de 1 mW (Figura 69).

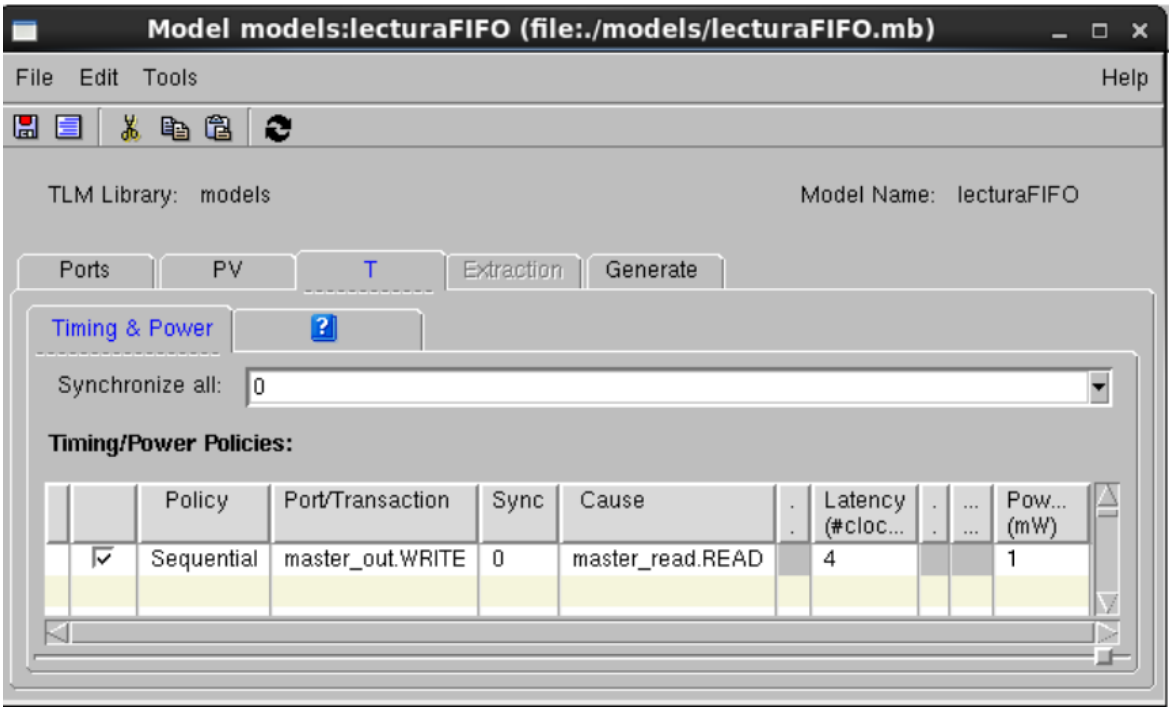



**Figura 69: Definición de políticas temporales y de potencia de *Lectura FIFO***

El comportamiento de este bloque se define en el Código 8. Cuando se recibe el resultado de la búsqueda del patrón se evalúa. Si se ha encontrado el patrón se vacía la FIFO desechando el paquete. En el caso contrario, si no se ha encontrado el patrón, se realiza una lectura a la FIFO para obtener el paquete y se transmite hacia la red.

```
1  #include "lecturaFIFO_pv.h"
2  #include <iostream>
3
4  using namespace sc_core;
5  using namespace sc_dt;
6  using namespace std;
7  //constructor
8  lecturaFIFO_pv::lecturaFIFO_pv(sc_module_name module_name)
9    : lecturaFIFO_pv_base(module_name) {
10 }
11 // callback for any change in signal: slave of type:
12 sc_in<unsigned int>
13 void lecturaFIFO_pv::sresult_callback() {
14
```

```
15    unsigned int flush[1024];
16    unsigned int size = (fsize.read())/4;
17    unsigned int mem[size];
18    if(sresult.read()==5){
19      master_read_write (0x0, flush, size);
20    }
21    if(sresult.read() == 1){
22      master_read_read (0x0, mem, size);
23      master_out_write (0x0, mem, size);
24    }
25  }
```

**Código 8: Función *callback* del puerto *sresult* de la *Lectura FIFO***

## 6.8 Conclusiones

Durante este capítulo se ha explicado el modelado en TLM-2.0 de los bloques que compone la plataforma DPI, desde el modelado de las interfaces, pasando por el modelado funcional y terminando con la aplicación de políticas temporales y de potencia. Se ha creado un generador de tráfico que permita emular tráfico de red en la plataforma permitiendo así comprobar el funcionamiento de los demás bloques. Los modelos creados para el *Header Analyzer* y el *Boyer-Moore* se han modelado en un único bloque en comparación con los de referencia.

# Capítulo 7. Creación de la Plataforma Virtual

Este capítulo contiene las distintas etapas que conforma la creación de la plataforma virtual. Con los bloques modelados en TLM se crea un diagrama de bloques de la plataforma, se configura la misma a través de un fichero de parámetros, se crea la aplicación *software* para realizar la configuración de los campos TCP/IP necesarios en el bloque *Header Analyzer* y el patrón que utilizará el *Boyer-Moore* en los motores de búsqueda y por último se constituye la plataforma virtual del sistema DPI.

## 7.1 Ensamblaje de la plataforma

Los modelos creados en el Capítulo 6: Modelado en TLM-2.0 de la arquitectura DPI, se encuentran almacenados en la librería *Models* que previamente se había creado. En la librería *zynq_models* se encuentran todos los modelos que definen el comportamiento real del dispositivo Zynq-7000. Con los modelos almacenados en esta librería, la herramienta Vista proporciona un diagrama de bloques del dispositivo que se encuentra listo para poder ser usada la parte de sistema de procesamiento. A través de las interfaces AXI_GP y AXI_HP que conectan el PS con el PL se incorpora los modelos TLM que han sido creados para ser implementados en la parte lógica de la plataforma. Este diagrama se encuentra en la librería *zynq_schematics* y se denomina *Zynq_SoC*. En la Figura 70 se muestra a través del editor de diagrama de bloques de Vista la plataforma Zynq-7000, donde da una visión general de la complejidad de la plataforma.

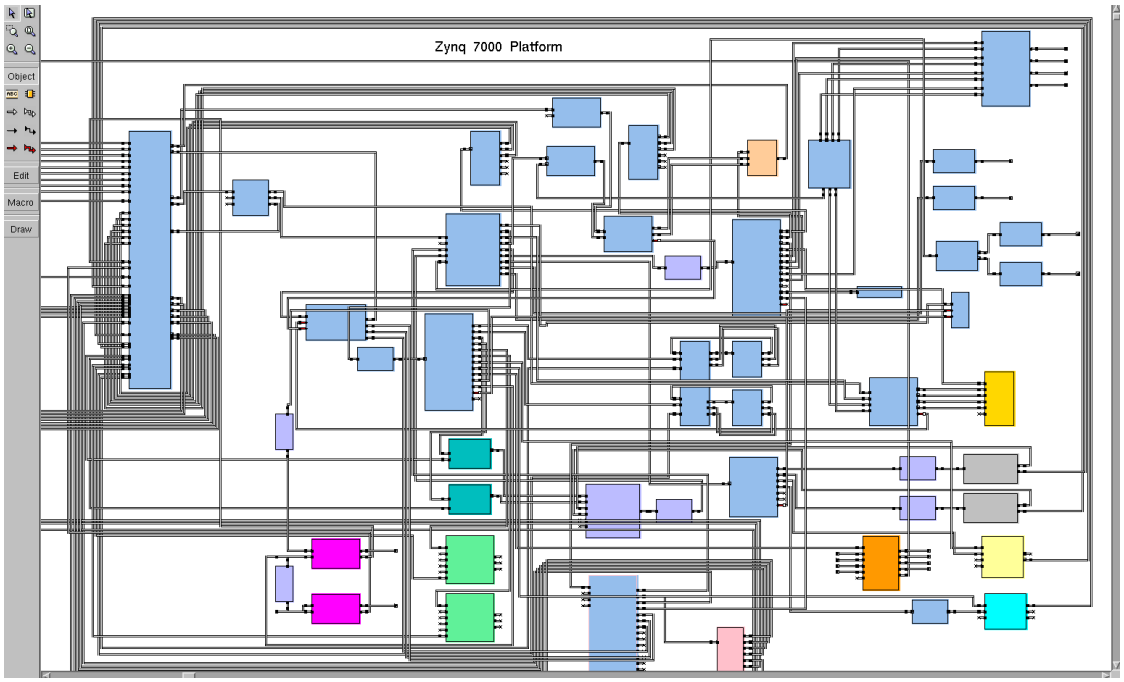



**Figura 70: Diagrama de bloques modelados en TLM de la plataforma Zynq-7000**

La herramienta Vista permite la creación de jerarquías en el editor de diagrama de bloques, consiguiendo que grandes diseños sean más manejables. Esto se demuestra en la Figura 71, donde el bloque denominado *Zynq_SoC* representa la plataforma a alto nivel, conteniendo dentro de él el diagrama mostrado en la figura anterior.

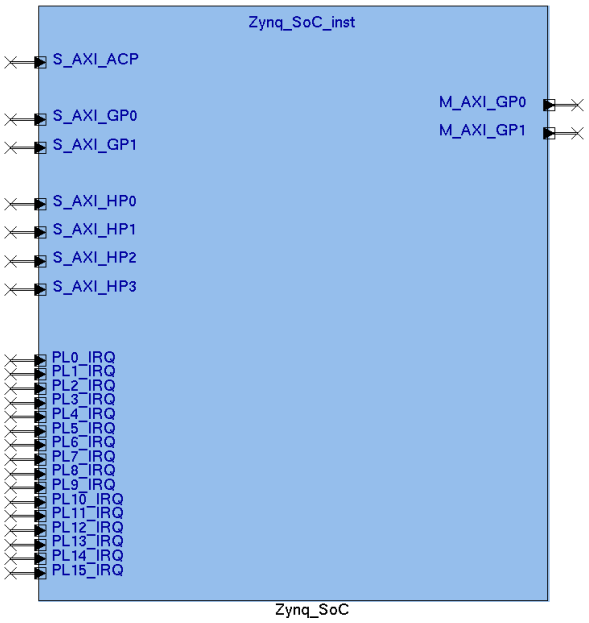



**Figura 71: Bloque en alto nivel de la plataforma Zynq SoC**

Siguiendo este recurso de jerarquía, en un nuevo editor de diagrama de bloques se integran los bloques que conforman la parte *hardware* de la arquitectura DPI y que

se presentaron en el Capítulo 6: Modelado en TLM-2.0 de la arquitectura DPI. En la Figura 72 se muestra el resultado de la integración.

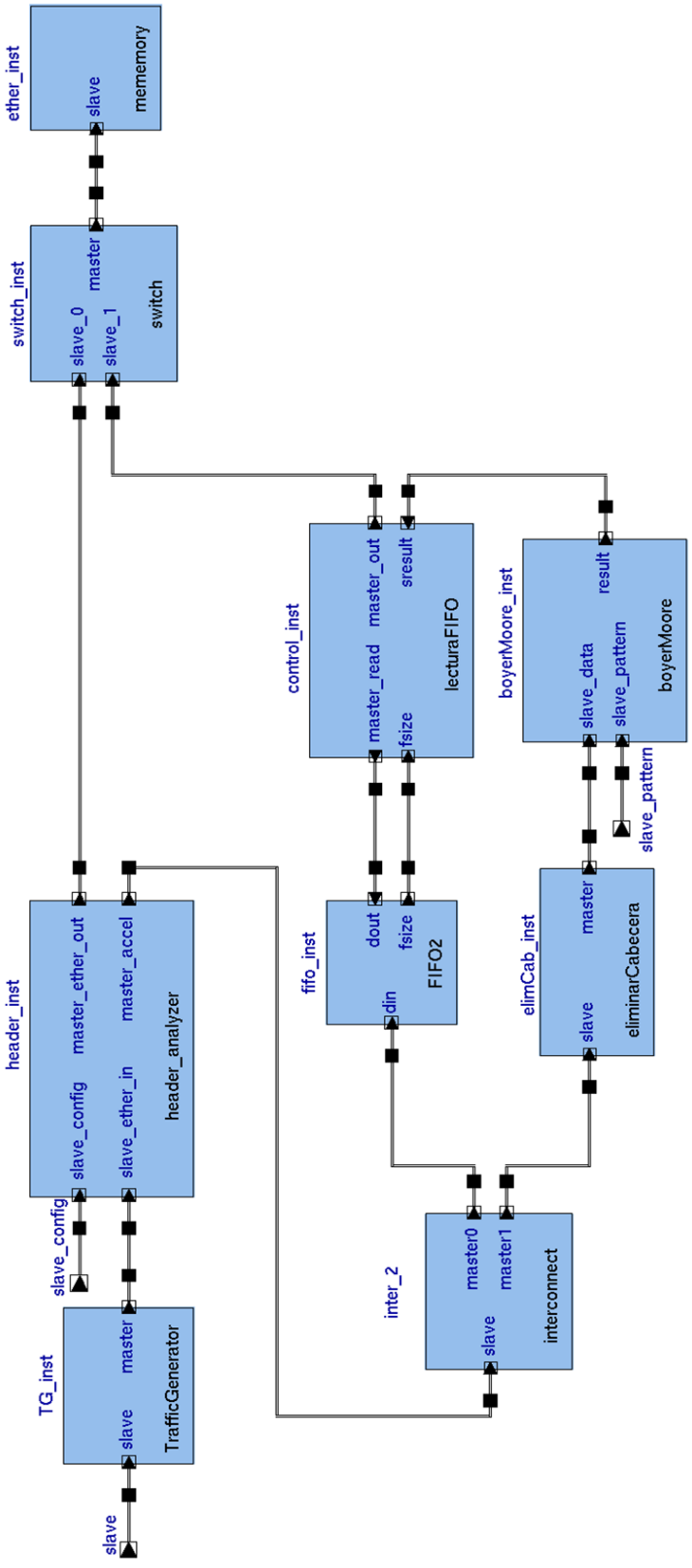



**Figura 72: Diagrama de bloques de la plataforma DPI modelado en TLM**

Una vez validado y guardado el diagrama, se puede incorporar como único bloque (Figura 73) al diagrama del dispositivo.

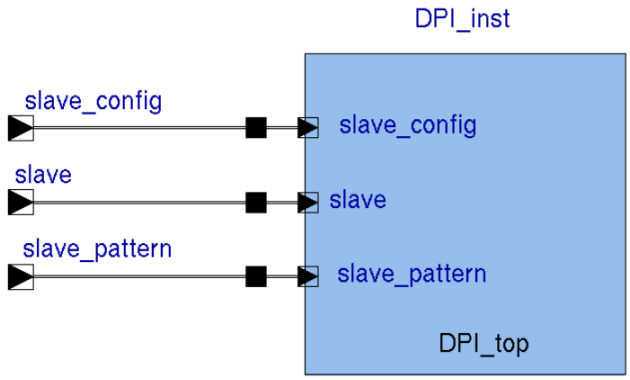



**Figura 73: Bloque en alto nivel del DPI**

El DPI se comunica con la plataforma a través de la interfaz M_AXI_GP0 permitiendo así la configuración del análisis de la cabecera y la definición del patrón de búsqueda desde la aplicación empotrada de la plataforma. Puesto que se desea conectar tres esclavos a un único maestro, es necesario la incorporación de un bus que permita asignar la salida correspondiente dependiendo de la dirección definida en la transacción. Esto se muestra en la Figura 74.

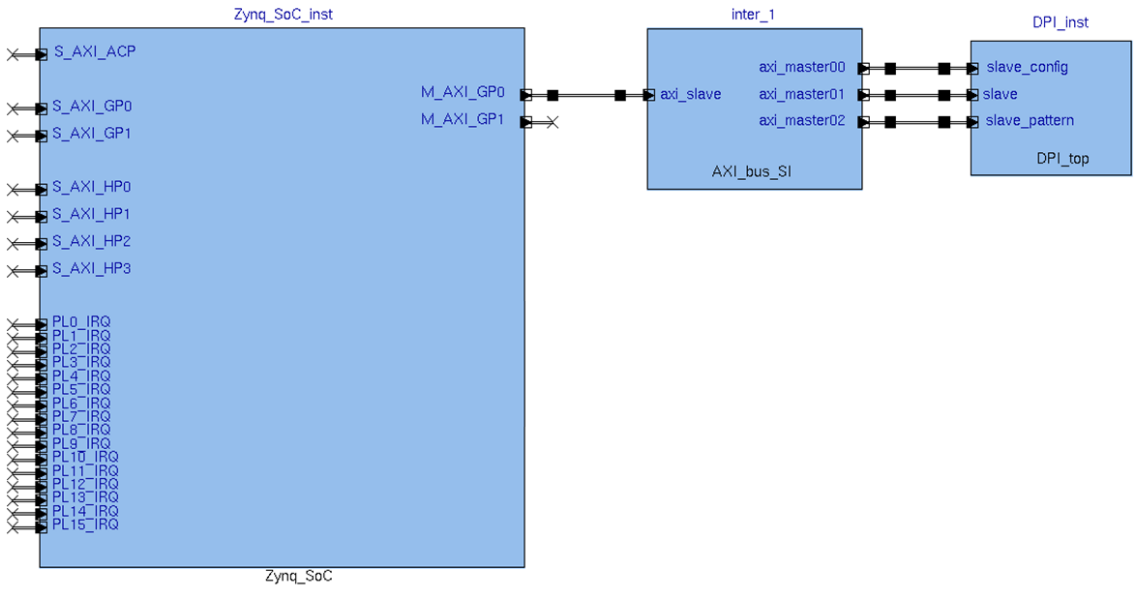



**Figura 74: Plataforma Zynq con DPI**

En el fichero de parámetros, donde se especifica los valores de los parámetros locales usados en la simulación, se debe indicar la configuración del bus que se ha

añadido entre el PS y el DPI (Código 9). Estos valores corresponden a las distintas direcciones que puede manejar el bus y deben coincidir con la dirección de asignadas a los registros a la hora de crear la aplicación, teniendo en cuenta que la dirección base del M_AXI_GP0 tiene el valor 0x40000000.

```
1  #HeaderAnalyzer
2  top.Zynq_SoC_inst.inter_1.axi_master00_base_address = 0x1000000
3  top.Zynq_SoC_inst.inter_1.axi_master00_size = 0x1000000
4
5  #TrafficGenerator
6  top.Zynq_SoC_inst.inter_1.axi_master01_base_address = 0x000000
7  top.Zynq_SoC_inst.inter_1.axi_master01_size = 0x100000
8
9  #BoyerMoore
10 top.Zynq_SoC_inst.inter_1.axi_master02_base_address = 0x2000000
11 top.Zynq_SoC_inst.inter_1.axi_master02_size = 0x100000
```

**Código 9: Direcciones del bus en el fichero de parámetros**

## 7.2 Desarrollo del *software*

Una de las múltiples ventajas que cuenta el uso de plataformas virtuales es la posibilidad de desarrollar el *software* mientras se está modelando los diversos bloques que conforma la plataforma, creciendo de manera paralela y adaptándose a las necesidades de cada parte desde que surgen.

Para empezar el desarrollo de la aplicación se debe conocer qué se pretende ejecutar en ella, así como conocer qué bloques están en comunicación directa. Conforme se comienza a modelar los bloques de la plataforma se empieza a deducir qué se puede incorporar en la aplicación. Para que los parámetros del análisis de la cabecera puedan ser reconfigurables su asignación se realiza desde la aplicación. De la misma manera, el patrón para la inspección del *payload* se puede reconfigurar desde la aplicación actualizando su valor para adaptarse a cada situación. Por último, cuando la integración *hardware-software* se ha realizado satisfactoriamente, se continúa con el refinamiento de la aplicación.

La aplicación cuenta con cuatro etapas que se encuentran contenidas en la función principal de la aplicación y se muestra en el Código 10:

1. Inicialización de la UART.
2. Configuración del bloque *Header Analyzer.*
3. Configuración del patrón del bloque *Boyer-Moore.*
4. Inicialización del *Traffic Generator*.

```
1  int main (){
2        unsigned char character;
3        UART0_Init();
4        HeaderAnalyzer_Init();
5        config_boyerMoore();
6
7        START_TRAFFIC_GENERATOR = 1;
8
9        UART0_Send_String("Press Enter to terminate the
10 VP...\r\n");
11       /* Wait until user press Enter */
12       character = UART0_Receive_Byte();
13       while((character != '\n') && (character != '\r')){
14             character = UART0_Receive_Byte();
15       }
16       /* Terminate the VP */
17       mb_stop(0);
18       return 0;
19 }
```

**Código 10: Programa principal de la aplicación**

Haciendo uso de una de las dos UART que cuenta la plataforma el usuario puede definir el patrón que se busca en el *payload* de los paquetes recibidos en la plataforma DPI. Se usa la UART0 y la configuración de sus parámetros se realiza a través de la función *UART0_Init()* que se muestra en el Código 11 :

```
1  void UART0_Init(void)
2  {
3        UART0_Mode_Reg0 = 0;
4        UART0_Baud_Rate_Gen_Reg0 = 9; /* CD = 9 */
5        UART0_Baud_Rate_Divider_Reg0 = 5; /* BDIV = 9 */
6        UART0_Control_Reg0 = 0x00000014;
7  }
```

**Código 11: Inicialización de la UART0**

Posteriormente se configura los parámetros de la cabecera que analiza el *Header Analyzer*. Esto se realiza en la función *HeaderAnalyzer_Init()* que se muestra en el Código 12.

```
1  void HeaderAnalyzer_Init(void)
2  {
3        CHECK_ETHER_REG_ADDR = 0x1;
4
5        DEST_MAC_REG_ADDR_LSB = 0x147461E0;
6        DEST_MAC_REG_ADDR_MSB = 0xFCAA;
7  }
```

**Código 12: Configuración del *Header Analyzer***

Desde el terminal que genera la UART se asigna el patrón de búsqueda. Esto se realiza a través de la función *config_BoyerMoore()*, donde cada carácter recibido se envía al bloque *BoyerMoore* formando el patrón y cuando se recibe un retorno de carro indica el final del patrón (Código 13).

```
1  void config_boyerMoore(){
2
3        char * volatile patron;
4        unsigned int i=0;
5
6        UART0_Send_String("Enter pattern to search:\r\n");
7
8        patron[i] = UART0_Receive_Byte();
9        UART0_Send_Byte(patron[i]);
10       PATTERN = (unsigned int) patron[i];
11       PATTERN_CONT = (unsigned int) i;
12       while((patron[i] != '\n') && (patron[i] != '\r')){
13             i++;
14             patron[i] = UART0_Receive_Byte();
15             UART0_Send_Byte(patron[i]); //Echo
16
17             PATTERN_CONT = (unsigned int) i;
18             PATTERN = (unsigned int) patron[i];
19       }
20       PATTERN_SIZE = (i);
21 }
```

**Código 13: Configuración del *Boyer Moore***

## 7.3 Comprobación de la funcionalidad de los modelos

Con el objetivo de conocer que los bloques que se han modelado en TLM-2.0 cumple con los requisitos funcionales marcados en su descripción, se realiza la simulación de la plataforma y se analiza la forma de onda de cada modelo desde la interfaz gráfica de Vista.

### 7.3.1 Simulación del modelo Header Analyzer

Desde el bloque *Traffic Generator* se envían dos paquetes, uno generado aleatoriamente y otro que contiene un parámetro de la cabecera que coincide con el parámetro que se envía desde la aplicación *software*. El bloque se ha modelado de la manera que si algún parámetro coincide se envíe hacia el puerto *master_accel*, en caso contrario se envía a través del *master_ether_out.* El parámetro que se analiza en esta simulación es la dirección MAC de destino que se encuentra en los 48 primeros bits del paquete y se ha definido como `FC:AA:14:74:61:E0`.

En la Figura 75 se muestra la forma de onda del *Header Analyzer* donde se muestran los puertos *slave_ether_in, master_ether_out* y *master_accel.* El puerto *slave_config* no se ha añadido debido a que en la forma de onda no se muestra los datos que se introducen desde la aplicación para ser escritos en sus registros. En cambio, se han añadido esos registros a la forma de onda y así poder comprobar que la comunicación entre la aplicación y los bloques modelados es correcta.

Las tres últimas señales corresponden a los registros. Se observa que se ha escrito sobre el registro *check_ether_reg_addr* un "1" activando así el análisis de la capa Ethernet. También se ha escrito sobre los registros *dest_mac_reg_addr_lsb* y *dest_mac_reg_addr_msb* que en su conjunto forma la dirección MAC de destino (`FC:AA:14:74:61:E0`). Se puede observar con el desglose de las transacción que el primer paquete que se recibe a través del puerto *slave_ether_in*, las dos primeras palabra son 0x2d530028 y 0x58e396f, formando la dirección MAC de destino `2D:53:00:28:58:E3` que no corresponde con la definida por el usuario. Por lo tanto, ese paquete se debe redirigir por el puerto *master_ether_out* y se verifica en la forma de onda. El siguiente paquete recibido contiene en las dos primeras palabras los

datos 0xfcaa1474 y 0x61e00007 que corresponden a la dirección que se ha definido desde la aplicación y se transmite a través del puerto *master_out.*

Desde el terminal (Figura 76) también se puede comprobar el correcto funcionamiento con los mensajes que se han incluido en el código.

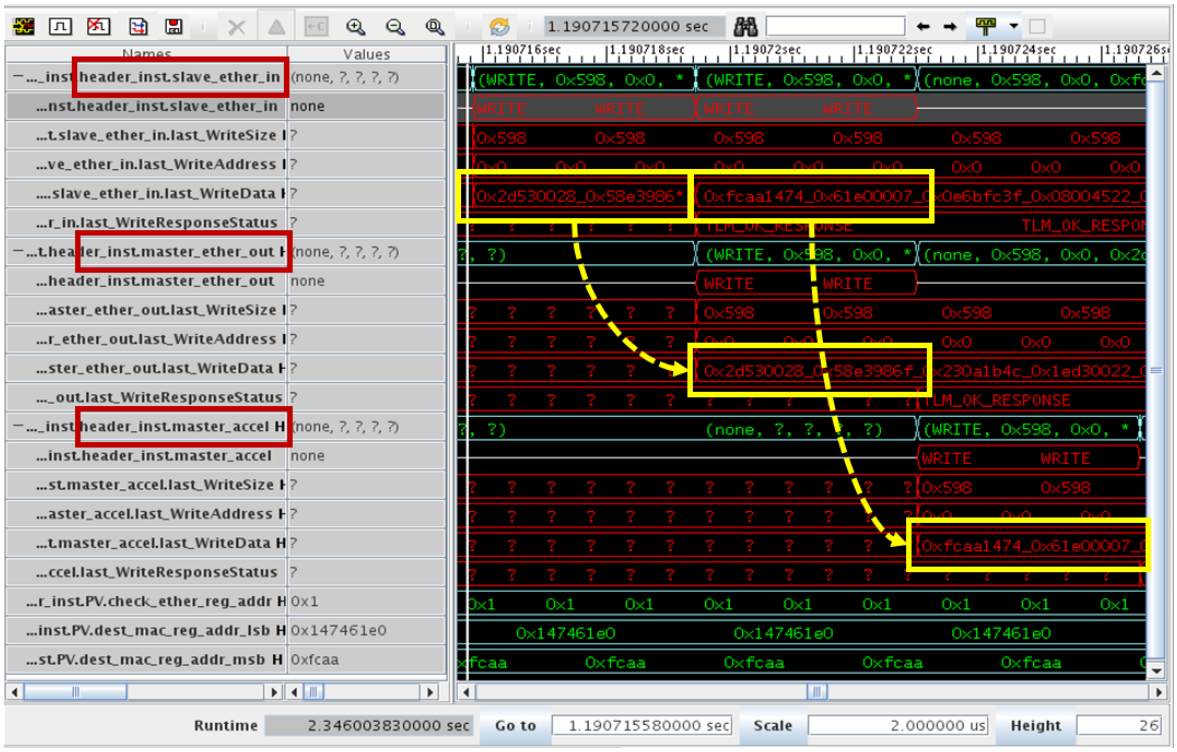



**Figura 75: Forma de onda del modelo *Header Analyzer***

```
#8  0x0000000000469d88 in sc_main (argc=1, argv=0x7fff6e668278) at /home/users/PFC2
/divsicad/slmartin/MentorVP/Plataformas/Virtual-Platforms/zynq7000_main.cpp:38
(vista) continue
Usuario ha definido analizar la capa Ethernet

Usuario ha definido analizar la direccion MAC destino
----------------------------------------
top.Zynq_SoC_inst.TG_inst.PV @ 1190715800 ns Traffic Generator activo 0
----------------------------------------
----------------------------------------
top.Zynq_SoC_inst.header_inst.PV @ 1190715820 ns Paquete recibido
----------------------------------------
Direccion MAC destino definida FC:AA:14:74:61:E0

Direccion MAC destino del paquete 0x02d53002858e3

Sin coincidencias

Salida a la red

----------------------------------------
top.Zynq_SoC_inst.header_inst.PV @ 1190719410 ns Paquete recibido
----------------------------------------
Direccion MAC destino definida FC:AA:14:74:61:E0

Direccion MAC destino del paquete 0x0fcaa147461e0

Existe coincidencia

Salida a acelerador

----------------------------------------
Simulation: /home/users/PFC/divsicad/slmartin/MentorVP/Plataformas/Virtual-Platforms
```

**Figura 76: Simulación *Header Analyzer***

### 7.3.2 Simulación del modelo Eliminar Cabecera

Cuando existe coincidencia en el análisis de la cabecera se envía el paquete al bloque *Eliminar Cabecera* para desechar las 13 primeras palabras de 32 bits del paquete que corresponde con la cabecera. El paquete que llega al bloque en esta simulación es conocido, por lo tanto se sabe que la primera palabra del paquete que recibe el bloque es `FC:AA:14:74:61:E0` y que la primera palabra que debe enviar después de eliminar las 13 primera palabras es `00:00:00:00:01:01`

En la Figura 77 se muestra el paquete recibido a través del puerto *slave* y con el desglose de señales se observa las primeras palabras del paquete. A la salida del bloque la primera palabra tiene el valor 0x00000101 que corresponde con la palabra decimotercera. Desde el terminal también se puede comprobar el funcionamiento del bloque como se muestra en la Figura 78.

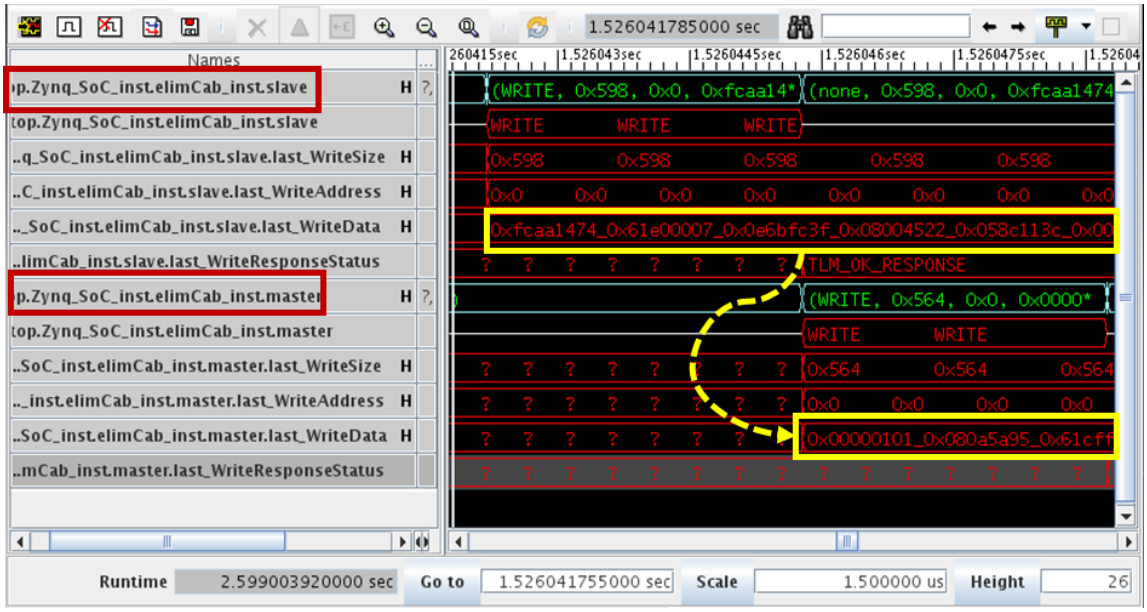



**Figura 77: Forma de onda del modelo *Eliminar Cabecera***

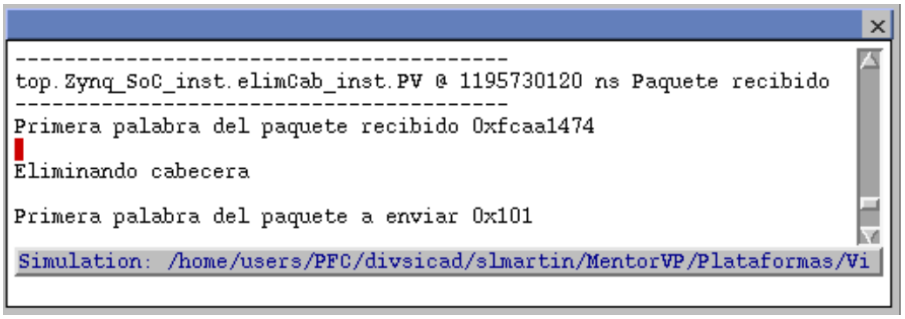



**Figura 78: Simulación *Eliminar Cabecera***

### 7.3.3 Simulación del modelo Boyer-Moore

Después de eliminar la cabecera del paquete el *payload* puede ser tratado por el *Boyer-Moore* con el objetivo de buscar un patrón definido por el usuario. Siguiendo con la línea de las comprobaciones anteriores, el paquete recibido por el bloque es conocido y contiene en su *payload* el patrón "raquel leon" definido en hexadecimal como `72:61:71:75:65:6C:20:6C:65:6F:6E`.

Desde el terminal, el usuario define el patrón de búsqueda. En la Figura 79 se muestra como se ha introducido el patrón "raquel leon".

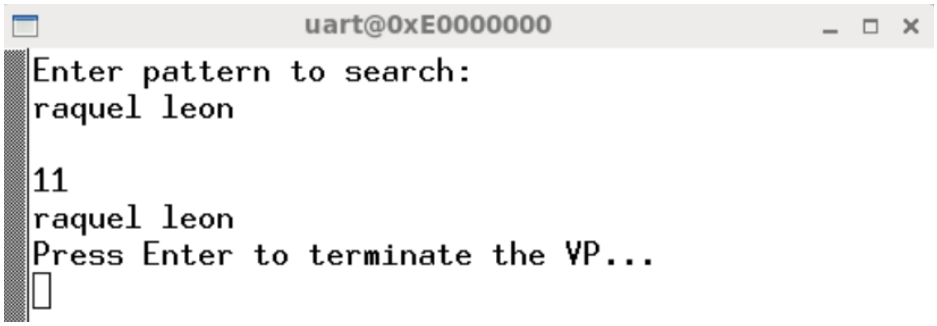



**Figura 79: Definir el patrón desde la aplicación (I)**

En la Figura 80, donde la primera señal corresponde con el registro *PATTERN* encargado de almacenar el patrón recibido desde la aplicación, se recibe uno a uno los caracteres introducidos conformando `72:61:71:75:65:6C:20:6C:65:6F:6E` que se corresponde con el patrón.

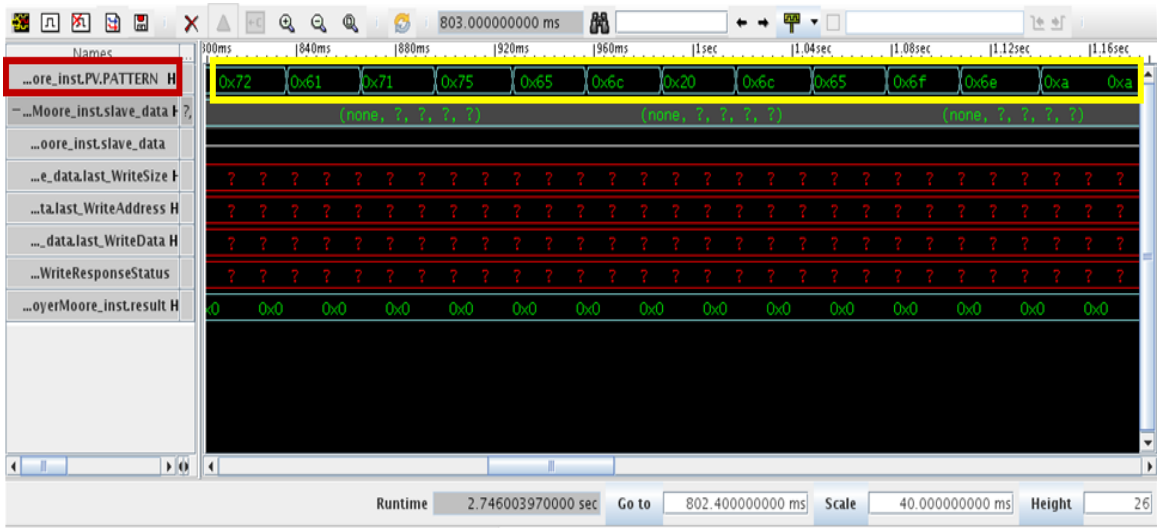



**Figura 80: Forma de onda del modelo *Boyer-Moore.* Definición del patrón (I)**

Si el patrón se encuentra, el paquete debe ser desechado y para eso se ha definido que se envíe un “5” por el puerto de resultado. Como el patrón introducido se corresponde con el que se encuentra en el *payload,* se verifica que el resultado que emite el bloque es un “5” y eso se muestra en la Figura 81.

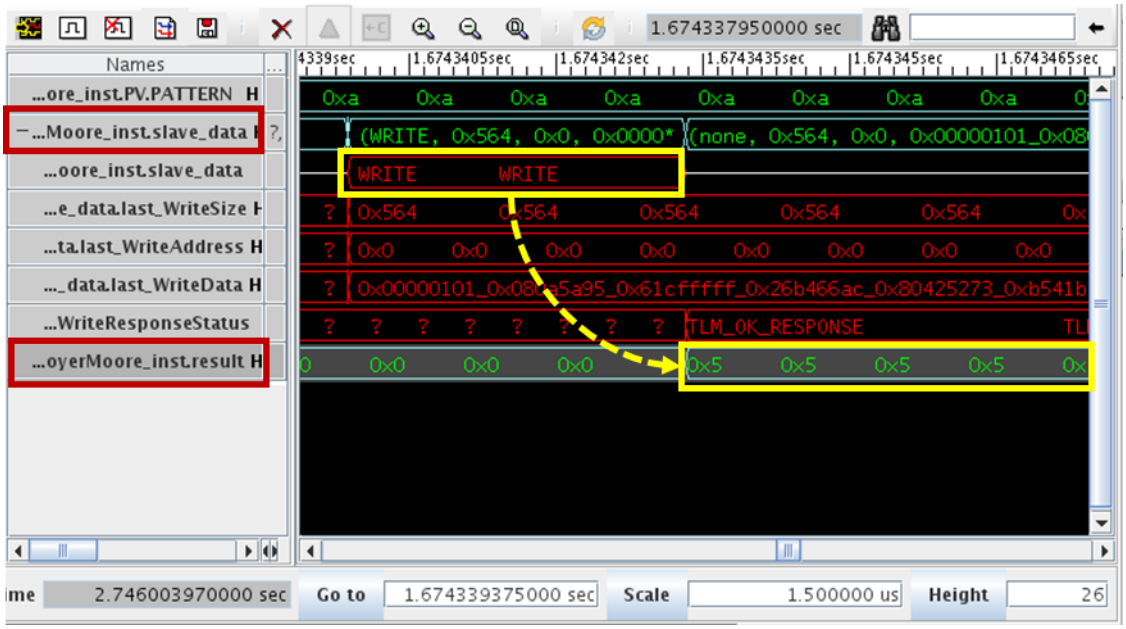



**Figura 81: Forma de onda del modelo *Boyer-Moore.* Patrón encontrado**

Para verificar que el algoritmo implementado realiza la búsqueda de manera correcta se recurre a la consola de la simulación donde se muestra por pantalla todas las iteraciones de búsqueda del patrón que realiza el modelo y se muestra en la Figura 82.

Si se define un patrón que no se encuentra en *payload* el paquete es reenviado hacia la red. Desde la aplicación se define el patrón *“sonia leon”* (Figura 83) que en hexadecimal corresponde a 73:6F:6E:69:61:20:6C:65:6F:6E y se muestra en la Figura 84 los caracteres del patrón recibidos en el registro *PATTERN.*

Al no coincidir el patrón incorporado por el usuario con el que se encuentra en *payload,* el *Boyer-Moore* genera en la señal *result* un “1” para que el paquete siga por la red. Esto se muestra en la Figura 85.

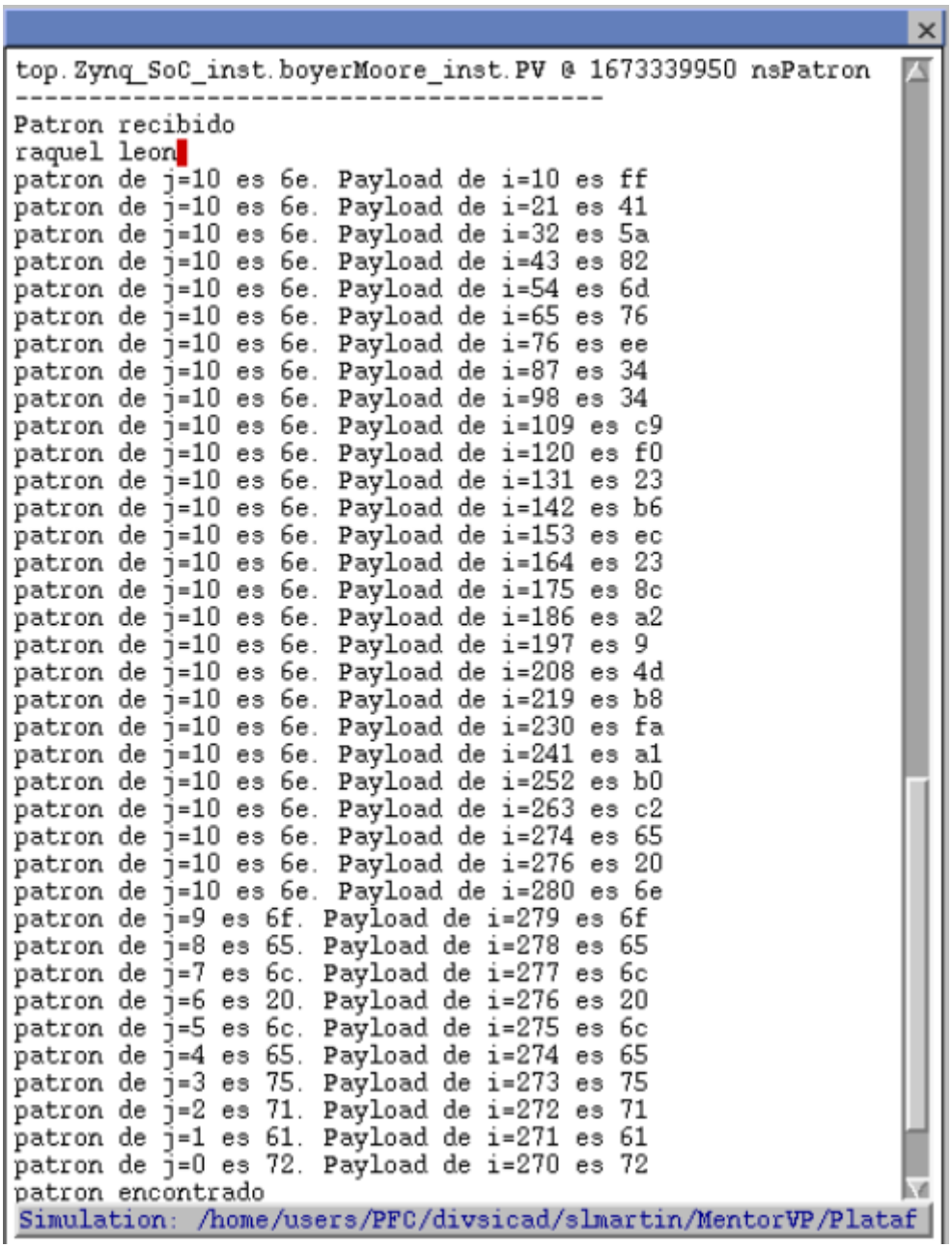



**Figura 82: Búsqueda del patrón. Patrón encontrado**

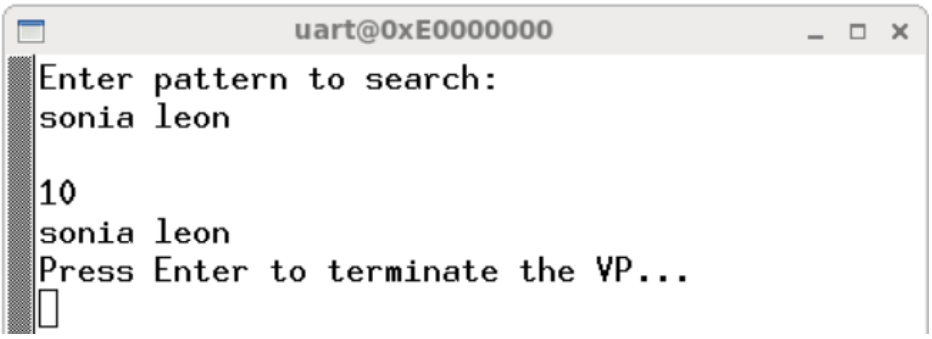



**Figura 83: Definir el patrón desde la aplicación (II)**

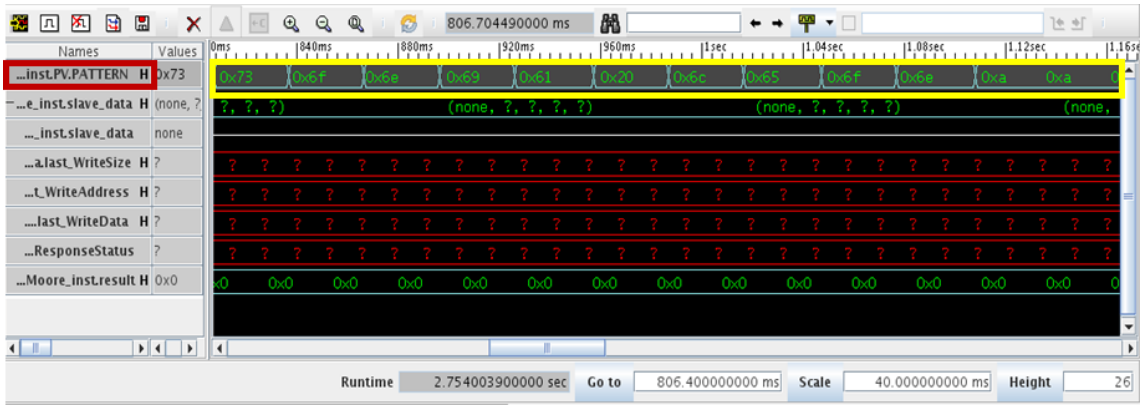



**Figura 84: Forma de onda del modelo *Boyer-Moore.* Definición del patrón (II)**

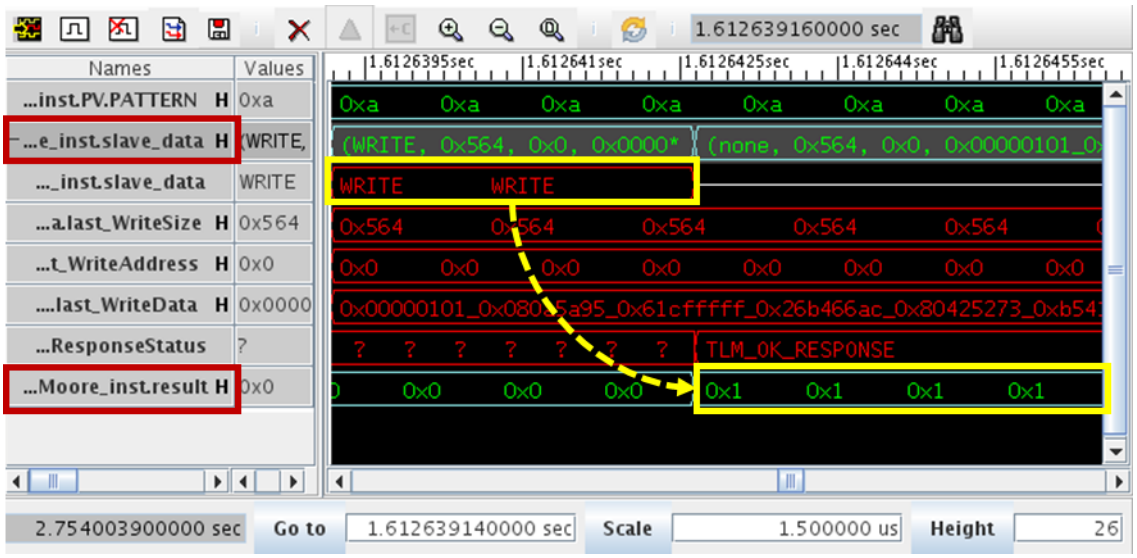



**Figura 85: Forma de onda del modelo *Boyer-Moore.* Patrón no encontrado**

Con la consola y los mensajes que se muestra se puede observar paso a paso la búsqueda que realiza. Al no encontrar el patrón recorre todo el paquete que cuenta con un tamaño de 1.380 bytes (Figura 86).

### 7.3.4 Simulación del modelo Lectura FIFO

El bloque *Lectura FIFO* se encarga de analizar el resultado del *Boyer-Moore* y dependiendo de este desechar el paquete almacenado en la FIFO o transmitirlo hacia la red. El bloque además recibe el tamaño del paquete almacenado, asegurando que se leerá el paquete completo. En la Figura 87 se muestra la forma de onda donde se recibe el resultado "1" que indica que el paquete puede ser transmitido. Por el puerto *master_read* se hace la lectura y por el puerto *master_out* se envía hacia el segundo bloque.

```
----------------------------------------
top.Zynq_SoC_inst.boyerMoore_inst.PV @ 1612639160 nsPatron
----------------------------------------
Patron recibido
sonia leon
patron de j=9 es 6e. Payload de i=9 es cf
patron de j=9 es 6e. Payload de i=19 es 73
patron de j=9 es 6e. Payload de i=28 es 4b
patron de j=9 es 6e. Payload de i=38 es be
patron de j=9 es 6e. Payload de i=48 es e5
patron de j=9 es 6e. Payload de i=58 es 21
patron de j=9 es 6e. Payload de i=68 es b5
patron de j=9 es 6e. Payload de i=78 es 71
patron de j=9 es 6e. Payload de i=88 es ce
  ...
patron de j=9 es 6e. Payload de i=1261 es 70
patron de j=9 es 6e. Payload de i=1271 es cf
patron de j=9 es 6e. Payload de i=1281 es 37
patron de j=9 es 6e. Payload de i=1291 es 2c
patron de j=9 es 6e. Payload de i=1301 es e9
patron de j=9 es 6e. Payload de i=1311 es 75
patron de j=9 es 6e. Payload de i=1321 es 74
patron de j=9 es 6e. Payload de i=1331 es 2c
patron de j=9 es 6e. Payload de i=1341 es ff
patron de j=9 es 6e. Payload de i=1351 es f2
patron de j=9 es 6e. Payload de i=1361 es ee
patron de j=9 es 6e. Payload de i=1371 es 95
patron no encontrado
```

**Figura 86: Búsqueda del patrón. Patrón no encontrado**

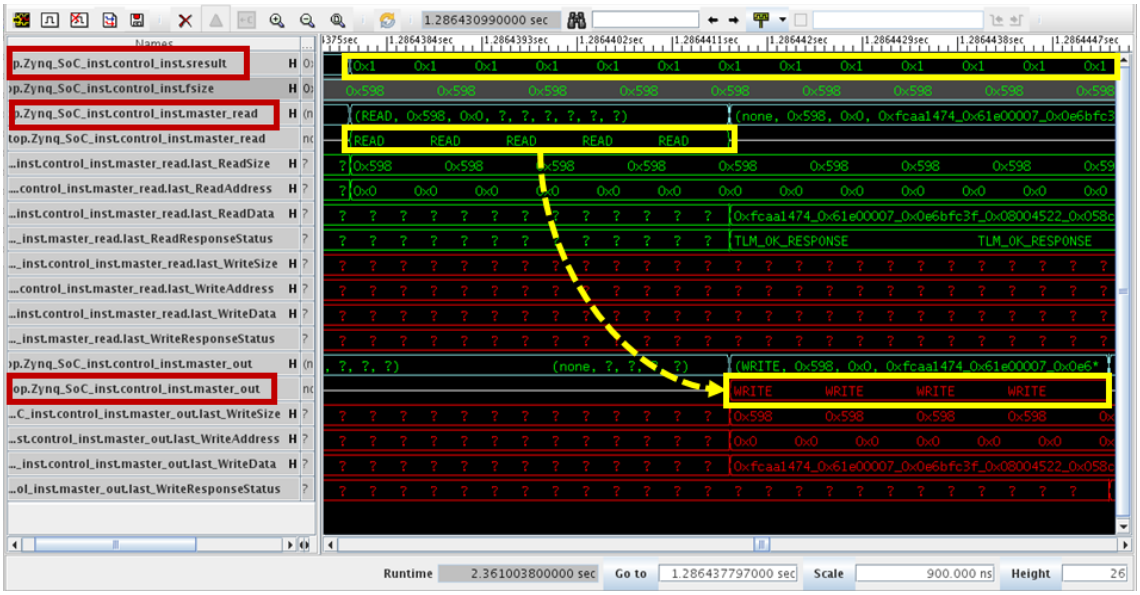



**Figura 87: Forma de onda del modelo *Lectura FIFO.* Transmitir el paquete**

En la Figura 88 se recibe el resultado "5" que significa que el patrón se ha encontrado en el *payload* y el paquete debe ser desechado. Por el puerto *master_read* se realiza una escritura sobre la FIFO para limpiarla. Por el puerto *master_out* no se envía nada.

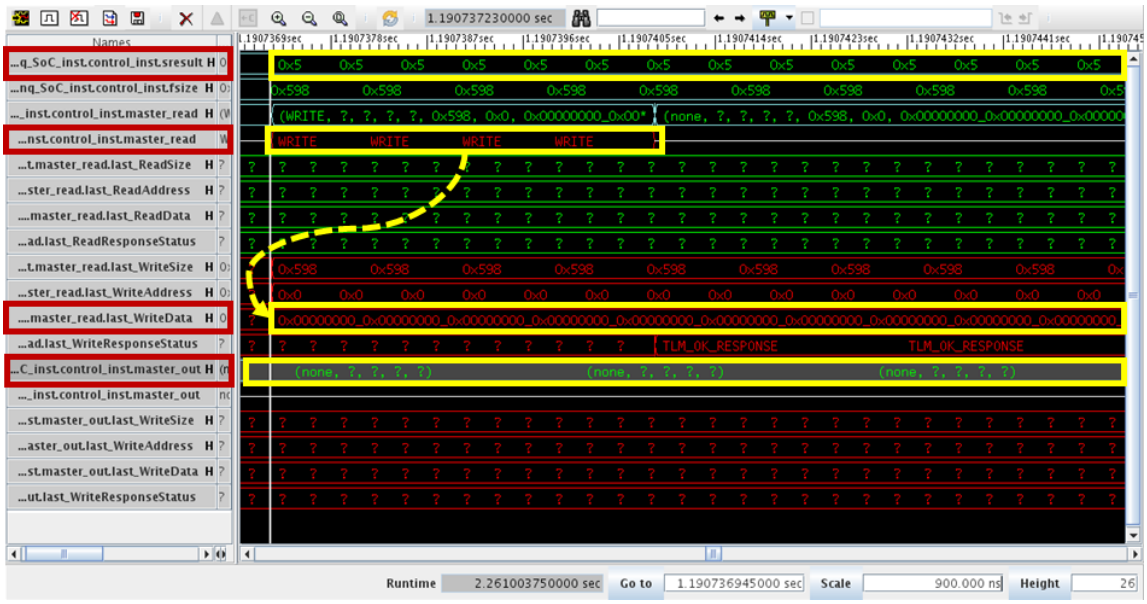



**Figura 88: Forma de onda del modelo *Lectura FIFO.* Desechar el paquete**

## 7.4 Comprobación de la funcionalidad del DPI

Tras demostrar la correcta funcionalidad de los bloques modelados en TLM-2.0, una manera rápida de verificar la funcionalidad de la plataforma modelada al completo incluyendo la comunicación *hardware/software* es realizar la simulación en modo LT. Este modo implementa interfaces bloqueantes donde el grado de detalle temporal es bajo. Se usa para realizar el desarrollo *software* de manera rápida ya que incluye los detalles temporales esenciales para añadir una aplicación, pero también se usa para realizar la verificación funcional del *hardware.*

La primera verificación que se realiza en este modo LT es sin aplicar ninguna política temporal a los modelos. Esto hace que la plataforma se esté simulando en modo *untimed*, es decir, basándose en el modelo funcional puro donde las transacciones se realizan en tiempo cero.

En la Figura 89 se muestra en el visor de forma de onda las acciones que realiza el DPI, desde que el *Traffic Generator* comienza a transmitir paquetes hasta que llegan a bloque que emula la salida hacia la red. Se observa que la plataforma funciona adecuadamente. Para realizar la verificación se ha reducido el número de paquetes enviados por el *Traffic Generator,* enviando únicamente cuatro paquetes, donde el primero y el último son aleatorios y el tercero y el cuarto son iguales, con patrón y

cabecera conocida. Desde la aplicación se ha asignado un patrón que no corresponde con el que se encuentra en el *payload.*

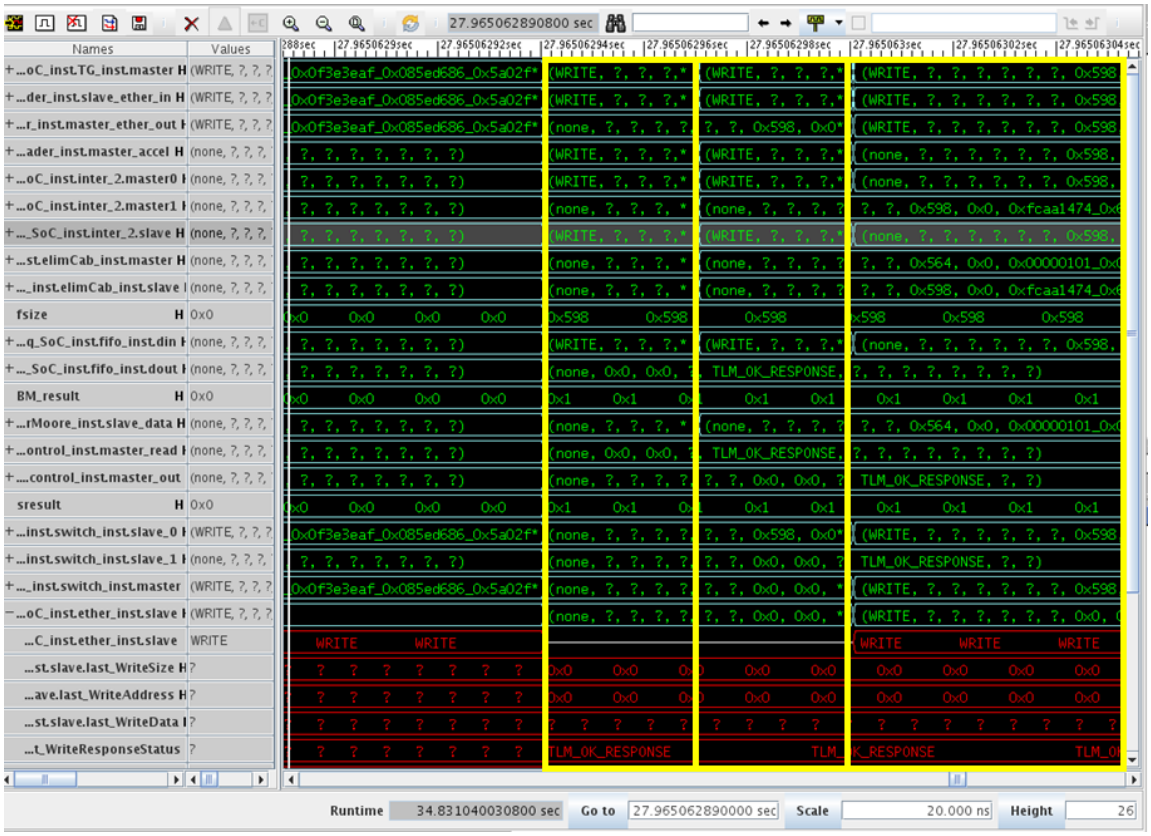



**Figura 89: Forma de onda de la arquitectura DPI. Modo LT**

Tras comprobar la funcionalidad de la arquitectura DPI en modo LT se pasa al modo AT, donde se tiene más detalles temporales y el DPI ya no se ejecuta en tiempo cero. Esto se demuestra en la Figura 90, donde además se observa que el bloque de *Lectura FIFO* no realiza una segunda lectura de la FIFO tras recibir la segunda respuesta del *Boyer-Moore.* Esto es debido a que el bloque está recibiendo más transacciones de las que puede manejar (Figura 91). Por este motivo es necesario la incorporación de políticas temporales y evitar colapsar los *sockets.*

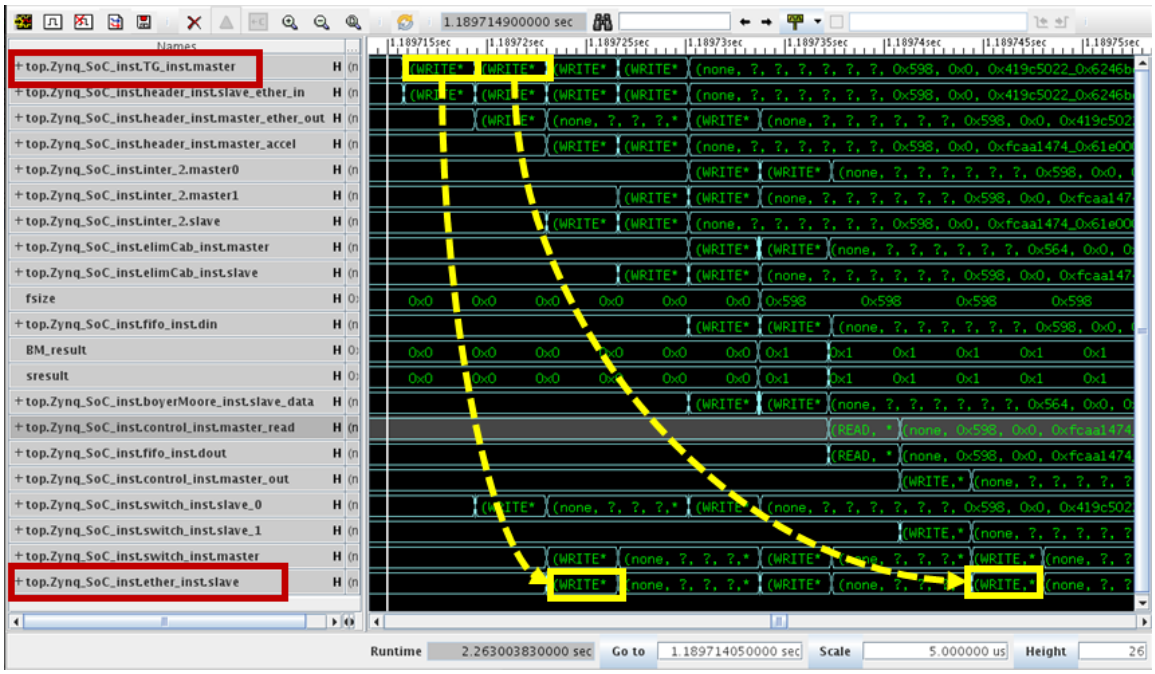



**Figura 90: Forma de onda de la arquitectura DPI. Modo AT sin política temporal**

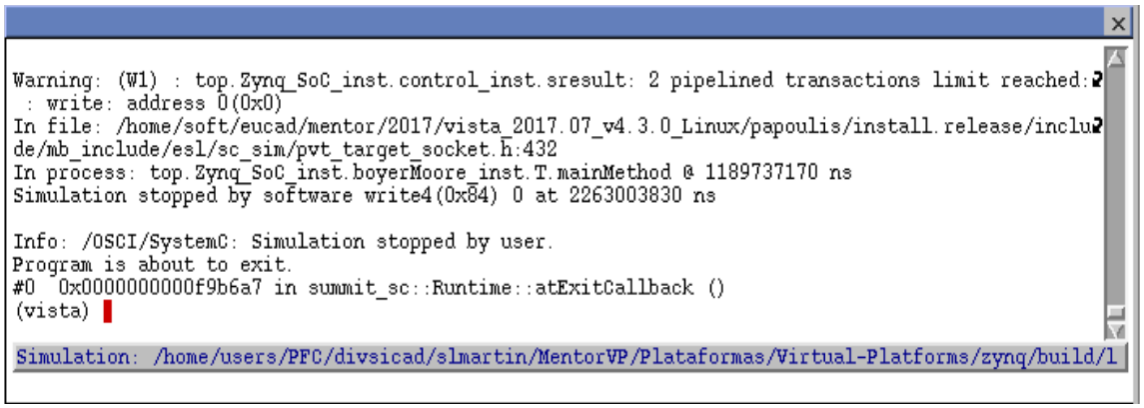



**Figura 91: Mensaje de aviso del bloque *Lectura FIFO***

A continuación, se activan las políticas temporales de los modelos y se simula en modo AT. En la Figura 92 se presenta la forma de onda de esta simulación donde se envían dos paquetes que son inspeccionados en profundidad que no incluyen el patrón. Cabría pensar que el segundo paquete se perdería debido a que el *Traffic Generator* envía paquetes más rápido de lo que el *Boyer-Moore* puede procesar, pero al estar utilizando interfaces bloqueantes la plataforma se ejecuta de manera secuencial debido a que hasta no se reciba confirmación de la transacción completada por parte del último bloque, el bloque anterior no confirmará su transacción.

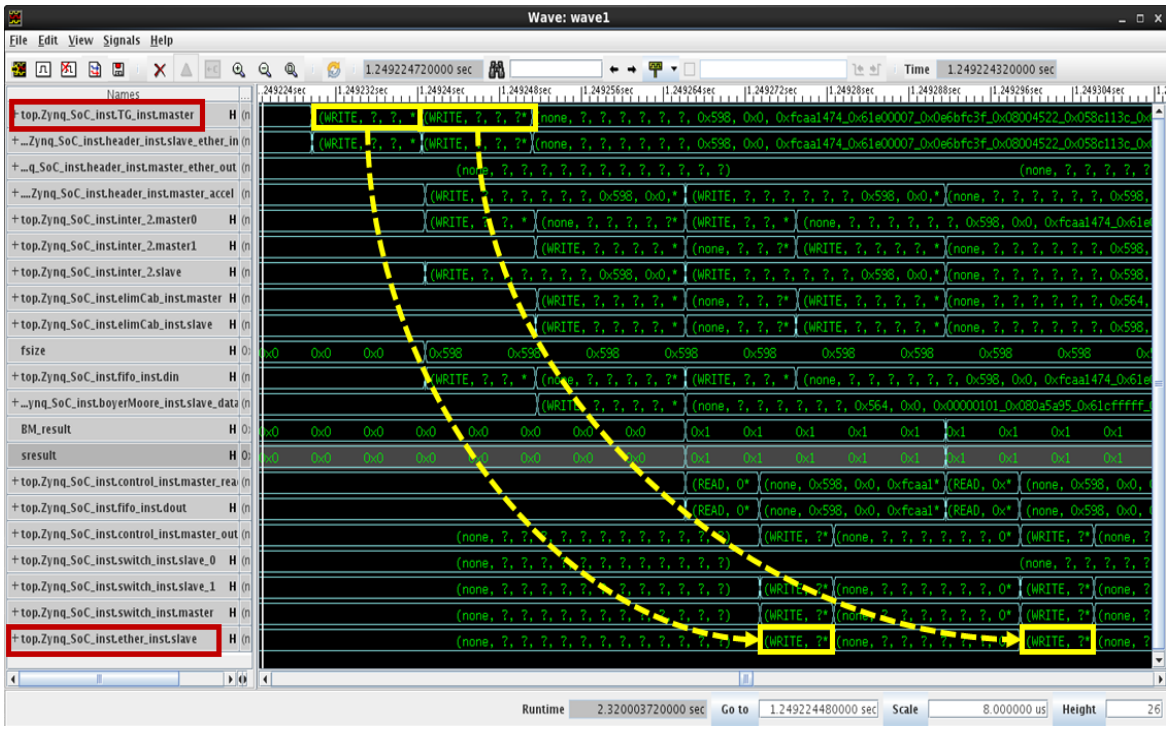



**Figura 92: Forma de onda de la arquitectura DPI. Modo AT con política temporal (I)**

En la Figura 93 se muestra la forma de onda donde el *Boyer-Moore* recibe dos paquetes donde en uno contiene el patrón y en otro no. Con esto se verifica el funcionamiento del bloque *Lectura FIFO* que limpia la FIFO y también del *Boyer-Moore*.

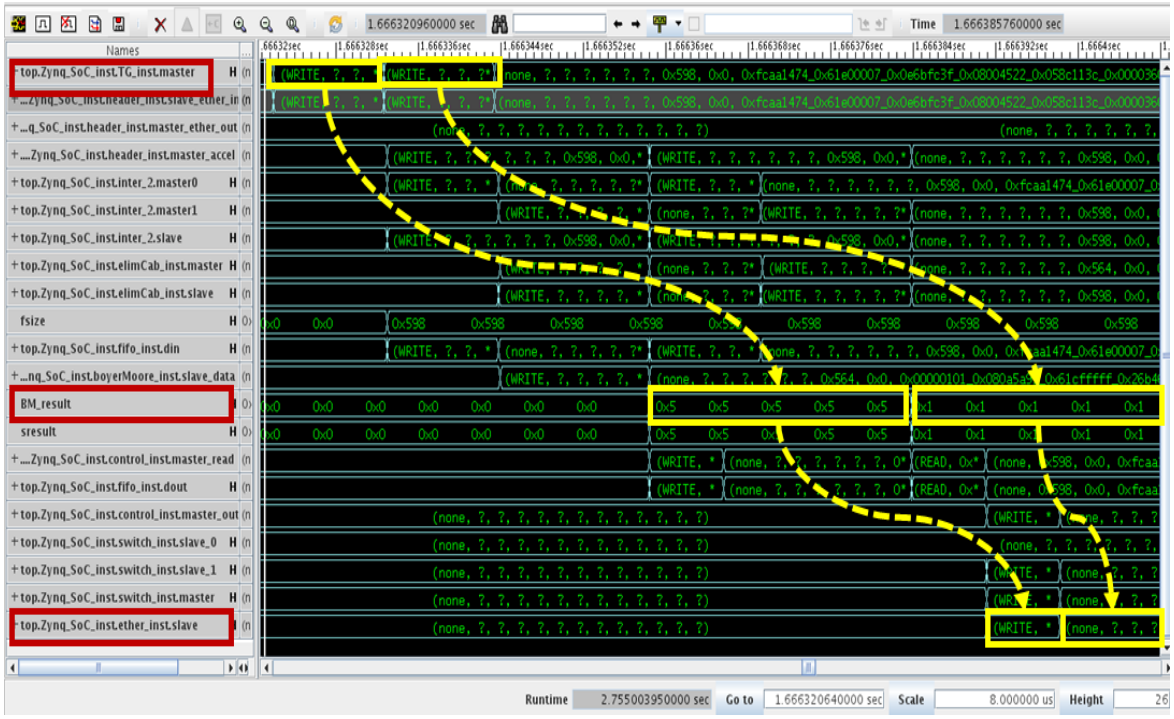



**Figura 93: Forma de onda de la arquitectura DPI. Modo AT con política temporal (II)**

Por último, en la Figura 94 se verifica a través de la forma de onda cuando se recibe un paquete que no es necesario realizar una inspección profunda y otra que sí. Con esto se verifica la funcionalidad del *Header Analyzer.*

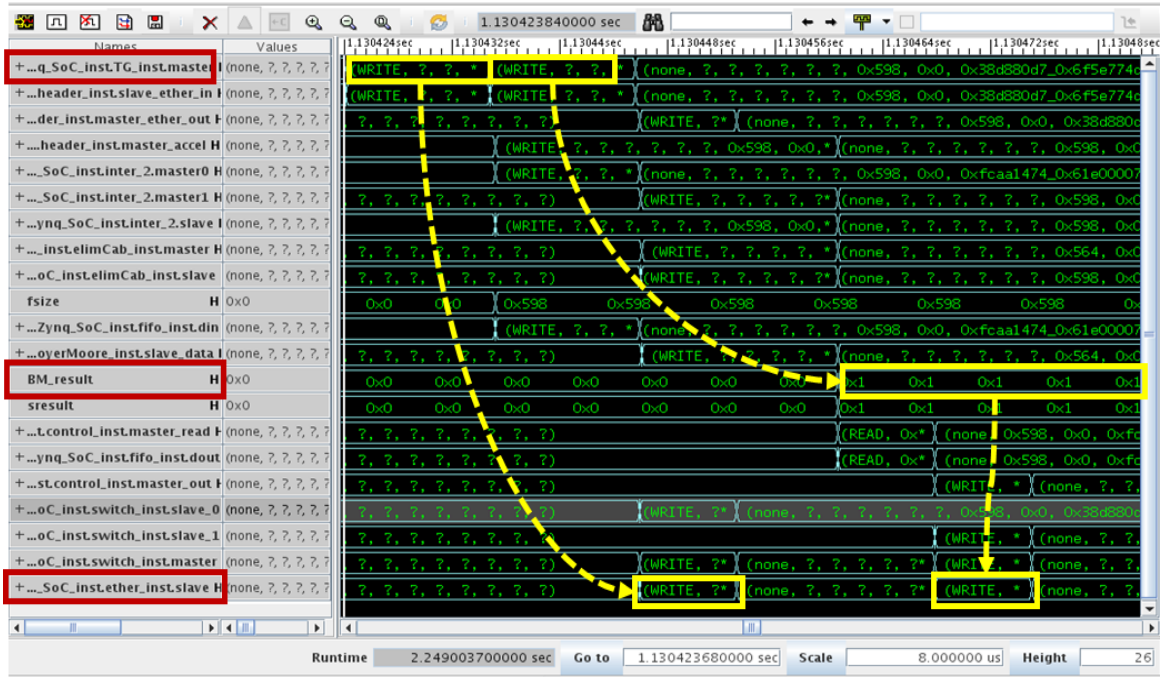



**Figura 94: Forma de onda de la arquitectura DPI. Modo AT con política temporal (III)**

Después de realizar estas verificaciones y confirmar el correcto funcionamiento se puede crear la plataforma virtual que se explica en el siguiente punto.

## 7.5 Generación de la Plataforma Virtual

La plataforma virtual es un ejecutable *stand-alone* que permite invocar la simulación de un diseño que ha sido creado con la herramienta Vista. Este fichero se crea a partir del fichero *sc_main* y contiene los siguientes elementos:

- Ejecutable de la simulación.
- Librerías compartidas necesarias.
- Ficheros de tiempo de ejecución para ejecutar la simulación.
- Ficheros adicionales.

Desde el menú *Tools* de la interfaz gráfica de la herramienta se puede generar la plataforma virtual o haciendo uso de la línea de comando. En la Figura 95 se muestra la

ventana emergente para la configuración de la plataforma desde la interfaz gráfica. Una vez indicado el nombre y los ficheros que se requieran incorporar, se genera la plataforma. El proceso se muestra a través de la consola del programa y se muestra en la Figura 96. Si existiera algún error durante su generación se mostraría en la consola.

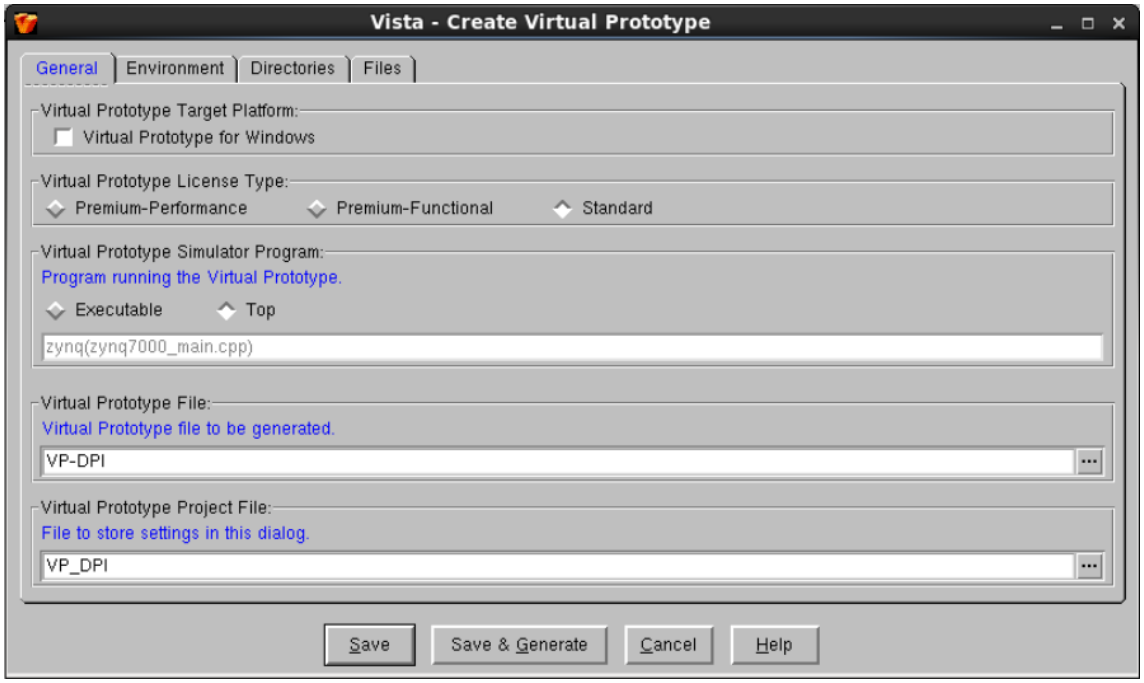



**Figura 95: Creación de la plataforma virtual**

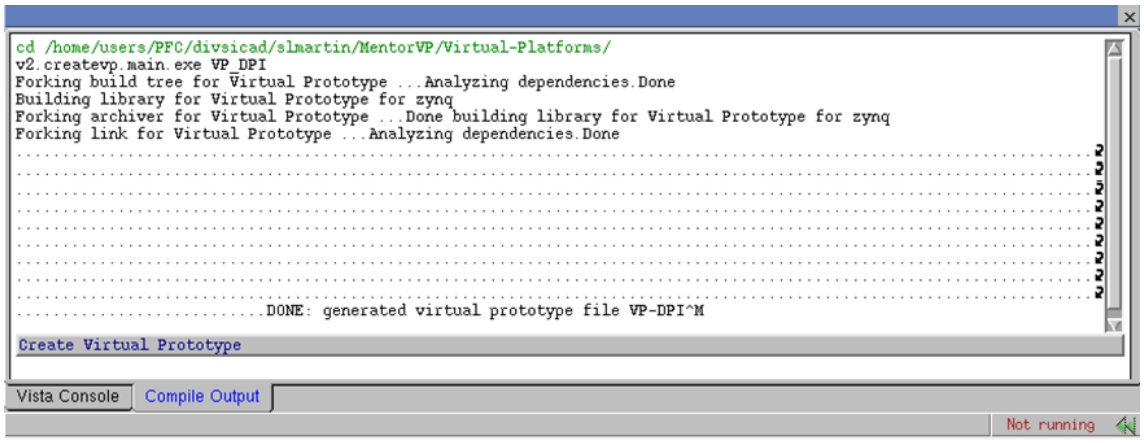



**Figura 96: Plataforma virtual generada**

Una vez se ha generado el fichero se puede proceder a la exploración de la plataforma, que se tratará en el capítulo siguiente.

## 7.6 Conclusiones

Se ha explicado el proceso de modelado y generación de la plataforma virtual, incluyendo el ensamblaje de los distintos bloques que conforman la plataforma, el desarrollo de las aplicaciones que se ejecutan en el procesador (finalmente en el ISS) y la verificación funcional de los modelos creados, incluyendo el bloque Header *Analyzer*, el bloque *Eliminar Cabecera,* el bloque de búsqueda de patrones *Boyer-Moore* y los bloques de gestión de la FIFO. Con todo ello la plataforma virtual queda preparada para su generación y posterior análisis.

# Capítulo 8. Exploración de la plataforma virtual

En este capítulo se analiza la plataforma virtual creada del DPI con el objetivo de estudiar su comportamiento y funcionalidad aplicando diversas políticas temporales.

## 8.1 Instalación y ejecución de la plataforma virtual

Una plataforma virtual es un ejecutable que permite invocar la simulación de un diseño para realizar el análisis completo de la plataforma. Con el fichero de la plataforma virtual que se ha generado en Vista en el capítulo anterior se realizan los análisis del DPI. Para ello es necesario realizar la instalación de la plataforma virtual, especificando un directorio, que no debe existir previamente, donde se almacenarán los ejecutables de la plataforma, documentación, librerías para la ejecución del *software* y de la propia plataforma. La instalación se realiza a través del comando *-install* añadiendo el directorio que creará para la instalación. Si el directorio existiera previamente o no fuera especificado se mostraría un mensaje de error.

En la Figura 97 se muestra el proceso de instalación, donde *VP-DPI* es el nombre de la plataforma virtual creada y *VirtualPlatform_DPI* es el directorio.

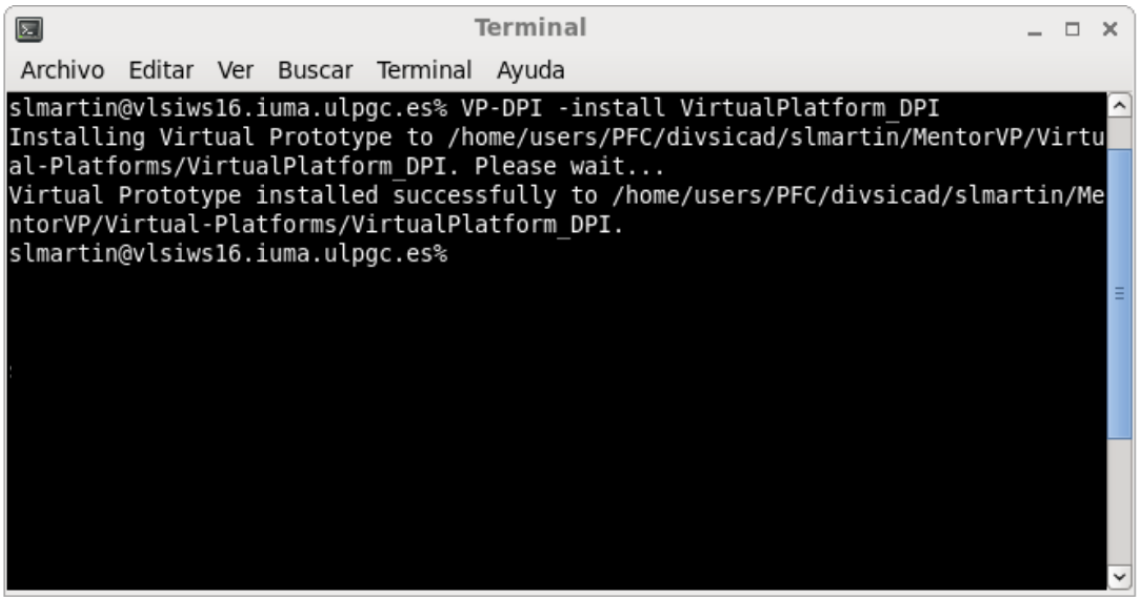



**Figura 97: Instalación de la plataforma virtual**

Después de que la instalación se realice con éxito, en el directorio de instalación que se ha indicado se crea el ejecutable *run.exe* que es una copia del ejecutable original de la plataforma virtual creada. En la Figura 98 se muestra el directorio que se crea tras la instalación.

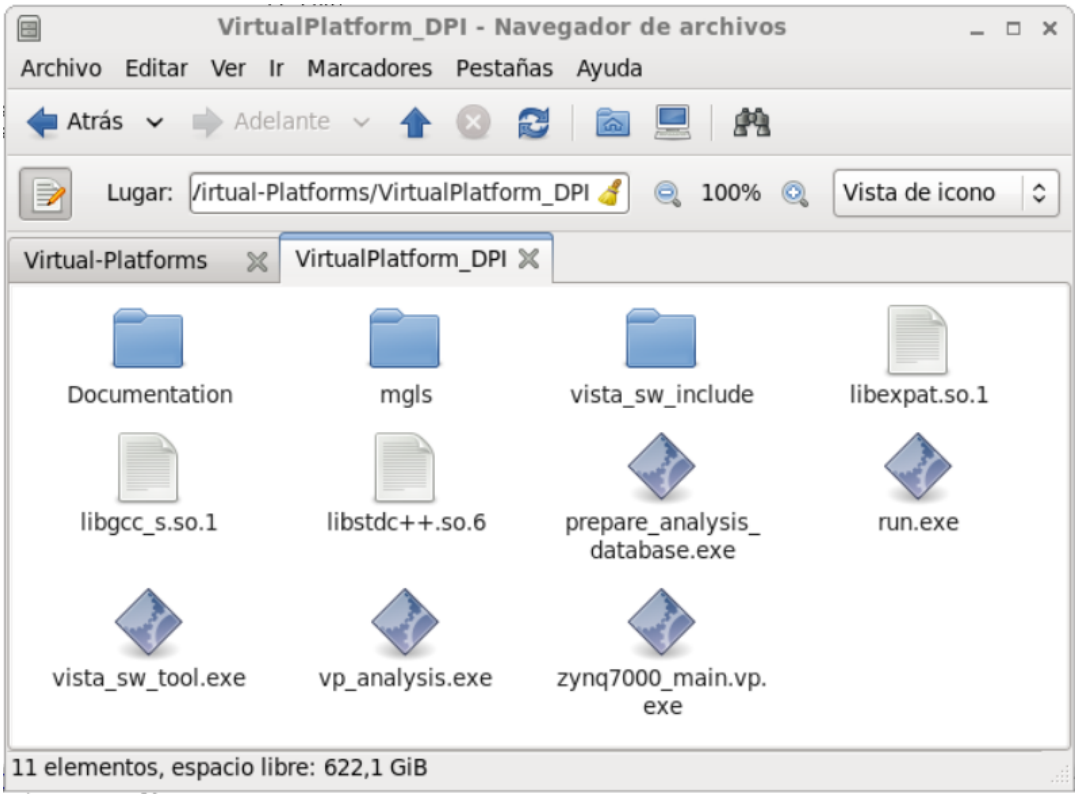



**Figura 98: Directorio creado tras la instalación de la plataforma virtual**

Para ejecutar la plataforma virtual se ejecuta el fichero generado tras la instalación que se encarga de invocar la simulación, cargar los ficheros de parámetros y abrir un nuevo terminal para la UART. En el mismo terminal se muestra los mensajes que se imprimen por pantalla con el uso del *printf* o *cout* en el código. En la Figura 99 se muestra la ejecución de la plataforma virtual.

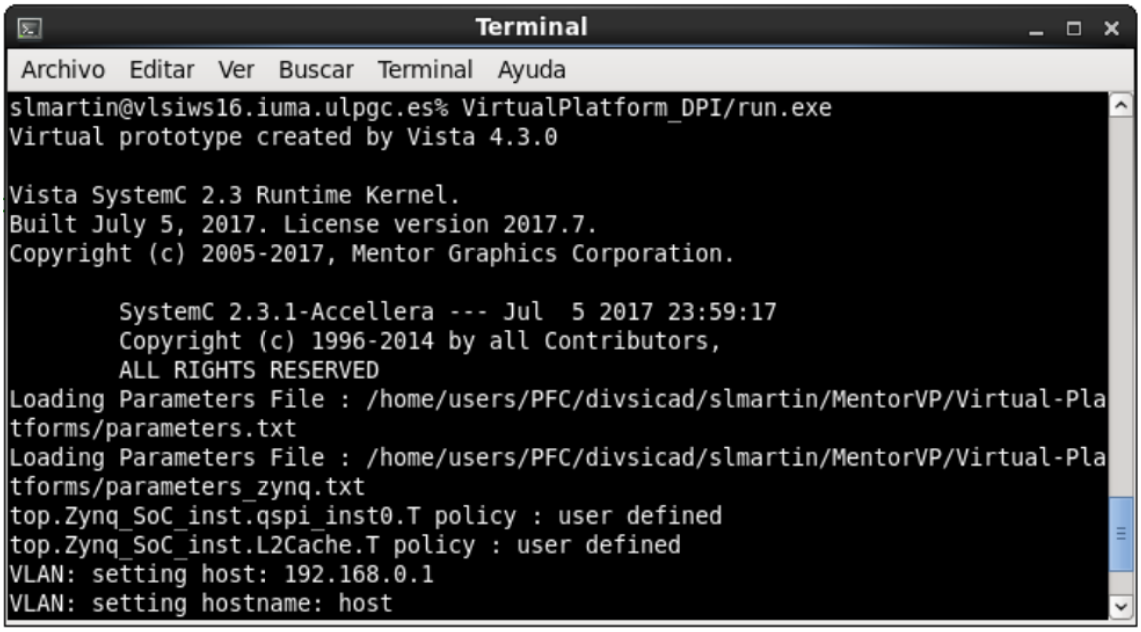



**Figura 99: Ejecución la plataforma virtual**

Los diferentes análisis que se pueden realizar en la plataforma virtual son cuatro:

1. Potencia por cada instancia de modelo y diseño completo con el comando *-vp-trace-power*
2. Tiempo de las transacciones en cada *socket* con el comando *-vp-trace_sockets*
3. Atributos definidos por el usuario con el comando *-vp-trace-attributes*
4. Tasa de *hit/miss* de la cache con el comando *-vp-trace-cache*

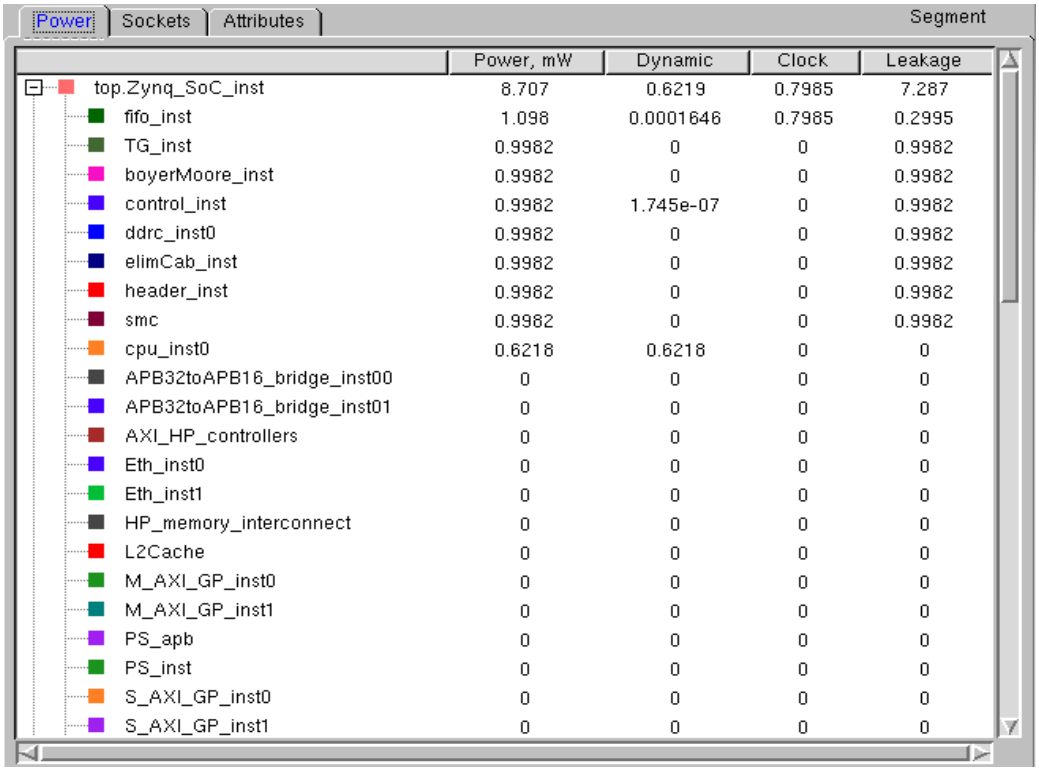
Al ejecutar la plataforma virtual se añade el comando correspondiente al análisis que se desea realizar. El resultado se almacena en la carpeta de simulación que se crea automáticamente en el directorio que contiene la plataforma virtual, si el usuario no ha indicado otro. Si no se especifica ninguna opción de análisis, la plataforma virtual no creará ningún directorio de simulación.

Ejecutando el fichero *prepare_analysis_database.exe* que se genera al instalar la plataforma virtual se puede crear una base de datos con todos los análisis que genera la simulación y se muestran con el Visor de Análisis de la interfaz de usuario de Vista al lanzar el ejecutable *vp_analysis.exe* también generado en la instalación.

## 8.2 Análisis de la plataforma virtual

Con la plataforma virtual creada e instalada, se realiza su ejecución definiendo los análisis a realizar, que son los de potencia y de *sockets*. Después de preparar la base de datos del análisis, se lanza el Visor de Análisis para realizar la exploración de la plataforma.

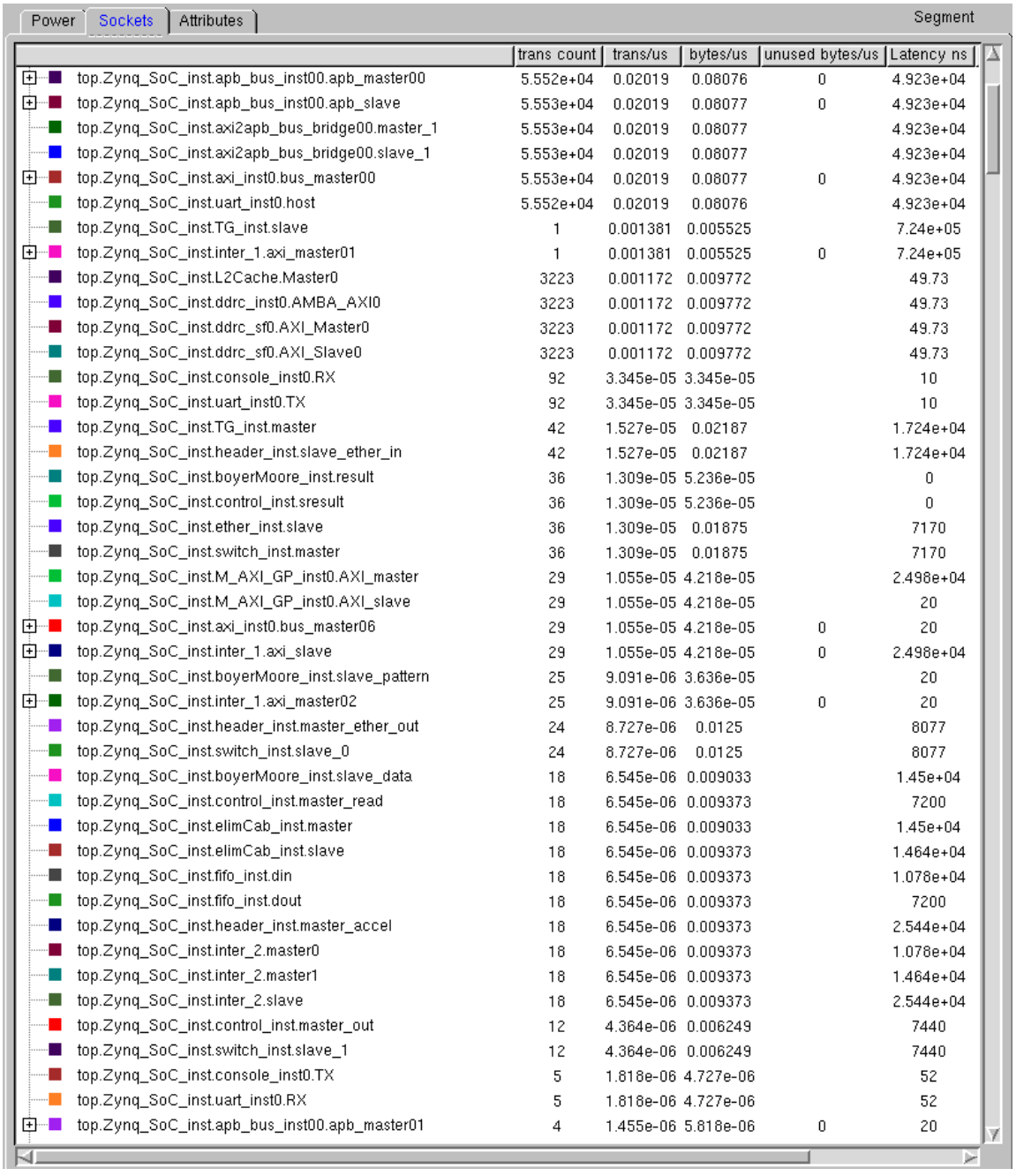
En la Figura 100 se muestra la estimación de la potencia media que cuenta la plataforma durante el tiempo que dura la simulación. De media, la plataforma virtual tiene un consumo potencia de 8,707 mW, mientras que los distintos modelos tienen 1 mW que corresponde con la potencia estática de los modelos. Estos valores están promediados con respecto al tiempo de simulación efectivo del modelo. Es preciso tomar valores relativos y no absolutos de las cifras finales.




| | Power, mW | Dynamic | Clock | Leakage |
|---|---|---|---|---|
| top.Zynq_SoC_inst | 8.707 | 0.6219 | 0.7985 | 7.287 |
| fifo_inst | 1.098 | 0.0001646 | 0.7985 | 0.2995 |
| TG_inst | 0.9982 | 0 | 0 | 0.9982 |
| boyerMoore_inst | 0.9982 | 0 | 0 | 0.9982 |
| control_inst | 0.9982 | 1.745e-07 | 0 | 0.9982 |
| ddrc_inst0 | 0.9982 | 0 | 0 | 0.9982 |
| elimCab_inst | 0.9982 | 0 | 0 | 0.9982 |
| header_inst | 0.9982 | 0 | 0 | 0.9982 |
| smc | 0.9982 | 0 | 0 | 0.9982 |
| cpu_inst0 | 0.6218 | 0.6218 | 0 | 0 |
| APB32toAPB16_bridge_inst00 | 0 | 0 | 0 | 0 |
| APB32toAPB16_bridge_inst01 | 0 | 0 | 0 | 0 |
| AXI_HP_controllers | 0 | 0 | 0 | 0 |
| Eth_inst0 | 0 | 0 | 0 | 0 |
| Eth_inst1 | 0 | 0 | 0 | 0 |
| HP_memory_interconnect | 0 | 0 | 0 | 0 |
| L2Cache | 0 | 0 | 0 | 0 |
| M_AXI_GP_inst0 | 0 | 0 | 0 | 0 |
| M_AXI_GP_inst1 | 0 | 0 | 0 | 0 |
| PS_apb | 0 | 0 | 0 | 0 |
| PS_inst | 0 | 0 | 0 | 0 |
| S_AXI_GP_inst0 | 0 | 0 | 0 | 0 |
| S_AXI_GP_inst1 | 0 | 0 | 0 | 0 |

**Figura 100: Análisis del consumo de potencia de la plataforma virtual**

En la Figura 101 se muestra el análisis realizado a todos los *sockets* que conforman la plataforma, mostrando el número de transacciones realizadas durante todo el tiempo de la simulación, el rendimiento de cada *socket* medido en transacciones/us y byte/us y su latencia (ns).


Power | Sockets | Attributes — Segment

| | trans count | trans/us | bytes/us | unused bytes/us | Latency ns |
|---|---|---|---|---|---|
| top.Zynq_SoC_inst.apb_bus_inst00.apb_master00 | 5.552e+04 | 0.02019 | 0.08076 | 0 | 4.923e+04 |
| top.Zynq_SoC_inst.apb_bus_inst00.apb_slave | 5.553e+04 | 0.02019 | 0.08077 | 0 | 4.923e+04 |
| top.Zynq_SoC_inst.axi2apb_bus_bridge00.master_1 | 5.553e+04 | 0.02019 | 0.08077 | | 4.923e+04 |
| top.Zynq_SoC_inst.axi2apb_bus_bridge00.slave_1 | 5.553e+04 | 0.02019 | 0.08077 | | 4.923e+04 |
| top.Zynq_SoC_inst.axi_inst0.bus_master00 | 5.553e+04 | 0.02019 | 0.08077 | 0 | 4.923e+04 |
| top.Zynq_SoC_inst.uart_inst0.host | 5.552e+04 | 0.02019 | 0.08076 | | 4.923e+04 |
| top.Zynq_SoC_inst.TG_inst.slave | 1 | 0.001381 | 0.005525 | | 7.24e+05 |
| top.Zynq_SoC_inst.inter_1.axi_master01 | 1 | 0.001381 | 0.005525 | 0 | 7.24e+05 |
| top.Zynq_SoC_inst.L2Cache.Master0 | 3223 | 0.001172 | 0.009772 | | 49.73 |
| top.Zynq_SoC_inst.ddrc_inst0.AMBA_AXI0 | 3223 | 0.001172 | 0.009772 | | 49.73 |
| top.Zynq_SoC_inst.ddrc_sf0.AXI_Master0 | 3223 | 0.001172 | 0.009772 | | 49.73 |
| top.Zynq_SoC_inst.ddrc_sf0.AXI_Slave0 | 3223 | 0.001172 | 0.009772 | | 49.73 |
| top.Zynq_SoC_inst.console_inst0.RX | 92 | 3.345e-05 | 3.345e-05 | | 10 |
| top.Zynq_SoC_inst.uart_inst0.TX | 92 | 3.345e-05 | 3.345e-05 | | 10 |
| top.Zynq_SoC_inst.TG_inst.master | 42 | 1.527e-05 | 0.02187 | | 1.724e+04 |
| top.Zynq_SoC_inst.header_inst.slave_ether_in | 42 | 1.527e-05 | 0.02187 | | 1.724e+04 |
| top.Zynq_SoC_inst.boyerMoore_inst.result | 36 | 1.309e-05 | 5.236e-05 | | 0 |
| top.Zynq_SoC_inst.control_inst.sresult | 36 | 1.309e-05 | 5.236e-05 | | 0 |
| top.Zynq_SoC_inst.ether_inst.slave | 36 | 1.309e-05 | 0.01875 | | 7170 |
| top.Zynq_SoC_inst.switch_inst.master | 36 | 1.309e-05 | 0.01875 | | 7170 |
| top.Zynq_SoC_inst.M_AXI_GP_inst0.AXI_master | 29 | 1.055e-05 | 4.218e-05 | | 2.498e+04 |
| top.Zynq_SoC_inst.M_AXI_GP_inst0.AXI_slave | 29 | 1.055e-05 | 4.218e-05 | | 20 |
| top.Zynq_SoC_inst.axi_inst0.bus_master06 | 29 | 1.055e-05 | 4.218e-05 | 0 | 20 |
| top.Zynq_SoC_inst.inter_1.axi_slave | 29 | 1.055e-05 | 4.218e-05 | 0 | 2.498e+04 |
| top.Zynq_SoC_inst.boyerMoore_inst.slave_pattern | 25 | 9.091e-06 | 3.636e-05 | | 20 |
| top.Zynq_SoC_inst.inter_1.axi_master02 | 25 | 9.091e-06 | 3.636e-05 | 0 | 20 |
| top.Zynq_SoC_inst.header_inst.master_ether_out | 24 | 8.727e-06 | 0.0125 | | 8077 |
| top.Zynq_SoC_inst.switch_inst.slave_0 | 24 | 8.727e-06 | 0.0125 | | 8077 |
| top.Zynq_SoC_inst.boyerMoore_inst.slave_data | 18 | 6.545e-06 | 0.009033 | | 1.45e+04 |
| top.Zynq_SoC_inst.control_inst.master_read | 18 | 6.545e-06 | 0.009373 | | 7200 |
| top.Zynq_SoC_inst.elimCab_inst.master | 18 | 6.545e-06 | 0.009033 | | 1.45e+04 |
| top.Zynq_SoC_inst.elimCab_inst.slave | 18 | 6.545e-06 | 0.009373 | | 1.464e+04 |
| top.Zynq_SoC_inst.fifo_inst.din | 18 | 6.545e-06 | 0.009373 | | 1.078e+04 |
| top.Zynq_SoC_inst.fifo_inst.dout | 18 | 6.545e-06 | 0.009373 | | 7200 |
| top.Zynq_SoC_inst.header_inst.master_accel | 18 | 6.545e-06 | 0.009373 | | 2.544e+04 |
| top.Zynq_SoC_inst.inter_2.master0 | 18 | 6.545e-06 | 0.009373 | | 1.078e+04 |
| top.Zynq_SoC_inst.inter_2.master1 | 18 | 6.545e-06 | 0.009373 | | 1.464e+04 |
| top.Zynq_SoC_inst.inter_2.slave | 18 | 6.545e-06 | 0.009373 | | 2.544e+04 |
| top.Zynq_SoC_inst.control_inst.master_out | 12 | 4.364e-06 | 0.006249 | | 7440 |
| top.Zynq_SoC_inst.switch_inst.slave_1 | 12 | 4.364e-06 | 0.006249 | | 7440 |
| top.Zynq_SoC_inst.console_inst0.TX | 5 | 1.818e-06 | 4.727e-06 | | 52 |
| top.Zynq_SoC_inst.uart_inst0.RX | 5 | 1.818e-06 | 4.727e-06 | | 52 |
| top.Zynq_SoC_inst.apb_bus_inst00.apb_master01 | 4 | 1.455e-06 | 5.818e-06 | 0 | 20 |

**Figura 101: Análisis de los *sockets* de la plataforma virtual**

Estos análisis se pueden mostrar también de manera gráfica. En la Figura 102 se muestra el consumo de potencia de la plataforma durante la simulación donde el eje X representa el tiempo de la simulación y el eje Y representa la potencia en mW.

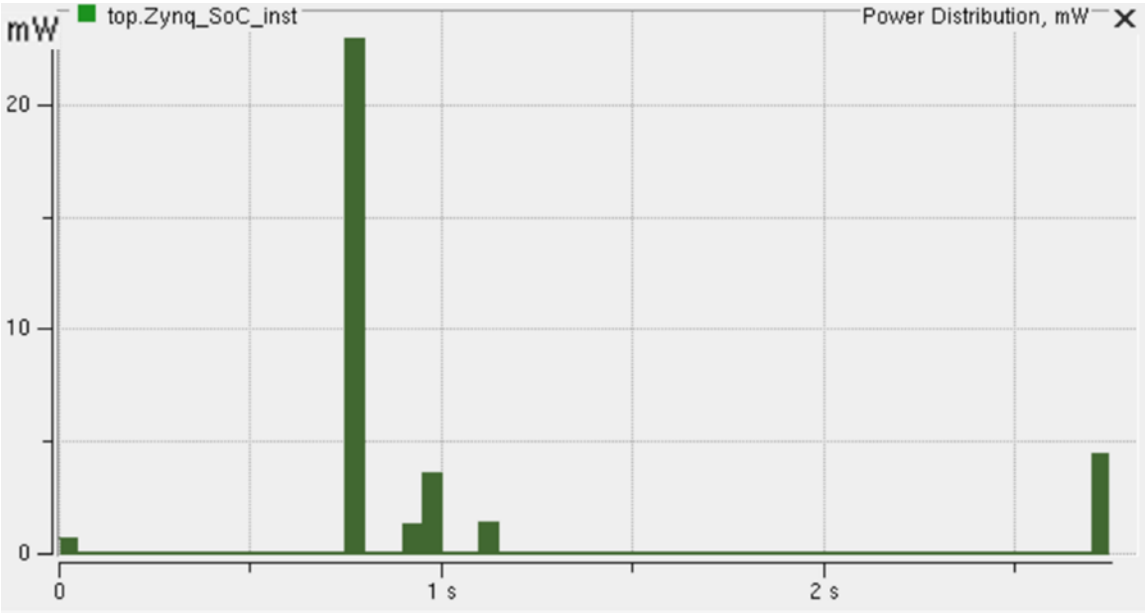



**Figura 102: Distribución de potencia de la plataforma virtual durante el tiempo de simulación**

La duración de la simulación en este análisis es de 2,74 segundos. Es un tiempo grande considerando que la arquitectura DPI trabaja a frecuencias de MHz. Realizar el análisis de la plataforma tomando todo el tiempo de la solución, como se ha mostrado en las figuras anteriores, hace que la interpretación de los resultados sea complicada. Por lo tanto, a continuación, se realiza un análisis en detalle de la plataforma virtual, donde se divide el análisis en cuatro partes:

1. Configuración de la plataforma
2. Incorporación del patrón de búsqueda
3. Inspección de los paquetes
4. Finalización de la plataforma

### 8.2.1 Análisis de la configuración de la plataforma

En la primera etapa, se realiza la configuración la plataforma DPI, donde se arranca la plataforma, se inicializa la UART y se configuran los parámetros de análisis de cabecera, realizándose desde la aplicación *software* desarrollada. En la Figura 103 se muestra la potencia invertida en esta etapa, donde se observa que el mayor consumo lo realiza la CPU, entorno a los 100 mW, y que en total la plataforma en ese momento alcanza un consumo de 110 mW. A partir de los 300 µs, la potencia consumida disminuye.

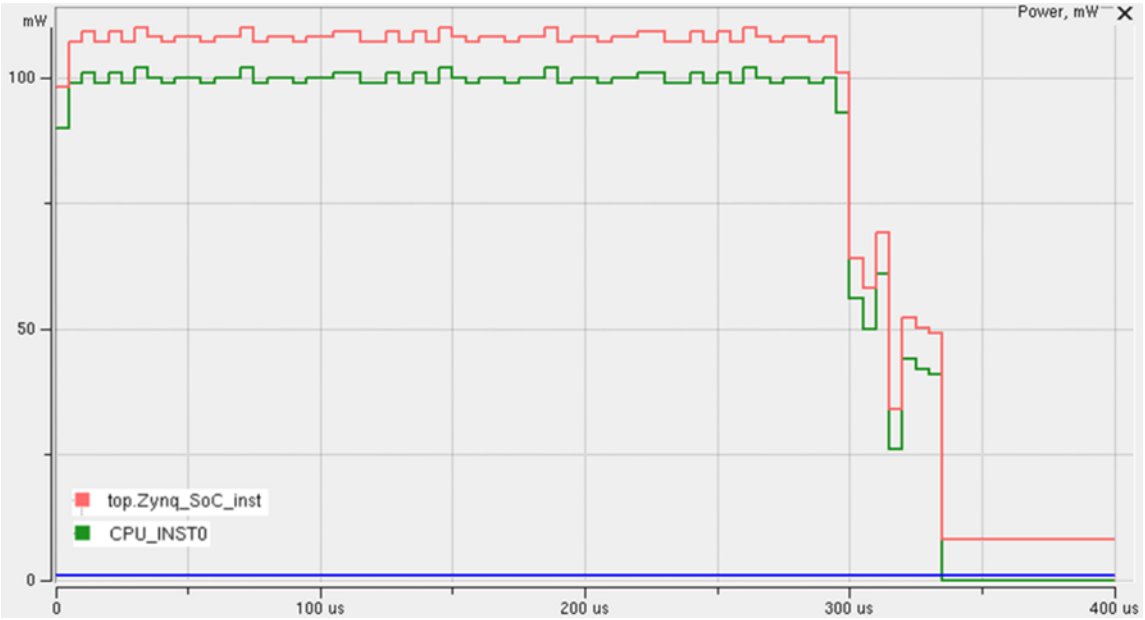



**Figura 103: Análisis de potencia de la etapa de configuración de la plataforma**

Las transacciones que ocurren durante los primeros 400 µs de la simulación muestran de dos partes. La primera, es la que corresponde con la Figura 104 y corresponde desde el tiempo cero hasta 300 µs. La segunda parte, se muestra en la Figura 105 y engloba desde 300 µs hasta 340 µs.

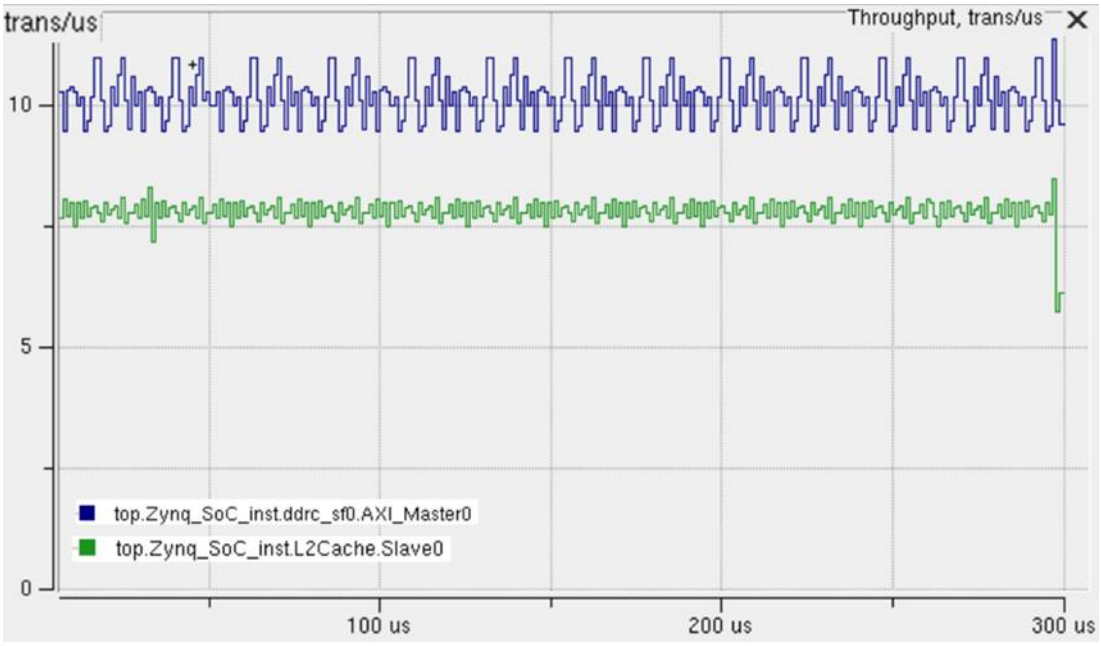



**Figura 104: T*hroughput* en la etapa de configuración de la plataforma medido en trans/µs (I)**

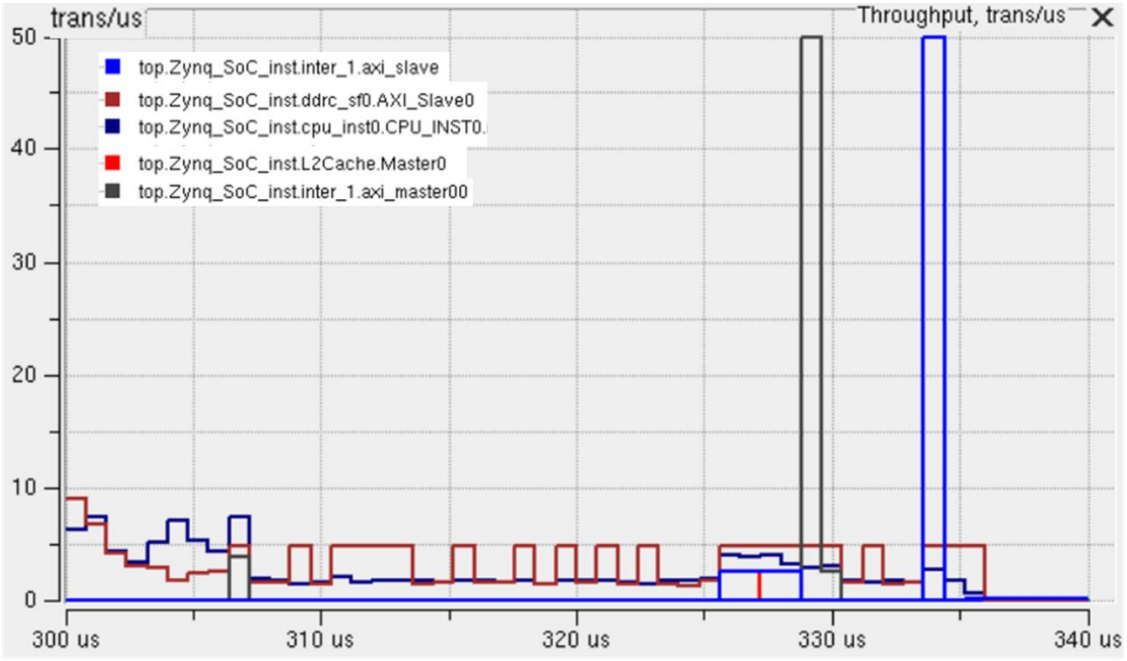
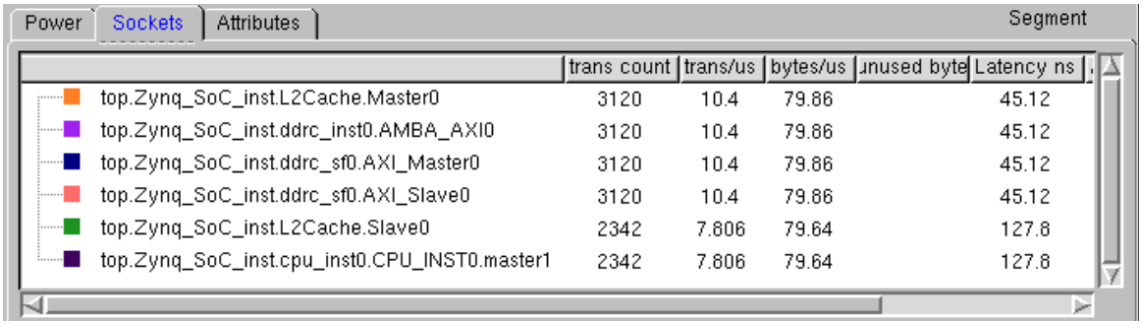



**Figura 105: *Throughput* en la etapa de configuración de la plataforma medido en trans/µs (II)**

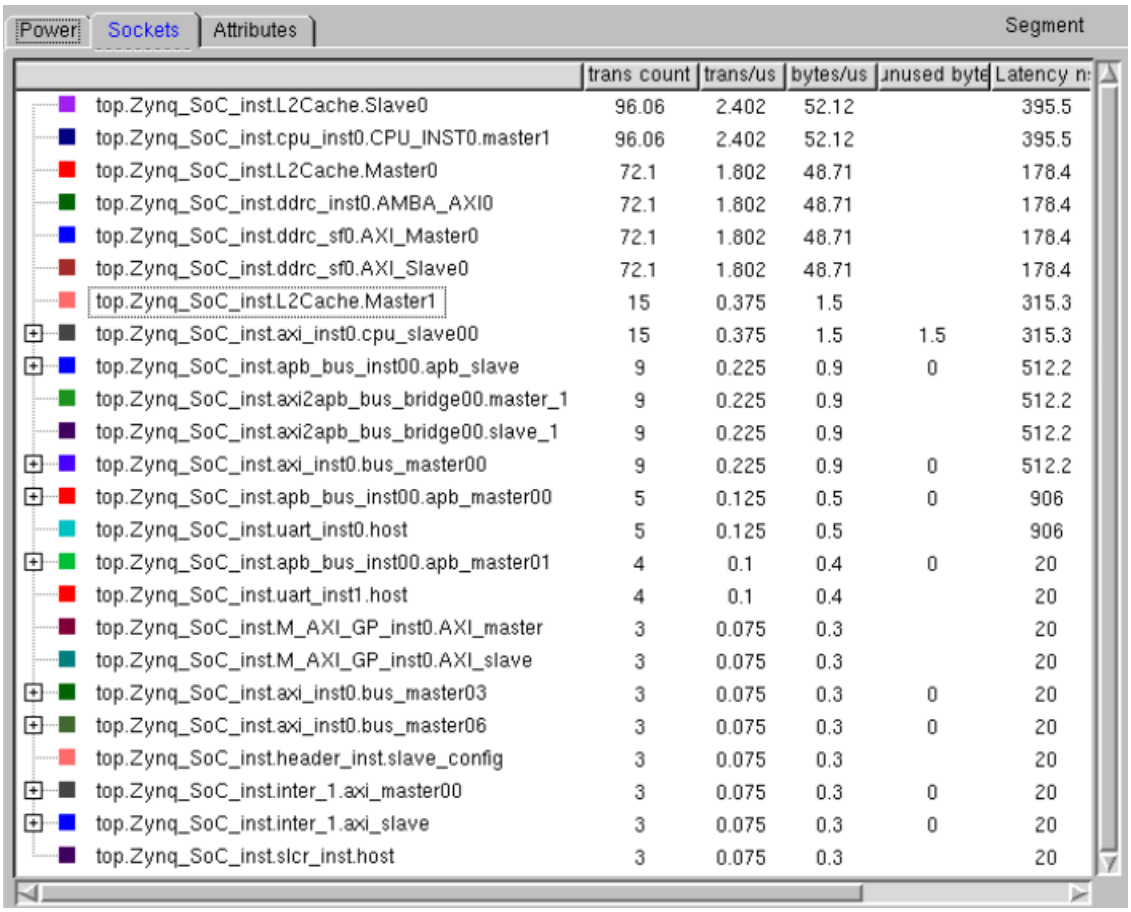
De la Figura 104 y con ayuda de la Figura 106 se observa que, durante este periodo de la simulación, la CPU está realizando escrituras y lecturas sobre la cache de nivel 2 y el DDR del Zynq. Desde el procesador se envía una media de 2.342 transacciones con un rendimiento de 7,8 transacciones/µs, justificando así el alto consumo de este bloque.




| | trans count | trans/us | bytes/us | unused byte | Latency ns |
|---|---|---|---|---|---|
| top.Zynq_SoC_inst.L2Cache.Master0 | 3120 | 10.4 | 79.86 | | 45.12 |
| top.Zynq_SoC_inst.ddrc_inst0.AMBA_AXI0 | 3120 | 10.4 | 79.86 | | 45.12 |
| top.Zynq_SoC_inst.ddrc_sf0.AXI_Master0 | 3120 | 10.4 | 79.86 | | 45.12 |
| top.Zynq_SoC_inst.ddrc_sf0.AXI_Slave0 | 3120 | 10.4 | 79.86 | | 45.12 |
| top.Zynq_SoC_inst.L2Cache.Slave0 | 2342 | 7.806 | 79.64 | | 127.8 |
| top.Zynq_SoC_inst.cpu_inst0.CPU_INST0.master1 | 2342 | 7.806 | 79.64 | | 127.8 |

**Figura 106: Análisis de las transacciones en la etapa de configuración de la plataforma (I)**

Con la Figura 105 y analizando la Figura 107 se observa que desde 300 µs hasta los 335 µs aproximadamente la CPU sigue enviando transacciones y la cache L2 y el DDR siguen operativas con una media de 2 transacciones/µs.




| | trans count | trans/us | bytes/us | unused byte | Latency n |
|---|---|---|---|---|---|
| top.Zynq_SoC_inst.L2Cache.Slave0 | 96.06 | 2.402 | 52.12 | | 395.5 |
| top.Zynq_SoC_inst.cpu_inst0.CPU_INST0.master1 | 96.06 | 2.402 | 52.12 | | 395.5 |
| top.Zynq_SoC_inst.L2Cache.Master0 | 72.1 | 1.802 | 48.71 | | 178.4 |
| top.Zynq_SoC_inst.ddrc_inst0.AMBA_AXI0 | 72.1 | 1.802 | 48.71 | | 178.4 |
| top.Zynq_SoC_inst.ddrc_sf0.AXI_Master0 | 72.1 | 1.802 | 48.71 | | 178.4 |
| top.Zynq_SoC_inst.ddrc_sf0.AXI_Slave0 | 72.1 | 1.802 | 48.71 | | 178.4 |
| top.Zynq_SoC_inst.L2Cache.Master1 | 15 | 0.375 | 1.5 | | 315.3 |
| top.Zynq_SoC_inst.axi_inst0.cpu_slave00 | 15 | 0.375 | 1.5 | 1.5 | 315.3 |
| top.Zynq_SoC_inst.apb_bus_inst00.apb_slave | 9 | 0.225 | 0.9 | 0 | 512.2 |
| top.Zynq_SoC_inst.axi2apb_bus_bridge00.master_1 | 9 | 0.225 | 0.9 | | 512.2 |
| top.Zynq_SoC_inst.axi2apb_bus_bridge00.slave_1 | 9 | 0.225 | 0.9 | | 512.2 |
| top.Zynq_SoC_inst.axi_inst0.bus_master00 | 9 | 0.225 | 0.9 | 0 | 512.2 |
| top.Zynq_SoC_inst.apb_bus_inst00.apb_master00 | 5 | 0.125 | 0.5 | 0 | 906 |
| top.Zynq_SoC_inst.uart_inst0.host | 5 | 0.125 | 0.5 | | 906 |
| top.Zynq_SoC_inst.apb_bus_inst00.apb_master01 | 4 | 0.1 | 0.4 | 0 | 20 |
| top.Zynq_SoC_inst.uart_inst1.host | 4 | 0.1 | 0.4 | | 20 |
| top.Zynq_SoC_inst.M_AXI_GP_inst0.AXI_master | 3 | 0.075 | 0.3 | | 20 |
| top.Zynq_SoC_inst.M_AXI_GP_inst0.AXI_slave | 3 | 0.075 | 0.3 | | 20 |
| top.Zynq_SoC_inst.axi_inst0.bus_master03 | 3 | 0.075 | 0.3 | 0 | 20 |
| top.Zynq_SoC_inst.axi_inst0.bus_master06 | 3 | 0.075 | 0.3 | 0 | 20 |
| top.Zynq_SoC_inst.header_inst.slave_config | 3 | 0.075 | 0.3 | | 20 |
| top.Zynq_SoC_inst.inter_1.axi_master00 | 3 | 0.075 | 0.3 | 0 | 20 |
| top.Zynq_SoC_inst.inter_1.axi_slave | 3 | 0.075 | 0.3 | 0 | 20 |
| top.Zynq_SoC_inst.slcr_inst.host | 3 | 0.075 | 0.3 | | 20 |

**Figura 107: Análisis de las transacciones en la etapa de configuración de la plataforma (II)**

A partir de los 325 µs de simulación se eleva la cantidad de transacciones enviadas. Con más detalle se puede observar en la Figura 108 que lo que produce este aumento de transacciones corresponde a la UART y al *Header Analyzer.* En verde y en rojo se muestran la UART0 y UART1, respectivamente, donde entre el rango de tiempos 325 µs y 330 µs las transacciones que se producen corresponde con la inicialización de las UART a través de la aplicación. El último pico de transacciones de la UART0 corresponde con el mensaje de escritura del patrón que se muestra en la consola. En azul, se muestra las transacciones que recibe el puerto de configuración del *Header Analyzer*, que corresponde con la escritura sobre los registros de este bloque los parámetros de la cabecera que se asigna desde la aplicación.

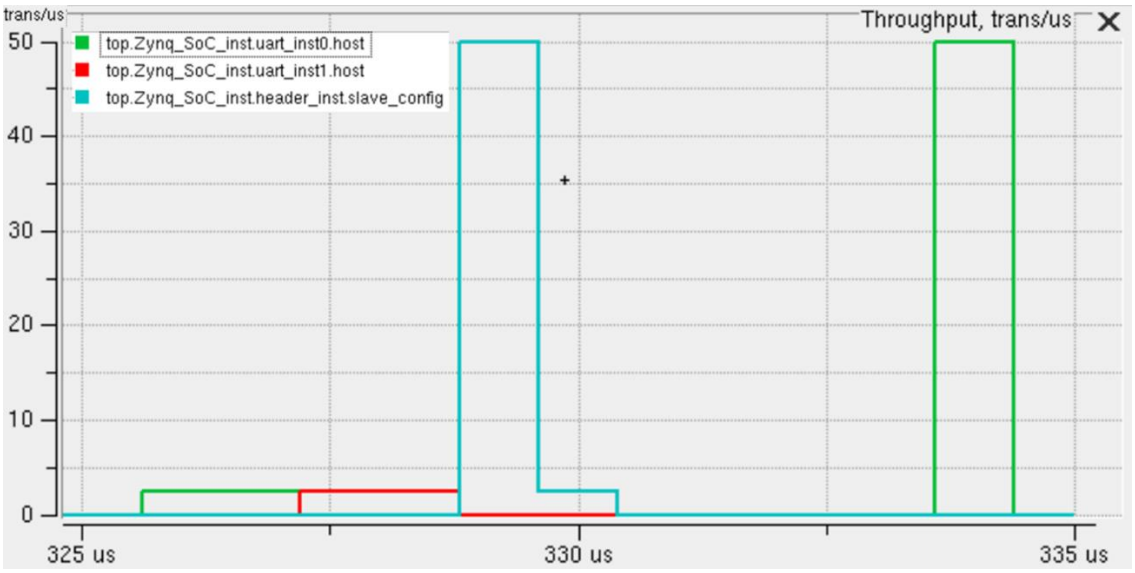



**Figura 108: *Throughput* en la etapa de configuración de la plataforma medido en trans/µs (III)**

### 8.2.2 Análisis de la incorporación del patrón

Avanzando en el tiempo de la simulación se encuentra la incorporación del patrón del *Boyer-Moore* por parte del usuario a través de la UART. Esto ocurre entre los 800 ms y 1,1 segundos. En la Figura 109 se muestra las transacciones que ocurren en este periodo en el puerto *slave_pattern* que recibe el patrón y lo registra. Se cuentan doce transacciones que corresponde con los diez caracteres del patrón "*raquel leon",* el espacio entre las dos palabras y el retorno de carro para finalizar el proceso.

En la Figura 110 se muestra el análisis de la potencia consumida durante esta etapa. El único consumo a gran escala es causado por la CPU, debido a que la recepción del patrón por parte del bloque *Boyer-Moore* no se le añadió ninguna política de consumo de potencia y la UART del modelo de Zynq-7000 proporcionado por Vista no cuenta tampoco con política de potencia.

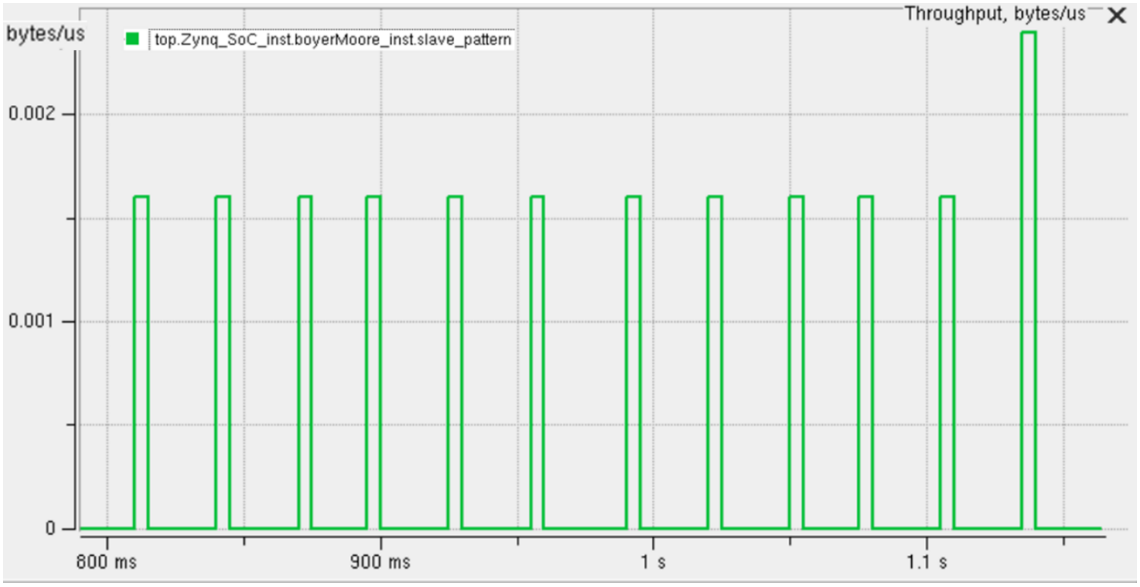



**Figura 109: *Throughput* en la etapa de incorporación del patrón medido en bytes/µs**

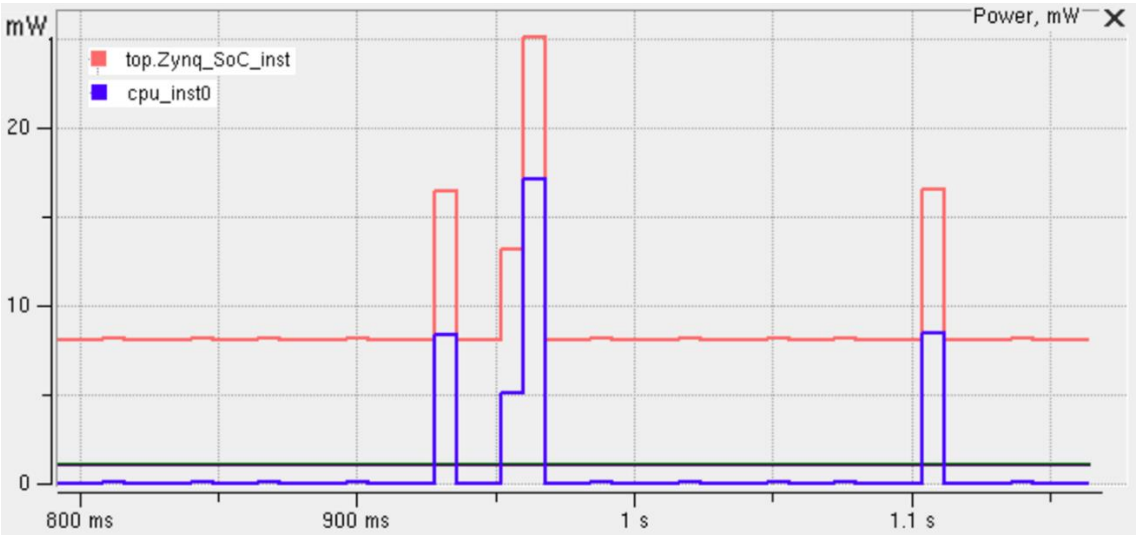



**Figura 110: Análisis de potencia de la etapa de incorporación del patrón**

### 8.2.3 Análisis de la inspección de paquetes

En esta etapa se analiza la parte que compone la arquitectura *hardware* del DPI con los modelos creados en el Capítulo 6: Modelado en TLM-2.0 de la arquitectura DPI.

En la Figura 111 se muestra el análisis de potencia durante el periodo de la inspección de los paquetes por parte del DPI. Se observa que la FIFO es el bloque que más consume y de forma variada, debido a las políticas de potencia que cuenta este modelo. Los modelos creados se mantienen con un consumo de potencia constante de 1 mW.

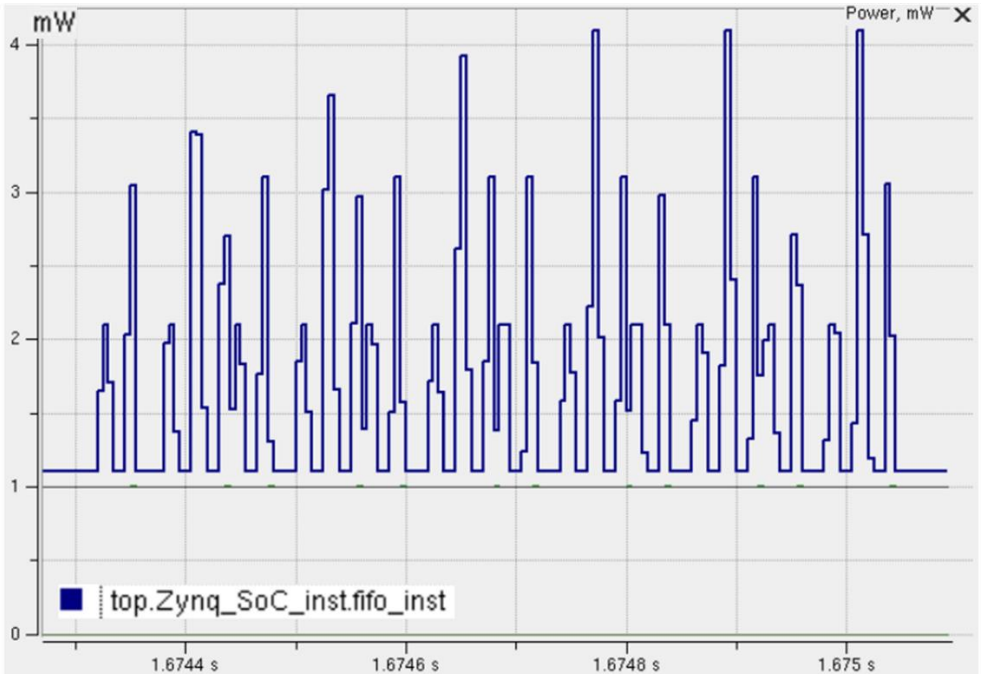



**Figura 111: Análisis de potencia de la etapa de inspección de paquetes**

En la Figura 112 se muestra en la gráfica todas las transacciones producidas por los modelos del DPI que da lugar a la inspección de los paquetes. En la Figura 113 se muestra un resumen de lo que ocurre en este periodo en todos los *sockets.*

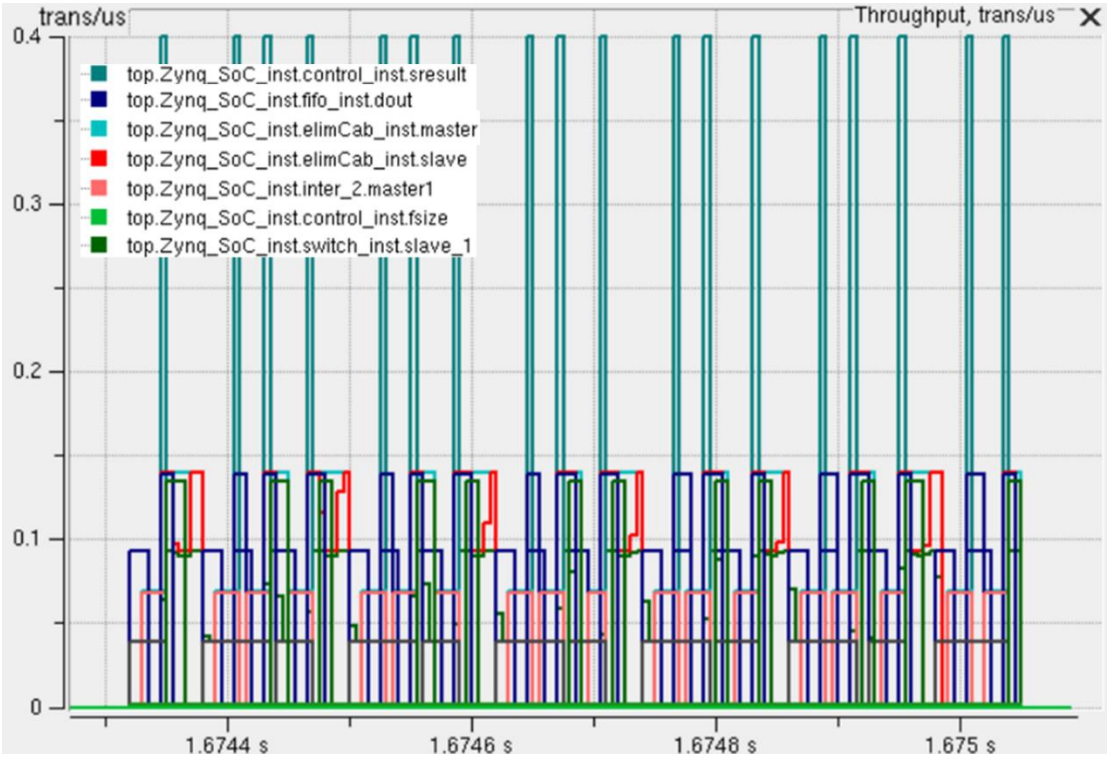
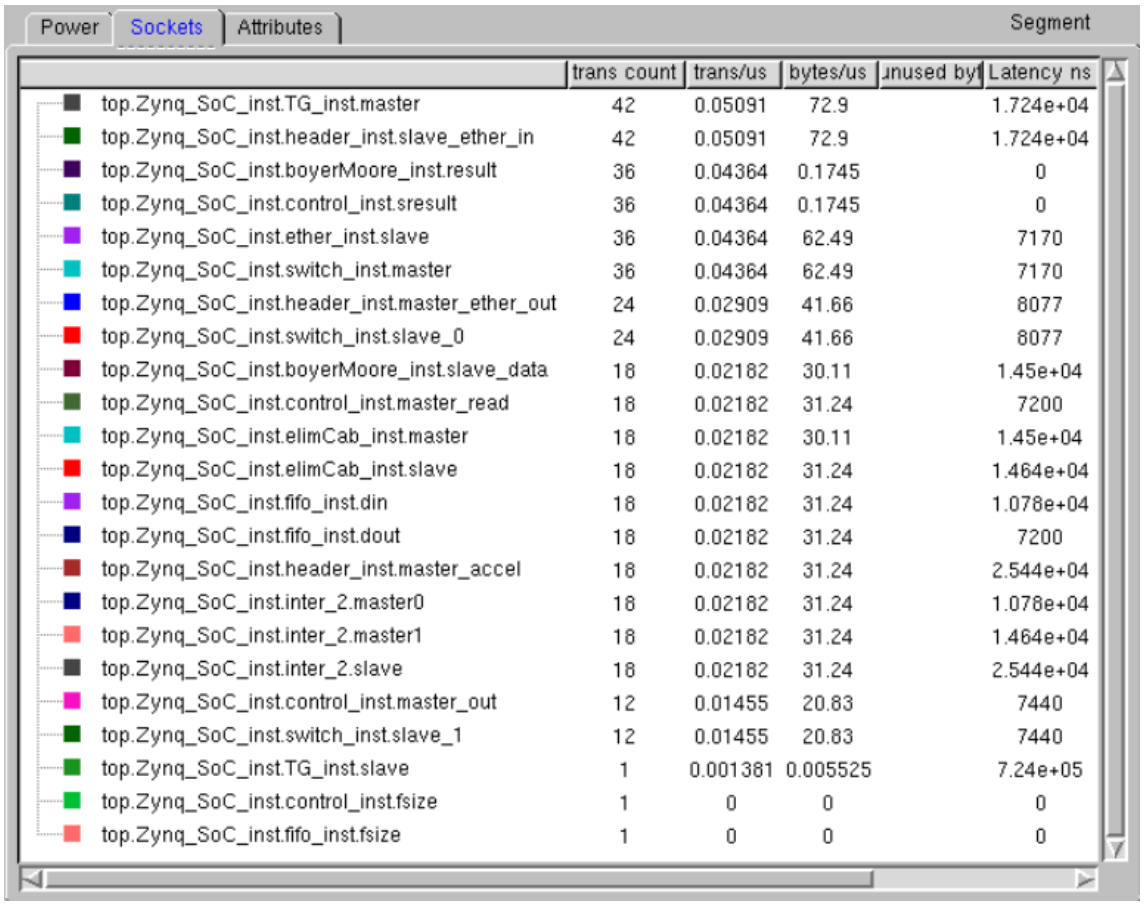



**Figura 112: *Throughput* en la etapa de inspección de los paquetes medido en trans/µs**




| | trans count | trans/us | bytes/us | unused byt | Latency ns |
|---|---|---|---|---|---|
| top.Zynq_SoC_inst.TG_inst.master | 42 | 0.05091 | 72.9 | | 1.724e+04 |
| top.Zynq_SoC_inst.header_inst.slave_ether_in | 42 | 0.05091 | 72.9 | | 1.724e+04 |
| top.Zynq_SoC_inst.boyerMoore_inst.result | 36 | 0.04364 | 0.1745 | | 0 |
| top.Zynq_SoC_inst.control_inst.sresult | 36 | 0.04364 | 0.1745 | | 0 |
| top.Zynq_SoC_inst.ether_inst.slave | 36 | 0.04364 | 62.49 | | 7170 |
| top.Zynq_SoC_inst.switch_inst.master | 36 | 0.04364 | 62.49 | | 7170 |
| top.Zynq_SoC_inst.header_inst.master_ether_out | 24 | 0.02909 | 41.66 | | 8077 |
| top.Zynq_SoC_inst.switch_inst.slave_0 | 24 | 0.02909 | 41.66 | | 8077 |
| top.Zynq_SoC_inst.boyerMoore_inst.slave_data | 18 | 0.02182 | 30.11 | | 1.45e+04 |
| top.Zynq_SoC_inst.control_inst.master_read | 18 | 0.02182 | 31.24 | | 7200 |
| top.Zynq_SoC_inst.elimCab_inst.master | 18 | 0.02182 | 30.11 | | 1.45e+04 |
| top.Zynq_SoC_inst.elimCab_inst.slave | 18 | 0.02182 | 31.24 | | 1.464e+04 |
| top.Zynq_SoC_inst.fifo_inst.din | 18 | 0.02182 | 31.24 | | 1.078e+04 |
| top.Zynq_SoC_inst.fifo_inst.dout | 18 | 0.02182 | 31.24 | | 7200 |
| top.Zynq_SoC_inst.header_inst.master_accel | 18 | 0.02182 | 31.24 | | 2.544e+04 |
| top.Zynq_SoC_inst.inter_2.master0 | 18 | 0.02182 | 31.24 | | 1.078e+04 |
| top.Zynq_SoC_inst.inter_2.master1 | 18 | 0.02182 | 31.24 | | 1.464e+04 |
| top.Zynq_SoC_inst.inter_2.slave | 18 | 0.02182 | 31.24 | | 2.544e+04 |
| top.Zynq_SoC_inst.control_inst.master_out | 12 | 0.01455 | 20.83 | | 7440 |
| top.Zynq_SoC_inst.switch_inst.slave_1 | 12 | 0.01455 | 20.83 | | 7440 |
| top.Zynq_SoC_inst.TG_inst.slave | 1 | 0.001381 | 0.005525 | | 7.24e+05 |
| top.Zynq_SoC_inst.control_inst.fsize | 1 | 0 | 0 | | 0 |
| top.Zynq_SoC_inst.fifo_inst.fsize | 1 | 0 | 0 | | 0 |

**Figura 113: Análisis de las transacciones en la etapa de inspección de los paquetes**

De aquí se puede obtener una monitorización de lo que ocurre en el DPI. Se observa que el *Traffic Generator* ha generado 42 paquetes, de los cuales 24 pasan la inspección de la cabecera sin necesidad de inspeccionar su *payload* y los 18 restante se inspeccionan en profundidad por el *Boyer-Moore.* De estos 18 paquetes, 6 paquetes contienen el patrón *"raquel leon"* en su payload y son desechados, los 12 restantes son transmitidos hacia la red. Por lo tanto, de los 42 paquetes, 36 de ellos han continuado por la red y 6 han sido eliminados de esta.

De manera más detalla se muestra en la Figura 114 las transacciones que recibe el *Header Analyzer* y que, tras inspeccionar su cabecera, son enviadas de nuevo a la red. En la Figura 115 se muestra la cantidad de bytes por cada microsegundo.

En la Figura 116 contiene un resume de las figuras mostradas anteriormente. El *Header Analyzer* recibe 42 transacciones, una media de 0,0506 transacciones/µs a 72,46 bytes/µs, y envía a la red 24 transacciones.

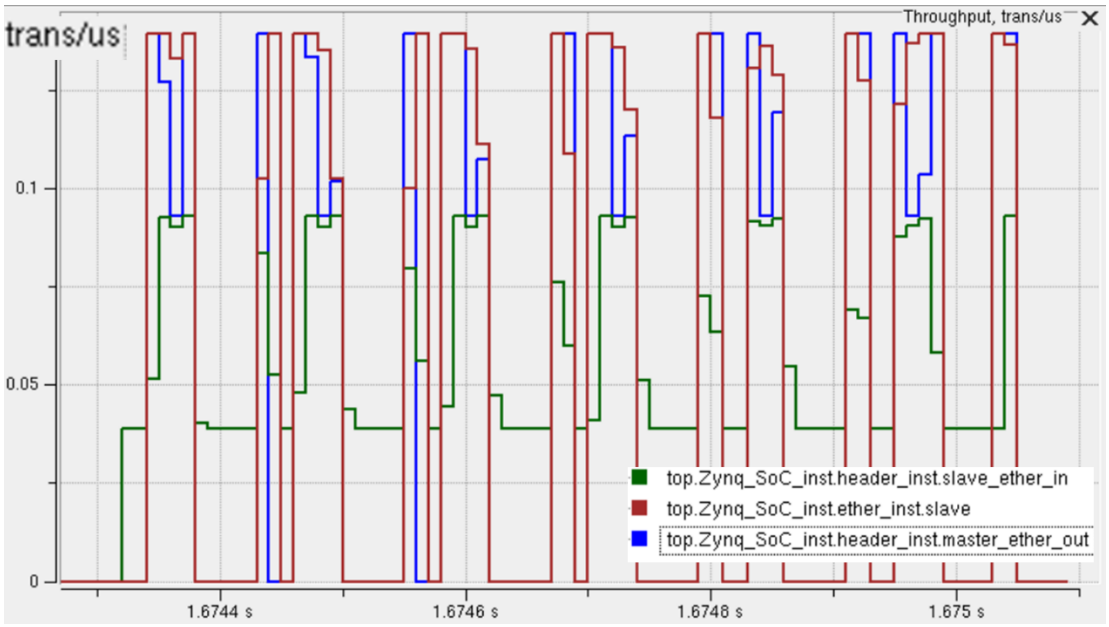



**Figura 114: *Throughput* en el *Header Analyzer* sin coincidencia de cabecera medido en trans/µs**

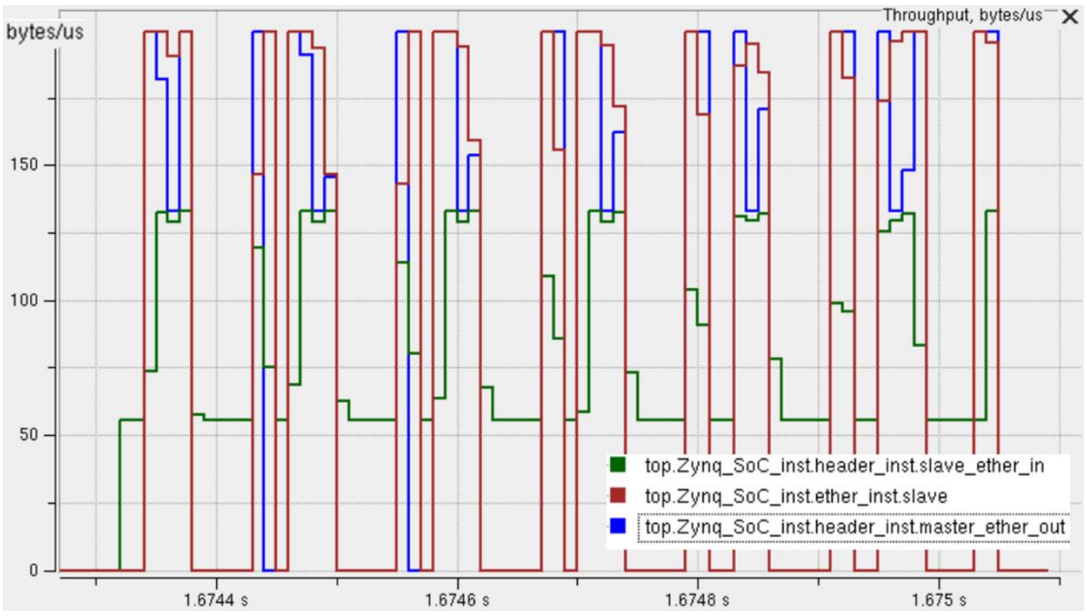



**Figura 115 *: Throughput* en el *Header Analyzer* sin coincidencia de cabecera medido en bytes/µs**

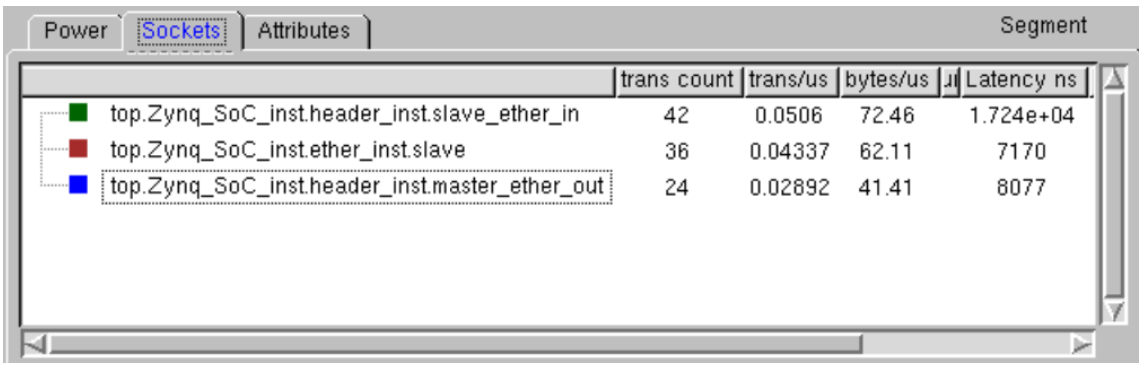



**Figura 116: Análisis de las transacciones en *Header Analyzer* sin coincidencia de cabecera**

En la Figura 117 y Figura 118 se muestran las transacciones/µs y bytes/µs que ocurren en el modelo de *Eliminar Cabecera,* donde en el primero se observa que la cantidad de transacciones no varía a la salida respecto la entrada y en la segunda imagen se observa una disminución de los bytes transmitidos. Esto es debido a la eliminación de la cabecera.

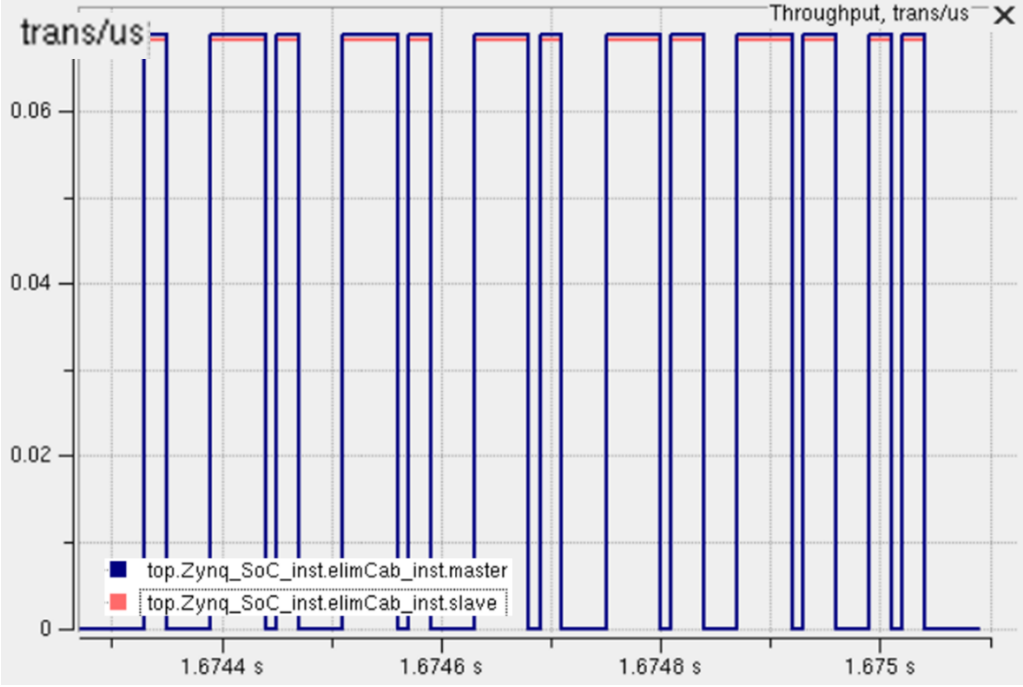



**Figura 117: *Throughput* en *Eliminar Cabecera* medido en trans/µs**

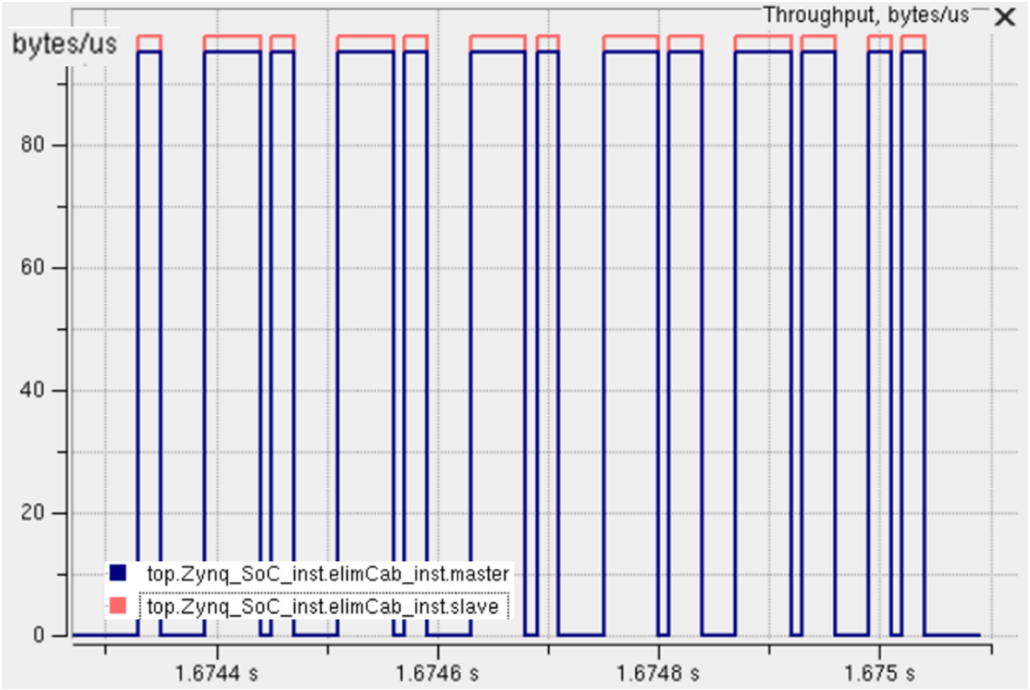



**Figura 118: *Throughput* en *Eliminar Cabecera* medido en bytes/µs**

A modo de resumen de lo comentado anteriormente se muestra en la Figura 119, donde se aprecia esta disminución de la media de los bytes transferidos en cada microsegundo.

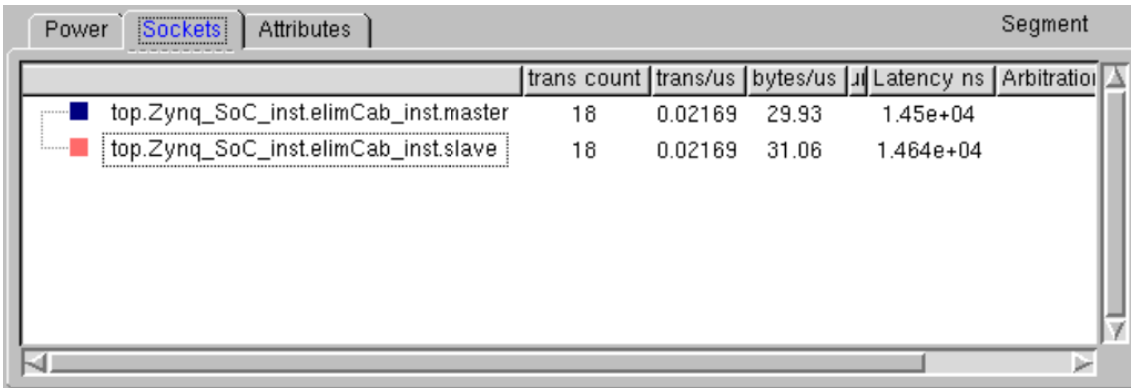



**Figura 119: Análisis de las transacciones en *Eliminar Cabecera***

De igual manera para el modelo del *Boyer-Moore*, se muestra en las Figura 120 y Figura 121 las transacciones/µs y bytes/µ. En verde se muestra el *socket* del resultado del *Boyer-Moore* y se aprecia que realiza más transacciones que datos recibidos en el bloque, esto es debido a que en cada búsqueda de un nuevo *payload,* el resultado se pone a cero, formando una transacción nueva.

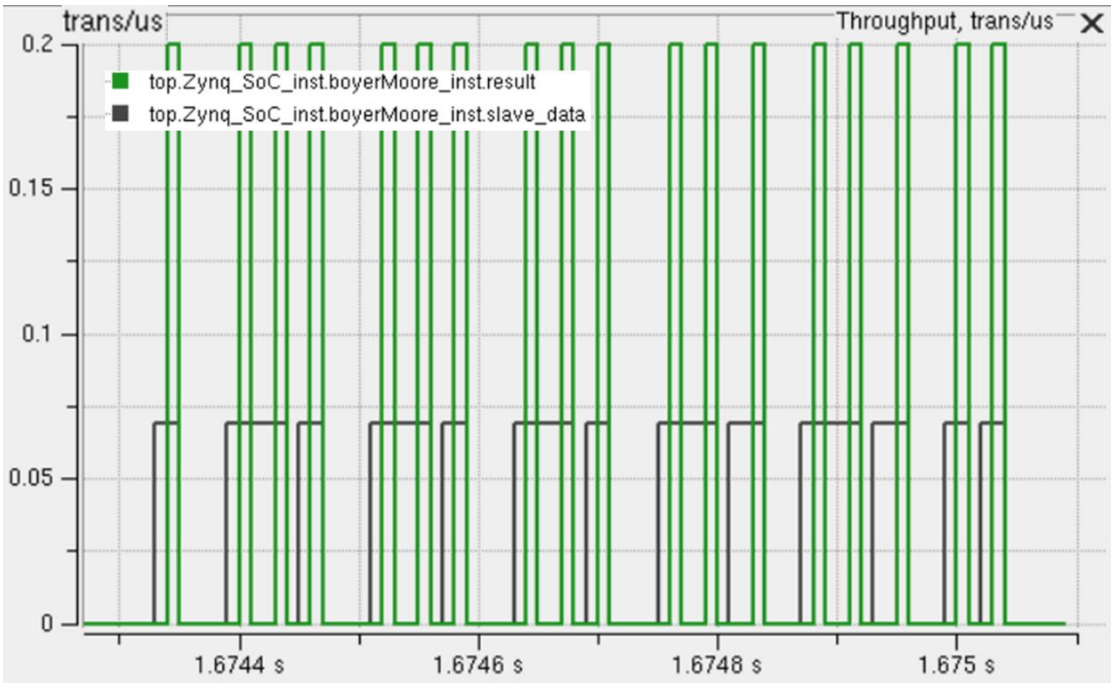



**Figura 120: *Throughput* en el *Boyer-Moore* medido en trans/µs**

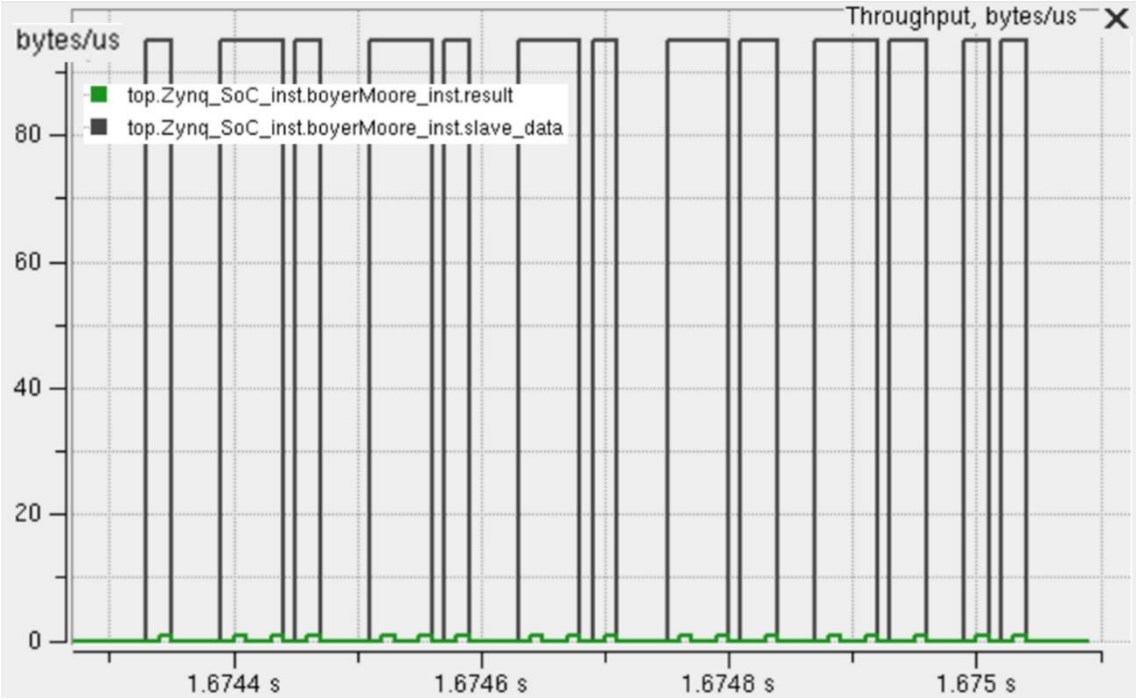



**Figura 121: *Throughput* del bloque *Boyer-Moore* medido en bytes/µs**

Un resumen de lo comentado anteriormente de las dos figuras se muestra en Figura 122, donde se observa que el bloque *Boyer-Moore* recibe 18 paquetes, pero se envían 36 respuestas. Como se comentó anteriormente, en cada análisis de *payload* se envían dos respuestas, antes de que empiece la búsqueda poniendo la señal de resultado a cero y otra del resultado. Además, en resultado tiene un tamaño de 3 bits, por lo que la cantidad media de bytes transmitidos en cada microsegundo es mucho menor que la recibida por el puerto esclavo.

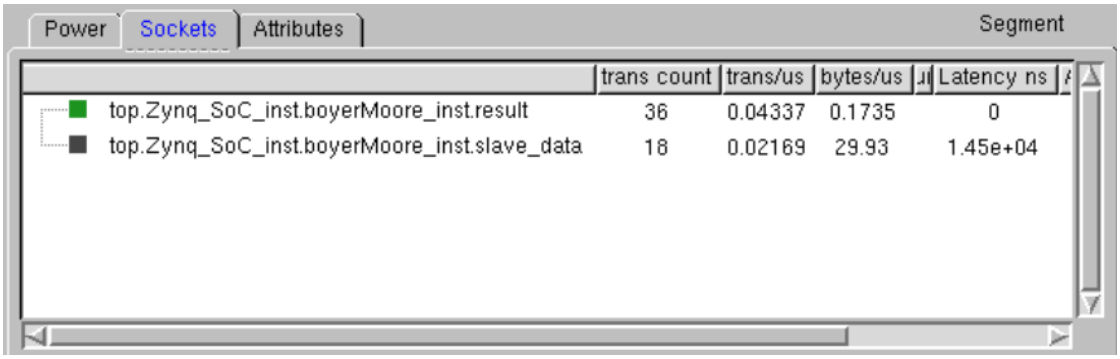



**Figura 122: Análisis de las transacciones en el *Boyer-Moore***

Para el bloque de lectura de la FIFO se muestra en las Figura 123 y Figura 124 los resultados de las transacciones/µs y bytes/µ, donde se muestra la escritura de la FIFO después salir el paquete del *Header Analyzer,* lectura/escritura de la FIFO después de

recibir el resultado del *Boyer-Moore* (rojo) y el envío a la red del paquete almacenado en la FIFO (verde claro).

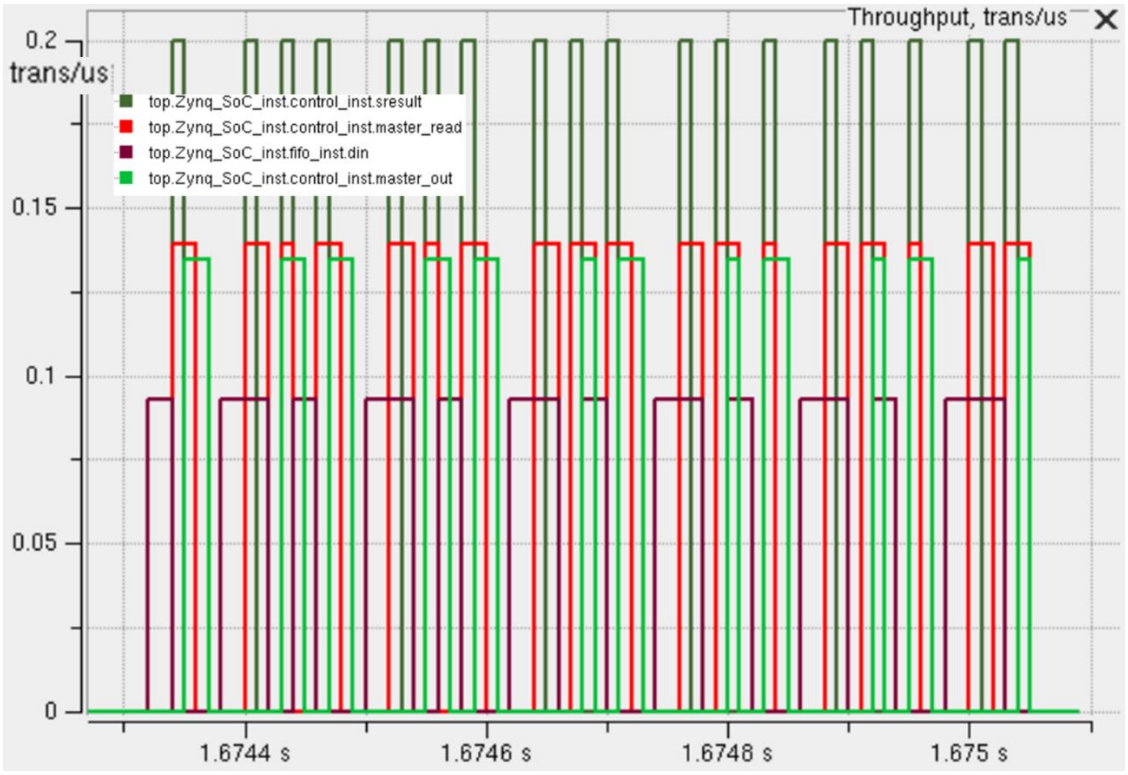



**Figura 123: *Throughput* en la lectura de la FIFO medido en trans/µs**

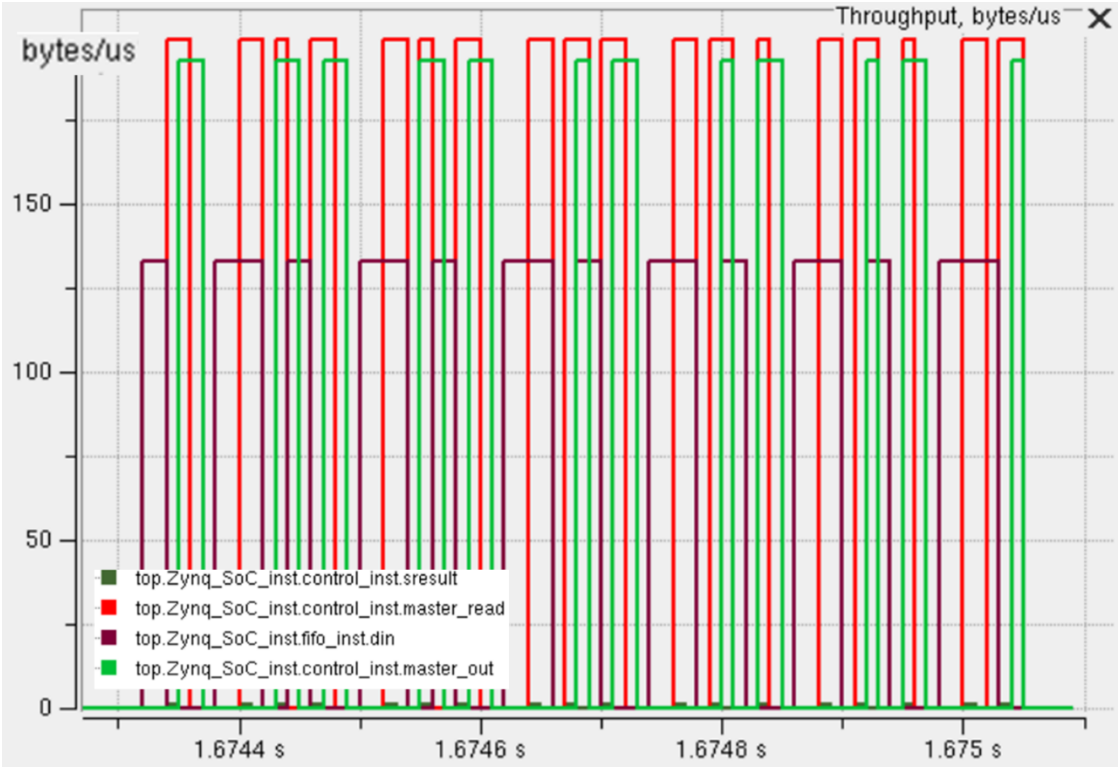



**Figura 124: *Throughput* alcanzado en la lectura de la FIFO medido en bytes/µs**

En la Figura 125 se resumen lo comentado anteriormente y se observa que después de recibir la respuesta del *Boyer-Moore,* se realizan 18 escrituras/lecturas a la FIFO que corresponden con los 18 *payload* que se analizan, pero sólo se transmite a la red 12.

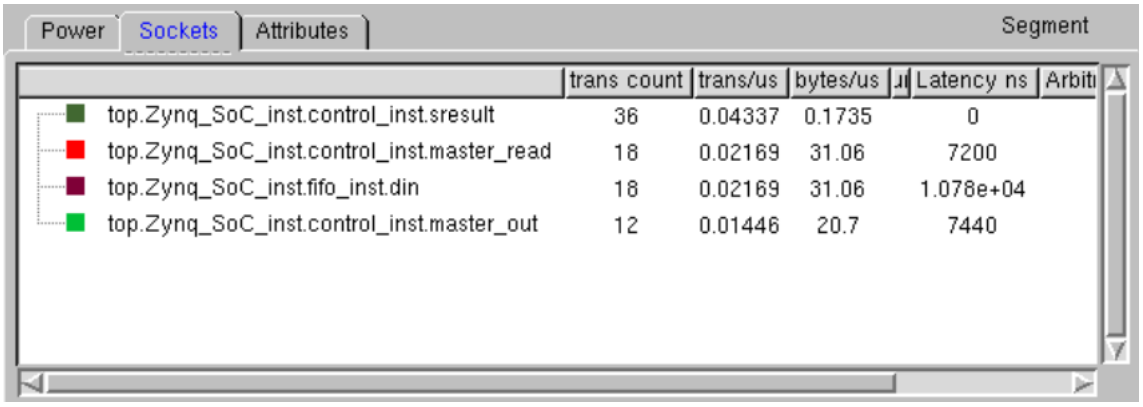



**Figura 125: Análisis de las transacciones en la lectura de la FIFO**

Por último, se muestra en la Figura 126 la latencia que tiene el DPI al analizar 42 paquetes, desde que el primer paquete es enviado por el *Traffic Generator* hasta que el último paquete es recibido por el bloque que emula la salida hacia la red. Esta latencia es de 724 µs.

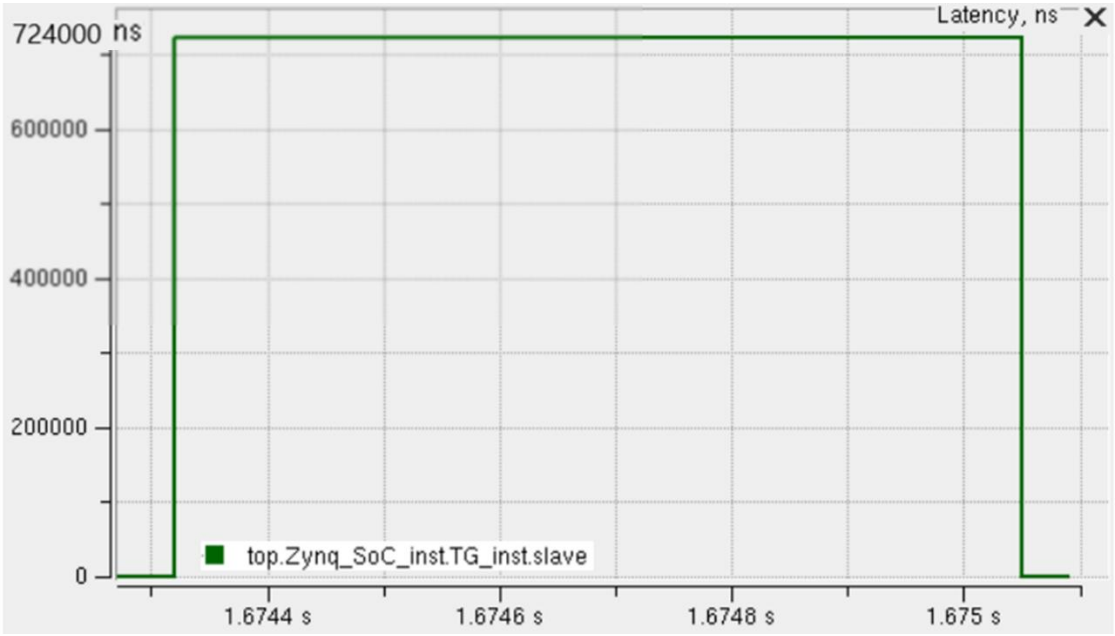



**Figura 126: Latencia del DPI**

### 8.2.4 Análisis de la finalización de la plataforma

La última etapa del análisis corresponde a la finalización de la plataforma virtual, donde es el usuario quién decide cuándo finalizarla a través de la UART, dando lugar al final de la simulación.

En la Figura 127 se muestra las transacciones que dan lugar a la finalización de la simulación, donde en azul se muestra la transacción que corresponde con el mensaje mostrado en la UART que indica que se puede finalizar la plataforma al pulsar la tecla *Enter* y en violeta se muestra la transacción que corresponde a esa tecla pulsada por el usuario.

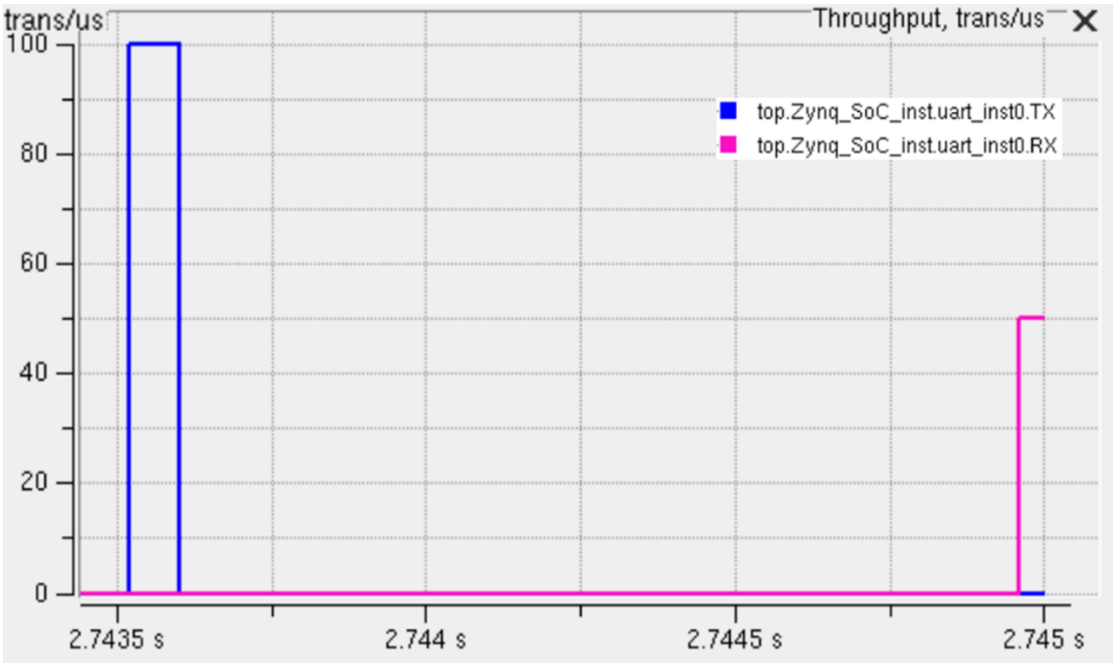



**Figura 127: *Throughput* en la etapa de finalización de la plataforma medido en trans/ µs**

En la Figura 128 se muestra la potencia consumida en esta etapa donde la mayor parte es debido a la utilización de la CPU. En la gráfica no se muestra la potencia de la UART o la consola debido a que en el modelo de la plataforma de Zynq-7000 no se definen.

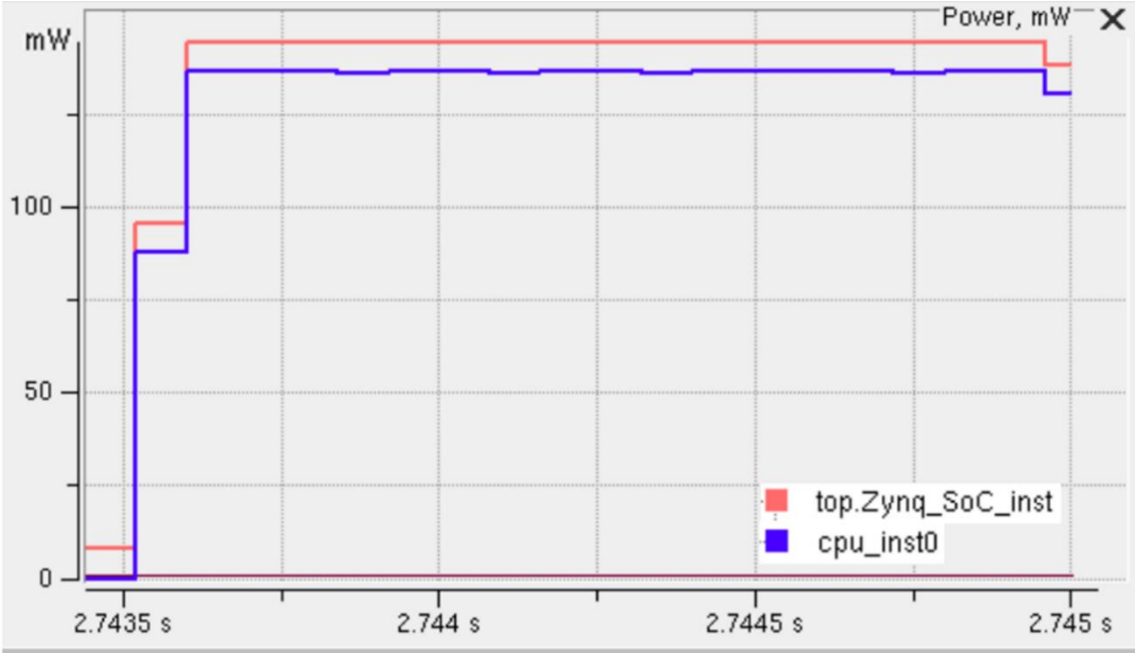



**Figura 128: Análisis de potencia de la etapa de finalización de la plataforma**

## 8.3 Conclusiones

Gracias a la creación de la plataforma virtual se ha podido realizar la exploración de la arquitectura completa del DPI, tanto de la parte *software* como del hardware. Se ha podido observar la potencia que se consume en cada instante de la simulación y ver las transacciones que ocurren en cualquier instante. Por último, se obtiene el tiempo que tarda el DPI en analizar una cierta cantidad de paquetes.

# Capítulo 9. Conclusiones y trabajos futuros

En este capítulo final se presentan las conclusiones a las que se ha llegado tras la realización de este Trabajo Fin de Máster y los trabajos futuros que tiene el mismo.

## 9.1 Conclusiones del proyecto

La motivación de este Trabajo de Fin de Máster es la creación de una plataforma virtual de un sistema Inspección Profunda de Paquetes (DPI) con objeto de realizar una exploración de la arquitectura.

Para la creación de esta plataforma se ha usado la herramienta Vista de Mentor Graphics que permite modelar arquitecturas a nivel de transacciones (TLM) y crear una plataforma virtual con la que se puede desarrollar el *software* del sistema. Además, permite el depurado y análisis del *hardware* modelado, y a través de la plataforma virtual, depurar el *software* y realizar una integración *hardware/software.*

Los bloques de la arquitectura del DPI han sido modelados en alto nivel, permitiendo un mayor nivel de abstracción que otros lenguajes y simplificando el modelado. Este modelado se ha realizado a nivel de transacciones (TLM-2.0), haciendo uso del estándar IEEE SystemC TLM-2.0, donde la funcionalidad del bloque está separada de las interfaces de comunicación, facilitando su modelado y el refinamiento de la arquitectura en cuanto a sus bloques funcionales y a la arquitectura de comunicaciones.

La herramienta Vista ofrece una arquitectura completa del dispositivo Zynq-7000 modelada a nivel de transacciones. Esto permite conectar la arquitectura DPI, modelada también en TLM-2.0, a la plataforma Zynq y crear una plataforma virtual completa para realizar análisis arquitecturales. Con ello se consigue realizar una exploración al detalle de la arquitectura, conociendo el consumo de potencia en cada instante y teniendo constancia de las transacciones que se realiza en la plataforma. Los resultados obtenidos permiten evolucionar la plataforma para ajustarla a diferentes velocidades de transferencia de datos fijando objetivos de diseño tanto para bloques funcionales como para la arquitectura de buses.

Durante el proceso de creación del modelo se han evaluado diferentes alternativas para la generación del modelo virtual, ya sea integrando QEMU con TLM-2.0 mediante puertos dedicados y otras opciones descritas en este documento. La opción elegida finalmente permite que el diseñador utilice los mecanismos descritos en el estándar IEEE 1666-2011 SystemC TLM-2.0 para crear su modelo sin necesidad de incluir mecanismos externos de comunicación entre el modelo del procesador y su simulador de juego de instrucciones (ISS). El diseñador crea sus modelos utilizando SystemC TLM-2.0 en las capas PV y T. El sistema permite parametrizar los modelos, como por ejemplo retardos, ciclos de acceso a memoria o potencia sin modificar los modelos funcionales.

Igualmente, la presencia ayudas gráficas que facilitan el depurado de las transacciones y de funciones de captura y de presentación de datos aportan a la metodología de trabajo la integración necesaria para la realización del análisis citado.

Por último, aunque no ha sido explotado en este trabajo, es posible la conexión con herramientas de síntesis de alto nivel (Mentor Catapult) para facilitar la implementación del sistema.

## 9.2 Trabajos futuros

En base a la experiencia conseguida con este proyecto, se proponen diversos trabajos futuros para seguir realizando la exploración de la arquitectura:

1. El desarrollo de un sistema con la complejidad implícita de este tipo de plataformas requiere que sea abordado por partes. En este proyecto se ha realizado una plataforma básica, y será necesario abordar la integración de más de un bloque de *Boyer-Moore* para conseguir tráficos MultiGigabit*,* por lo que se plantea la integración de un *Motor de Búsqueda* que disponga de varios bloques *Boyer-Moore* mejorando el rendimiento del DPI.
2. Para hacer análisis completos de las prestaciones de la plataforma se debe partir de tráficos de red más realistas. Se propone obtener el tráfico desde la red haciendo uso de los bloques *Gigabit Ethernet Control* de la Zynq. Por tanto, se plantea integrar la plataforma virtual con un generador de tráfico que soporte perfiles cercanos a los existentes en los *routers*, para distintas aplicaciones (web, email, etc.)
3. Las políticas de estimación de potencia utilizadas en este trabajo deben ser ajustadas para tener un modelo más preciso en cuanto a su análisis. Será necesario contrastar los modelos globales para ajustarlos a valores más realistas para cada bloque una vez se ha realizado su síntesis de alto nivel.
4. Para continuar con el flujo del modelo hacia la implementación es necesario explorar los procedimientos de conexión con las herramientas de implementación, especialmente con las herramientas de síntesis de alto nivel.

# Referencias


[1] G. E. Moore, «Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.», *IEEE Solid-State Circuits Soc. Newsl.*, vol. 11, n.º 3, pp. 33-35, 2006 [Online]. Disponible en: https://doi.org/10.1109/N-SSC.2006.4785860

[2] J. Zhu y N. Dutt, «Chapter 5 - Electronic system-level design and high-level synthesis», en *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, y K.-T. (Tim) Cheng, Eds. Boston: Morgan Kaufmann, 2009, pp. 235-297 [Online]. Disponible en: www.sciencedirect.com/science/article/pii/B9780123743640500126

[3] A. John, «The Transaction Level Modeling standard of the Open SystemC Initiative (OSCI)». Doulos, 2009 [Online]. Disponible en: https://pdfs.semanticscholar.org/presentation/40b7/9bb4e911fb5ae7de66ec2bff206e65d1d5d2.pdf

[4] A. B. Mehta, «ESL (Electronic System Level) Verification Methodology», en *ASIC/SoC Functional Design Verification*, Cham: Springer International Publishing, 2018, pp. 221-241 [Online]. Disponible en: http://link.springer.com/10.1007/978-3-319-59418-7_11. [Accedido: 21-feb-2018]

[5] S. Khandelwal, «World's largest 1 Tbps DDoS Attack launched from 152,000 hacked Smart Devices», *The Hacker News*, 2016. [Online]. Disponible en: http://thehackernews.com/2016/09/ddos-attack-iot.html. [Accedido: 03-feb-2018]

[6] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, y M. Rajarajan, «A survey of intrusion detection techniques in Cloud», *J. Netw. Comput. Appl.*, vol. 36, n.º 1, pp. 42-57, 2013 [Online]. Disponible en: http://dx.doi.org/10.1016/j.jnca.2012.05.003

[7] Open Information Security Foundation (OISF), «Suricata | Open Source IDS / IPS / NSM engine». .

[8] «The Bro Network Security Monitor», 2014. [Online]. Disponible en: https://www.bro.org/index.html. [Accedido: 02-feb-2018]

[9] M. Mueller, «DPI technology from the standpoint of Internet governance studies.», Nueva York, 2011 [Online]. Disponible en: http://dpi.ischool.syr.edu/Technology_files/WhatisDPI-2.pdf. [Accedido: 12-nov-2017]

[10] Y.-S. Lin, C.-L. Lee, y Y.-C. Chen, «Length-Bounded Hybrid CPU/GPU Pattern Matching Algorithm for Deep Packet Inspection», *Algorithms*, vol. 10, n.º 1, p. 16, ene. 2017 [Online]. Disponible en: http://www.mdpi.com/1999-4893/10/1/16. [Accedido: 15-mar-2018]

[11] C. Xu, S. Chen, J. Su, S. M. Yiu, y L. C. K. Hui, «A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms», *IEEE Commun. Surv. Tutorials*, vol. 18, n.º 4, pp. 2991-3029, 2016 [Online]. Disponible en: http://ieeexplore.ieee.org/document/7468531/. [Accedido: 16-feb-2018]

[12] J. Spahr, «Análisis de Expresiones Regulares Usando DFA/NFA. Aplicación a la Inspección de Paquetes de Datos en FPGA», Universidad de Las Palmas de Gran Canaria, Las Palmas de Gran Canaria, 2016.

[13] R. S. Boyer y J. S. Moore, «A Fast String Searching Algorithm», *Commun. ACM*, vol. 20, n.º 10, pp. 762-772, 1977 [Online]. Disponible en: http://doi.acm.org/10.1145/359842.359859

[14] S. Wu, U. Manber, y others, «A fast algorithm for multi-pattern searching», University of Arizona. Department of Computer Science, Tucson, AZ (USA), may 1994 [Online]. Disponible en: http://webglimpse.net/pubs/TR94-17.pdf

[15] U. Trivedi y M. Patel, «A fully automated deep packet inspection verification system with machine learning», en *2016 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, 2016, pp. 1-6 [Online]. Disponible en: https://doi.org/10.1109/ANTS.2016.7947802

[16] A. Domínguez, P. P. Carballo, y A. Núñez, «Programmable SoC platform for Deep Packet Inspection using enhanced Boyer-Moore algorithm», en *Proceedings of 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2017)*, 2017, pp. 1-8 [Online]. Disponible en: https://doi.org/10.1109/ReCoSoC.2017.8016159

[17] B. Vega, «Diseño e Implementación mediante Síntesis de Alto Nivel de un IP para el filtrado y clasificación de paquetes TCP/IP», Universidad de Las Palmas de Gran Canaria, Las Palmas de Gran Canaria, 2016 [Online]. Disponible en: https://en.iuma.ulpgc.es/tfm_tfg/documentos/2015/TFM_BenjaminVegadelPino_Memoria.pdf

[18] S. León, «Optimización de una Plataforma de Inspección Profunda de Paquetes basada en SoC FPGA para Gigabit Ethernet», Universidad de Las Palmas de Gran Canaria, 2017.

[19] M. Jung, «Introduction Virtual Platforms Motivation», *Microelectron. Syst. Des. Res. Gr.*, 2012 [Online]. Disponible en: https://ems.eit.uni-kl.de/fileadmin/ems/downloads/Introduction_Virtual_Platforms.pdf. [Accedido: 05-feb-2018]

[20] Synopsys, «Virtualizer - VDK». 2016 [Online]. Disponible en: https://www.synopsys.com/cgi-bin/proto/pdfdla/docsdl/virtualizer_ds.pdf?file=virtualizer_ds.pdf. [Accedido: 06-feb-2018]

[21] M. Montón, J. Carrabina, y M. Burton, «Mixed simulation kernels for high performance virtual platforms», en *2009 Forum on Specification & Design Languages (FDL)*, 2009, pp. 1-6 [Online]. Disponible en: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5404061&isnumber=5404035. [Accedido: 06-feb-2018]

[22] IEEE, «IEEE Standard for Standard SystemC Language Reference Manual», *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* -. pp. 1-638, 2012 [Online]. Disponible en: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6134619&isnumber=6134618. [Accedido: 04-feb-2018]

[23] B. Bailey, F. Balarin, M. McNamara, G. Mosenson, M. Stellfox, y Y. Watanabe, *TLM-Driven Design and Verification Methodology*. Cadence Design Systems, 2010.

[24] Accellera Systems Initiative, «SystemC». [Online]. Disponible en: http://www.accellera.org/downloads/standards/systemc. [Accedido: 17-jul-2018]

[25] Cadence Design Systems, «SystemC Language Fundamentals». 2011.

[26] OSCI, «OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL». 2009 [Online]. Disponible en: http://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf. [Accedido: 06-feb-2018]

[27] M. Monton, «Checkpointing for virtual platforms and systemC-TLM», Universitat Autònoma de Barcelona, 2010.

[28] F. Ghenassia y A. Clouard, «TLM: An Overview and Brief History», en *Transaction Level Modeling with SystemC*, F. Ghenassia, Ed. Berlin/Heidelberg: Springer-Verlag, 2005, pp. 1-22 [Online]. Disponible en: http://link.springer.com/10.1007/0-387-26233-4_1. [Accedido: 27-abr-2018]

[29] A. Fasching, «Virtual Prototyping», 2016. [Online]. Disponible en: https://ti.tuwien.ac.at/ecs/teaching/courses/hwswcode_vu/hwsw-codesign-student-presentations/3-virtual-prototyping.pdf. [Accedido: 20-abr-2018]

[30] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. USA: Synopsys Press, 2014.

[31] A. Lonardi y G. Pravadelli, «On the Co-simulation of SystemC with QEMU and OVP Virtual Platforms», en *VLSI-SoC: Internet of Things Foundations*, 2015, pp. 110-128 [Online]. Disponible en: https://link.springer.com/chapter/10.1007/978-3-319-

25279-7_7

[32] T.-C. Yeh, G.-F. Tseng, y M.-C. Chiang, «A fast cycle-accurate instruction set simulator based on QEMU and SystemC for SoC development», en *Melecon 2010 - 2010 15th IEEE Mediterranean Electrotechnical Conference*, 2010, pp. 1033-1038 [Online]. Disponible en: http://ieeexplore.ieee.org/document/5475901/. [Accedido: 03-feb-2018]

[33] M. A. Qayum, «Design of a Mips Instruction Set Simulator for Multicore Processor Research in Systemc», Oklahoma State University, 2010 [Online]. Disponible en: https://shareok.org/handle/11244/10263

[34] C. Helmstetter y V. Joloboff, «SimSoC: A SystemC TLM integrated ISS for full system simulation», en *APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems*, 2008, pp. 1759-1762 [Online]. Disponible en: http://ieeexplore.ieee.org/document/4746381/. [Accedido: 07-jun-2018]

[35] Cadence, «Cadence Virtual System Platform for the Xilinx Zynq-7000 All Programmable SoC». 2012.

[36] J. Engblom, «Creating Virtual Platforms with Wind River Simics». 2011 [Online]. Disponible en: http://events.windriver.com/wrcd01/wrcm/2016/10/wr_creating-virtual-platforms_wp-1.pdf. [Accedido: 06-feb-2018]

[37] Imperas, «Open Virtual Platforms™ Benefits of Virtual Platforms for Embedded Software Development». 2011 [Online]. Disponible en: http://www.europractice.stfc.ac.uk/vendors/OVP_Overview_Datasheet.pdf. [Accedido: 06-feb-2018]

[38] S. Matalon, «Vista Virtual Prototyping», 2015. [Online]. Disponible en: http://s3.mentor.com/public_documents/whitepaper/resources/mentorpaper_77272.pdf. [Accedido: 15-may-2018]

[39] Bochs, «bochs: The Open Source IA-32 Emulator». [Online]. Disponible en: http://bochs.sourceforge.net/. [Accedido: 06-feb-2018]

[40] F. Bellard, «QEMU, a Fast and Portable Dynamic Translator», en *FREENIX Track: 2005*

*USENIX Annual Technical Conference*, 2005, pp. 41-46 [Online]. Disponible en: http://static.usenix.org/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf. [Accedido: 06-feb-2018]

[41] M. Monton, A. Portero, M. Moreno, B. Martinez, y J. Carrabina, «Mixed SW/SystemC SoC Emulation Framework», en *2007 IEEE International Symposium on Industrial Electronics*, 2007, pp. 2338-2341 [Online]. Disponible en: http://ieeexplore.ieee.org/document/4374971/. [Accedido: 06-feb-2018]

[42] C.-S. Peng, L.-C. Chang, C.-H. Kuo, y B.-D. Liu, «Dual-core virtual platform with QEMU and SystemC», en *2010 International Symposium on Next Generation Electronics*, 2010, pp. 69-72 [Online]. Disponible en: http://ieeexplore.ieee.org/document/5669196/. [Accedido: 03-feb-2018]

[43] T.-C. Yeh, Z.-Y. Lin, y M.-C. Chiang, «Enabling TLM-2.0 interface on QEMU and SystemC-based virtual platform», en *2011 IEEE International Conference on IC Design & Technology*, 2011, pp. 1-4 [Online]. Disponible en: http://ieeexplore.ieee.org/document/5783207/. [Accedido: 11-feb-2018]

[44] C. Sauer, H.-M. Bluethgen, y H.-P. Loeb, «Distributed, loosely-synchronized systemC/TLM simulations of many-processor platforms», en *Proceedings of the 2014 Forum on Specification and Design Languages (FDL)*, 2014, pp. 1-8 [Online]. Disponible en: http://ieeexplore.ieee.org/document/7119360/. [Accedido: 03-feb-2018]

[45] G. Delbergue, M. Burton, F. Konrad, B. Le Gal, y C. Jego, «QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0», ene. 2016 [Online]. Disponible en: https://hal.archives-ouvertes.fr/hal-01292317. [Accedido: 08-jun-2018]

[46] M. Carey, «HW/SW Performance Analysis using Virtual Prototyping Technologies», 2014.

[47] Mentor Graphics, «Vista™ User's Manual». Mentor Graphics Corporation, 2017.

[48] Mentor Graphics, «Vista Architect - System Level Design Solution for Performance and Power». Mentor Graphics Corporation, 2009 [Online]. Disponible en:

http://s3.mentor.com/public_documents/datasheet/esl/vista/vista-architect-ds.pdf

[49] Mentor Graphics, «Chapter 7: Creating TLM Models», en *Vista ™ User's Manual*, Mentor Graphics Corporation, 2017, pp. 97-224.

[50] Mentor Graphics, «Vista™ Modeling Guide». Mentor Graphics Corporation, 2017.

[51] Mentor Graphics, «Chapter 8: The Generic Models and Models Catalogue Libraries», en *Vista ™ User's Manual*, Mentor Graphics Corporation, 2017, pp. 225-446.

[52] Mentor Graphics, «Chapter 9: Vista Design Assembly», en *Vista ™ User's Manual*, Mentor Graphics Corporation, 2017, pp. 447-542.

[53] Mentor Graphics, «Chapter 10: Compiling and Building a Project», en *Vista ™ User's Manual*, Mentor Graphics Corporation, 2017, pp. 543-553.

[54] Mentor Graphics, «Chapter 12: Simulation and Verification», en *Vista ™ User's Manual*, Mentor Graphics Corporation, 2017, pp. 557-638.

[55] Mentor Graphics, «Chapter 14: Analysis Graphs and Reports», en *Vista ™ User's Manual*, Mentor Graphics Corporation, 2017, pp. 645-674.

[56] Mentor Graphics, «Chapter 15: Virtual Prototyping», en *Vista ™ User's Manual*, Mentor Graphics Corporation, 2017, pp. 675-692.

[57] Xilinx, «Zynq-7000 EPP», 2011 [Online]. Disponible en: http://www.uio.no/studier/emner/matnat/ifi/INF3430/h11/undervisningsmateriale/roarsk/Xilinx_ZYNQ_Product_Brief.pdf

[58] Xilinx Inc., «Zynq-7000 All Programmable SoC - Technical Reference Manual». pp. 1-1863, 2018 [Online]. Disponible en: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. [Accedido: 30-jun-2018]

[59] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, y R. W. Stewart, *The Zynq Book*, 1st Editio. Glasgow: Strathclyde Academic Media, 2014 [Online]. Disponible en: http://www.zynqbook.com/. [Accedido: 21-may-2018]

[60] Xilinx Inc., «Zynq-7000 All Programmable SoC Data Sheet: Overview», 2018 [Online]. Disponible en: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. [Accedido: 07-jul-2018]

[61] Mentor Graphics, *Model Description for DDRMC*. Mentor Graphics Corporation, 2017.

[62] Mentor Graphics, «Xilinx Zynq-7000 Virtual Platform». Mentor Graphics Corporation, pp. 1-28, 2015.

[63] N. de la Cruz, «Plataforma para inspección profunda de paquetes sobre Zynq UltraScale+ MPSoC», Universidad de Las Palmas de Gran Canaria, 2018.