



UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Carrera de Ingeniero Informático

**SISTEMA BASADO EN FPGA  
PARA LA SIMULACIÓN DINÁMICA DE  
CIRCUITOS ELÉCTRICOS LINEALES**

VICTORINA FERNÁNDEZ GONZÁLEZ

Dirigido por: Dr. ALFONSO URQUÍA MORALEDA

Curso: 2014/2015





## SISTEMA BASADO EN FPGA PARA LA SIMULACIÓN DINÁMICA DE CIRCUITOS ELÉCTRICOS LINEALES

Proyecto de Fin de Carrera de modalidad *oferta específica* (tipo B)

Realizado por: VICTORINA FERNÁNDEZ GONZÁLEZ (firma)

Dirigido por: Dr. ALFONSO URQUÍA MORALERA (firma)

Tribunal calificador:

Presidente: D./D<sup>a</sup>. .....  
(firma)

Secretario: D./D<sup>a</sup>. .....  
(firma)

Vocal: D./D<sup>a</sup>. .....  
(firma)

Fecha de lectura y defensa: .....

Calificación: .....



Dedicado a mi pequeña Elena, por prestarme su tiempo

*Un diseñador sabe que ha alcanzado la perfección no cuando ya no tiene nada más que  
añadir, sino cuando ya no le queda nada más que quitar*

Antoine de Saint-Exupery



## RESUMEN

En este proyecto fin de carrera se presenta el desarrollo de un sistema basado en FPGA para obtener simulaciones transitorias de circuitos eléctricos RLC. La arquitectura del sistema describe con VHDL tanto el método numérico para calcular los resultados de la simulación como el controlador Ethernet para intercambiar información con un PC. El método numérico elegido es el algoritmo de Runge-Kutta de cuarto orden, caracterizado por efectuar los cálculos con una precisión muy alta pero con la desventaja de requerir un gran número de operaciones.

Para el desarrollo de la arquitectura se sigue una metodología de diseño hardware con los procesos de especificación de requisitos, diseño, realización física, verificación y validación. Con la herramienta de desarrollo ISE de Xilinx se editan y analizan las descripciones VHDL, se verifican lanzando testbenches con el simulador ISim, se sintetiza y se implementa el diseño y se carga la configuración sobre una FPGA XC6SLX45 de Xilinx impresa en una tarjeta Atlys de Digilent.

En paralelo se diseña con la herramienta de modelado, procesado y simulación Scilab una aplicación de usuario que incluye una interfaz gráfica. Con esta aplicación se genera, empleando el Análisis Nodal Modificado, el modelo de estado del circuito a partir del esquema gráfico trazado con el editor Xcos. Después se pueden ejecutar simulaciones del modelo basadas en PC o bien activar simulaciones basadas en FPGA enviando por red Ethernet el modelo del circuito a la FPGA, que procesa el algoritmo de simulación y devuelve los resultados al PC. Por último, la interfaz gráfica representa los resultados de las diferentes simulaciones junto con los tiempos de procesado empleados.

La simulación dinámica obtenida a partir de la FPGA se valida comparándola con las basadas en PC. El cálculo de los tiempos de procesado de la simulación con ambos métodos demuestra la mejora de la eficiencia cuando se emplea el sistema basado en FPGA pero teniendo en cuenta que su uso está limitado a circuitos de orden menor o igual a 16.





# **PALABRAS CLAVE**

FPGA

VHDL

Simulación de sistemas lineales

Ecuaciones diferenciales

Matrices de estado

Circuitos RLC

Runge-Kutta

Scilab

Xcos

Xilinx



# SUMMARY

Title: *A FPGA-based system for the dynamic simulation of linear electric circuits*

This final year project presents the development of a FPGA-based system for transient simulations of RLC circuits. The system architecture describes with VHDL the numerical method to solve the simulation and the Ethernet driver to exchange data with a PC. The numerical method selected is the algorithm of 4<sup>th</sup>-order Runge-Kutta, characterized by very high precision, but with the disadvantage of requiring a large number of operations.

For the development of the architecture a hardware design methodology is followed with processes as requirements specification, design, implementation, verification and validation. Using the Xilinx ISE development tool, the VHDL descriptions are edited, analyzed and verified by launching testbenches with the ISim simulator, the design is synthesized and implemented and the configuration is loaded on the Xilinx XC6SLX45 FPGA in a Digilent Atlys board.

Simultaneously, a graphical user interface is designed with the modeling, processing and simulation tool Scilab. With this application, it is generated, by means of the Modified Nodal Analysis, the state-space of the circuit from its scheme traced with Xcos. Afterwards it's possible to run PC-based simulations or activate FPGA-based simulations sending through Ethernet network the model of the circuit to the FPGA, device that processes the algorithm simulation and returns the results to the PC. Finally, the graphics representation of the solution with the simulation processing times is displayed by the user interface.

The dynamic simulation obtained by the FPGA is validated comparing with the calculated simulation by Scilab. The simulation processing times show the improvement in efficiency when using the FPGA-based system in front of the PC-based system but taking into account that its use is limited to circuits with the order less than or equal to 16.



# KEYWORDS

FPGA

VHDL

Simulation of linear systems

Differential equations

State-space matrices

RLC circuits

Runge-Kutta

Scilab

Xcos

Xilinx



# ÍNDICE DE CONTENIDOS

<b>INDICE DE FIGURAS .....</b>	<b>15</b>
<b>INDICE DE TABLAS.....</b>	<b>19</b>
<b>1. INTRODUCCIÓN, OBJETIVOS Y ESTRUCTURA .....</b>	<b>21</b>
1.1. Introducción .....	21
1.2. Objetivos.....	25
1.3. Estructura .....	26
<b>2. METODOLOGÍA Y HERRAMIENTAS.....</b>	<b>29</b>
2.1. Introducción .....	29
2.2. Metodología .....	29
2.3. Herramientas .....	30
2.4. Conclusiones .....	31
<b>3. ESPECIFICACIÓN DE REQUISITOS .....</b>	<b>33</b>
3.1. Introducción .....	33
3.2. Funcionalidad general de la arquitectura .....	33
3.3. Flujo de datos de entrada y salida de la FPGA.....	34
3.4. Algoritmo de resolución numérica .....	36
3.5. Tamaño del problema.....	39
3.6. Representación de los datos.....	40
3.7. Controlador de comunicación Ethernet.....	40
3.8. Almacenado de datos de entrada y salida .....	41
3.9. Activación del procesado de la simulación.....	42
3.10. Señales de reloj y reseteado .....	42
3.11. Puertos de entrada y salida.....	42
3.12. Conclusiones .....	43
<b>4. DISEÑO CONCEPTUAL.....</b>	<b>45</b>
4.1. Introducción .....	45

4.2. Arquitectura del sistema .....	45
4.3. Controlador Ethernet .....	46
4.4. Memorias .....	49
4.5. Algoritmo RK4.....	50
4.6. Conclusiones .....	54
<b>5. DISEÑO DETALLADO .....</b>	<b>55</b>
5.1. Introducción .....	55
5.2. Descripciones VHDL .....	55
5.3. Memorias.....	75
5.4. Cores .....	79
5.5. Relojes.....	83
5.6. Conclusiones .....	88
<b>6. REALIZACIÓN FÍSICA .....</b>	<b>89</b>
6.1. Introducción .....	89
6.2. Instalación del entorno de desarrollo de Xilinx.....	89
6.3. Dispositivo y tarjeta de evaluación .....	90
6.4. Construcción de un proyecto.....	91
6.5. Fuentes VHDL .....	92
6.6. Definición de restricciones.....	94
6.7. Síntesis .....	95
6.8. Implementación .....	96
6.9. Configuración de la FPGA.....	98
6.9.1. Programación en modo JTAG.....	98
6.9.2. Configuración en modo ROM.....	99
6.10. Conclusiones .....	100
<b>7. DISEÑO DE LA INTERFAZ DE USUARIO .....</b>	<b>101</b>
7.1. Introducción .....	101
7.2. Uso de la aplicación .....	102
7.2.1. Instalación .....	102



7.2.2. Definición gráfica del modelo .....	102
7.2.3. Inicio de la aplicación .....	104
7.2.4. Carga del circuito.....	104
7.2.5. Obtención de las ecuaciones de estado .....	105
7.2.6. Configuración de la simulación .....	110
7.2.7. Conexión del PC con la FPGA.....	111
7.2.8. Lanzamiento y representación de la simulación .....	113
7.3. Diseño y codificación de la aplicación.....	114
7.3.1. Función principal .....	115
7.3.2. Función para cargar el circuito .....	117
7.3.3. Función de chequeo de comunicación.....	121
7.3.4. Función para cargar el modelo.....	121
7.3.5. Función para procesar las simulaciones.....	121
7.3.6. Función embebida para intercambio de datos con la FPGA .....	124
7.4. Verificación de la construcción del modelo .....	125
7.5. Conclusiones .....	126
<b>8. VERIFICACIÓN Y VALIDACIÓN.....</b>	<b>127</b>
8.1. Introducción .....	127
8.2. Verificación del algoritmo .....	127
8.2.1. Procedimiento.....	127
8.2.2. Chequeo .....	134
8.3. Verificación del controlador Ethernet .....	135
8.3.1. Procedimiento.....	135
8.3.2. Chequeo.....	138
8.4. Validación del sistema.....	139
8.4.1. Procedimiento.....	140
8.4.2. Chequeo .....	140
8.5. Conclusiones .....	140
<b>9. ANÁLISIS DE RESULTADOS.....</b>	<b>143</b>

9.1. Introducción .....	143
9.2. Crecimiento de la complejidad y del coste de la metodología .....	143
9.3. Tiempos de procesado de simulación .....	144
9.4. Conclusiones .....	148
<b>10.PLANIFICACIÓN Y COSTES DEL PROYECTO .....</b>	<b>149</b>
10.1. Introducción .....	149
10.2. Planificación.....	149
10.3. Costes del proyecto.....	152
10.4. Conclusiones .....	152
<b>11.CONCLUSIONES Y TRABAJOS FUTUROS .....</b>	<b>153</b>
11.1. Introducción .....	153
11.2. Conclusiones .....	153
11.3. Trabajos futuros.....	154
<b>BIBLIOGRAFÍA Y REFERENCIAS .....</b>	<b>157</b>
<b>SIGLAS, ABREVIATURAS Y ACRÓNIMOS .....</b>	<b>161</b>

## INDICE DE FIGURAS

Figura 1.1. Estructura básica de una FPGA .....	22
Figura 1.2. Estructura de carpetas del proyecto en el CD .....	27
Figura 2.1. Procesos de la metodología basada en VHDL.....	30
Figura 3.1. Funcionalidad general del sistema .....	33
Figura 3.2. Campo de datos de un datagrama UDP .....	41
Figura 4.1. Arquitectura del sistema.....	45
Figura 4.2. Campos de trama Ethernet.....	46
Figura 4.3. Controlador Ethernet.....	47
Figura 4.4. Generación del campo CRC .....	48
Figura 4.5. Descripción de las operaciones del algoritmo .....	50
Figura 5.1. Ventana de configuración del <i>logiCORE Floating-point Operator</i> .....	80
Figura 5.2. Definición de la precisión del <i>logiCORE Floating-point Operator</i> .....	80
Figura 5.3. Simulación de un sumador con latencia 7 .....	81
Figura 5.4. Configuración de la latencia del sumador <i>logiCORE Floating-point</i> .....	81
Figura 5.5. Control de la latencia de los <i>logiCORE Floating-point Operators</i> en pipeline .....	82
Figura 5.6. Recursos empleados del multiplicador <i>logiCORE Floating-point</i> .....	82
Figura 5.7. Estimación de los recursos en el <i>logiCORE Floating-point Operator</i> .....	83
Figura 5.8. Sincronizador de un pulso asíncrono.....	86
Figura 5.9. Cambio de dominio de reloj de la señal <i>start_algorithm</i> .....	86
Figura 5.10. Cambio de dominio de reloj de la señal <i>ready_algorithm</i> .....	87
Figura 6.1. Entorno de desarrollo de Xilinx .....	89
Figura 6.2. Tarjeta Atlys de Digilent .....	90
Figura 6.3. Creación del proyecto .....	91
Figura 6.4. Propiedades del proyecto .....	92
Figura 6.5. Ventana de creación de fuentes .....	92
Figura 6.6. Selección de cores.....	93

Figura 6.7. Jerarquía de componentes .....	93
Figura 6.8. Librerías .....	94
Figura 6.9. Ventana de procesos para la ejecución de la síntesis.....	96
Figura 6.10. Procesos para la implementación.....	97
Figura 6.11. ISE iMPACT para la programación de la FPGA en modo JTAG.....	98
Figura 6.12. Software <i>Adept</i> para la programación de la FPGA en modo ROM .....	99
Figura 7.1. Interfaz de usuario para la simulación de los circuitos .....	101
Figura 7.2. Trazado de un circuito con Xcos.....	103
Figura 7.3. Explorador de paletas de Xcos .....	103
Figura 7.4. Ventana de configuración de una resistencia con Xcos .....	104
Figura 7.5. Ventana de selección de un circuito .....	105
Figura 7.6. Nombre del circuito en la barra de estado.....	105
Figura 7.7. Configuración y activación de las simulaciones .....	110
Figura 7.8. Ventana de configuración de la conexión de área local .....	112
Figura 7.9. Ventana de estado de conexión de área local.....	113
Figura 7.10. Chequeo de la comunicación con la FPGA .....	113
Figura 7.11. Representación gráfica de las simulaciones seleccionadas.....	114
Figura 7.12. Estructura de ficheros de la aplicación <i>Simulaciones RK4</i> .....	114
Figura 7.13. Simulación de circuitos con Qucs .....	126
Figura 8.1. Jerarquía de ficheros para el testbench <i>algorithm_tb</i> .....	128
Figura 8.2. Circuito RL de orden 3 .....	129
Figura 8.3. Valores de configuración extraídos por consola.....	130
Figura 8.4. Ventana de procesos para el testbench .....	130
Figura 8.5. Propiedades de la simulación .....	131
Figura 8.6. Propiedades de la simulación .....	131
Figura 8.7. Carga del script desde la consola de ISim.....	131
Figura 8.8. Fichero del script <i>algo.tcl</i> .....	132
Figura 8.9. Ventana del simulador ISim .....	132
Figura 8.10. Simulación del circuito RL de orden 3 con RK4 definido con código Scilab .....	134

Figura 8.11. Simulación tipo <i>Algoritmo RK4 FPGA</i> .....	135
Figura 8.12. Ventana del analizador de paquetes Ethernet Wireshark .....	136
Figura 8.13. Jerarquía de ficheros para el testbench <i>rk4_tb</i> .....	136
Figura 8.14. Opciones de captura con Wireshark.....	138
Figura 8.15. Integración del sistema final .....	139
Figura 8.16. Comparación de las simulaciones del circuito RL de orden 3 .....	140
Figura 10.1. Diagrama de Gantt de progreso de tareas .....	151



## INDICE DE TABLAS

Tabla 4.1. Sumador que obtiene <i>res1</i> en función del orden .....	52
Tabla 4.2. Salida de multiplexor en función de la etapa para <i>res2</i> .....	52
Tabla 4.3. Salida de multiplexor en función de la etapa para <i>res3</i> .....	53
Tabla 5.1. Características de las memorias empleadas en la arquitectura .....	75
Tabla 6.1. Características de la FPGA adquirida.....	90
Tabla 6.2. Recursos empleados.....	97
Tabla 7.1. Configuración de la red .....	112
Tabla 8.1. Comportamiento de los leds .....	138
Tabla 9.1. Tiempos de procesado de las simulaciones .....	145
Tabla 9.2. Tiempos de procesado del algoritmo con FPGA .....	146





# 1. INTRODUCCIÓN, OBJETIVOS Y ESTRUCTURA

## 1.1. Introducción

Antes de trazar los objetivos de este proyecto y presentar su estructura se explican de forma resumida los conceptos con los que se va a trabajar y se expone la justificación de la elección del tema abordado.

### ***Modelado y simulación de sistemas***

El comportamiento de un sistema físico se puede aproximar por medio de un modelo matemático compuesto por un conjunto de entradas, salidas y variables de estado relacionadas por ecuaciones diferenciales lineales de primer orden (Wikipedia, 2015). Una forma de obtener este modelo en el caso de circuitos eléctricos tipo RLC es empleando el Análisis Nodal Modificado que parte de las leyes de Kirchhoff (Hanke, 2006).

Para obtener una simulación dinámica a partir del modelo matemático del sistema se requiere la ejecución de un algoritmo que genere la respuesta transitoria. El algoritmo Runge-Kutta de cuarto orden obtiene resultados muy próximos al comportamiento real del sistema pero exige la realización de una cantidad elevada de operaciones (Nizzo, 2009).

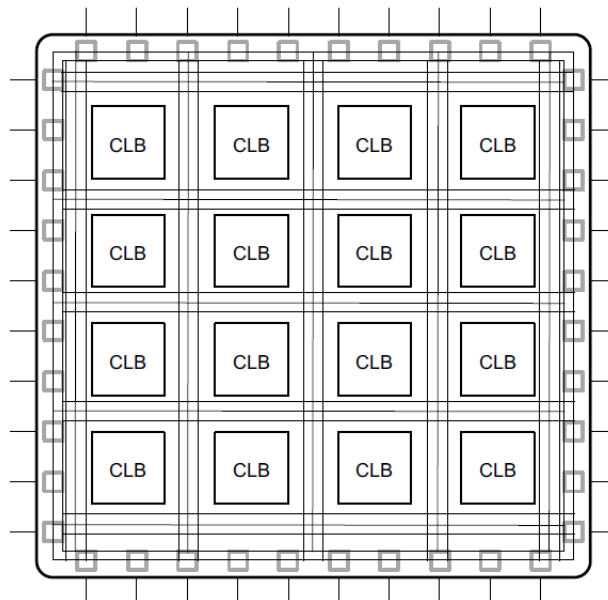
### **Scilab**

Scilab es una herramienta para cálculo numérico y simulación sobre PC de uso libre, con la que se pueden modelar sistemas físicos, ejecutar de forma eficiente código matemático desarrollado con un lenguaje de alto nivel y representar resultados numéricos en gráficas fácilmente configurables. Además incluye herramientas como Xcos que permite la edición gráfica de circuitos y las librerías JIMS con las que se puede embeber código Java en el propio

código Scilab. La herramienta también dispone de funciones para la creación de interfaces gráficas de usuario.

### **Lógica programable y FPGAs**

Una FPGA es un dispositivo formado por un gran número de pequeños bloques lógicos configurables (CLBs o Configurable Logic Blocks) interconectados entre sí y capaces de reproducir operaciones lógicas y aritméticas de una forma optimizada (Xilinx, 2015b).



**Figura 1.1.** Estructura básica de una FPGA

En la estructura de la FPGA destacan los siguientes elementos (Xilinx, 2014):

- Look-up Tables (LUTs): bloques que ejecutan operaciones lógicas.
- Flip-flops (FFs): elementos que registran el resultado de las operaciones realizadas por las LUTs.
- Hilos o buses que conectan los distintos elementos.
- Pads de entrada, salida o ambos (I/O) que son los puertos que intercambian información con el exterior.

- Bloques DSP o unidades aritmético-lógicas embebidas.
- Bloques de memoria.

Esta arquitectura de la lógica programable hace posible el paralelizado de operaciones empleando frecuencias superiores a 100 MHz por lo que favorece el desarrollo de algoritmos que requieren la ejecución de muchas operaciones en un tiempo limitado.

### **VHDL**

VHDL (Very High Speed Integrated Circuit Hardware Description Language) es un lenguaje estandarizado por el IEEE (*Institute of Electrical and Electronics Engineers*) en la norma ANSI/IEEE 1076-1993 y que permite describir el comportamiento de un circuito digital (Floyd, 2006: 250).

Una de las formas que tiene VHDL de describir un circuito es siguiendo la Lógica de Transferencia de Registros (RTL o Register-Transfer Logic). Considerando que un circuito digital síncrono consiste en registros implementados con flip-flops (FFs) y lógica combinacional formada por puertas lógicas, una descripción RTL significa que en cada paso el circuito debe realizar una transferencia de datos entre registros y evaluar un conjunto de condiciones expresada con la lógica combinacional para pasar al siguiente paso. La descripción RTL es comprendida por los sintetizadores de VHDL que la convierten en otra descripción a nivel de puertas lógicas (Terés, 1998:182-184).

En la sintaxis de VHDL destacan elementos (Floyd, 2006: 250) como la entidad, que describe los puertos de entrada y salida del circuito lógico, la arquitectura, que describe la operación interna del circuito, el componente, una forma de predefinir un circuito que puede instanciarse repetidas veces, y la señal, la forma de especificar una conexión entre componentes (Floyd, 2006: 296). Dentro del cuerpo de la arquitectura se definen sentencias concurrentes y dentro del cuerpo de los procesos sentencias secuenciales (Wikilibros, 2015).

### ***Metodología basada en VHDL***

Fabricantes como Xilinx distribuyen distintas familias de FPGAs y ofrecen potentes herramientas de diseño que facilitan las tareas de descripción hardware, síntesis, verificación, implementación y configuración.

Para el desarrollo de un sistema sobre FPGA se establece una metodología basada en VHDL en la que son esenciales la especificación de requisitos, los diseños conceptual y detallado, la realización física, la verificación con generación de estímulos y la validación (Terés, 1998: 355-359).

Es interesante el uso de Intellectual Properties (IP) o cores, bloques ya construidos con una funcionalidad perfectamente definida, proporcionados por los fabricantes de FPGAs y configurables con las herramientas de diseño. Estos cores permiten al desarrollador centrarse en el propio diseño ganando en tiempo y eficiencia (Xilinx, 2015a).

### ***Elección del tema del proyecto***

El tema abordado en este proyecto surge del interés personal y profesional por el empleo de lógica programable para el diseño de sistemas digitales. También se pretende trabajar en un desarrollo que requiera un cierto grado de análisis matemático y que se relacione con herramientas de simulación PC y diseño de interfaces de usuario orientado a objetos. La concreción del tema parte del trabajo descrito en el artículo (Chen, 2009). Aunque este trabajo es similar al propuesto en este proyecto existen ciertas diferencias destacadas a continuación.

- El tipo de sistema físico a modelar y simular es distinto.
- Se emplean otros métodos y herramientas de validación (Matlab/Simulink vs Scilab/Xcos).
- Cambia la forma de extraer los datos de la simulación de la FPGA hacia el PC.

- En este proyecto se añade la posibilidad de poder cargar en la FPGA distintos modelos configurables desde una aplicación de usuario.

## 1.2. Objetivos

Este proyecto plantea los siguientes objetivos.

- Definir el modelo matemático de un circuito eléctrico RLC con corriente continua y optar por el método numérico idóneo para calcular la respuesta transitoria de la señal de tensión en uno de sus nodos.
- Evaluar la aplicabilidad de la metodología basada en VHDL para la obtención de la simulación dinámica del modelo matemático del circuito RLC. Debe demostrarse que son factibles la descripción correcta y eficiente del algoritmo, su síntesis, implementación y el cumplimiento de los requisitos temporales y lógicos exigidos por la arquitectura de la FPGA.

Para ello se planteará una especificación de requisitos, se realizarán el diseño de la arquitectura y su descripción con VHDL, se elegirán la tecnología de lógica programable y la tarjeta de evaluación con la FPGA adecuada, se verificarán las descripciones por medio de testbenches, se implantará físicamente el sistema hardware y se validará.

- Crear un entorno de usuario desde el que se pueda extraer el modelo matemático de un circuito a partir de su representación gráfica. Una vez configurada la FPGA, que exista la posibilidad de cargar el modelo en la FPGA y que los resultados de la simulación dinámica puedan ser adecuadamente transferidos al PC y representados en una interfaz gráfica.

Es necesario previamente elegir y adaptarse a una herramienta de modelado y simulación que incluya la posibilidad de diseñar interfaces gráficas de usuario y codificar a alto nivel.

También se debe definir una forma eficiente de comunicación entre el PC y la FPGA y preparar los controladores tanto en la aplicación de usuario como en la arquitectura de la FPGA para hacer posible el intercambio de información.

- Comprobar la mejora de la velocidad de procesado de la simulación dinámica del modelo sobre la FPGA en comparación con otros sistemas de simulación basados en PC.

Para ello, con la herramienta de modelado, se codificará el algoritmo de simulación elegido para obtener simulaciones basadas en PC. Posteriormente se lanzarán los dos tipos de simulación, basada en PC y basada en FPGA, se calcularán los respectivos tiempos de procesado y se representarán gráficamente para el posterior análisis de la diferencia entre ambos. Se espera obtener tiempos de simulación para el sistema basado en FPGA como mínimo inferiores en un orden de magnitud.

- Evaluar el crecimiento de la complejidad y coste de la metodología a medida que aumenta la dimensión del circuito. Este objetivo implica analizar cómo influye en el desarrollo del sistema el aumento de la complejidad del sistema modelado. Se deben conocer las dimensiones a partir de las cuales el desarrollo ya no es posible debido a limitaciones, como por ejemplo, la falta de disponibilidad de elementos lógicos en la FPGA.

### **1.3. Estructura**

#### ***Memoria***

Una vez entendidos los conceptos a emplear y trazados los objetivos, en el siguiente capítulo se van a describir las herramientas usadas y la metodología a seguir. Posteriormente a lo largo de varios capítulos se explican las diferentes etapas del desarrollo del sistema: especificación de requisitos, diseños conceptual y detallado, realización física, verificación y validación. Se intercala un capítulo para explicar la funcionalidad y diseño de la interfaz de usuario. Se

continúa con el análisis de resultados, con la planificación y costes del proyecto y con las conclusiones y trabajos futuros. Se finaliza con un apartado de bibliografía y referencias y otro para siglas y abreviaturas.

## CD

En el CD adjunto a esta memoria se proporciona la documentación, el código y ficheros generados en el desarrollo del sistema así como los esquemas de los circuitos empleados para el testeo, todos ellos ordenados en la estructura de carpetas de la Figura 1.2.

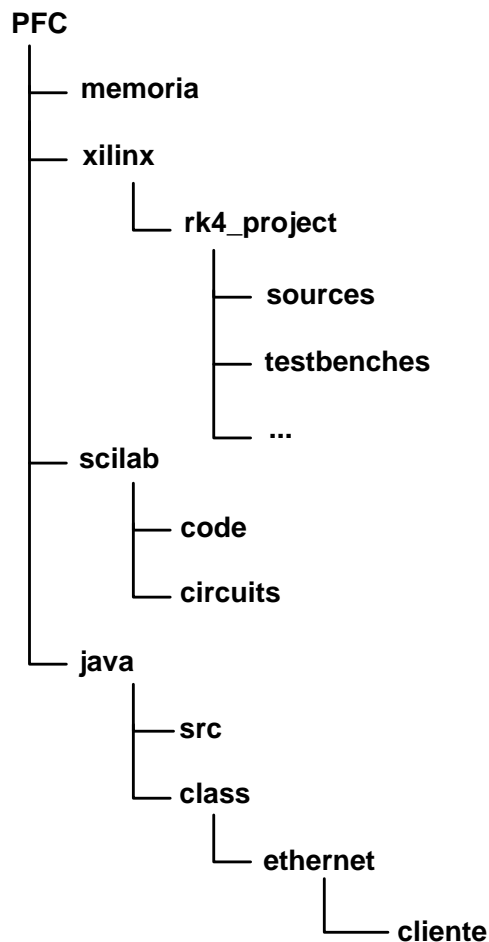


Figura 1.2. Estructura de carpetas del proyecto en el CD

En la carpeta **memoria** aparece este documento en formato pdf.

En la carpeta **xilinx** se encuentra el proyecto *rk4\_project* generado con la herramienta ISE de Xilinx y que contiene todos los ficheros (fuentes con las descripciones VHDL, testbenches, constraints, scripts, binarios, etc.) creados a lo largo de los procesos de síntesis, testado e implementación.

En la carpeta **scilab** se encuentra la subcarpeta *code* con el conjunto de ficheros .sce y .sci que permiten lanzar y ejecutar la aplicación de usuario en Scilab. La subcarpeta *circuits* almacena ficheros .zcos con la representación gráfica de diferentes circuitos RLC.

En la carpeta **java** está salvado el código Java y el código intermedio .class para embeber el control de envío y recepción de tramas Ethernet en Scilab.



## 2. METODOLOGÍA Y HERRAMIENTAS

### 2.1. Introducción

Una vez trazados los objetivos es fundamental establecer qué metodología se va a seguir para el desarrollo del sistema y qué herramientas se requieren. En los siguientes apartados se tratan estos puntos.

### 2.2. Metodología

Para el desarrollo del simulador de circuitos RLC sobre una FPGA se sigue una metodología basada en VHDL en el que se llevan a cabo los procesos de especificación de requisitos, diseño conceptual, diseño detallado, realización física, verificación y validación (Terés, 1998: 355-394).

En esta metodología el diseño es tipo “top-down” (Terés, 1998: 351) lo que significa que el sistema que se desea desarrollar se va subdividiendo en módulos más sencillos con funcionalidad independiente. Estos módulos forman una jerarquía constituida por diferentes niveles siendo los niveles inferiores especializaciones del nivel superior. Esta forma de diseñar lleva a que la descripción del hardware y su verificación sean más eficientes.

En la Figura 2.1 pueden verse los diferentes procesos seguidos por la metodología. Aunque el diseño detallado, la realización física y la verificación se presenten de forma secuencial, realmente son procesos que se solapan en el tiempo.

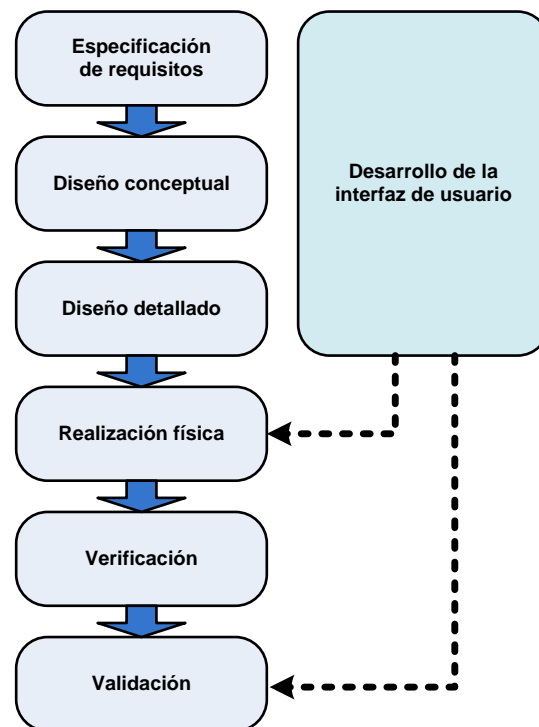


Figura 2.1. Procesos de la metodología basada en VHDL

En paralelo al diseño de la arquitectura sobre la FPGA se genera otro flujo de diseño para crear la interfaz gráfica de usuario con la que el usuario simulará sus circuitos en el sistema final pero que además agiliza los procesos de verificación, validación y análisis de resultados.

### 2.3. Herramientas

A continuación se enumeran todas las herramientas software y hardware que posibilitan el desarrollo del sistema junto con la funcionalidad y la forma de adquisición.

Herramienta de desarrollo **ISE (Integrated Software Environment) Design Suite 14.7** de Xilinx de libre distribución. En la metodología de diseño elegida permite editar y analizar las descripciones VHDL, sintetizar e implementar. Incluye el simulador **ISim** para lanzar simulaciones comportamentales y el software de programación **IMPACT** para configurar la FPGA en modo JTAG.

Placa de evaluación **Atlys de Digilent** con la FPGA de Xilinx XC6SLX45 y un transceptor Ethernet para el nivel físico a 1GHz. Es adquirida por compra. Desde la página web de Digilent se puede descargar de forma libre el programa **Digilent Adept** para programar la FPGA de la placa Atlys en modo ROM.

Herramienta de computación numérica, modelado y simulación **Scilab 5.5.0** de código abierto. Incluye **Xcos** que posibilita la edición de circuitos. Desde el entorno Scilab se instalan las librerías **JIMS 1.1** de forma libre que permiten el uso de código Java en programas escritos con el lenguaje de Scilab. Para poder emplear las librerías JIMS se requiere el entorno libre **JDK (Java Development Kit) 1.8.0** que incluye JRE (Java Runtime Environment) y JVM (Java Virtual Machine).

Herramienta de simulación de circuitos **Quite Universal Circuit Simulator (Qucs) 0.0.18** de código abierto. Hace posible la visualización del comportamiento de circuitos eléctricos.

Portátil PC con procesador Intel 4-Core 2.4 GHz y tarjeta de red Realtek PCI GBE. Se emplea un cable de red CAT-6 para unir el PC con la tarjeta Atlys. Son adquiridos por compra.

Analizador de paquetes Ethernet **Wireshark 1.12** para el chequeo del tráfico de información entre el PC y la FPGA. Desde la página <https://www.wireshark.org/download.html> puede descargarse de forma libre.

## 2.4. Conclusiones

Está definida una metodología para guiar el desarrollo del sistema y se dispone de un conjunto de herramientas. En los siguientes capítulos se describen las tareas para construir el sistema. Se comienza con la especificación de requisitos a la que sigue el diseño, la realización física, la verificación y la validación.



### 3. ESPECIFICACIÓN DE REQUISITOS

#### 3.1. Introducción

Elegida una metodología basada en VHDL se establecen una serie de procesos a seguir para el desarrollo del sistema. El primer proceso es la especificación de requisitos que debe definir qué requerimientos cumple la arquitectura a crear, es decir, qué funciones realiza. A continuación se enumeran los requisitos que cumple la arquitectura sobre la FPGA desarrollada en este proyecto.

#### 3.2. Funcionalidad general de la arquitectura

La arquitectura recibe como datos de entrada, a través de un controlador Ethernet, el modelo matemático que describe un sistema físico, en este caso un circuito eléctrico lineal.

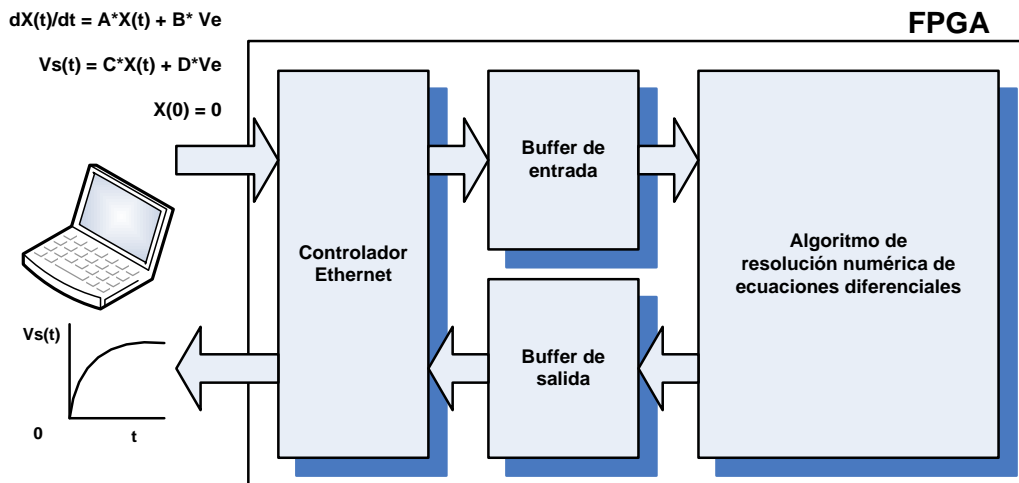


Figura 3.1. Funcionalidad general del sistema

Los datos de entrada quedan almacenados y son usados por un algoritmo de resolución numérica de ecuaciones diferenciales que obtiene una solución. Esta solución, que representa la simulación del circuito eléctrico lineal, queda acumulada en un buffer y posteriormente es extraída por el controlador Ethernet hacia el PC, para su posterior representación y análisis.

### **3.3. Flujo de datos de entrada y salida de la FPGA**

Los sistemas a simular son circuitos lineales que tienen las siguientes características.

- Como elemento activo se emplea una fuente de tensión  $V_e$  continua (constante en el tiempo), la cual representa la única entrada del sistema.
  
- Los posibles elementos pasivos del circuito son:
  - resistencias con conductancia  $G = 1/R$ ,
  - condensadores con capacitancia  $C$  y carga inicial nula, y además,
  - inducciones con auto-inductancia  $L$  y atravesadas por una corriente inicial nula.
  
- La salida del sistema o señal que se quiere simular es  $V_s(t)$ , también única y una de las tensiones en los nodos interiores de la malla.

Los valores de  $R$ ,  $L$  y  $C$  permanecen constantes.

El modelo matemático de estos sistemas físicos (Wikipedia, 2015) viene dado por el espacio de estados:

$$\begin{aligned}\frac{dX(t)}{dt} &= A \cdot X(t) + B \cdot V_e \\ V_s(t) &= C \cdot X(t) + D \cdot V_e \\ X(0) &= 0\end{aligned}\quad [3.1]$$

donde

N es el orden del problema,

A es la matriz de estado NxN (N filas x N columnas),

B es la matriz de entrada Nx1 (N filas x 1 columna),

C es la matriz de salida 1xN (1 fila x N columnas),

D es la matriz de transmisión directa 1x1 y

X(t) es la matriz de las variables de estado Nx1. Las variables de estado representan el estado dinámico del sistema en un instante de tiempo t y forman un conjunto mínimo.

Normalmente estas variables en circuitos eléctricos representan tensiones en nudos que conexionan ramas con condensadores o corrientes en ramas con inducciones.

La ecuación  $X(0) = 0$  define las condiciones iniciales del problema, en las que, para simplificar, se consideran nulos todos los elementos de la matriz X en el instante 0.

Por lo tanto, la salida  $V_s(t)$  queda definida en función de las variables de estado representadas en la matriz X(t) y de la entrada  $V_e$ .

En la Sección 7.2.5 se desarrolla un procedimiento sistemático para obtener la representación [3.1] a partir del esquema del circuito.

De la formulación del modelo [3.1] a simular se deduce que el flujo de datos de entrada de la FPGA viene dado por los coeficientes de las matrices A, B, C y D y por  $V_e$ , todos ellos constantes en el tiempo.

El flujo de datos de salida lo forman los valores reales de  $V_s(t)$  para los instantes  $t = 0, \dots, S-1$  donde  $S$  es el número de muestras de la simulación.

### 3.4. Algoritmo de resolución numérica

La arquitectura debe describir un algoritmo de resolución numérica con el que a partir del modelo matemático dado por el espacio de estados [3.1] se obtiene la solución  $V_s(t)$  para los instantes  $t = 0, \dots, S-1$ . De los algoritmos de resolución diferencial lineal existentes se elige el algoritmo de Runge-Kutta de orden 4 (RK4) que permite obtener  $V_s(t)$  con una aproximación muy alta con respecto a lo medido en un modelo físico real. El algoritmo se describe a continuación (Nizzo, 2009).

Dada una ecuación diferencial ordinaria de primer orden  $\frac{dX(t)}{dt} = f(t, X(t))$ , donde  $f$  es una función  $\mathfrak{R} \times \mathfrak{R}^N \rightarrow \mathfrak{R}^N$ ,  $X(t)$  es una matriz de variables y  $X(0) = X_0$  representa las condiciones iniciales. La solución  $X(t)$  se obtiene de la siguiente forma:

Para  $t = 0, \dots, S-1$ :

$$K_1(t) = f(t, X(t))$$

$$K_2(t) = f(t + h/2, X(t) + K_1(t) \cdot h/2)$$

$$K_3(t) = f(t + h/2, X(t) + K_2(t) \cdot h/2)$$

$$K_4(t) = f(t + h, X(t) + K_3(t))$$

$$X(t + 1) = X(t) + \frac{[K_1(t) + 2 \cdot (K_2(t) + K_3(t)) + K_4(t)]}{6}$$

$$t = t + h$$

El principio de este método numérico es recoger información de cuatro puntos alrededor de la última aproximación para definir el siguiente paso.



La constante  $h$  es el paso temporal entre dos muestras y determina la precisión de la solución. Su valor también está relacionado con la estabilidad del resultado, puesto que para valores grandes de  $h$  este método numérico puede hacer que el error al hacer la aproximación crezca a medida que el cálculo avanza (Nizzo, 2009).

Si se aplica el algoritmo al espacio de estados [3.1], se tiene que  $f$  es una función independiente de  $t$ , puesto que la entrada  $V_e$  es constante:

$$f(X(t)) = A \cdot X(t) + B \cdot V_e$$

y se puede obtener la solución  $V_s(t)$  de la siguiente forma:

Para  $t = 0, \dots, S - 1$ :

$$K_1(t) = f(X(t))$$

$$K_2(t) = f(X(t) + K_1(t) \cdot h/2)$$

$$K_3(t) = f(X(t) + K_2(t) \cdot h/2)$$

$$K_4(t) = f(X(t) + K_3(t) \cdot h)$$

$$X(t+1) = X(t) + h \cdot \frac{[K_1(t) + 2 \cdot (K_2(t) + K_3(t)) + K_4(t)]}{6}$$

$$V_s(t) = C \cdot X(t) + D \cdot V_e$$

donde  $K_1(t)$ ,  $K_2(t)$ ,  $K_3(t)$  y  $K_4(t)$  son matrices  $N \times 1$ .

A continuación se realiza un desglose del algoritmo de desarrollo propio para obtener una mejor comprensión del mismo. Previamente se definen los siguientes valores constantes de entrada:

$$BV_e = B \cdot V_e \quad DV_e = D \cdot V_e$$

$$H = h \quad H_{div2} = \frac{h}{2} \quad H_{div3} = \frac{h}{3} \quad H_{div6} = \frac{h}{6}$$

donde  $BV_e$  es una matriz  $N \times 1$  y lo demás son valores simples.

Para  $t = 0, \dots, S-1$ :

### **Etapa 0**

<i>Paso 0:</i>	$res1 = A \cdot X(t)$
<i>Paso 1:</i>	$K = res1 + BV_e$
<i>Paso 2:</i>	$res2 = H_{div2} \cdot K$ $res3 = H_{div6} \cdot K$
<i>Paso 3:</i>	$X' = X(t) + res2$ $X(t+1) = X(t) + res3$

### **Etapa 1**

<i>Paso 0:</i>	$res1 = A \cdot X'$
<i>Paso 1:</i>	$K = res1 + BV_e$
<i>Paso 2:</i>	$res2 = H_{div2} \cdot K$ $res3 = H_{div3} \cdot K$
<i>Paso 3:</i>	$X' = X(t) + res2$ $X(t+1) = X(t+1) + res3$

### **Etapa 2**

<i>Paso 0:</i>	$res1 = A \cdot X'$
<i>Paso 1:</i>	$K = res1 + BV_e$

$$\begin{aligned}
 \text{Paso 2:} & \quad \text{res2} = H \cdot K \\
 & \quad \text{res3} = H\text{div3} \cdot K \\
 \text{Paso 3:} & \quad X' = X(t) + \text{res2} \\
 & \quad X(t+1) = X(t+1) + \text{res3}
 \end{aligned}$$

### **Etapa 3**

$$\begin{aligned}
 \text{Paso 0:} & \quad \text{res1} = A \cdot X' \\
 \text{Paso 1:} & \quad K = \text{res1} + BV_e \\
 \text{Paso 2:} & \quad \text{res3} = H\text{div6} \cdot K \\
 \text{Paso 3:} & \quad X(t+1) = X(t+1) + \text{res3} \\
 \text{Paso 4:} & \quad \text{res4} = C \cdot X(t+1) \\
 \text{Paso 5:} & \quad V_s(t) = \text{res4} + DV_e
 \end{aligned}$$

Donde  $K$ ,  $X'$ ,  $\text{res1}$ ,  $\text{res2}$ ,  $\text{res3}$  y  $\text{res4}$  representan matrices con los resultados de las correspondientes operaciones.

### **3.5. Tamaño del problema**

El sistema puede realizar la simulación de circuitos con un orden  $N$  en el rango de 1 a  $N_m$ , donde  $N_m$  representa el tamaño del problema y es igual a 16.

En este punto es importante aclarar que el valor del orden máximo  $N_m$  se limita a 16 para poder enviar los datos de configuración en una sola trama Ethernet y facilitar el diseño del sistema.

Para un análisis más detallado ver la Sección 3.7 y la Sección 9.1.

### **3.6. Representación de los datos**

Los elementos de las matrices  $A$ ,  $BV_e$ ,  $C$ ,  $X(t)$ ,  $X(t+1)$ ,  $X'$ ,  $K$ ,  $res1$ ,  $res2$ ,  $res3$  y  $res4$  y los valores  $DV_e$ ,  $H$ ,  $Hdiv2$ ,  $Hdiv3$  y  $Hdiv6$ , así como el resultado de las operaciones entre ellos, son números reales y su representación viene dada por el estándar IEEE-754 con precisión simple de 32 bits. El valor del orden  $N$  se representa como un número entero tipo byte.

### **3.7. Controlador de comunicación Ethernet**

Con respecto a la capa física, la FPGA se comunica con un PC a través de Ethernet con un enlace punto a punto y recibe y transmite los datos en modo GMII a 1 Gbps. La arquitectura también gestiona las capas de enlace, de red y de transporte. De la capa de enlace existe control del protocolo ARP (Address Resolution Protocol) y a nivel de transporte el envío de datos es con UDP (User Datagram Protocol).

La FPGA recibe datagramas UDP con un campo de datos o payload de longitud fija  $L = 1 + 16 * 16 * 4 + 16 * 4 + 16 * 4 + 4 + 4 * 4 = 1.173$  bytes con los sub-campos mostrados en la Figura 3.2. La recepción comienza con el valor  $N-1$ , donde  $N$  es el orden del sistema a simular, y acaba con el valor  $Hdiv6$ .

Con respecto a los valores de cada matriz, la FPGA comienza recibiendo la primera fila y dentro de cada fila se empieza con el elemento de la primera columna. De los elementos de las matrices  $A$ ,  $BV_e$  y  $C$  y de los valores  $DV_e$ ,  $H$ ,  $Hdiv2$ ,  $Hdiv3$  y  $Hdiv6$ , se recibe primero el byte menos significativo teniendo en cuenta que están formados por cuatro bytes (32 bits en coma flotante). El valor  $N-1$  es de un byte.

N-1
A(0, 0)
A(0, 1)
...
A(0, N-1)
...
A(N-1, 0)
A(N-1, 1)
...
A(N-1, N-1)
BV <sub>e</sub> (0)
BV <sub>e</sub> (1)
...
BV <sub>e</sub> (N-1)
C(0)
C(1)
...
C(N-1)
DV <sub>e</sub>
H
Hdiv2
Hdiv3
Hdiv6

**Figura 3.2. Campo de datos de un datagrama UDP**

La longitud del campo de datos o payload del datagrama de entrada es fijo, por lo que si el sistema a simular tiene un orden menor que 16, los valores útiles mantienen su posición y las demás celdas se rellenan con ceros.

El datagrama de salida UDP contiene los S valores del resultado de la simulación, enviando siempre del primero al último, según la representación en el tiempo de la simulación del circuito RLC, y dentro de un valor, comenzando siempre por el byte menos significativo. Para esta arquitectura se fija el valor de S a 200 muestras.

### 3.8. Almacenado de datos de entrada y salida

Los datos de entrada se almacenan en la FPGA de forma permanente durante el procesado del algoritmo. Los datos de salida se van generando y registrando en un buffer para ser enviados cuando se han completado.

### 3.9. Activación del procesado de la simulación

Una vez queda configurada la FPGA, cada vez que la arquitectura recibe un datagrama de entrada UDP se activa el algoritmo, se calcula el resultado y se transmite el datagrama de salida UDP con la solución. La FPGA debe estar preparada para recibir nuevos datagramas con las consiguientes solicitudes, incluso para simulaciones de sistemas con distinto orden.

### 3.10. Señales de reloj y reseteado

La arquitectura recibe dos relojes externos: el reloj CLK de 100 MHz para la arquitectura del algoritmo y el reloj RXCLK de 125 MHz proveniente de un transceptor Ethernet y con el que se sincronizan los datos de las tramas recibidas.

La FPGA genera un reloj CLK\_GMII para la transmisión de datos por el controlador Ethernet con el que deben estar sincronizados los datos de salida.

Se emplea un reset externo asíncrono que reinicializa todos los registros de la arquitectura.

### 3.11. Puertos de entrada y salida

En la Tabla 3.1 se definen los puertos de la arquitectura y las señales que introducen o extraen.

Puerto	Tipo	Función
CLK100MHZ	Entrada	Reloj de 100 MHz
RESET	Entrada	Señal de reset asíncrono
CLK_GMII	Salida	Reloj Ethernet de salida de 125 MHz para transmisión
RXCLK	Entrada	Reloj Ethernet de 125 MHz para recepción
RXDV	Entrada	Señal de validación de datos de entrada Ethernet
RXD(7:0)	Entrada	Datos de entrada Ethernet
TXEN	Salida	Señal de validación de datos de salida Ethernet
TXD(7:0)	Salida	Datos de salida Ethernet

Tabla 3.1. Puertos de la arquitectura

### **3.12. Conclusiones**

A partir de las funcionalidades del sistema va a realizarse el diseño que mostrará cómo construir la arquitectura, conceptualmente, definiendo un conjunto de bloques interconectados, y en detalle, describiendo cada bloque con VHDL, así como el uso de cores, memorias, relojes, etc.





## 4. DISEÑO CONCEPTUAL

### 4.1. Introducción

En la metodología basada en VHDL, el diseño top-down parte de un conjunto de bloques interconectados que se van de forma progresiva detallando o subdividiendo formando una jerarquía. En el diseño conceptual se define la estructura del sistema, los bloques que la forman y la conexión entre ellos. Posteriormente se hace una primera descripción de cada bloque. En las siguientes secciones se realizan estos pasos.

### 4.2. Arquitectura del sistema

En la siguiente figura puede verse el conjunto de bloques que forman la arquitectura de este proyecto sobre la FPGA, la conexión entre ellos y la interfaz hacia el exterior.

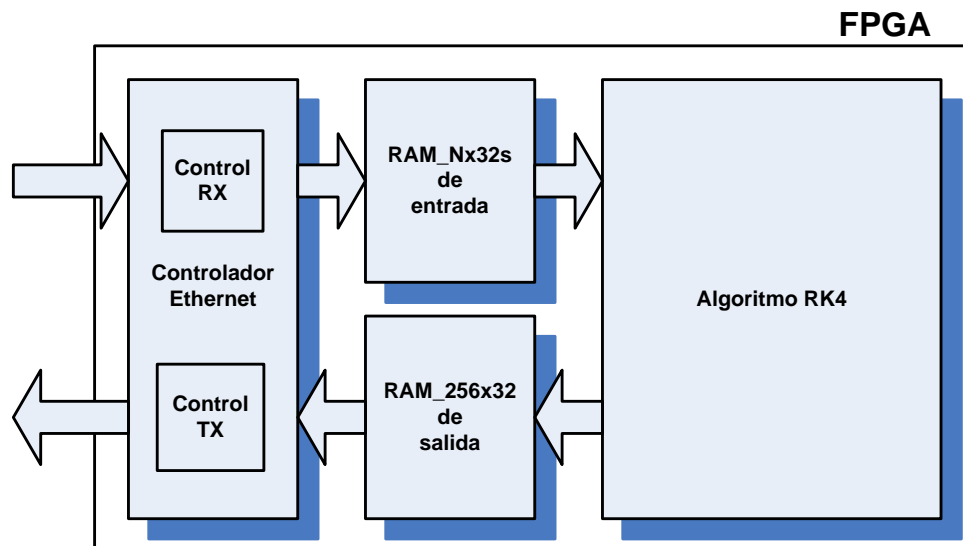


Figura 4.1. Arquitectura del sistema

El flujo de datos de la información de estado de un circuito a simular proveniente del PC es detectado y leído por un controlador Ethernet y almacenado en distintas memorias RAMs en función del tipo de información. Seguidamente se activa el procesado del algoritmo y los resultados alcanzados de la simulación del circuito se van registrando en otra memoria RAM. Cuando el algoritmo finaliza el procesado, el controlador Ethernet inicia la transmisión de los resultados hacia el PC.

### 4.3. Controlador Ethernet

El controlador Ethernet, siguiendo la norma IEEE 802.3, selecciona, de entre todas las tramas provenientes del tráfico de red, las tramas ARP y URP enviadas por el PC hacia la FPGA.

Las tramas Ethernet (Tanenbaum, 2010: 282) están formadas por los campos mostrados en la Figura 4.2. En el campo de datos se van encapsulando las cabeceras que controlan los protocolos, como ARP (Hall, 2000: 113-114) o UDP (Hall, 2000:258-262), de los distintos niveles de la arquitectura de red.

<b>Preámbulo</b> 7 bytes	<b>Delimitador de inicio de trama</b> 1 byte	<b>Dirección MAC de destino</b> 6 bytes	<b>Dirección MAC de origen</b> 6 bytes	<b>Longitud</b> 2 bytes	<b>Datos</b> 46-1.500 bytes	<b>CRC</b> 4 bytes
-----------------------------	---	--	---	----------------------------	--------------------------------	-----------------------

**Figura 4.2. Campos de trama Ethernet**

El PC actúa como cliente que inicia la comunicación y la FPGA como servidor que atiende las peticiones del cliente. El controlador Ethernet recibe los bytes de las tramas de entrada sincronizados con el reloj *rxclk* y con un contador va identificando cada byte y a qué campo pertenece. El controlador también debe generar la trama respuesta para cada caso.

La primera vez que la aplicación cliente envía datos UDP se transmite previamente una trama ARP a la red para preguntar quién es el dispositivo con dirección IP (Internet Protocol) 192.168.0.1 y cuál es su dirección MAC (Media Access Control). La FPGA debe reaccionar a esta trama ARP que pide su identificación, almacenar en registros la IP y la MAC del PC cliente que inicia la comunicación y enviar otra trama ARP de respuesta con su IP y su MAC como direcciones fuente y con la IP y la MAC registradas del cliente como direcciones destino.

Cuando el controlador Ethernet recibe una trama UDP de datos, comprueba la identificación del cliente con la IP registrada anteriormente y almacena los datos útiles de entrada en las correspondientes memorias RAM y registros. Estos datos son los valores de N-1, A, BVe, C, DVe, H, Hdiv2, Hdiv3 y Hdiv6. Una vez finalizada la recepción de la trama de datos de entrada se activa un pulso de inicio de procesamiento del algoritmo.

Cuando la arquitectura acaba de generar los resultados activa un pulso de finalización y la FPGA envía la trama de respuesta hacia el cliente con los valores de la simulación del circuito.

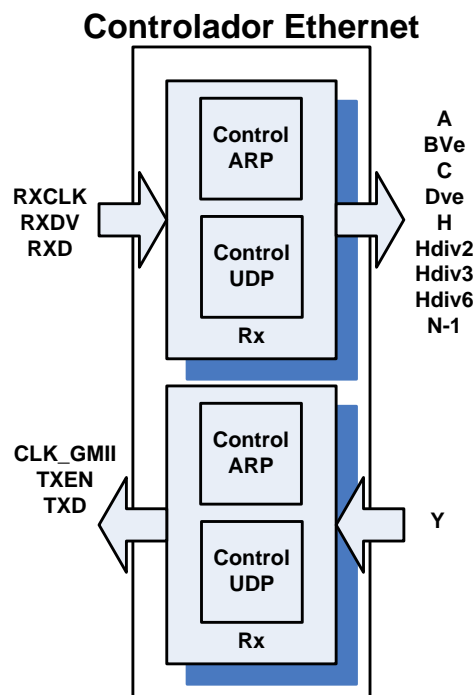


Figura 4.3. Controlador Ethernet

Por simplificación, el controlador Ethernet en la recepción no realiza la verificación de redundancia cíclica (CRC - Cyclic Redundancy Check) para detección de errores en la trama, pero en la transmisión es necesario añadir el campo CRC después del campo de datos para que el cliente no descarte la trama.

Consideremos que la trama a transmitir, sin los bits de preámbulo y byte delimitador de trama (Spurgeon, 2014: 52), se representa con el polinomio binario de orden k:

$$M(X) = m_k \cdot X^k + m_{k-1} \cdot X^{k-1} + \dots + m_1 \cdot X^1 + m_0 \cdot X^0 \quad \forall m_k = 0,1$$

donde cada byte que forma la trama se ha colocado comenzando con el bit menos significativo.

Se tiene entonces que el campo CRC-32 (32 bits) se forma con el resto de realizar la división binaria  $X^{32+k} \cdot M(X)$  por el polinomio binario de orden 32:

$$G(X) = X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X^1 + X^0$$

A nivel lógico el campo CRC se puede conseguir con la combinación de un registro de desplazamiento y operaciones xor, como puede verse en la Figura 4.4 (Warren, 2012: 319-323).

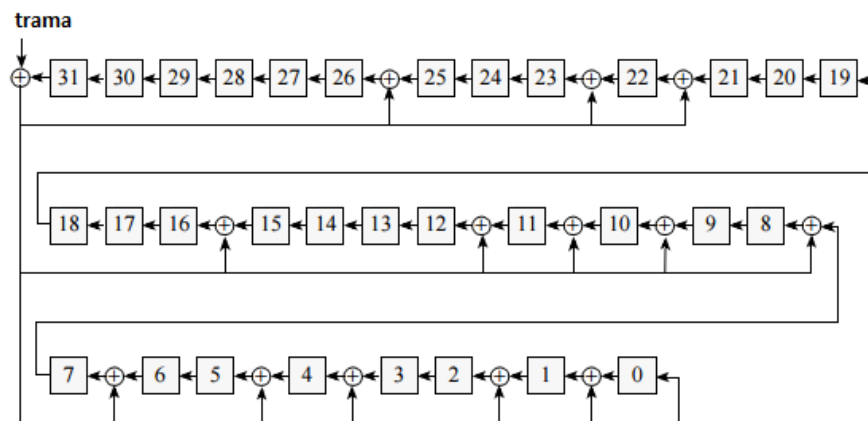


Figura 4.4. Generación del campo CRC

Inicialmente los flip-flops almacenan 1s y se van cargando con el mensaje de entrada M por desplazamiento. Si el bit de entrada es 1 se aplican las operaciones xor. El valor del CRC lo forman los bits complementarios en los flip-flops después de entrar el último bit. El pseudocódigo del algoritmo completo se muestra a continuación.

```

VECTOR DE 1s Y 0s: mensaje(N)

crc = 0xFFFFFFFF;

PARA i DESDE 0 HASTA N-1 HACER
    SI (crc[15] XOR mensaje[i] = 1) ENTONCES
        crc = DESPLAZAR_HACIA_IZDA(crc) XOR 0x04C11DB7;
    SINO
        crc = DESPLAZAR_HACIA_IZDA(crc);
    FINAL SI
FIN PARA

crc = INVERTIR_ORDEN(COMPLEMENTAR(crc));

```

#### 4.4. Memorias

Los valores de los elementos de las matrices A,  $BV_e$  y C se almacenan en memorias RAM  $N \times 32$  (N filas x 32 bits) dentro de la arquitectura de la FPGA. Las matrices  $BV_e$  y C emplean una memoria cada una, pero la matriz A requiere N memorias RAM  $N \times 32$  de tal forma que el elemento  $A(i, j)$  de la fila i y columna j, donde  $i, j = 0, \dots, N-1$ , se almacena en la fila i de la RAM j. Al encontrarse todos los elementos de una fila de la matriz A en memorias independientes se permite la lectura de todos ellos simultáneamente.

Las distintas muestras de la solución  $V_s(t)$  se van registrando en una memoria RAM  $256 \times 32$ .

El algoritmo RK4 hace uso de tres memorias RAM  $N \times 32$ , para los dos contextos  $X(t)$  y  $X(t+1)$  y para la matriz auxiliar  $X'$ .

Los registros de todas estas memorias almacenan palabras de 32 bits para contener valores en coma flotante siendo el bit-31 el MSB y el bit-0 el LSB.

#### 4.5. Algoritmo RK4

En la Figura 4.5 se describen los bloques empleados para realizar las distintas operaciones aritméticas del algoritmo. Las operaciones son sumas y productos en coma flotante sincronizadas con escrituras y lecturas en memorias RAM.

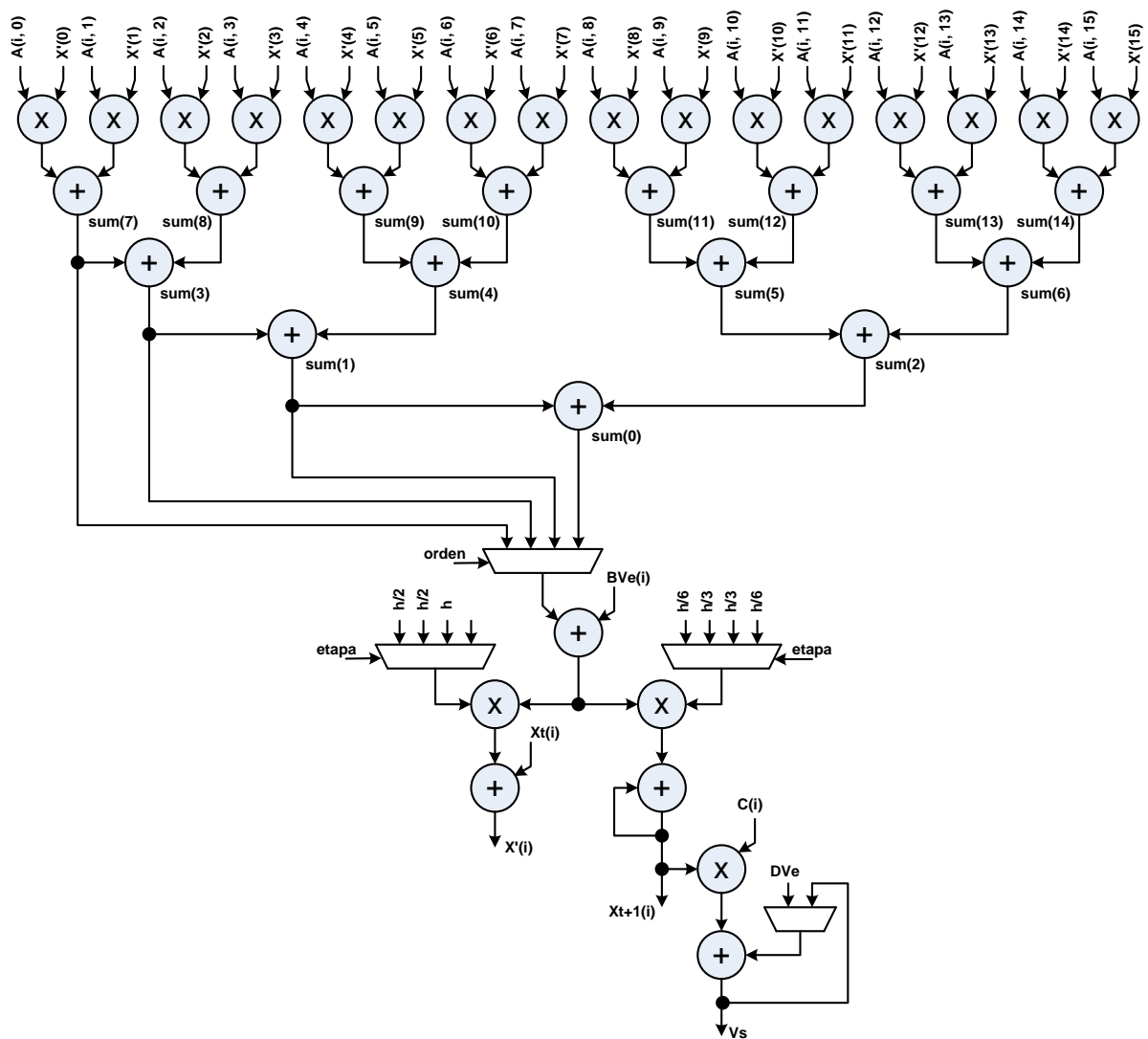


Figura 4.5. Descripción de las operaciones del algoritmo

Operaciones de diferentes pasos del algoritmo se pueden realizar simultáneamente, por lo que no es necesario finalizar todas las operaciones de un paso para iniciar otro (pipeline). Se emplean contadores para sincronizar las operaciones en cada momento y multiplexores para seleccionar el valor de ciertos operadores según la etapa.

A continuación se detallan los pasos de una etapa para el cálculo de cada muestra del algoritmo. Se denomina *ciclo de operación* al número de ciclos de reloj (100 MHz) que tarda un sumador o un multiplicador de coma flotante en realizar una operación.

**Paso 0: si etapa = 0 entonces  $res1 = A * X(t)$ , si etapa = 1, 2 ó 3 entonces  $res1 = A * X'$**

Se deben obtener N valores multiplicando N elementos de cada fila de A por los N elementos de la columna de  $X(t-1)$  o  $X'$  donde N es el orden del sistema. Este producto de matrices se realiza para cada muestra  $t = 0, \dots, S-1$  de la simulación durante las cuatro etapas del algoritmo, es decir  $4 * S$  veces, por lo que es necesario que su diseño se realice de una forma eficiente por el número de operaciones que implica.

En este diseño, para esta operación, se emplean N multiplicadores y N-1 sumadores. Al final de la etapa anterior se cargan durante N ciclos de reloj los operadores de los multiplicadores extrayendo los valores de la matriz  $X(t)$  cuando la etapa es 0 y de la matriz  $X'$  cuando son las etapas 1, 2 ó 3. En cada ciclo de operación se cargan los elementos de una fila de A en los puertos de los N multiplicadores y se multiplican por los valores de los coeficientes de  $X(t)$  o de  $X'$ , que permanecen fijos en cada etapa.

Posteriormente se realizan las sumas en  $\log_2 N$  ciclos de operación, de tal forma que mientras se están realizando los productos para la fila 1 de A se están calculando las primeras sumas de la fila 0 y así sucesivamente. Para  $N = 16$  la suma final se extrae del sumador 0, pero para evitar retardos innecesarios cuando el orden del sistema es menor que N, la suma final se obtiene del

correspondiente sumador en función del orden del sistema a simular. La siguiente tabla muestra el sumador que extrae el resultado final de *res1* en función del orden del sistema.

Orden del sistema	Sumador
1, 2	Sum7
3, 4	Sum3
5, 6, 7, 8	Sum1
9, 10, 11, 12, 13, 14, 15, 16	Sum0

**Tabla 4.1.** Sumador que obtiene *res1* en función del orden

**Paso 1: suma de matrices para obtener  $K = res1 + BV_e$**

Para realizar esta operación de matrices Nx1 se emplea un sumador que en cada ciclo de operación va sumando al valor de una fila de *res1* el correspondiente elemento de la columna de  $BV_e$ .

**Paso 2: productos de escalar por matriz para obtener *res2* y *res3***

En las tres primeras etapas de cada muestra, para obtener *res2* se realiza el producto de una constante por la matriz K. Para ello se emplea un multiplicador que va obteniendo en cada ciclo de operación el producto de un elemento de K por la constante. Un multiplexor determina el valor de la constante en función de la etapa, como puede verse en la siguiente tabla.

Etapa	Salida del multiplexor
0	$h/2$
1	$h/2$
2	$h$
3	-

**Tabla 4.2.** Salida de multiplexor en función de la etapa para *res2*



Para obtener  $res3$  se realiza el mismo procedimiento empleando otro multiplicador pero ahora el multiplexor extrae diferentes valores para las constantes.

Etapa	Salida del multiplexor
0	$h/6$
1	$h/3$
2	$h/3$
3	$h/6$

**Tabla 4.3.** Salida de multiplexor en función de la etapa para  $res3$

**Paso 3: suma de matrices  $X' = X(t) + res2$  y  $X(t+1) = X(t+1) + res3$**

También simultáneamente y haciendo uso de dos sumadores, se obtienen los valores de una fila de  $X'$  y  $X(t+1)$  en cada ciclo de operación y se almacenan en una RAM  $N \times 32$ . Al final de las etapas 0, 1 y 2 los elementos de la memoria  $X'$  se cargan en los multiplicadores para los cálculos de la siguiente etapa. La matriz  $X(t+1)$  parte de los valores de  $X(t)$  para la anterior muestra y va sumando  $res3$  en las distintas etapas.

**Paso 4: producto de matrices  $res4 = C * X(t+1)$**

Con un multiplicador se va obteniendo en cada ciclo de operación el producto de un elemento de  $C$  por un elemento de  $X(t+1)$ . El resultado pasa a un sumador que actúa de acumulador.

**Paso 5:  $V_s(t) = res4 + DV_e$**

Al final de la realización de todos los productos se acaba sumando al resultado de la acumulación el valor  $DV_e$ . Posteriormente se almacena este valor final en la memoria RAM de  $V_s(t)$ .

***Inicio de cálculo para la nueva muestra***

Al final de la etapa 3 en las dos matrices  $X(t)$  y  $X(t+1)$  quedan almacenados los valores del actual valor de  $X$  por lo que en la siguiente etapa 0, para la nueva muestra, ya está preparada la matriz  $X(t)$  y  $X(t+1)$  contiene el valor de partida para iniciar el resto de sumas.

## **4.6. Conclusiones**

Establecida la arquitectura y descritos los distintos bloques a alto nivel se pasa al siguiente nivel de diseño en el que se aumenta el detalle de cómo el sistema realiza sus funciones y se añaden las descripciones VHDL.

## 5. DISEÑO DETALLADO

### 5.1. Introducción

Del diseño conceptual se obtiene una estructura jerárquica de bloques interconectados entre sí y una primera descripción de cada componente. Ahora en el diseño detallado se describe con VHDL cada uno de los componentes, se define la configuración de las memorias y cores a emplear y se aclaran los puntos referentes a sincronización y dominio de relojes.

### 5.2. Descripciones VHDL

A continuación se exponen las partes de VHDL que se consideran más ilustrativas para la descripción del sistema con sus correspondientes comentarios.

#### *Entidad rk4*

La entidad *rk4* describe los puertos para la arquitectura principal del sistema.

```

16 entity rk4 is
17     port( CLK100MHZ: in std_logic;
18           RESET: in std_logic;
19           --
20           CLK_GMII: out std_logic;
21           TXEN: out std_logic;
22           TXER: out std_logic;
23           TXD: out std_logic_vector(7 downto 0);
24           --
25           RXCLK: in std_logic;
26           RXDV: in std_logic;
27           RXER: in std_logic;
28           RXD: in std_logic_vector(7 downto 0);
29           --
30           MDC: out std_logic;
31           MDIO: out std_logic;
32           NRST: out std_logic;
33           --
34           LED: out std_logic_vector(3 downto 0));
35 end rk4;
```

### Arquitectura *rk4\_arch*

La arquitectura principal del sistema *rk4\_arch* instancia el componente *udp* con el controlador Ethernet y el componente *algorithm* con la descripción del algoritmo.

Son instanciadas además en la arquitectura *rk4\_arch* las memorias RAM de doble puerto *dpram\_Nx32* y *dpram\_256x32* que almacenan los datos de las matrices A,  $BV_e$ , C y  $V_s(t)$ , como puede observarse a continuación.

```
293 inst_generate_dpram_a: for i in 0 to N-1 generate
294   inst_dpram_a_i: dpram_Nx32 port map(
295     CLK_W => RXCLK,
296     WR => wr_ram_a,
297     ADDR_W => addr_ram_a_udp,
298     DIN => din_ram_a(i),
299     CLK_R => CLK100MHZ,
300     ADDR_R => addr_ram_a_alg,
301     DOUT => dout_ram_a(i));
302 end generate;
303
304 inst_dpram_bve: dpram_Nx32 port map(
305   CLK_W => RXCLK,
306   WR => wr_ram_bve,
307   ADDR_W => addr_ram_bve_udp,
308   DIN => din_ram_bve,
309   CLK_R => CLK100MHZ,
310   ADDR_R => addr_ram_bve_alg,
311   DOUT => dout_ram_bve);
312
313 inst_dpram_c: dpram_Nx32 port map(
314   CLK_W => RXCLK,
315   WR => wr_ram_c,
316   ADDR_W => addr_ram_c_udp,
317   DIN => din_ram_c,
318   CLK_R => CLK100MHZ,
319   ADDR_R => addr_ram_c_alg,
320   DOUT => dout_ram_c);
321
322 inst_dpram_yt: dpram_256x32 port map(
323   CLK_W => CLK100MHZ,
324   WR => wr_ram_yt,
325   ADDR_W => addr_ram_yt_alg,
326   DIN => din_ram_yt,
327   CLK_R => RXCLK,
328   ADDR_R => addr_ram_yt_udp,
329   DOUT => dout_ram_yt);
```

### Entidad *udp*

Describe los puertos del bloque de control del controlador Ethernet.

```

19 entity udp is
20     port( RESET: in std_logic;
21         --
22         CLK_GMII: out std_logic;
23         TXEN: out std_logic;
24         TXER: out std_logic;
25         TXD: out std_logic_vector(7 downto 0);
26         --
27         RXCLK: in std_logic;
28         RXDV: in std_logic;
29         RXER: in std_logic;
30         RXD: in std_logic_vector(7 downto 0);
31         --
32         MDC: out std_logic;
33         MDIO: out std_logic;
34         NRST: out std_logic;
35         --
36         LED: out std_logic_vector(3 downto 0);
37         --
38         ORDER: out std_logic_vector(M-1 downto 0);
39         --
40         WR_RAM_A: out std_logic;
41         ADDR_RAM_A: out std_logic_vector(M-1 downto 0);
42         DIN_RAM_A: out reg_N_32_type;
43         --
44         WR_RAM_BVE: out std_logic;
45         ADDR_RAM_BVE: out std_logic_vector(M-1 downto 0);
46         DIN_RAM_BVE: out std_logic_vector(31 downto 0);
47         --
48         WR_RAM_C: out std_logic;
49         ADDR_RAM_C: out std_logic_vector(M-1 downto 0);
50         DIN_RAM_C: out std_logic_vector(31 downto 0);
51         --
52         DVE: out std_logic_vector(31 downto 0);
53         --
54         H: out std_logic_vector(31 downto 0);
55         H_DIV_2: out std_logic_vector(31 downto 0);
56         H_DIV_3: out std_logic_vector(31 downto 0);
57         H_DIV_6: out std_logic_vector(31 downto 0);
58         --
59         ADDR_RAM_YT: out std_logic_vector(7 downto 0);
60         DOUT_RAM_YT: in std_logic_vector(31 downto 0);
61         --
62         START_ALGORITHM: out std_logic;
63         READY_ALGORITHM: in std_logic);
64 end udp;

```

### Arquitectura *udp\_arch*

En la arquitectura *udp\_arch* el proceso *proc\_rx\_udp* contiene el control de la recepción de los datos por Ethernet. En este proceso se detecta la llegada del preámbulo (7 bytes de valor x55) y del delimitador de inicio de trama (1 byte de valor xD5) de una trama y se activa el contador *counter\_rx* que controla los bytes recibidos.

```

210         if (rxdv_reg = '1') then
211             --
212             if (counter_rx = 0) then
213                 --
214                 if ((check_mac = '0') and (buffer_rxd = x"5555555555555555D5")) then
215                     --
216                     counter_rx <= counter_rx + 1;
217                     check_mac <= '1';
218                     --
219                     LED(0) <= '1';
220                     --
221                 end if;
222             --
223         else
224             --
225             counter_rx <= counter_rx + 1;

```

Conociendo la posición de cada byte en la trama recibida y con el contador *counter\_rx* se registra la información útil, como las direcciones MAC e IP del PC transmisor o el tipo de trama Ethernet.

```

227         if (counter_rx = 6) then destin_mac(5) <= rxd_reg;
228         elsif (counter_rx = 7) then destin_mac(4) <= rxd_reg;
229         elsif (counter_rx = 8) then destin_mac(3) <= rxd_reg;
230         elsif (counter_rx = 9) then destin_mac(2) <= rxd_reg;
231         elsif (counter_rx = 10) then destin_mac(1) <= rxd_reg;
232         elsif (counter_rx = 11) then destin_mac(0) <= rxd_reg;
233         --
234         elsif (counter_rx = 12) then ether_type(15 downto 8) <= rxd_reg;
235         elsif (counter_rx = 13) then ether_type(7 downto 0) <= rxd_reg;
236         --
237         elsif (counter_rx = 20) then
238             --
239             if (ether_type = x"0806") then opcode_arp(15 downto 8) <= rxd_reg;
240             end if;
241             --
242         elsif (counter_rx = 21) then
243             --
244             if (ether_type = x"0806") then opcode_arp(7 downto 0) <= rxd_reg;
245             end if;
246             --
247         elsif (counter_rx = 26) then
248             --
249             if (ether_type = x"0800") then destin_ip(3) <= rxd_reg;
250             end if;

```

En el registro *data\_type* se almacena el identificador del tipo de trama de datos UDP.

```

311         elsif (counter_rx = 42) then
312             --
313             data_type <= rxd_reg;

```

En el proceso *proc\_rx\_udp* también se añade el control para cargar en las memorias RAM los valores de A, BV<sub>e</sub> y C según van llegando los bytes de la trama de datos UDP.

```

319     elsif ((counter_rx > 43) and (counter_rx <= 43 + 1024)) then
320         --
321         index_ram := counter_rx - 44;
322         --
323         if (counter_rx(1 downto 0) = 0) then DIN_RAM_A(conv_integer(index_ram(5 downto 2)))(7 downto 0) <= rxd_reg;
324         elsif (counter_rx(1 downto 0) = 1) then DIN_RAM_A(conv_integer(index_ram(5 downto 2)))(15 downto 8) <= rxd_reg;
325         elsif (counter_rx(1 downto 0) = 2) then DIN_RAM_A(conv_integer(index_ram(5 downto 2)))(23 downto 16) <= rxd_reg;
326         elsif (counter_rx(1 downto 0) = 3) then DIN_RAM_A(conv_integer(index_ram(5 downto 2)))(31 downto 24) <= rxd_reg;
327         end if;
328         --
329         if (counter_rx(5 downto 0) = 63) then
330             --
331             ADDR_RAM_A <= index_ram(9 downto 6);
332             WR_RAM_A <= '1';
333             --
334         end if;

336     elsif ((counter_rx > 43 + 1024) and (counter_rx <= 43 + 1024 + 64)) then
337         --
338         if (counter_rx(1 downto 0) = 0) then DIN_RAM_BVE(7 downto 0) <= rxd_reg;
339         elsif (counter_rx(1 downto 0) = 1) then DIN_RAM_BVE(15 downto 8) <= rxd_reg;
340         elsif (counter_rx(1 downto 0) = 2) then DIN_RAM_BVE(23 downto 16) <= rxd_reg;
341         elsif (counter_rx(1 downto 0) = 3) then DIN_RAM_BVE(31 downto 24) <= rxd_reg;
342         end if;
343         --
344         if (counter_rx(1 downto 0) = 3) then
345             --
346             index_ram := counter_rx - (44 + 1024);
347             ADDR_RAM_BVE <= index_ram(5 downto 2);
348             WR_RAM_BVE <= '1';
349             --
350         end if;

352     elsif ((counter_rx > 43 + 1024 + 64) and (counter_rx <= 43 + 1024 + 64 + 64)) then
353         --
354         if (counter_rx(1 downto 0) = 0) then DIN_RAM_C(7 downto 0) <= rxd_reg;
355         elsif (counter_rx(1 downto 0) = 1) then DIN_RAM_C(15 downto 8) <= rxd_reg;
356         elsif (counter_rx(1 downto 0) = 2) then DIN_RAM_C(23 downto 16) <= rxd_reg;
357         elsif (counter_rx(1 downto 0) = 3) then DIN_RAM_C(31 downto 24) <= rxd_reg;
358         end if;
359         --
360         if (counter_rx(1 downto 0) = 3) then
361             --
362             index_ram := counter_rx - (44 + 1024 + 64);
363             ADDR_RAM_C <= index_ram(5 downto 2);
364             WR_RAM_C <= '1';
365             --
366         end if;

```

Además se encuentra el control para cargar los valores N-1, DV<sub>e</sub>, H, H/2, H/3 y H/6.

```

315     elsif (counter_rx = 43) then
316         --
317         ORDER <= rxd_reg(M-1 downto 0);

368     elsif ((counter_rx > 43 + 1024 + 64 + 64) and (counter_rx <= 43 + 1024 + 64 + 64 + 4)) then
369         --
370         if (counter_rx(1 downto 0) = 0) then DVE(7 downto 0) <= rxd_reg;
371         elsif (counter_rx(1 downto 0) = 1) then DVE(15 downto 8) <= rxd_reg;
372         elsif (counter_rx(1 downto 0) = 2) then DVE(23 downto 16) <= rxd_reg;
373         elsif (counter_rx(1 downto 0) = 3) then DVE(31 downto 24) <= rxd_reg;
374         --
375     end if;

```

```

377         elsif ((counter_rx > 43 + 1024 + 64 + 64 + 4) and (counter_rx <= 43 + 1024 + 64 + 64 + 4 + 16)) then
378             --
379             if (counter_rx(3 downto 0) = 0) then H(7 downto 0) <= rxd_reg;
380             elsif (counter_rx(3 downto 0) = 1) then H(15 downto 8) <= rxd_reg;
381             elsif (counter_rx(3 downto 0) = 2) then H(23 downto 16) <= rxd_reg;
382             elsif (counter_rx(3 downto 0) = 3) then H(31 downto 24) <= rxd_reg;
383             --
384             elsif (counter_rx(3 downto 0) = 4) then H_DIV_2(7 downto 0) <= rxd_reg;
385             elsif (counter_rx(3 downto 0) = 5) then H_DIV_2(15 downto 8) <= rxd_reg;
386             elsif (counter_rx(3 downto 0) = 6) then H_DIV_2(23 downto 16) <= rxd_reg;
387             elsif (counter_rx(3 downto 0) = 7) then H_DIV_2(31 downto 24) <= rxd_reg;
388             --
389             elsif (counter_rx(3 downto 0) = 8) then H_DIV_3(7 downto 0) <= rxd_reg;
390             elsif (counter_rx(3 downto 0) = 9) then H_DIV_3(15 downto 8) <= rxd_reg;
391             elsif (counter_rx(3 downto 0) = 10) then H_DIV_3(23 downto 16) <= rxd_reg;
392             elsif (counter_rx(3 downto 0) = 11) then H_DIV_3(31 downto 24) <= rxd_reg;
393             --
394             elsif (counter_rx(3 downto 0) = 12) then H_DIV_6(7 downto 0) <= rxd_reg;
395             elsif (counter_rx(3 downto 0) = 13) then H_DIV_6(15 downto 8) <= rxd_reg;
396             elsif (counter_rx(3 downto 0) = 14) then H_DIV_6(23 downto 16) <= rxd_reg;
397             elsif (counter_rx(3 downto 0) = 15) then H_DIV_6(31 downto 24) <= rxd_reg;
398             --
399             end if;

```

Al final de la trama recibida se determina si ésta es de tipo ARP o es un datagrama UDP. Si es una trama ARP se define la cabecera de la trama ARP de respuesta y se activa su transmisión con el pulso *start\_tx*.

```

407         if (check_mac = '1') then
408             --
409             if (ether_type = x"0806") then
410                 --
411                 if ((opcode_arp = x"0001") and (target_ip = source_ip)) then
412                     --
413                     packet_arp <= '1';
414                     --
415                     LED(1) <= '1';
416                     --
417                     -----
418                     --// cabecera MAC para IEEE 802.3 Ethernet (14 bytes) //
419                     -----
420
421                     header_tx(0) <= destin_mac(5);      -- DA --> MAC del destino (cliente)
422                     header_tx(1) <= destin_mac(4);
423                     header_tx(2) <= destin_mac(3);
424                     header_tx(3) <= destin_mac(2);
425                     header_tx(4) <= destin_mac(1);
426                     header_tx(5) <= destin_mac(0);
427
428                     header_tx(6) <= source_mac(5);     -- SA --> MAC de la fuente (fpga)
429                     header_tx(7) <= source_mac(4);
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473                     header_tx(38) <= destin_ip(3);
474                     header_tx(39) <= destin_ip(2);
475                     header_tx(40) <= destin_ip(1);
476                     header_tx(41) <= destin_ip(0);
477                     --
478                     size_header_tx <= 42;
479                     size_data_tx <= 18;
480                     --
481                     start_tx <= '1';

```

Si la trama recibida es un datagrama UDP, se debe diferenciar entre tramas de tipo solicitud de reconocimiento de comunicación (*data\_type = x06*) y tramas de activación del algoritmo de



simulación (*data\_type* = xFF). Cuando se recibe una trama del primer tipo se activa la transmisión directamente.

```

485         elsif (ether_type = x"0800") then
486             --
487             if (target_ip = source_ip) then
488                 --
489                 LED(2) <= '1';
490                 --
491                 if (data_type = x"06") then           -- solicita reconocimiento de comunicación
492                     --
493                     start_tx <= '1';
494                     --
495                 elsif (data_type = x"FF") then       -- solicita activación de algoritmo
496                     --
497                     wait_tx <= '1';
498                     START_ALGORITHM <= '1';
499                     --
500                 end if;
501                 --
502                 packet_arp <= '0';
503                 --
504                 --////////////////////////////////////////////////////
505                 --// cabecera MAC para IEEE 802.3 Ethernet (14 bytes) //
506                 --////////////////////////////////////////////////////
507
508                 header_tx(0) <= destin_mac(5);      -- DA --> MAC del destino (cliente)
509                 header_tx(1) <= destin_mac(4);
510                 header_tx(2) <= destin_mac(3);
511                 header_tx(3) <= destin_mac(2);
512                 header_tx(4) <= destin_mac(1);
513                 header_tx(5) <= destin_mac(0);
514
515                 header_tx(6) <= source_mac(5);      -- SA --> MAC de la fuente (fpga)
516                 header_tx(7) <= source_mac(4);
517                 header_tx(8) <= source_mac(3);
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601

```

En el caso de un datagrama UDP de tipo solicitud de activación del algoritmo se habilita la señal *start\_algorithm* y se espera a que el algoritmo finalice su procesado para iniciar la transmisión del datagrama UDP con los resultados.

```

614         if ((wait_tx = '1') and (READY_ALGORITHM = '1')) then
615             --
616             wait_tx <= '0';
617             start_tx <= '1';
618             --
619         end if;

```

El proceso *proc\_tx\_udp* controla la transmisión de tramas ARP y UDP de respuesta al cliente. La transmisión se activa con el pulso *start\_tx* que inicia el contador *counter\_tx*.

```

659     if (counter_tx = 0) then
660         --
661         if (start_tx = '1') then
662             --
663             counter_tx <= counter_tx + 1;
664             txen_reg <= '1';
665             reg_crc <= (others => '1');
666             --
667             txd_reg <= x"55";
668             --
669         else
670             --
671             txen_reg <= '0';
672             txd_reg <= (others => '0');
673             --
674         end if;
675         --
676     else
677         --
678         counter_tx <= counter_tx + 1;

714
715     if (counter_tx = 7 + size_header_tx + size_data_tx + 4) then counter_tx <= (others => '0');
716     else counter_tx <= counter_tx + 1;
717     end if;

747         counter_tx <= (others => '0');

```

En función del valor de *counter\_tx* se define el bus *txd\_reg* con los bytes de salida, comenzando con el preámbulo y delimitador de inicio de trama, seguido de la cabecera, los datos útiles o payload (y relleno de ceros cuando es necesario) y el campo CRC.

```

680     if (counter_tx <= 6) then txd_reg <= x"55";
681     elsif (counter_tx = 7) then txd_reg <= x"D5";
682     elsif (counter_tx <= 7 + size_header_tx + size_data_tx) then
683         --
684         if (counter_tx <= 7 + size_header_tx) then data_in := header_tx(conv_integer(counter_tx-8));
685         else
686             --
687             if (packet_arp = '1') then data_in := x"00";
688             else
689                 --
690                 if (data_type = x"06") then
691                     --
692                     if (counter_tx = 7 + size_header_tx + 1) then data_in := x"06";
693                     else data_in := x"00";
694                     end if;
695                 --
696                 elsif (data_type = x"FF") then
697                     --
698                     if (counter_tx(1 downto 0) = 2) then data_in := reg_data_tx(7 downto 0);
699                     elsif (counter_tx(1 downto 0) = 3) then data_in := reg_data_tx(15 downto 8);
700                     elsif (counter_tx(1 downto 0) = 0) then data_in := reg_data_tx(23 downto 16);
701                     elsif (counter_tx(1 downto 0) = 1) then data_in := reg_data_tx(31 downto 24);
702                     end if;
703                     --
704                     LED(3) <= '1';
705                     --
706                 end if;
707                 --
708             end if;
709             --
710         end if;

739         txd_reg <= data_in;
740         --
741     elsif (counter_tx = 7 + size_header_tx + size_data_tx + 1) then txd_reg <= reg_crc(7 downto 0);
742     elsif (counter_tx = 7 + size_header_tx + size_data_tx + 2) then txd_reg <= reg_crc(15 downto 8);
743     elsif (counter_tx = 7 + size_header_tx + size_data_tx + 3) then txd_reg <= reg_crc(23 downto 16);
744     elsif (counter_tx = 7 + size_header_tx + size_data_tx + 4) then
745         --
746         txd_reg <= reg_crc(31 downto 24);
747         counter_tx <= (others => '0');
748         --
749     end if;

```

La trama UDP transmitida está formada únicamente por un byte de valor x06 si es la respuesta a la solicitud de reconocimiento de comunicación. Si es la respuesta a una solicitud de activación de algoritmo la trama transmitida contiene los datos contenidos en la memoria RAM de doble puerta YT.

```

651         if (counter_tx(1 downto 0) = "01") then
652             --
653             reg_data_tx <= DOUT_RAM_YT;
654             --
655         end if;
656         --
657         addr_ram_yt_reg <= conv_std_logic_vector(conv_integer(counter_tx) - (2 + size_header_tx), 12);

757     ADDR_RAM_YT <= addr_ram_yt_reg(9 downto 2);

```

Para el cálculo del campo CRC se emplea el registro de desplazamiento *reg\_crc* que se rellena inicialmente con 1s y en el que se van introduciendo los bytes de la trama de salida comenzando por el LSB. Si el bit introducido es 1 se realiza la operación xor con el polinomio generador *g* del CRC-32.

```

665         reg_crc <= (others => '1');

712         reg_crc_var := reg_crc;
713         --
714         for i in 0 to 7 loop
715             --
716             if ((reg_crc_var(31) xor data_in(i)) = '0') then
717                 --
718                 reg_crc_var := reg_crc_var(30 downto 0) & '0';
719                 --
720             else
721                 --
722                 reg_crc_var := (reg_crc_var(30 downto 0) & '0') xor g;
723                 --
724             end if;
725             --
726         end loop;

```

Finalmente se obtiene como campo CRC los últimos cuatro bytes complementados a 1. Se invierte el orden de los bits en el registro para que sean transmitidos en primer lugar los bits más significativos.

```
728         if (counter_tx = 7 + size_header_tx + size_data_tx) then
729             --
730             for index in 0 to 31 loop
731                 --
732                 reg_crc(index) <= not(reg_crc_var(31-index));
733                 --
734             end loop;
735             --
736         else reg_crc <= reg_crc_var;
737         end if;
```

### Entidad *algorithm*

Contiene la descripción de los puertos de la arquitectura que controla el algoritmo.

```
16  entity algorithm is
17      port( CLK: in std_logic;
18            RESET: in std_logic;
19            START: in std_logic;
20            --
21            ADDR_RAM_A: out std_logic_vector(M-1 downto 0);
22            DOUT_RAM_A: in reg_N_32_type;
23            --
24            ADDR_RAM_BVE: out std_logic_vector(M-1 downto 0);
25            DOUT_RAM_BVE: in std_logic_vector(31 downto 0);
26            --
27            ADDR_RAM_C: out std_logic_vector(M-1 downto 0);
28            DOUT_RAM_C: in std_logic_vector(31 downto 0);
29            --
30            DVE: in std_logic_vector(31 downto 0);
31            --
32            H: in std_logic_vector(31 downto 0);
33            H_DIV_2: in std_logic_vector(31 downto 0);
34            H_DIV_3: in std_logic_vector(31 downto 0);
35            H_DIV_6: in std_logic_vector(31 downto 0);
36            --
37            ORDER: in std_logic_vector(M-1 downto 0);
38            --
39            WR_RAM_YT: out std_logic;
40            ADDR_RAM_YT: out std_logic_vector(7 downto 0);
41            DIN_RAM_YT: out std_logic_vector(31 downto 0);
42            --
43            RDY: out std_logic);
44  end algorithm;
```

### Arquitectura *algorithm\_arch*

La arquitectura *algorithm\_arch* describe el comportamiento del algoritmo rk4.

Dentro de la arquitectura el proceso *proc\_stages* genera el contador de muestras *counter\_samples*, el contador *counter\_stages* que contabiliza las etapas del algoritmo para cada muestra y el pulso de final de etapa *rdy\_stage*.

```

183     rdy_stage <= '0';
184     --
185     if (enable_load_xaux = '1') then
186         --
187         if (counter_rows_xaux = ORDER) then rdy_stage <= '1';
188         end if;
189         --
190     end if;
191     --
192     if (rdy_stage = '1') then
193         --
194         if (counter_stages = 3) then
195             --
196             counter_stages <= 0;
197             counter_samples <= counter_samples + 1;
198             --
199             else counter_stages <= counter_stages + 1;
200         end if;
201         --
202     end if;
203     --
204     if (START = '1') then
205         --
206         counter_stages <= 0;
207         counter_samples <= (others => '0');
208         --
209     end if;

```

La sentencia *inst\_generate\_float\_mult\_ax* genera N multiplicadores en coma flotante para el producto de matrices  $res1 = A * X(t)$  en la etapa 0 y  $res1 = A * X'$  en las etapas 1, 2 ó 3.

```

219     inst_generate_float_mult_ax: for i in 0 to N-1 generate
220
221         inst_float_mult_ax_i: float_mult port map(
222             A => DOUT_RAM_A(i),
223             B => b_float_mult_ax(i),
224             OPERATION_ND => start_float_mult_ax,
225             CLK => CLK,
226             RESULT => result_float_mult_ax(i),
227             RDY => rdy_float_mult_ax(i));
228
229     end generate;

```

La sentencia *inst\_generate\_float\_mult\_ax* genera N-1 sumadores en coma flotante para el producto de matrices *res1*. Con la sentencia *inst\_generate\_upper\_sum\_i* se instancian los sumadores conectados con los multiplicadores y con *inst\_generate\_lower\_sum\_i* se genera el resto de sumadores en pipeline.

```

231 inst_generate_float_sum_ax: for i in 0 to N-2 generate
232
233     inst_generate_lower_sum_i: if (i <= N/2-2) generate
234         --
235         a_float_sum_ax(i) <= result_float_sum_ax(2*i+1);
236         b_float_sum_ax(i) <= result_float_sum_ax(2*i+2);
237         --
238         start_float_sum_ax(i) <= rdy_float_sum_ax(2*i+1);
239         --
240     end generate;
241
242     inst_generate_upper_sum_i: if (i > N/2-2) generate
243         --
244         a_float_sum_ax(i) <= result_float_mult_ax(2*i-N+2);
245         b_float_sum_ax(i) <= result_float_mult_ax(2*i-N+3);
246         --
247         start_float_sum_ax(i) <= rdy_float_mult_ax(2*i-N+2);
248         --
249     end generate;
250
251     inst_float_sum_ax_i: float_sum port map(
252         A => a_float_sum_ax(i),
253         B => b_float_sum_ax(i),
254         OPERATION_ND => start_float_sum_ax(i),
255         CLK => CLK,
256         RESULT => result_float_sum_ax(i),
257         RDY => rdy_float_sum_ax(i));
258
259 end generate;

```

En el proceso *proc\_ax* se genera la señal de control *start\_float\_mult\_ax* para los multiplicadores y el operando *b\_float\_mult\_ax* que se carga al final de cada etapa anterior con los valores de  $X(t)$  para la etapa 0 ó con  $X'$  para el resto de etapas.

```

273 start_float_mult_ax <= '0';
274 --
275 if ((START = '1') or
276     ((rdy_float_mult_ax(0) = '1') and (counter_rows_a > 0)) or
277     ((rdy_stage = '1') and not((counter_samples = S-1) and (counter_stages = 3)))) then
278     start_float_mult_ax <= '1';
279 end if;
280 --
281 if (start_float_mult_ax = '1') then
282     --
283     if (counter_rows_a = ORDER) then counter_rows_a <= 0;
284     else counter_rows_a <= counter_rows_a + 1;
285     end if;
286     --
287 end if;
288 --
289 if (enable_load_xt = '1') then
290     --
291     b_float_mult_ax(counter_rows_r_xt) <= dout_dpram_xt(0);
292     --
293 elsif (enable_load_xaux = '1') then
294     --
295     b_float_mult_ax(counter_rows_xaux) <= dout_ram_xaux;
296     --
297 end if;

```

Con el contador *counter\_rows\_a* se direccionan las memorias *dpram\_Nx32* para leer los coeficientes de la matriz A.

```
217 ADDR_RAM_A <= conv_std_logic_vector(counter_rows_a, M);
```

En el proceso *proc\_ax* también se añade el control para asegurar que al lanzarse de nuevo el algoritmo los valores de  $X(0)$  son nulos. Para ello se inicializa el operando *b\_float\_mult\_ax* de los multiplicadores con el pulso de fin del algoritmo *rdy\_reg*.

```
299         if (rdy_reg = '1') then
300             --
301             b_float_mult_ax <= (others => (others => '0'));
302             --
303         end if;
```

Con *inst\_float\_sum\_axbve* se instancia el sumador en coma flotante para la operación  $K = res1 + BV_e$ .

```
313 inst_float_sum_axbve: float_sum port map(
314     A => a_float_sum_axbve,
315     B => DOUT_RAM_BVE,
316     OPERATION_ND => start_float_sum_axbve,
317     CLK => CLK,
318     RESULT => result_float_sum_axbve,
319     RDY => rdy_float_sum_axbve);
```

En el proceso *proc\_axbve* se genera la señal de activación *start\_float\_sum\_axbve* del sumador. El operando *a\_float\_sum\_axbve* se conecta con la salida del correspondiente sumador del bloque anterior en función del orden del sistema.

```

335     case conv_integer(ORDER) is
336         --
337         when 0 | 1 => index_a_float_sum_ax <= 7;
338         when 2 | 3 => index_a_float_sum_ax <= 3;
339         when 4 | 5 | 6 | 7 => index_a_float_sum_ax <= 1;
340         when others => index_a_float_sum_ax <= 0;
341         --
342     end case;
343     --
344     a_float_sum_axbve <= result_float_sum_ax(index_a_float_sum_ax);
345     start_float_sum_axbve <= rdy_float_sum_ax(index_a_float_sum_ax);
346     --
347     if (start_float_sum_axbve = '1') then
348         --
349         if (counter_rows_bve = ORDER) then counter_rows_bve <= 0;
350         else counter_rows_bve <= counter_rows_bve + 1;
351         end if;
352         --
353     end if;

```

Con el contador *counter\_row\_bve* se direcciona la memoria *dpram\_Nx32* para leer los coeficientes de la matriz  $BV_e$ .

```

311     ADDR_RAM_BVE <= conv_std_logic_vector(counter_rows_bve, M);

```

Con *inst\_float\_mult\_xaux* y *inst\_float\_sum\_xaux* se instancian respectivamente el multiplicador y el sumador en coma flotante para la operación  $X' = X(t) + K * (h \text{ or } h/2 \text{ or } h/2 \text{ or } h)$ .

```

361     inst_float_mult_xaux: float_mult port map(
362         A => result_float_sum_axbve,
363         B => b_float_mult_xaux,
364         OPERATION_ND => rdy_float_sum_axbve,
365         CLK => CLK,
366         RESULT => result_float_mult_xaux,
367         RDY => rdy_float_mult_xaux);
368
369     inst_float_sum_xaux: float_sum port map(
370         A => result_float_mult_xaux,
371         B => b_float_sum_xaux,
372         OPERATION_ND => rdy_float_mult_xaux,
373         CLK => CLK,
374         RESULT => result_float_sum_xaux,
375         RDY => rdy_float_sum_xaux);

```

El resultado  $X'$  se guarda en una memoria auxiliar *ram\_Nx32*.



```

377     addr_ram_xaux <= conv_std_logic_vector(counter_rows_xaux, M);
378
379     inst_ram_xaux: ram_Nx32 port map(
380         CLK => CLK,
381         WR => rdy_float_sum_xaux,
382         ADDR => addr_ram_xaux,
383         DIN => result_float_sum_xaux,
384         DOUT => dout_ram_xaux);

```

En el proceso *proc\_xaux* se generan los operandos del multiplicador y del sumador y las señales de control para la escritura en la memoria *ram\_Nx32*.

```

400     case counter_stages is
401         --
402         when 0 => b_float_mult_xaux <= H_DIV_2;
403         when 1 => b_float_mult_xaux <= H_DIV_2;
404         when others => b_float_mult_xaux <= H;
405         --
406     end case;
407     --
408     b_float_sum_xaux <= dout_dpram_xt(0);
409     --
410     if (rdy_float_sum_xaux = '1') then
411         --
412         if (counter_rows_xaux = ORDER) then
413             --
414             counter_rows_xaux <= 0;
415             enable_load_xaux <= '1';
416             --
417             else counter_rows_xaux <= counter_rows_xaux + 1;
418             end if;
419         --
420     end if;
421     --
422     if (enable_load_xaux = '1') then
423         --
424         if (counter_rows_xaux = ORDER) then
425             --
426             counter_rows_xaux <= 0;
427             enable_load_xaux <= '0';
428             --
429             else counter_rows_xaux <= counter_rows_xaux + 1;
430             end if;
431         --
432     end if;

```

Con *inst\_float\_mult\_xt* y *inst\_float\_sum\_xt* se instancian el multiplicador y el sumador en coma flotante para la operación  $X(t+1) = X(t) + K * (h/6 \text{ or } h/3 \text{ or } h/3 \text{ or } h/6)$ .

```
440     inst_float_mult_xt: float_mult port map(  
441       A => result_float_sum_axbve,  
442       B => b_float_mult_xt,  
443       OPERATION_ND => rdy_float_sum_axbve,  
444       CLK => CLK,  
445       RESULT => result_float_mult_xt,  
446       RDY => rdy_float_mult_xt);  
447  
448     inst_float_sum_xt: float_sum port map(  
449       A => result_float_mult_xt,  
450       B => b_float_sum_xt,  
451       OPERATION_ND => rdy_float_mult_xt,  
452       CLK => CLK,  
453       RESULT => result_float_sum_xt,  
454       RDY => rdy_float_sum_xt);
```

El resultado se guarda en dos memorias dpram\_Nx32 para X(t) y X(t+1).

```
456     inst_generate_xt_rams: for i in 0 to 1 generate  
457  
458       addr_dpram_w_xt(i) <= conv_std_logic_vector(counter_rows_w_xt, M);  
459       addr_dpram_r_xt(i) <= conv_std_logic_vector(counter_rows_r_xt, M);  
460  
461       inst_xt_dpram_asyn_i: dpram_Nx32_asyn port map(  
462         CLK_W => CLK,  
463         WR => wr_dpram_xt(i),  
464         ADDR_W => addr_dpram_w_xt(i),  
465         DIN => din_dpram_xt(i),  
466         --  
467         CLK_R => CLK,  
468         ADDR_R => addr_dpram_r_xt(i),  
469         DOUT => dout_dpram_xt(i));  
470  
471     end generate;
```

En el proceso *proc\_xt* se controlan los operandos del multiplicador y del sumador y las señales de control para la lectura de las memorias X(t) y X(t+1).

```

493     case counter_stages is
494         --
495         when 0 | 3 => b_float_mult_xt <= H_DIV_6;
496         when 1 | 2 => b_float_mult_xt <= H_DIV_3;
497         --
498     end case;
499     --
500     b_float_sum_xt <= dout_dpram_xt(1);
501     --
502     if (rdy_float_mult_xt = '1') then
503         --
504         if (counter_rows_r_xt = ORDER) then counter_rows_r_xt <= 0;
505         else counter_rows_r_xt <= counter_rows_r_xt + 1;
506         end if;
507         --
508     end if;
509     --
510     if ((rdy_float_sum_xt = '1') and (counter_rows_xaux = ORDER) and (counter_stages = 3)) then
511         --
512         enable_load_xt <= '1';
513         --
514     end if;
515     --
516     if (enable_load_xt = '1') then
517         --
518         if (counter_rows_r_xt = ORDER) then
519             --
520             counter_rows_r_xt <= 0;
521             enable_load_xt <= '0';
522             --
523         else counter_rows_r_xt <= counter_rows_r_xt + 1;
524         end if;
525         --
526     end if;

```

En el proceso *proc\_xt* también se controla la escritura en la *dpram\_Nx32* del resultado  $X(t+1)$  en cada etapa y del resultado  $X(t)$  sólo en la etapa 3 para que esté preparada para la siguiente muestra.

```

528     if ((wr_dpram_xt(0) = '1') or (wr_dpram_xt(1) = '1')) then
529         --
530         if (counter_rows_w_xt = ORDER) then counter_rows_w_xt <= 0;
531         else counter_rows_w_xt <= counter_rows_w_xt + 1;
532         end if;
533         --
534     end if;
535     --
536     wr_dpram_xt(1) <= rdy_float_sum_xt;
537     din_dpram_xt(1) <= result_float_sum_xt;
538     --
539     if (counter_stages = 3) then
540         --
541         wr_dpram_xt(0) <= rdy_float_sum_xt;
542         din_dpram_xt(0) <= result_float_sum_xt;
543         --
544     end if;

```

Al final de cada ejecución del algoritmo se resetean las dos memorias para  $X(t)$  y  $X(t+1)$  y que estén preparadas para un nuevo procesado del algoritmo.

```

546         if (rdy_reg = '1') then
547             --
548             wr_dpram_xt <= (others => '1');
549             din_dpram_xt <= (others => (others => '0'));
550             --
551             reset_dpram_xt <= '1';
552             --
553         end if;
554         --
555         if (reset_dpram_xt = '1') then
556             --
557             wr_dpram_xt <= (others => '1');
558             din_dpram_xt <= (others => (others => '0'));
559             --
560             if (counter_rows_w_xt = N-1) then
561                 --
562                 counter_rows_w_xt <= 0;
563                 wr_dpram_xt <= (others => '0');
564                 reset_dpram_xt <= '0';
565                 --
566             else counter_rows_w_xt <= counter_rows_w_xt + 1;
567             end if;
568             --
569         end if;

```

En *inst\_float\_mult\_cx* y *inst\_float\_sum\_cxdve* se instancian el multiplicador y el sumador en coma flotante para la operación  $V_s(t) = C * X(t+1) + DV_e$ .

```

579     inst_float_mult_cx: float_mult port map(
580         A => a_float_mult_cx,
581         B => DOUT_RAM_C,
582         OPERATION_ND => start_float_mult_cx,
583         CLK => CLK,
584         RESULT => result_float_mult_cx,
585         RDY => rdy_float_mult_cx);
586
587     inst_float_sum_cxdve: float_sum port map(
588         A => a_float_sum_cxdve,
589         B => b_float_sum_cxdve,
590         OPERATION_ND => start_float_sum_cxdve,
591         CLK => CLK,
592         RESULT => result_float_sum_cxdve,
593         RDY => rdy_float_sum_cxdve);

```

En el proceso *inst\_process\_cxdve* se generan las señales de activación *start\_float\_mult\_cx* y *start\_float\_sum\_cxdve* del multiplicador y del sumador respectivamente. También se obtienen los operandos *a\_float\_mult\_cx* y *a\_float\_sum\_cxdve* a partir de los resultados de las operaciones anteriores.

```

620     start_float_mult_cx <= '0';
621     start_float_sum_cxdve <= '0';
622     --
623     if ((rdy_float_sum_xt = '1') and (counter_stages = 3)) then
624         --
625         start_float_mult_cx <= '1';
626         --
627         a_float_mult_cx <= result_float_sum_xt;
628         --
629         if (counter_rows_c = ORDER) then counter_rows_c <= 0;
630         else counter_rows_c <= counter_rows_c + 1;
631         end if;
632         --
633     end if;
634     --
635     if (rdy_float_mult_cx = '1') then
636         --
637         start_float_sum_cxdve <= '1';
638         --
639         if (counter_rows_cx = ORDER) then counter_rows_cx <= 0;
640         else counter_rows_cx <= counter_rows_cx + 1;
641         end if;
642         --
643         a_float_sum_cxdve <= result_float_mult_cx;
644         --
645         if (counter_rows_cx = 0) then b_float_sum_cxdve <= (others => '0');
646         else b_float_sum_cxdve <= result_float_sum_cxdve;
647         end if;
648         --
649     end if;

```

El contador *counter\_rows\_cx* controla el direccionamiento de la memoria *dpram\_Nx32* para la lectura de los coeficientes de la matriz C.

```

577     ADDR_RAM_C <= conv_std_logic_vector(counter_rows_c, M);

```

La última suma es para añadir el valor  $DV_e$  al valor acumulado.

```

654     if (rdy_float_sum_cxdve = '1') then
655         --
656         if (counter_rows_cxdve = ORDER) then
657             --
658             start_float_sum_cxdve <= '1';
659             --
660             a_float_sum_cxdve <= DVE;
661             b_float_sum_cxdve <= result_float_sum_cxdve;
662             --
663         end if;

```

El proceso *proc\_cxdve* también controla la escritura en la memoria *dpram\_256x32* del resultado de la simulación y genera el pulso *rdy\_reg* para indicar el final del algoritmo.

```
651     WR_RAM_YT <= '0';
652     rdy_reg <= '0';
653     --
654     if (rdy_float_sum_cxdve = '1') then
655         --
656
665         if (counter_rows_cxdve = ('0' & ORDER) + 1) then
666             --
667             counter_rows_cxdve <= 0;
668             --
669             WR_RAM_YT <= '1';
670             DIN_RAM_YT <= result_float_sum_cxdve(31 downto 0);
671             ADDR_RAM_YT <= counter_samples - 1;
672             --
673             if (counter_samples = S) then rdy_reg <= '1';
674             end if;
675             --
676             else counter_rows_cxdve <= counter_rows_cxdve + 1;
677             end if;
678             --
679         end if;
```

### Paquete *algorithm\_package*

Contiene la parte declarativa y el cuerpo del paquete *algorithm\_package* incluido en la librería *mylib*.

En la parte declarativa se definen los parámetros *N* ( $N_m$  en las especificaciones de requisitos), *M* y *S* del algoritmo y se declara el tipo *reg\_N\_32\_type* para los puertos arrays de buses de datos entre las entidades *udp* y *algorithm*.

```
13 package algorithm_package is
14
15     constant N: integer := 16;           -- orden máximo del problema de ecuaciones diferenciales
16     constant M: integer := 4;           -- M = log2(N)
17     constant S: integer := 200;        -- número de muestras
18
19     type reg_N_32_type is array (0 to N-1) of std_logic_vector(31 downto 0);
20
21 end algorithm_package;
22
23 package body algorithm_package is
24 end algorithm_package;
```

### 5.3. Memorias

En la siguiente tabla se resumen las características principales de las memorias empleadas en el diseño. En el resto de la sección se justifican las causas de su elección y se exponen su funcionalidad dentro de la arquitectura y las descripciones VHDL que definen su comportamiento.

Matrices de la arquitectura	Ancho de palabra (bits)	Ancho de bus de direcciones (bits)	Tipo de memoria	Tipo de lectura	Componente VHDL
A, BV <sub>e</sub> , C	32	16	Dual port RAM	Síncrona	dpram_Nx32
X(t), X(t+1)	32	16	Dual port RAM	Asíncrona	dpram_Nx32_asyn
X'	32	16	Single port RAM	Asíncrona	ram_Nx32
V <sub>s</sub> (t)	32	256	Dual port RAM	Síncrona	dpram_256x32

**Tabla 5.1. Características de las memorias empleadas en la arquitectura**

Para las matrices A, BV<sub>e</sub> y C se emplean RAMs de doble puerto que se escriben con el reloj de 125 MHz en el controlador Ethernet y que se leen en el bloque del algoritmo con el reloj de 100 MHz.

En el caso de las memorias X(t) y X(t+1) el reloj de ambas puertas es el mismo pero se emplea el doble puerto porque se necesita leer y escribir simultáneamente.

La matriz X(t) se actualiza con los valores calculados de X(t+1) en la etapa 3 del algoritmo. La matriz X(t) se lee al final de la etapa 3 para cargar los operandos de los multiplicadores usados en la operación A\*X(t) durante la etapa 0 de la siguiente muestra. Se lee también en las etapas 0, 1 y 2 para cargar uno de los operandos del sumador y obtener X'.

La matriz  $X(t+1)$  actúa de acumulador: se leen sus elementos en todas las etapas, se les suma un valor y se escriben de nuevo en la matriz. Al pasar a la etapa 0 de la siguiente muestra del algoritmo ya contiene  $X(t)$  y puede reiniciar la acumulación para la nueva muestra.

Para la matriz  $V_s(t)$  se emplea una RAM de doble puerto que se escribe con el reloj de 100 MHz en el bloque del algoritmo y se lee con el reloj RXCLK en el controlador Ethernet.

A continuación se añaden las descripciones VHDL para las memorias empleadas.

### **Memoria dpram\_Nx32**

```
16 entity dpram_Nx32 is
17
18     port( CLK_W: in std_logic;
19           WR: in std_logic;
20           ADDR_W: in std_logic_vector(M-1 downto 0);
21           DIN: in std_logic_vector(31 downto 0);
22           --
23           CLK_R: in std_logic;
24           ADDR_R: in std_logic_vector(M-1 downto 0);
25           DOUT: out std_logic_vector(31 downto 0));
26
27 end dpram_Nx32;
```



```

29 architecture dpram_Nx32_arch of dpram_Nx32 is
30
31     type ram_type is array (0 to N-1) of std_logic_vector(31 downto 0);
32
33     signal sram: ram_type := (others => (others => '0'));
34
35 begin
36
37     process(CLK_W)
38     begin
39         --
40         if (CLK_W'event and CLK_W = '1') then
41             --
42             if (WR = '1') then sram(conv_integer(ADDR_W)) <= DIN;
43                 end if;
44             --
45         end if;
46         --
47     end process;
48
49     process(CLK_R)
50     begin
51         --
52         if (CLK_R'event and CLK_R = '1') then
53             --
54             DOUT <= sram(conv_integer(ADDR_R));
55             --
56         end if;
57         --
58     end process;
59
60 end dpram_Nx32_arch;

```

### Memoria dpram\_Nx32\_asyn

```

16 entity dpram_Nx32_asyn is
17
18     port( CLK_W: in std_logic;
19           WR: in std_logic;
20           ADDR_W: in std_logic_vector(M-1 downto 0);
21           DIN: in std_logic_vector(31 downto 0);
22           --
23           ADDR_R: in std_logic_vector(M-1 downto 0);
24           DOUT: out std_logic_vector(31 downto 0));
25
26 end dpram_Nx32_asyn;

```

```

28 architecture dpram_Nx32_asyn_arch of dpram_Nx32_asyn is
29
30     type ram_type is array (0 to N-1) of std_logic_vector(31 downto 0);
31
32     signal sram: ram_type := (others => (others => '0'));
33
34 begin
35
36     process(CLK_W)
37     begin
38         --
39         if (CLK_W'event and CLK_W = '1') then
40             --
41             if (WR = '1') then sram(conv_integer(ADDR_W)) <= DIN;
42                 end if;
43             --
44         end if;
45         --
46     end process;
47
48     DOUT <= sram(conv_integer(ADDR_R));
49
50 end dpram_Nx32_asyn_arch;

```

### Memoria ram\_Nx32

```

16 entity ram_Nx32 is
17     port( CLK: in std_logic;
18           WR: in std_logic;
19           ADDR: in std_logic_vector(M-1 downto 0);
20           DIN: in std_logic_vector(31 downto 0);
21           DOUT: out std_logic_vector(31 downto 0));
22 end ram_Nx32;

```

```

24 architecture arch_ram_Nx32 of ram_Nx32 is
25
26     type ram_type is array (0 to N-1) of std_logic_vector(31 downto 0);
27
28     signal sram: ram_type := (others => (others => '0'));
29
30 begin
31
32     process(CLK)
33     begin
34         --
35         if (CLK'event and CLK = '1') then
36             --
37             if (WR = '1') then sram(conv_integer(ADDR)) <= DIN;
38                 end if;
39             --
40         end if;
41         --
42     end process;
43
44     DOUT <= sram(conv_integer(ADDR));
45
46 end arch_ram_Nx32;

```

**Memoria dpram\_256x32**

```

16 entity dpram_256x32 is
17     port( CLK_W: in std_logic;
18           WR: in std_logic;
19           ADDR_W: in std_logic_vector(7 downto 0);
20           DIN: in std_logic_vector(31 downto 0);
21           CLK_R: in std_logic;
22           ADDR_R: in std_logic_vector(7 downto 0);
23           DOUT: out std_logic_vector(31 downto 0));
24 end dpram_256x32;

26 architecture dpram_256x32_arch of dpram_256x32 is
27
28     type ram_type is array (0 to 255) of std_logic_vector(31 downto 0);
29
30     signal sram: ram_type := (others => (others => '0'));
31
32 begin
33
34     process(CLK_W)
35     begin
36         --
37         if (CLK_W'event and CLK_W = '1') then
38             --
39             if (WR = '1') then sram(conv_integer(ADDR_W)) <= DIN;
40             end if;
41             --
42         end if;
43         --
44     end process;
45
46     process(CLK_R)
47     begin
48         --
49         if (CLK_R'event and CLK_R = '1') then
50             --
51             DOUT <= sram(conv_integer(ADDR_R));
52             --
53         end if;
54         --
55     end process;
56
57 end dpram_256x32_arch;

```

**5.4. Cores**

Las operaciones de suma y producto en coma flotante se realizan con el bloque *LogiCORE IP Floating-Point Operator* de Xilinx que se configura como sumador o multiplicador según el caso.

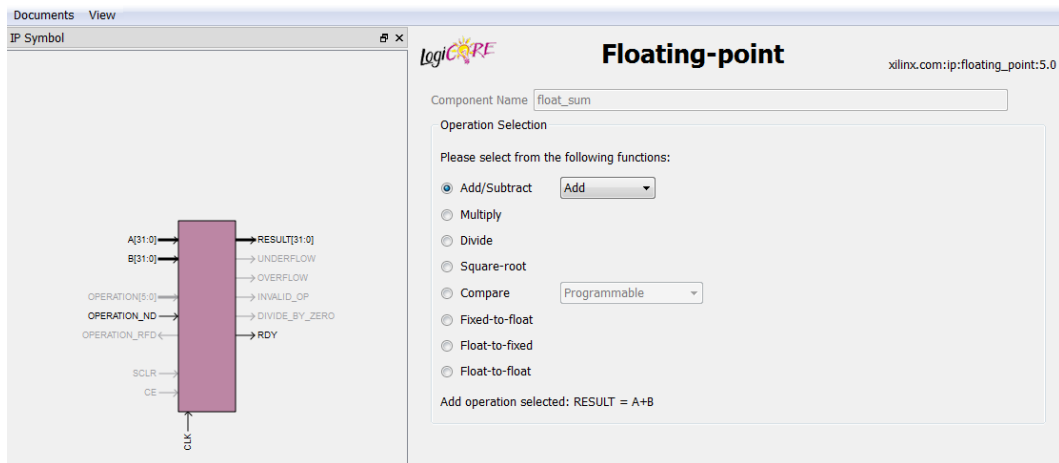


Figura 5.1. Ventana de configuración del *logiCORE Floating-point Operator*

Se selecciona para los operandos y resultado una precisión simple en coma flotante que sigue el estándar IEEE-754 con 32 bits (8 bits para el exponente y 24 bits para la parte fraccionaria con un bit de signo).

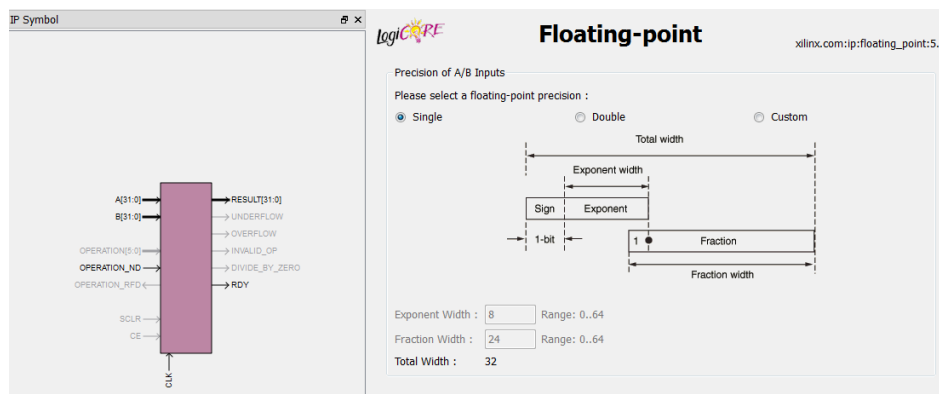


Figura 5.2. Definición de la precisión del *logiCORE Floating-point Operator*

Se configura la latencia del componente, es decir, el número de ciclos que tarda en mostrar el resultado desde la señal de activación de la operación. En la Figura 5.3 se muestra la simulación de un sumador con latencia de 7 ciclos, que se activa con la señal *start\_float\_sum* y

los valores de los operandos *a\_float\_sum* y *b\_float\_sum* y muestra el resultado en el bus *result\_float\_sum* sincronizado con la señal *rdy\_float\_sum*.

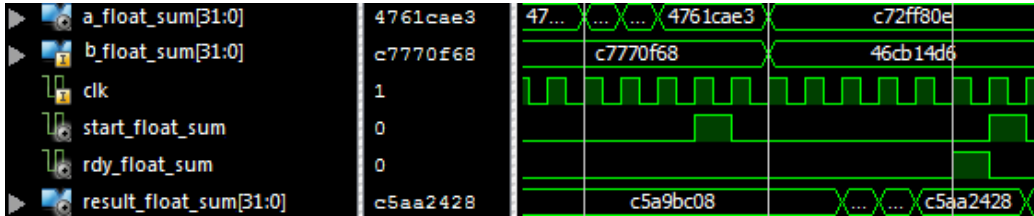


Figura 5.3. Simulación de un sumador con latencia 7

En los sumadores la latencia se puede definir en un rango invariable de 0 a 12. En los multiplicadores el rango varía con el tipo de optimización seleccionada para la implementación. En general, la elección de la latencia está condicionada por el valor de la frecuencia a la que procesa el core y por el consumo de LUTs y FFs de la FPGA.

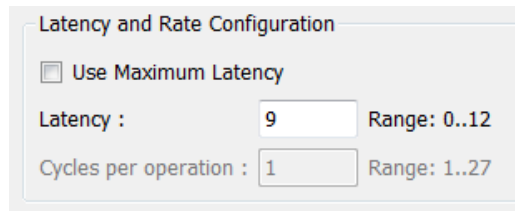


Figura 5.4. Configuración de la latencia del sumador *logiCORE Floating-point*

En este diseño se define una latencia igual para sumadores que para multiplicadores. Esto es debido a que las operaciones en los distintos bloques se realizan en pipeline y para su correcta realización la latencia de un componente debe ser menor o igual que la latencia del bloque que le suministra los datos. En la simulación puede observarse un multiplicador en pipeline con un sumador. Si el sumador tuviese una latencia superior no podría procesar todos los datos extraídos por el multiplicador durante la activación de la señal *rdy\_float\_mult*.

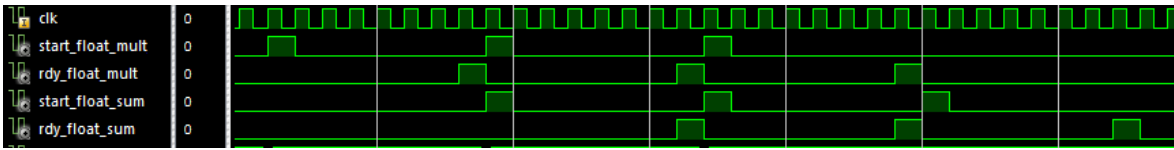


Figura 5.5. Control de la latencia de los *logiCORE Floating-point Operators* en pipeline

La implementación de los cores configurados como sumadores sobre la FPGA se realiza siempre con lógica (LUTs y FFs) pero en el caso de los multiplicadores se puede elegir el tipo de recurso a utilizar. En este diseño se necesitan  $N_m+3 = 19$  multiplicadores cada uno de los cuales emplea un bloque DSP, haciendo un uso medio de los 58 bloques DSP48A que posee la FPGA. Puede observarse que con esta opción el rango de la latencia se define de 0 a 9. A mayor uso de bloques DSP48A menor es el gasto realizado de LUTs y FFs.

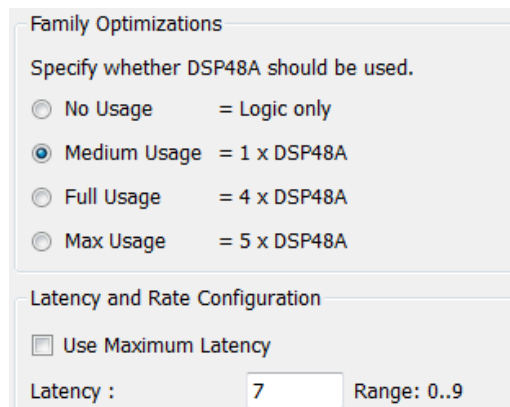


Figura 5.6. Recursos empleados del multiplicador *logiCORE Floating-point*

La ventana de configuración de los cores ofrece una gráfica para la estimación de los recursos empleados en función de la latencia. Para el multiplicador se puede observar que con un reloj de 100 MHz se puede bajar la latencia a 7 ciclos de reloj con un consumo aproximado de 400 LUTs y 450 FFs. Puede remarcarse en la gráfica de la izquierda que una latencia de 7 ciclos no es perjudicial en los sumadores.

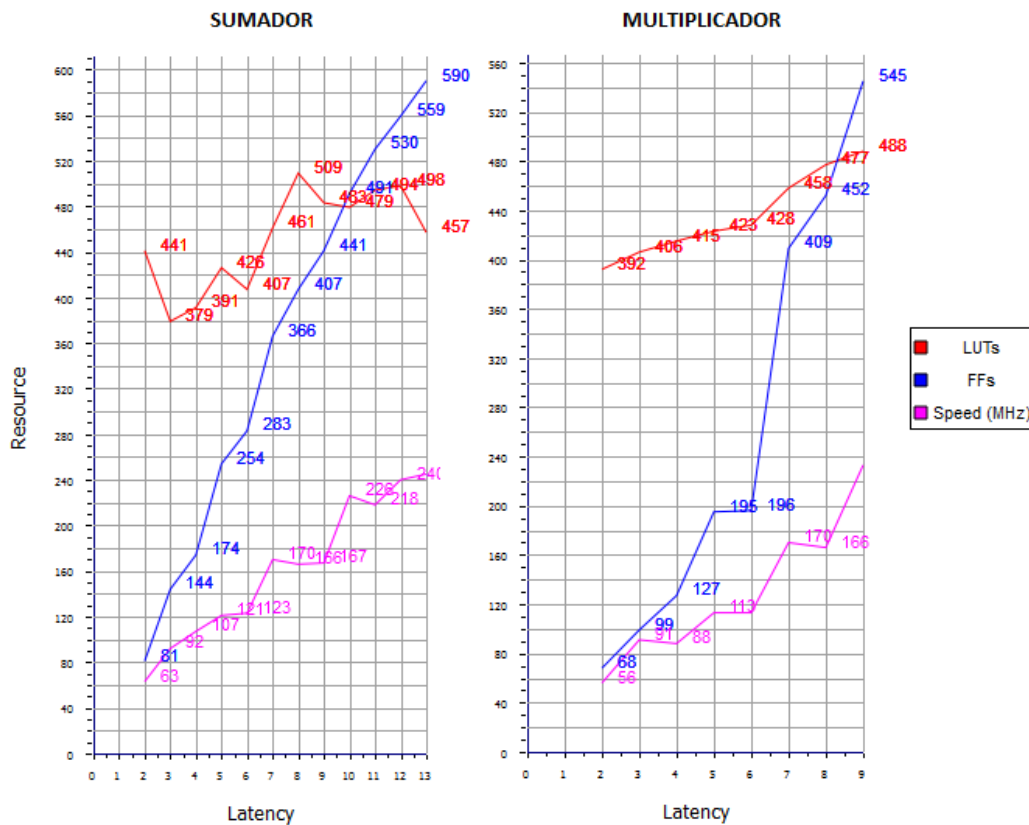


Figura 5.7. Estimación de los recursos en el *logiCORE Floating-point Operator*

## 5.5. Relojes

### Entrada de datos en el controlador Ethernet

En la recepción de datos Ethernet se deben evitar retardos en la señal de control *rxdrv* y en el bus de datos *rxd* con respecto al flanco de subida del reloj *rxclk* de 125 MHz por lo que interesa que los flip-flops que registran estas señales de entrada estén adyacentes a los pads de entrada. Para ello en la arquitectura *udp\_arch* se instancian flip-flops FDCE que registran las señales de entrada y se les fija el atributo IOB a TRUE.

```
107 attribute IOB: string;
108 attribute IOB of generate_FDCE_rxd: label is "TRUE";
109 attribute IOB of inst_FDCE_rxdv: label is "TRUE";

115 generate_FDCE_rxd: for i in 0 to 7 generate
116   attribute IOB of inst_FDCE_rxd_i: label is "TRUE";
117 begin
118   inst_FDCE_rxd_i: FDCE
119     generic map(
120       INIT => '0')
121     port map(
122       D => RXD(i),
123       C => RXCLK,
124       CE => '1',
125       CLR => '0',
126       Q => rxd_reg(i));
127 end generate;
128
129 inst_FDCE_rxdv: FDCE
130   generic map(
131     INIT => '0')
132   port map(
133     D => RXDV,
134     C => RXCLK,
135     CE => '1',
136     CLR => '0',
137     Q => rxdv_reg);
```

### Salida de datos en el controlador Ethernet

Para la salida Ethernet en la arquitectura *udp\_arch* la señal de control *txen* y el bus de datos *txd* deben estar sincronizados con el flanco de subida del reloj *clk\_gmii* de 125 MHz generado en la FPGA. Para ello *txen* y *txd* se producen con el reloj de entrada *rxclk* de 125 MHz y *clk\_gmii* se sincroniza con *rxclk* con la mayor precisión empleando el componente ODDR2 (Xilinx, 2014). Además se registran las salidas con flip-flops adyacentes a los pads de salida.

```
110 attribute IOB of generate_FDCE_txd: label is "TRUE";
111 attribute IOB of inst_FDCE_txen: label is "TRUE";
```



```

759 not_RXCLK <= not (RXCLK);
760 not_reset <= not (RESET);
761
762 inst_ODDR2_clk_gmii: ODDR2
763     generic map(
764         DDR_ALIGNMENT => "NONE",
765         INIT => '0',
766         SRTYPE => "SYNC")
767     port map(
768         Q => CLK_GMII,
769         C0 => RXCLK,
770         C1 => not_RXCLK,
771         D0 => '1',
772         D1 => '0',
773         CE => '1',
774         R => not_reset,
775         S => '0');

777 generate_FDCE_txd: for i in 0 to 7 generate
778     attribute IOB of inst_FDCE_txd_i: label is "TRUE";
779     begin
780         inst_FDCE_txd_i: FDCE
781             generic map(
782                 INIT => '0')
783             port map(
784                 D => txd_reg(i),
785                 C => RXCLK,
786                 CE => '1',
787                 CLR => '0',
788                 Q => TXD(i));
789     end generate;

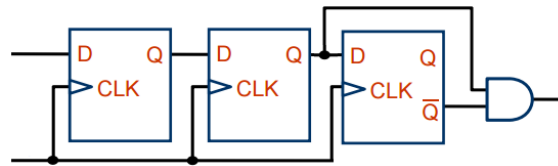
790
791 inst_FDCE_txen: FDCE
792     generic map(
793         INIT => '0')
794     port map(
795         D => txen_reg,
796         C => RXCLK,
797         CE => '1',
798         CLR => '0',
799         Q => TXEN);

```

### Cambios de dominio de reloj

Otra cuestión a tener en cuenta es el cambio de dominio de relojes de las señales de control *start\_algorithm* y *ready\_algorithm*. La señal *start\_algorithm* es un pulso generado en el controlador Ethernet con el reloj *rxclk* y es una señal de entrada del bloque que describe el algoritmo con el reloj de 100 MHz. Como la frecuencia de *rxclk* es mayor que 100 MHz se genera un pulso de doble ciclo para asegurar la detección del pulso con el reloj de 100 MHz. Este pulso es asíncrono con respecto al reloj de 100 MHz así que requiere un doble registro para prevenir la aparición de metaestabilidad,

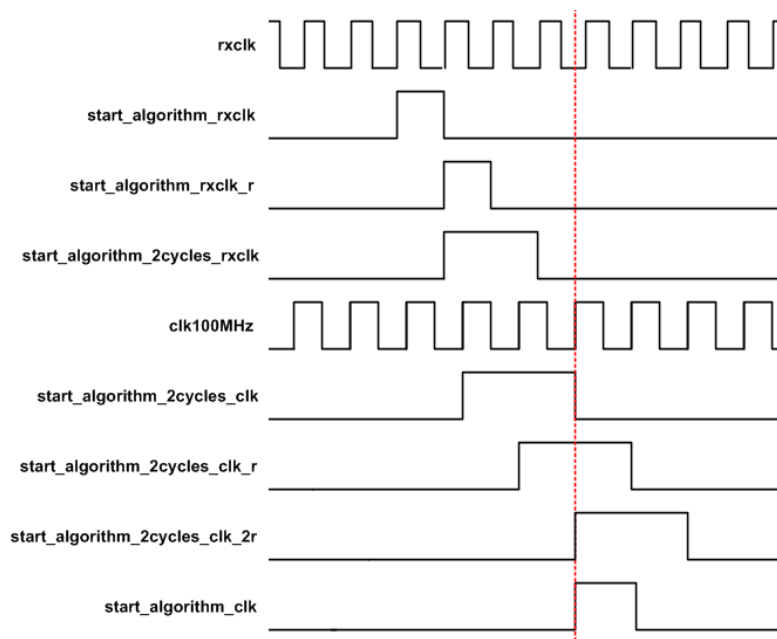
producida cuando no se cumplen los tiempos de set-up y de hold de un flip-flop y su salida puede tomar un valor intermedio entre 0 y 1.



**Figura 5.8.** Sincronizador de un pulso asíncrono

En la Figura 5.8 se muestra el sincronizador empleado formado por tres flip-flops en serie y una puerta AND. El primer flip-flop registra la señal, con el segundo flip-flop se asegura la espera de un ciclo a que decaiga un posible estado de metaestabilidad antes de usar la señal. Como el pulso tiene un ancho mayor que un período de reloj se introduce un tercer flip-flop y una puerta AND para producir un pulso de ancho un ciclo de 100 MHz.

En la siguiente representación pueden verse las diferentes señales en el cambio de dominio de relojes para el pulso *start\_algorithm*.



**Figura 5.9.** Cambio de dominio de reloj de la señal *start\_algorithm*

Ocurre algo parecido con la señal *ready\_algorithm* que es producida en el bloque del algoritmo a 100 MHz y es conducida al controlador Ethernet sincronizada con el reloj *rxclk*.

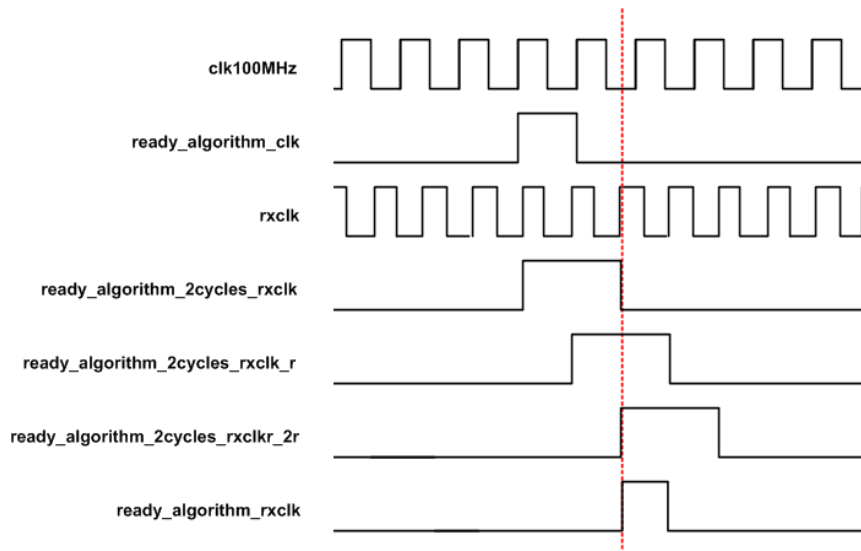


Figura 5.10. Cambio de dominio de reloj de la señal *ready\_algorithm*

En la arquitectura *rk4\_arch* se encuentran descritos los procesos para el cambio de dominio de relojes de las señales *start\_algorithm* y *ready\_algorithm*. La descripción VHDL del proceso *proc\_clk* para el cambio de dominio al reloj de 100 MHz se muestra a continuación.

```

180  proc_clk: process (CLK100MHZ, RESET)
181  begin
182  --
183  if (RESET = '0') then
184  --
185  start_algorithm_2cycles_clk <= '0';
186  start_algorithm_2cycles_clk_r <= '0';
187  start_algorithm_2cycles_clk_2r <= '0';
188  --
189  start_algorithm_clk <= '0';
190  --
191  elsif (CLK100MHZ = '1' and CLK100MHZ'event) then
192  --
193  start_algorithm_2cycles_clk <= start_algorithm_2cycles_rxclk;
194  start_algorithm_2cycles_clk_r <= start_algorithm_2cycles_clk;
195  start_algorithm_2cycles_clk_2r <= start_algorithm_2cycles_clk_r;
196  --
197  start_algorithm_clk <= '0';
198  --
199  if ((start_algorithm_2cycles_clk_r = '1') and (start_algorithm_2cycles_clk_2r = '0')) then
200  --
201  start_algorithm_clk <= '1';
202  --
203  end if;
204  --
205  end if;
206  --
207  end process;

```

Para el proceso *inst\_process\_rxclk* con el cambio de dominio al reloj de 125 MHz se tiene:

```
209 proc_rxclk: process (RXCLK, RESET)
210 begin
211 --
212 if (RESET = '0') then
213 --
214 start_algorithm_rxclk_r <= '0';
215 start_algorithm_2cycles_rxclk <= '0';
216 --
217 ready_algorithm_2cycles_rxclk <= '0';
218 ready_algorithm_2cycles_rxclk_r <= '0';
219 ready_algorithm_2cycles_rxclk_2r <= '0';
220 --
221 ready_algorithm_rxclk <= '0';
222 --
223 elsif (RXCLK = '1' and RXCLK'event) then
224 --
225 start_algorithm_rxclk_r <= start_algorithm_rxclk;
226 --
227 start_algorithm_2cycles_rxclk <= start_algorithm_rxclk or start_algorithm_rxclk_r;
228 --
229 ready_algorithm_2cycles_rxclk <= ready_algorithm_clk;
230 ready_algorithm_2cycles_rxclk_r <= ready_algorithm_2cycles_rxclk;
231 ready_algorithm_2cycles_rxclk_2r <= ready_algorithm_2cycles_rxclk_r;
232 --
233 ready_algorithm_rxclk <= '0';
234 --
235 if ((ready_algorithm_2cycles_rxclk_r = '1') and (ready_algorithm_2cycles_rxclk_2r = '0')) then
236 --
237 ready_algorithm_rxclk <= '1';
238 --
239 end if;
240 --
241 end if;
242 --
243 end process;
```

## 5.6. Conclusiones

Una vez completado el diseño ya se conoce como es su arquitectura, qué componentes la forman, las interconexiones entre ellos y como se describen a nivel RTL o empleando cores. En las siguientes etapas se construye físicamente el sistema y simultáneamente se verifica el diseño.

## 6. REALIZACIÓN FÍSICA

### 6.1. Introducción

Una vez realizado el diseño se construye físicamente el sistema hardware. Para ello se crea un proyecto con las herramientas de diseño de Xilinx, se añaden las descripciones VHDL, se llevan a cabo la síntesis y la implementación y finalmente se configura la FPGA. En paralelo a este proceso se realiza la verificación del diseño.

### 6.2. Instalación del entorno de desarrollo de Xilinx

Desde la página web de Xilinx [www.xilinx.com](http://www.xilinx.com) se descarga la herramienta *ISE Design Suite* 14.7 y se instala en el PC. Se crea una cuenta de usuario de Xilinx y desde la página web de solicitud de licencias se obtiene el fichero .lic por correo electrónico. Se abre el entorno y se carga el fichero con la licencia desde el menú *Help* con la opción *Manage License*.

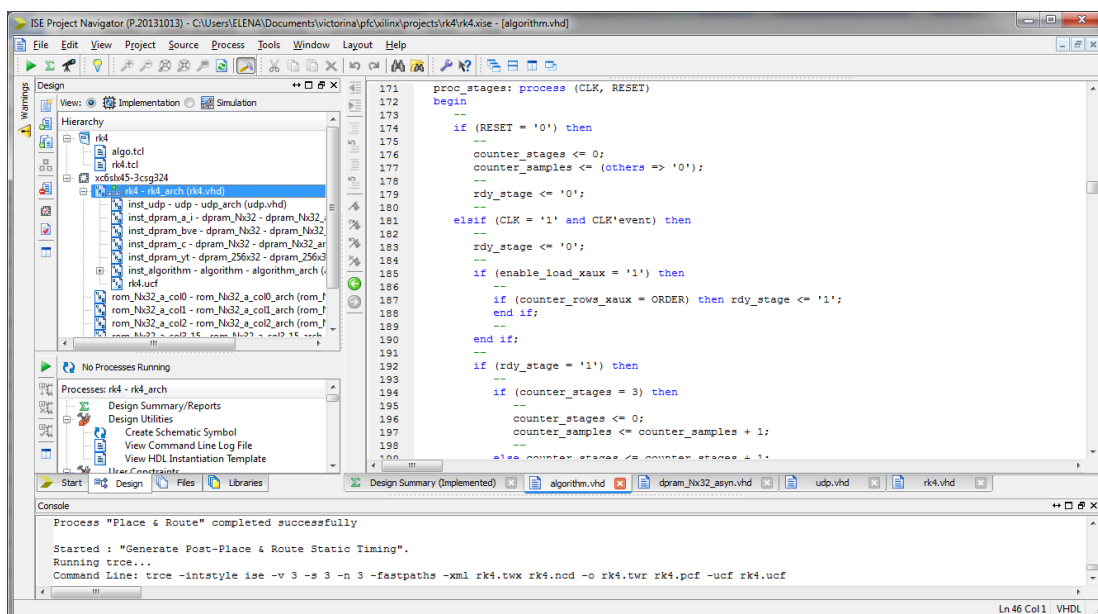


Figura 6.1. Entorno de desarrollo de Xilinx

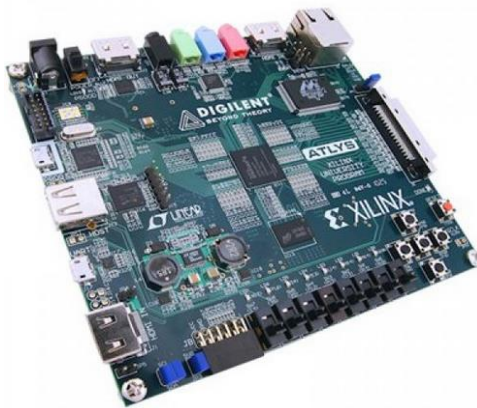
### 6.3. Dispositivo y tarjeta de evaluación

El sistema se implementa sobre una FPGA con las características mostradas en la siguiente tabla.

Características de la FPGA	
Fabricante	Xilinx
Familia	Spartan-6
Dispositivo	XC6SLX45
Encapsulado	CSG324
Grado de velocidad	-3C

**Tabla 6.1.** Características de la FPGA adquirida

La FPGA se encuentra impresa en la tarjeta de evaluación Atlys de Digilent. Esta tarjeta, asequible económicamente, es elegida por tener una FPGA con una cantidad alta de bloques DSP que favorece la realización de un número grande de operaciones en coma flotante y además dispone de un transceptor Marvell Alaska 88E1111 que permite el enlace punto a punto con el PC en modo GMII a 1.000 Mbps.



**Figura 6.2.** Tarjeta Atlys de Digilent

## 6.4. Construcción de un proyecto

Desde el entorno de desarrollo de Xilinx se crea el proyecto *rk4\_project* seleccionando desde el menú *File* la opción *New Project*.

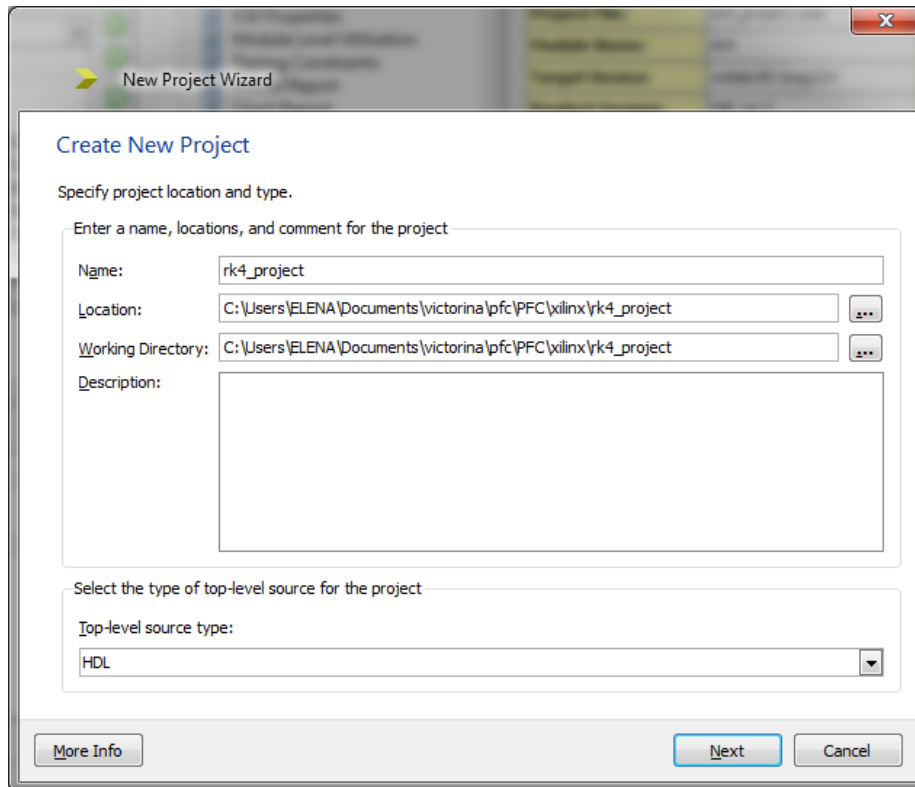


Figura 6.3. Creación del proyecto

Se selecciona la familia, el modelo y el encapsulado de la FPGA que va a emplearse. En esta versión libre de desarrollo se emplea por defecto el sintetizador XST y el simulador ISim.

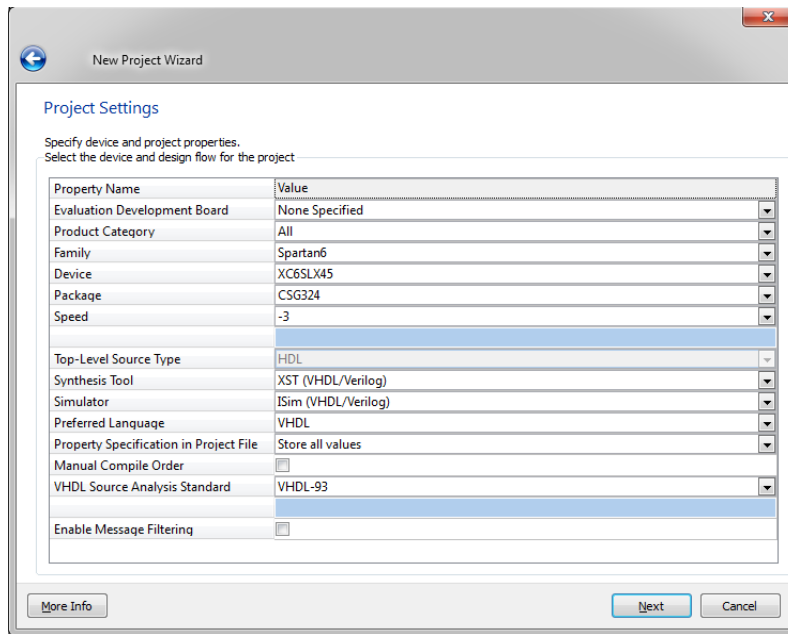


Figura 6.4. Propiedades del proyecto

## 6.5. Fuentes VHDL

Desde el menú *Project* se selecciona *New Source* y aparece la ventana que permite crear los ficheros VHDL, los cores (IPs), el paquete *algorithm\_package* y la librería *mylib*.

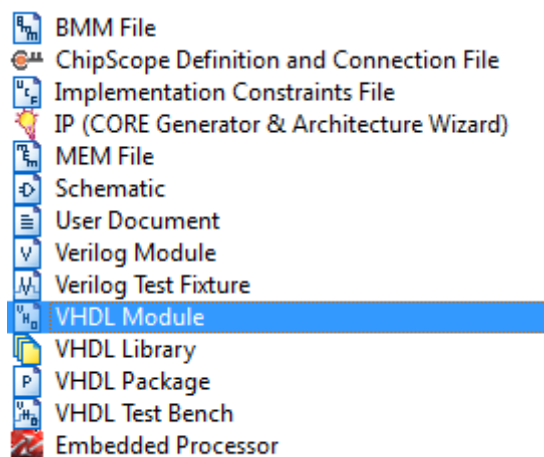


Figura 6.5. Ventana de creación de fuentes



Si se elige la opción IP se abre la herramienta *CORE Generator* de Xilinx para seleccionar el core desde un catálogo de IPs, configurar sus características y crear las fuentes que permiten implementarlo.

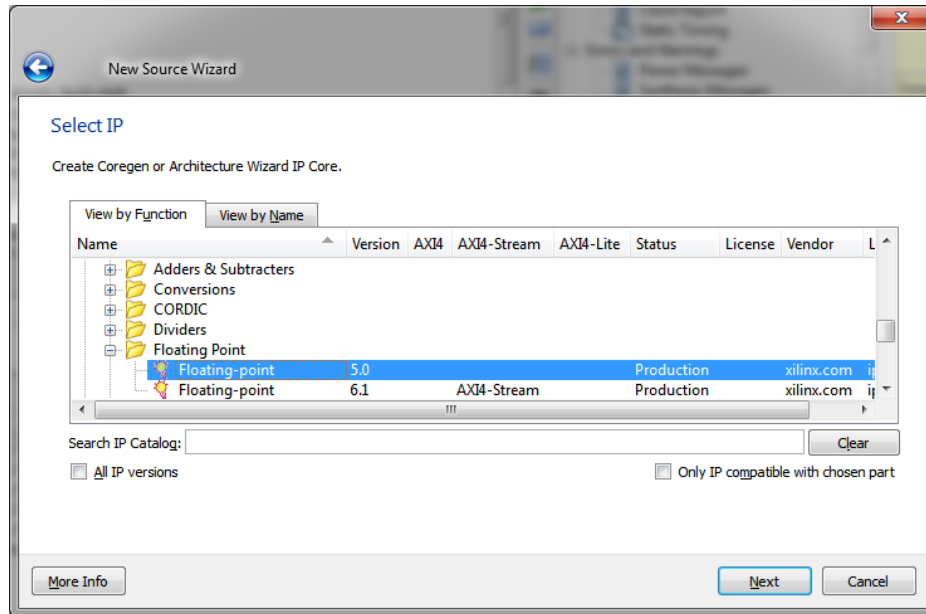


Figura 6.6. Selección de cores

A continuación se muestran los ficheros que contienen las diferentes descripciones VHDL y cores.

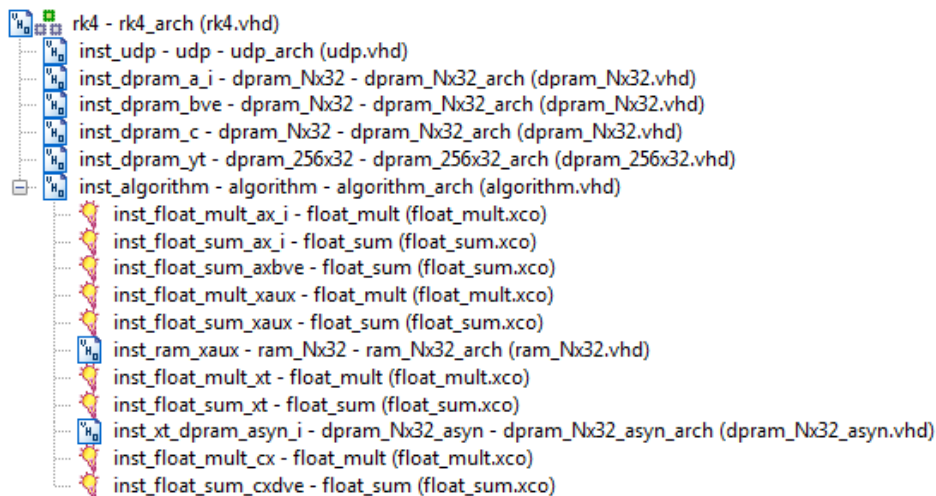


Figura 6.7. Jerarquía de componentes

Existen dos librerías, la librería por defecto *work* donde se almacenan los componentes creados y *mylib*, la generada para contener el paquete *algorithm\_package*.

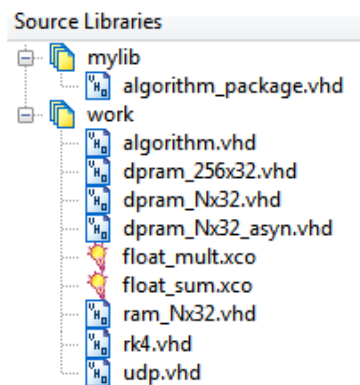


Figura 6.8. Librerías

## 6.6. Definición de restricciones

Las restricciones (*constraints*) se definen en el fichero *rk4.ucf* incluido en el proyecto.

### **Restricciones temporales**

Se incluyen las restricciones impuestas por las frecuencias mínimas requeridas de los dos relojes de entrada de 100 MHz y 125 MHz.

```
51 NET "CLK100MHZ" TNM_NET = CLK100MHZ;  
52 TIMESPEC TS_CLK100MHZ = PERIOD "CLK100MHZ" 100 MHz HIGH 50%;  
53  
54 NET "RXCLK" TNM_NET = RXCLK;  
55 TIMESPEC TS_RXCLK = PERIOD "RXCLK" 125 MHz HIGH 50%;
```

### **Asignación de pines**

Las siguientes restricciones permiten asignar a los puertos de la entidad *rk4* los correspondientes pines de la FPGA.

```

9 NET "CLK100MHZ" LOC = "L15";
10 NET "RESET" LOC = "T15";
11
12 #####
13
14 NET "CLK_GMII" LOC = "L12";
15 NET "TXD<0>" LOC = "H16";
16 NET "TXD<1>" LOC = "H13";
17 NET "TXD<2>" LOC = "K14";
18 NET "TXD<3>" LOC = "K13";
19 NET "TXD<4>" LOC = "J13";
20 NET "TXD<5>" LOC = "G14";
21 NET "TXD<6>" LOC = "H12";
22 NET "TXD<7>" LOC = "K12";
23 NET "TXEN" LOC = "H15";
24 NET "TXER" LOC = "G18";
25
26 NET "RXCLK" LOC = "K15";
27 NET "RXDV" LOC = "F17";
28 NET "RXD<0>" LOC = "G16";
29 NET "RXD<1>" LOC = "H14";
30 NET "RXD<2>" LOC = "E16";
31 NET "RXD<3>" LOC = "F15";
32 NET "RXD<4>" LOC = "F14";
33 NET "RXD<5>" LOC = "E18";
34 NET "RXD<6>" LOC = "D18";
35 NET "RXD<7>" LOC = "D17";
36 NET "RXER" LOC = "F18";
37
38 NET "MDC" LOC = "F16";
39 NET "MDIO" LOC = "N17";
40 NET "NRST" LOC = "G13";
41
42 #####
43
44 NET "LED<0>" LOC = "U18";
45 NET "LED<1>" LOC = "M14";
46 NET "LED<2>" LOC = "N14";
47 NET "LED<3>" LOC = "L14";

```

Estas localizaciones están impuestas por la tarjeta de evaluación Atlys de Digilent y se obtienen consultando el manual de la tarjeta (Digilent, 2013: 12). Se tiene en cuenta que *CLK100MHZ* es el reloj base proporcionado, *RESET* parte de uno de los botones de la placa, *LEDS* se conecta al conjunto de leds y el resto forman parte de la interfaz Ethernet conectada con el transceptor Marvell Alaska 88E1111.

## 6.7. Síntesis

Desde la opción *Check Syntax* de la ventana *Processes* se analiza la sintaxis de las descripciones VHDL realizadas. Posteriormente se ejecuta la síntesis, proceso por el cual las descripciones RTL del circuito digital se convierten en una netlist que contiene las entradas y salidas de éste, las puertas que lo componen y sus interconexiones.

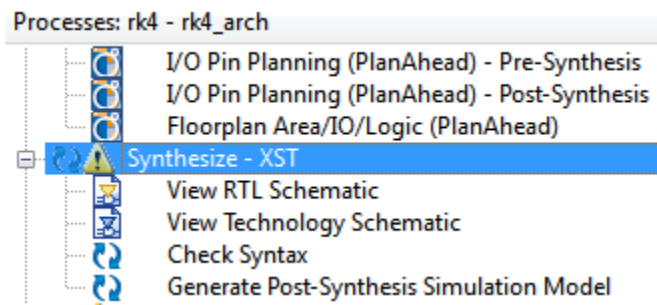


Figura 6.9. Ventana de procesos para la ejecución de la síntesis

Se analizan los informes creados durante la síntesis para detectar posibles problemas.

## 6.8. Implementación

La implementación está distribuida en varios procesos (Xilinx, 2008):

### ***Traducción (Translate)***

A partir de la información obtenida en la síntesis y de las restricciones temporales y lógicas definidas (*constraints*) se genera una descripción del diseño lógico a nivel de primitivas de Xilinx.

### ***Mapeo (Map)***

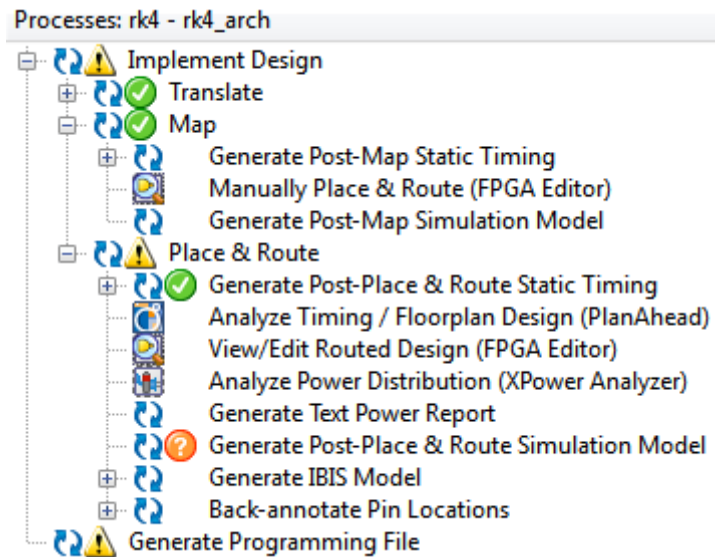
Se identifica la lógica producida durante la traducción con elementos de los CLBs e IOBs de la FPGA.

### ***Ubicación y conexionado (Place & Route)***

Se emplazan los CLBs y IOBs y se conectan para que cumplan con las restricciones (*constraints*) impuestas.

**Generación del fichero de configuración (Generate Programming File)**

Se crea un fichero bitstream para configurar la FPGA.



**Figura 6.10. Procesos para la implementación**

De los informes generados por la herramienta de diseño durante la implementación se comprueba que las restricciones temporales son cumplidas y se hace un análisis de los recursos empleados de la FPGA.

XC6SLX45		
Recurso	Disponibles	Usados
Registros (FFs)	54.576	17.349
LUTs para lógica	27.288	15.955
LUTs para memoria	6.408	858
Bloques DSP48A	58	19
User I/O	218	31

**Tabla 6.2. Recursos empleados**

## 6.9. Configuración de la FPGA

Se emplean dos procedimientos diferentes para programar la FPGA, uno para ser usado durante las pruebas y otro para la configuración del sistema final (Digilent, 2013: 2-5).

### 6.9.1. Programación en modo JTAG

Se realiza durante los procesos de verificación del diseño y validación. Se siguen los siguientes pasos:

- Se conecta la FPGA al PC a través del cable USB.
- Se unen las conexiones del jumper J11 de la tarjeta.
- Se enciende la tarjeta.
- Se abre el software iMPACT de Xilinx y se detecta la cadena del JTAG con la FPGA.

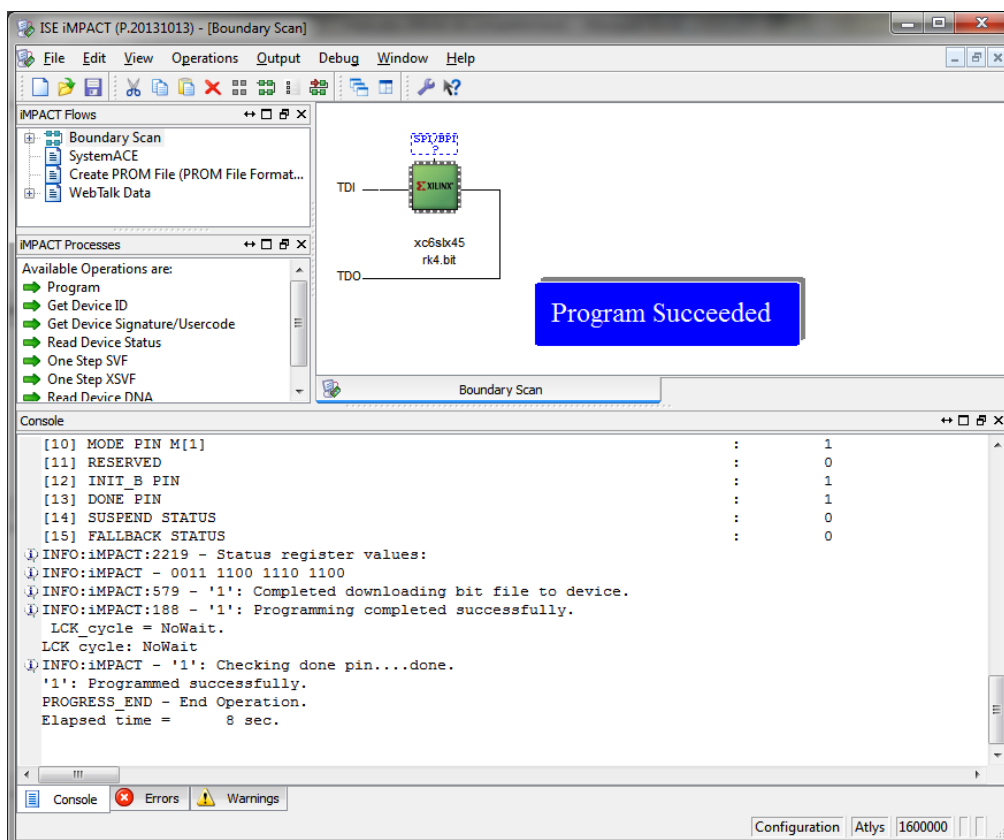


Figura 6.11. ISE iMPACT para la programación de la FPGA en modo JTAG

- Se selecciona con iMPACT el fichero *rk4.bit*.
- Se ejecuta la programación de la FPGA para que el bitstream almacenado en el fichero *.bit* se cargue en la FPGA. Cuando la tarjeta se apaga la FPGA se desconfigura siendo necesaria la reprogramación.

### 6.9.2. Configuración en modo ROM

Se realiza una vez validado el sistema. Es necesario realizar una serie de pasos:

- Se dejan libres las conexiones del jumper J11 de la tarjeta.
- Se descarga de la página web de Digilent <https://www.digilentinc.com> el programa *Adept* y se instala.
- Se abre el entorno y se entra en la ventana *Flash*.

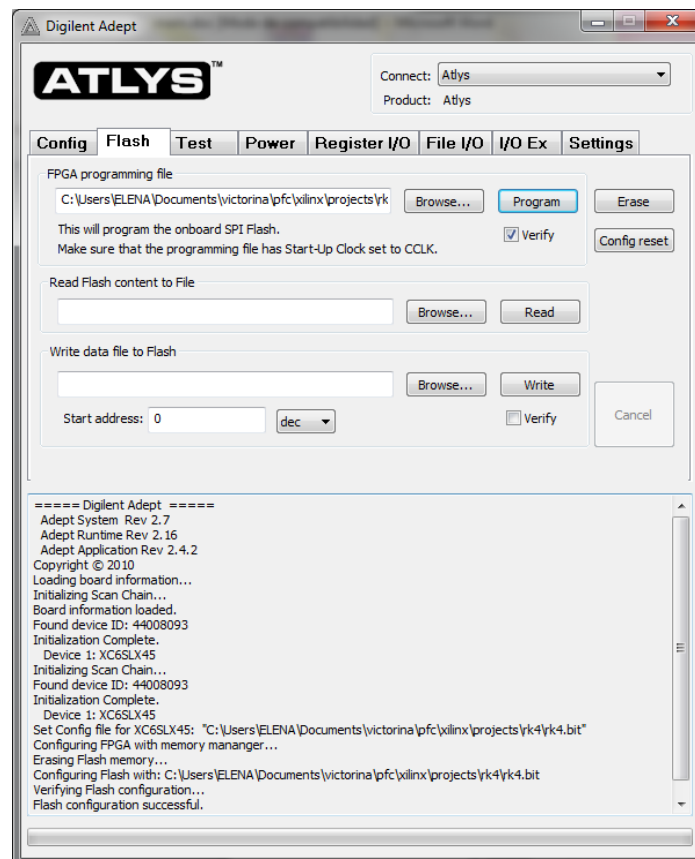


Figura 6.12. Software *Adept* para la programación de la FPGA en modo ROM

- Se selecciona el fichero *rk4.bit* y la opción *Verify*.
- Si la FPGA ya había sido previamente programada se pulsa la opción *Erase*.
- Se pulsa la opción *Program*. La FPGA es configurada con un circuito que programa la Flash ROM SPI con el bitstream del fichero *.bit*.
- Cada vez que se enciende la placa la FPGA es programada con la configuración almacenada en la memoria ROM.

## **6.10. Conclusiones**

La realización física del sistema incluye la creación de un proyecto y de los correspondientes ficheros (.vhd, scripts, netlists, .bin, .ucf, etc) con las herramientas de diseño de Xilinx. Posteriormente se realiza la síntesis, la implementación y la configuración de la FPGA. Durante la verificación debe estar parte del sistema ya construido parcial o totalmente. Antes de presentar el proceso de verificación se describe el diseño de la aplicación de usuario.



## 7. DISEÑO DE LA INTERFAZ DE USUARIO

### 7.1. Introducción

En paralelo al flujo de diseño del algoritmo de simulación de circuitos sobre la FPGA se define otro flujo para el desarrollo de la aplicación *Simulaciones RK4* que permite al usuario cargar el modelo de un circuito RLC, obtener las matrices de estado, configurar y ejecutar la simulación de su comportamiento eléctrico así como visualizar de forma gráfica los resultados de ésta.

En la siguiente figura se muestra la ventana de la interfaz gráfica de la aplicación.

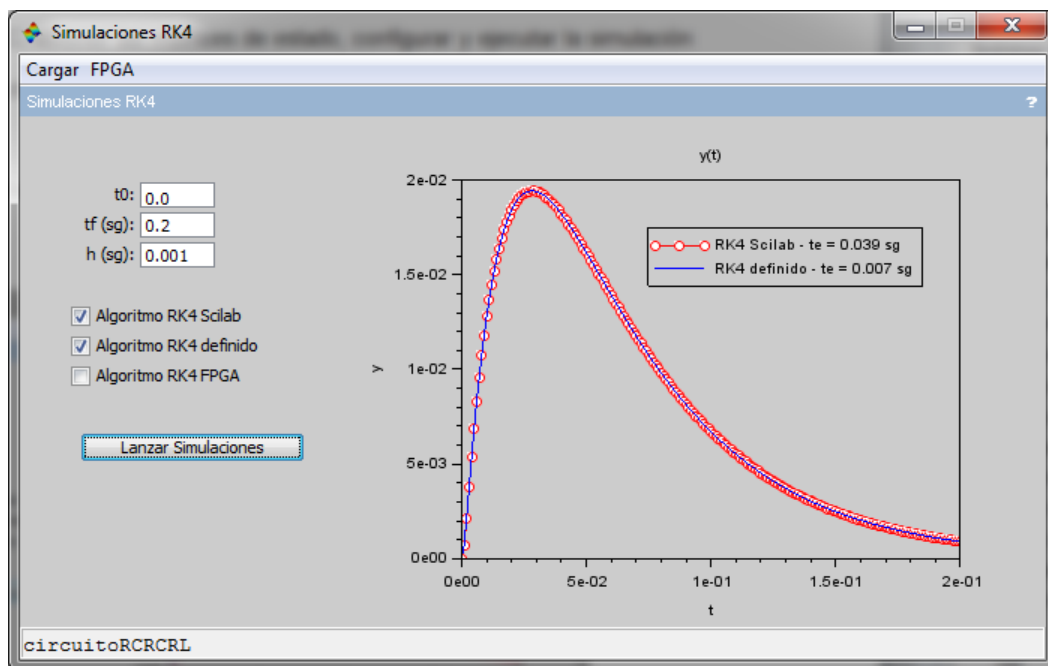


Figura 7.1. Interfaz de usuario para la simulación de los circuitos

Esta aplicación es también fundamental, como se verá en el Capítulo 8, para la etapa de verificación del diseño del algoritmo sobre la FPGA porque extrae valores por consola de simulaciones basadas en PC que son comparados con los obtenidos con la FPGA.

Para el desarrollo de la aplicación *Simulaciones RK4* se emplea el entorno de cálculo y simulación Scilab, que permite definir y trabajar con modelos, mostrar gráficos y definir interfaces.

A continuación se describen las funcionalidades de esta aplicación y los pasos de diseño y codificación seguidos durante su desarrollo.

## **7.2. Uso de la aplicación**

### **7.2.1. Instalación**

Se siguen los siguientes pasos:

- Desde la página de Oracle [www.oracle.org](http://www.oracle.org) se descarga JDK 1.8.0 y se instala.
- Desde la página de Scilab [www.scilab.org](http://www.scilab.org) se descarga Scilab y se instala.
- Se abre el entorno Scilab.
- Desde el menú *Aplicaciones* se selecciona la opción *Administrador de módulos (ATOMS)* y se instala JIMS. En la página web [forge.scilab.org/index.php/p/JIMS](http://forge.scilab.org/index.php/p/JIMS) puede obtenerse información del uso de estas librerías.
- Desde la consola de Scilab se define la variable de entorno PATH\_PFC introduciendo  
`-->setenv('PATH_PFC', 'dirección')`  
donde *dirección* es la trayectoria de la carpeta PFC de este proyecto.

### **7.2.2. Definición gráfica del modelo**

Para definir el modelo del sistema a simular, un circuito RLC, se emplea el editor gráfico de circuitos Xcos del entorno Scilab que se abre ejecutando desde la consola `-->xcos`.

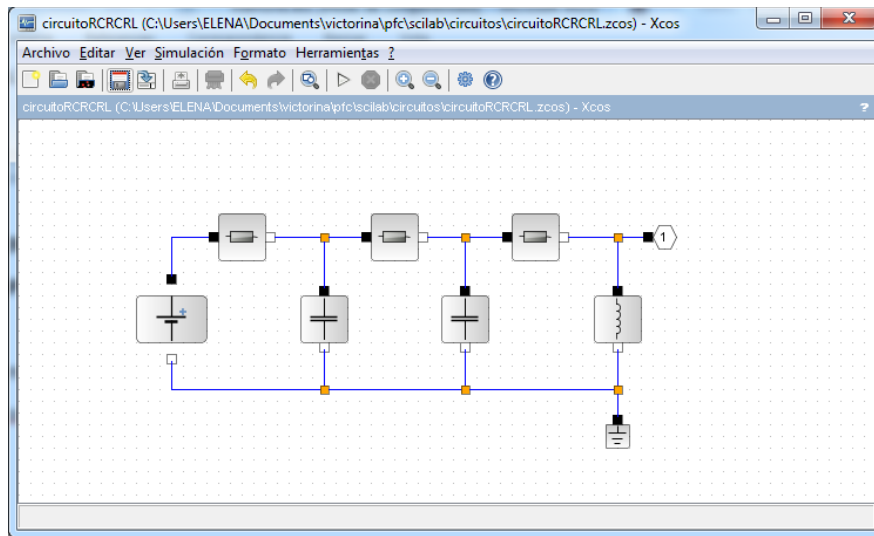


Figura 7.2. Trazado de un circuito con Xcos

Desde la ventana del *Explorador de paletas* de Xcos pueden extraerse de la paleta *Eléctrica* los componentes *Resistor*, *Capacitor*, *Inductor*, *ConstantVoltage* y *Ground*. De la paleta *Puerto* y *Subsistema* se obtiene el componente *OUTIMPL\_f* que permite definir el nodo de salida del que se desea medir el voltaje.

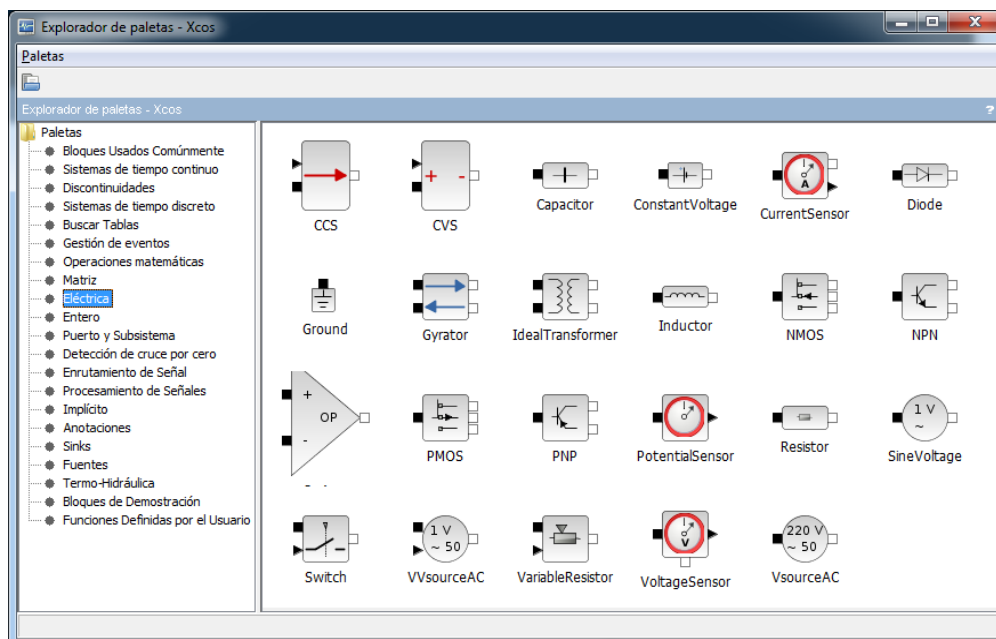
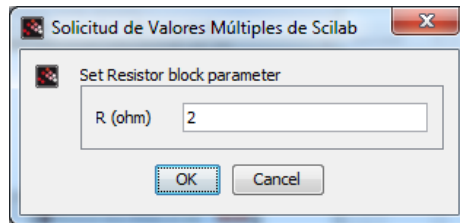


Figura 7.3. Explorador de paletas de Xcos

Pinchando sobre los componentes *Resistor*, *Capacitor*, *Inductor* y *ConstantVoltage* aparece su ventana de configuración desde la cual puede definirse el valor de sus correspondientes propiedades.



**Figura 7.4.** Ventana de configuración de una resistencia con Xcos

Posteriormente se trazan las conexiones entre componentes y se guarda el circuito como un fichero .zcos. Se genera una librería con varios circuitos RLC creados con diversas estructuras y órdenes.

### **7.2.3. Inicio de la aplicación**

Desde el menú *Archivo* del entorno Scilab se pulsa la opción *Ejecutar* y desde la ventana de selección de fichero se elige el archivo `PATH_PFC\scilab\code\principal.sce` y se ejecuta. Aparece la ventana de usuario de la aplicación.

### **7.2.4. Carga del circuito**

Desde el menú *Cargar* de la interfaz de usuario y pulsando la opción *Circuito* se abre la ventana de selección de circuito desde la que se elige el fichero .zcos que guarda la información gráfica del circuito RLC a simular.

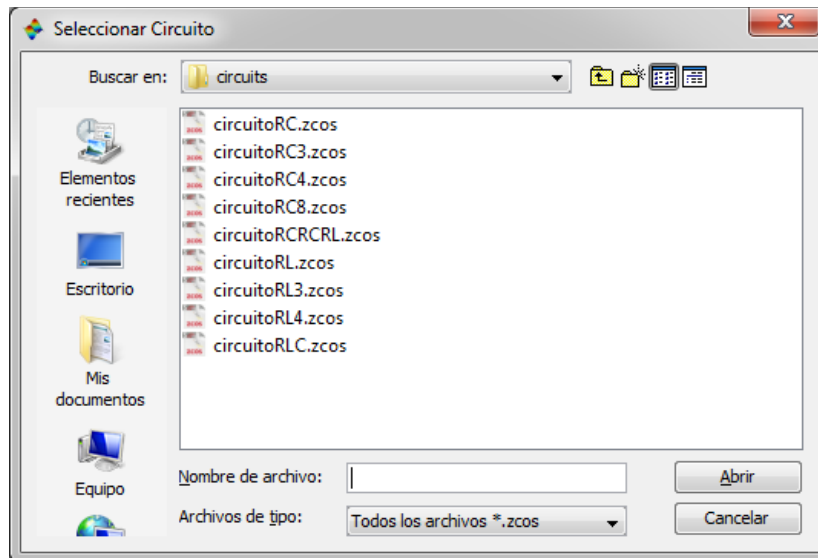


Figura 7.5. Ventana de selección de un circuito

Si el circuito se carga correctamente aparece su nombre en la parte izquierda de la barra de estado de la interfaz de usuario.

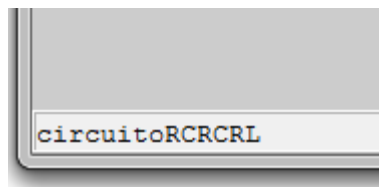


Figura 7.6. Nombre del circuito en la barra de estado

La carga del circuito implica la obtención de las matrices de estado del circuito RLC a partir de la información gráfica extraída del fichero .zcos. Para este paso no trivial se sigue el método desarrollado a continuación.

### 7.2.5. Obtención de las ecuaciones de estado

A continuación se desarrolla un método para obtener el espacio de estados de un circuito lineal a partir de la malla de componentes que lo forman. El espacio de estados de los circuitos que interesan se representa por la formulación [3.1].

Para obtener las matrices A, B, C y D se debe partir de ecuaciones diferenciales que definen de una forma completa el comportamiento del circuito. El comportamiento de las redes eléctricas está descrito por las leyes de Kirchhoff, que se enuncian a continuación (Attia, 1999).

***Ley de nodos o primera ley de Kirchhoff***

La suma de todas las corrientes que pasan por un nodo es igual a cero:

$$\sum_{n=1}^N I_n = 0 \quad [7.1]$$

donde n representa cada uno de los N nodos del circuito distintos del nodo 0 de referencia o nodo a tierra.

***Ley de mallas o segunda ley de Kirchhoff***

La suma de las diferencias de potencial eléctrico en un lazo es igual a cero:

$$\sum_{r=1}^R V_r = 0 \quad [7.2]$$

donde r representa cada una de las R ramas del circuito.

Una forma sistemática de resolver el problema de la obtención del espacio de estados partiendo de las leyes fundamentales de Kirchhoff es empleando el *Análisis Nodal Modificado* (Hanke, 2006). Para ello se realizan los siguientes pasos:

***Se numeran los nodos del circuito***

Se elige el nodo a tierra como nodo 0 de referencia. Todos los demás nodos se enumeran del 1 a N donde N es el orden del circuito.

**Se numeran las ramas del circuito**

La numeración se realiza en el siguiente orden: primero las ramas resistivas, después las capacitivas, las inductivas y se acaba con la rama que contiene la fuente de tensión  $V_e$ . Se define una orientación para cada rama manteniendo fijo el criterio de nodo origen y nodo destino.

**Se definen las matrices de incidencia**

Son matrices que crean para los elementos resistivos ( $A_R$ ), capacitivos ( $A_C$ ), inductivos ( $A_L$ ) y de fuente de tensión ( $A_V$ ). Estas matrices recogen el efecto de cada tipo de elemento en el circuito y tienen tantas filas como nodos distintos de cero tiene el circuito, y tantas columnas como ramas con un elemento de ese tipo tiene el circuito.

$$A_R = \begin{cases} -1 & \text{si el nodo es de entrada de R} \\ 1 & \text{si el nodo es de salida de R} \\ 0 & \text{si es otro nodo cualquiera} \end{cases}$$

$$A_C = \begin{cases} -1 & \text{si el nodo es de entrada de C} \\ 1 & \text{si el nodo es de salida de C} \\ 0 & \text{si es otro nodo cualquiera} \end{cases}$$

$$A_L = \begin{cases} -1 & \text{si el nodo es de entrada de L} \\ 1 & \text{si el nodo es de salida de L} \\ 0 & \text{si es otro nodo cualquiera} \end{cases}$$

$$A_V = \begin{cases} -1 & \text{si el nodo es de entrada de la fuente } V_e \\ 1 & \text{si el nodo es de salida de la fuente } V_e \\ 0 & \text{si es otro nodo cualquiera} \end{cases}$$

**Se obtiene un espacio de estados a partir de las leyes de Kirchhoff**

De la Ley de Kirchhoff de los nodos [7.1] se deduce que:

$$A_C \cdot C \cdot A'_C \cdot \frac{dV_N(t)}{dt} + A_R \cdot G \cdot A'_R \cdot V_N(t) + A_L \cdot i_L(t) + A_V \cdot i_{V_e}(t) = 0$$

donde

C es una matriz con los valores de las capacitancias en la diagonal y el resto a 0,

G es una matriz con los valores de las inversas de las resistencias en la diagonal y el resto a 0,

L es una matriz con los valores de las inductancias en la diagonal y el resto a 0,

$V_N(t)$  es la matriz con las tensiones de los nodos,

$i_L(t)$  es la matriz de las corrientes a través de las ramas inductivas,

$i_{V_e}(t)$  es la corriente a través de la fuente de alimentación  $V_e$ .

$A'$  es la matriz transpuesta de A.

De la Ley de Kirchhoff de las mallas [7.2] se tiene que:

$$L \cdot \frac{di_L(t)}{dt} - A'_L \cdot V_N(t) = 0$$

$$A'_V \cdot V_N(t) = V_e$$

Expresado todo en modo matricial:



$$\begin{bmatrix} A_C \cdot C \cdot A'_C & 0 & 0 \\ 0 & L & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \frac{d}{dt} \begin{bmatrix} V_N(t) \\ I_L(t) \\ i_{V_e}(t) \end{bmatrix} + \begin{bmatrix} A_R \cdot G \cdot A'_R & A_L & A_V \\ -A'_L & 0 & 0 \\ A'_V & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} V_N(t) \\ I_L(t) \\ i_{V_e}(t) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ V_e \end{bmatrix} \quad [7.3]$$

Al intentar despejar  $\frac{dX(t)}{dt}$  queda:

$$\frac{d}{dt} \begin{bmatrix} V_N(t) \\ I_L(t) \\ i_{V_e}(t) \end{bmatrix} = - \begin{bmatrix} A_C \cdot C \cdot A'_C & 0 & 0 \\ 0 & L & 0 \\ 0 & 0 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} A_R \cdot G \cdot A'_R & A_L & A_V \\ -A'_L & 0 & 0 \\ A'_V & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} V_N(t) \\ I_L(t) \\ i_{V_e}(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ V_e \end{bmatrix}$$

El problema es que aparece el cálculo de una matriz no singular, no existe su inversa, por lo que se resuelve en el dominio de  $s$  (Yildiz, 2010), para lo cual se llevan a cabo las transformaciones de Laplace:

$$X(t) \Rightarrow X(s) \quad \text{y} \quad \frac{dX(t)}{dt} \Rightarrow s \cdot X(s)$$

quedando la expresión [7.3] de la forma

$$\begin{bmatrix} A_C \cdot s \cdot C \cdot A'_C & 0 & 0 \\ 0 & s \cdot L & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} V_N(s) \\ I_L(s) \\ i_{V_e}(s) \end{bmatrix} + \begin{bmatrix} A_R \cdot G \cdot A'_R & A_L & A_V \\ -A'_L & 0 & 0 \\ A'_V & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} V_N(s) \\ I_L(s) \\ i_{V_e}(s) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ V_e \end{bmatrix}$$

Sumando y despejando las variables de salida se tiene que:

$$\begin{bmatrix} A_C \cdot s \cdot C \cdot A'_C + A_R \cdot G \cdot A'_R & A_L & A_V \\ -A'_L & s \cdot L & 0 \\ A'_V & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} V_N(s) \\ I_L(s) \\ i_{V_e}(s) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ V_e \end{bmatrix}$$

$$\begin{bmatrix} V_N(s) \\ I_L(s) \\ i_{V_e}(s) \end{bmatrix} = \begin{bmatrix} A_C \cdot s \cdot C \cdot A'_C + A_R \cdot G \cdot A'_R & A_L & A_V \\ -A'_L & s \cdot L & 0 \\ A'_V & 0 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 0 \\ 0 \\ V_e \end{bmatrix}$$

Esta expresión puede escribirse de la forma:

$$Y(s) = H(s) \cdot V_e \quad [7.4]$$

A partir de la función de transferencia  $H(s)$  se pueden obtener las matrices A, B, C y D del modelo de estado [3.1] por varios métodos, no siendo la solución única (Wikipedia, 2015).

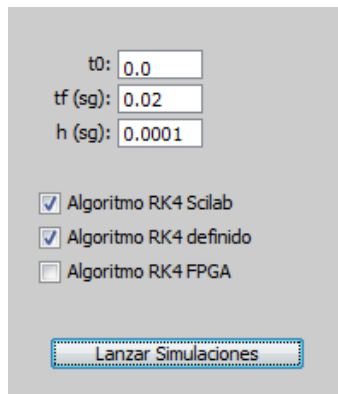
Teniendo en cuenta que la salida del sistema es la tensión en un nodo  $n$  distinto de 0, se tiene que [7.4] puede expresarse como:

$$V_s(t) = Y(t)|_n$$

por lo que las matrices C y D se simplifican para considerar sólo el elemento  $V_s(t)$  dentro de la matriz  $Y(t)$ .

### 7.2.6. Configuración de la simulación

La representación gráfica de la simulación se representa desde el instante inicial  $t_0$  hasta el instante final  $t_f$  expresado en segundos (tiempo de simulación). Debe configurarse la constante  $h$  que representa el paso temporal del algoritmo.



t0: 0.0  
tf (sg): 0.02  
h (sg): 0.0001

Algoritmo RK4 Scilab  
 Algoritmo RK4 definido  
 Algoritmo RK4 FPGA

Lanzar Simulaciones

Figura 7.7. Configuración y activación de las simulaciones

La aplicación obtiene la simulación a partir del algoritmo de Runge-Kutta de orden 4 en tres versiones distintas:

#### **Algoritmo RK4 Scilab**

Esta opción emplea la función `ode()` de Scilab.

#### **Algoritmo RK4 definido**

En este caso se emplea código Scilab para desarrollar el algoritmo paso a paso. Este caso resulta muy positivo a la hora de entender el algoritmo e iniciar el planteamiento de la descripción con VHDL.

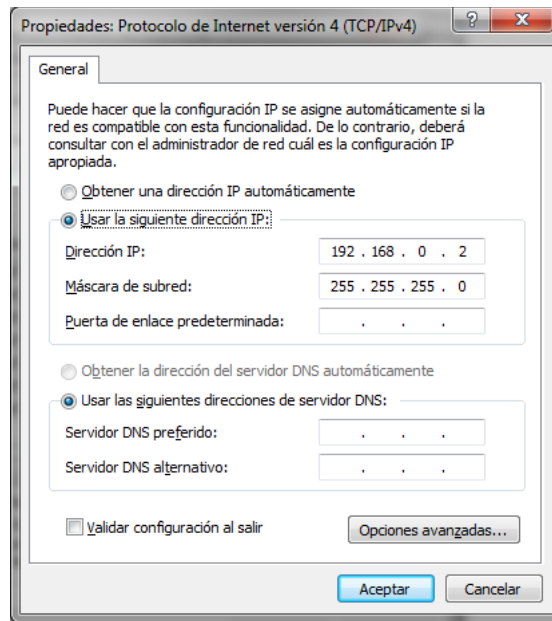
#### **Algoritmo RK4 FPGA**

La simulación se lanza sobre la FPGA que recibe la información de configuración, procesa el algoritmo y transmite la trama con los resultados finales. En este caso es importante tener en cuenta que sólo se recuperan 200 muestras de la simulación por lo que se debe respetar la relación  $tf/h = 200$ .

#### **7.2.7. Conexión del PC con la FPGA**

Para realizar la simulación del algoritmo con la FPGA es necesario previamente que el PC que ejecuta la aplicación de usuario esté correctamente conectado con la FPGA que procesa el algoritmo. Para ello deben realizarse los siguientes pasos.

- Unir, empleando el cable cruzado CAT-6, el conector RJ-45 de la tarjeta Atlys que contiene impresa la FPGA con el conector RJ-45 del PC.
- Desactivar la red WiFi en el PC si está activa.
- Se configura la dirección IP del PC desde la ventana *Propiedades de conexión de Área Local*.



**Figura 7.8.** Ventana de configuración de la conexión de área local

Es importante que la IP del PC pertenezca a la misma red que la IP de la FPGA.

Función	Elemento	Máscara/IP
Servidor	FPGA	255.255.255.0/192.168.0.1
Cliente	PC	255.255.255.0/192.168.0.2

**Tabla 7.1.** Configuración de la red

- Se enciende la tarjeta Atilys de Digilent y se programa la FPGA en modo JTAG.
- Debe comprobarse si la velocidad de la red está a 1 Gbps chequeando los leds Ethernet de la tarjeta o desde la ventana *Estado de Conexión de área local*.

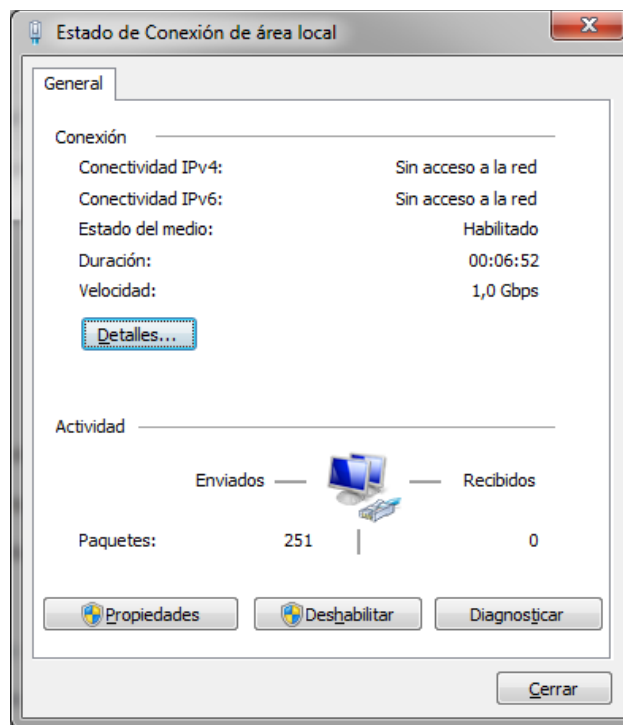


Figura 7.9. Ventana de estado de conexión de área local

- Se chequea la comunicación desde la aplicación pulsando desde el menú *FPGA* la opción *Comprobar comunicación*. Si la comunicación es correcta aparece en la parte derecha de la barra de estado un mensaje de aprobación o de error en caso contrario.

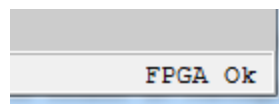


Figura 7.10. Chequeo de la comunicación con la FPGA

### 7.2.8. Lanzamiento y representación de la simulación

Pulsando el botón *Lanzar Simulaciones* se inician los cálculos de las simulaciones seleccionadas. Posteriormente una parte gráfica muestra superpuestas las curvas, obtenidas por los diferentes algoritmos, de la tensión en el nodo de salida del circuito cargado. Los tres tipos de simulación deben proporcionar exactamente la misma solución.

Sobre la gráfica aparece una leyenda donde se muestran los tiempos de ejecución empleados por cada algoritmo expresados en segundos. Esta leyenda se fija sobre el gráfico pulsando con el botón izquierdo del ratón.

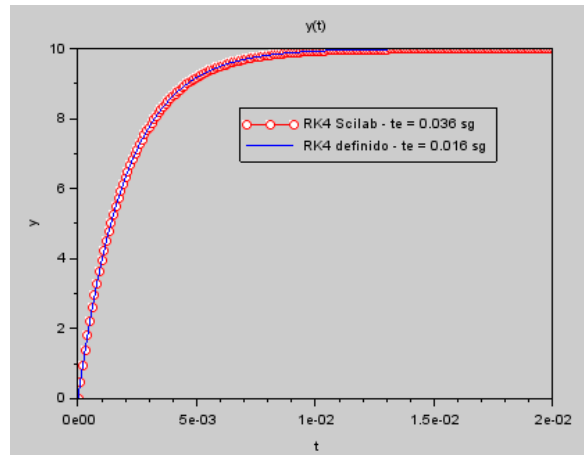


Figura 7.11. Representación gráfica de las simulaciones seleccionadas

### 7.3. Diseño y codificación de la aplicación

La aplicación está codificada con el lenguaje de alto nivel y librería de funciones que ofrece Scilab y sigue la siguiente estructura de ficheros almacenados en `PATH_PFC\scilab\code`.

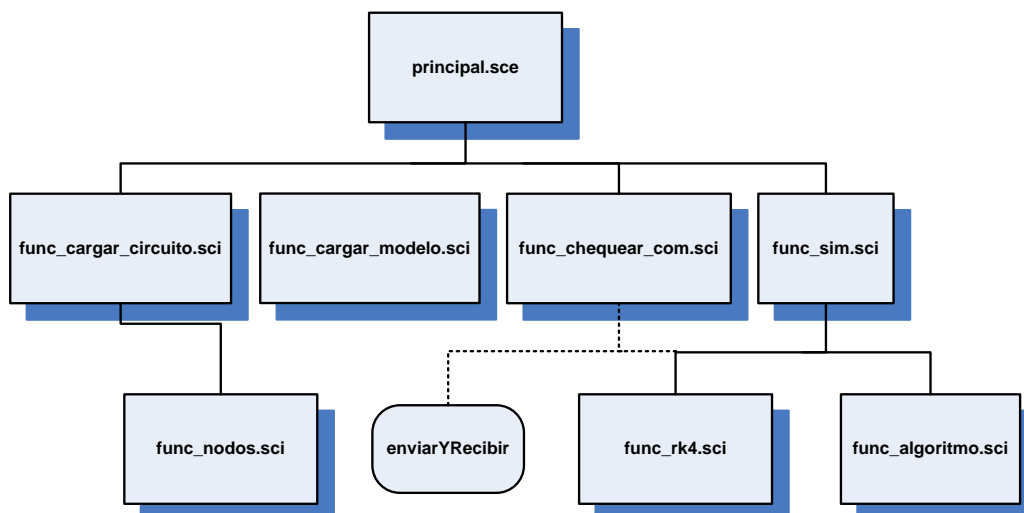


Figura 7.12. Estructura de ficheros de la aplicación *Simulaciones RK4*

### 7.3.1. Función principal

El script *principal.sce* define la ventana que forma la interfaz gráfica de usuario (GUI), sus controles, apariencia y eventos aplicados a los controles. Para ello se crea una figura y a partir de ella una ventana.

```
figura = figure('figure_name', gettext('Simulaciones RK4'));
w = scf(figura);

w.figure_size = [700, altura];
w.resize = 'off';

w.menubar_visible = 'on';
w.toolbar_visible = 'off';
w.infobar_visible = 'on';
```

Se emplea la función *uimenu* para crear la barra de menús. El parámetro *parent* es el identificador del elemento padre, el parámetro *label* el texto visible sobre el elemento y el parámetro *callback* el código que se ejecuta cuando se pulsa el botón del menú o submenú.

```
menu1 = uimenu('parent', w, 'label', gettext('Cargar'));
menu2 = uimenu('parent', w, 'label', gettext('FPGA'));

uimenu('parent', menu1, 'label', gettext('Circuito'), 'callback', 'circuito =
uigetfile('*.zcos', getenv('PATH_PFC') + '\\scilab\circuitos\\', 'Seleccionar
Circuito'); [A, B, C, D, X0, ve] = func_cargar_circuito(circuito); [path, fname,
ext] = fileparts(circuito); circuitoCargado = fname; xinfo(circuitoCargado +
blanks(77 - length(circuitoCargado)) + comFpga); set(button1, 'enable', 'on');');
uimenu('parent', menu1, 'label', gettext('Modelo'), 'callback', '[A, B, C, D, X0, ve]
= func_cargar_modelo();xinfo(''modelo cargado'');');
```

```
uimenu('parent', menu1, 'label', gettext('Salir'), 'callback', 'delete(gcf());');
uimenu('parent', menu2, 'label', gettext('Comprobar comunicación'), 'callback',
'jimport(''ethernet.cliente.ClienteUdp''); miClase = ClienteUdp.new(); comFpga =
func_chequear_com(miClase); xinfo(circuitoCargado + blanks(77 -
length(circuitoCargado)) + comFpga);');
```

Se emplea la función *uicontrol* para definir el resto de controles. El parámetro *style* determina el tipo de control; para este diseño, cuadro de texto, cuadro de edición, casilla de verificación o botón. Destacan el parámetro *position* que introduce la posición del elemento de control en la ventana y el parámetro *string* que define el texto sobre el elemento.

```
text1 = uicontrol('parent', w, 'style', 'text', 'BackgroundColor', [0.8 0.8 0.8],
'position', [30 altura-120 50 17], 'horizontalalignment', 'right', 'string', 't0: ');
text2 = uicontrol('parent', w, 'style', 'text', 'BackgroundColor', [0.8 0.8 0.8],
'position', [30 altura-140 50 17], 'horizontalalignment', 'right', 'string', 'tf
(sg): ');
text3 = uicontrol('parent', w, 'style', 'text', 'BackgroundColor', [0.8 0.8 0.8],
'position', [30 altura-160 50 17], 'horizontalalignment', 'right', 'string', 'h (sg):
');
```

```
edit1 = uicontrol('parent', w, 'style', 'edit', 'BackgroundColor', [1 1 1],
'position', [80 altura-120 50 17], 'string', '0.0');
edit2 = uicontrol('parent', w, 'style', 'edit', 'BackgroundColor', [1 1 1],
'position', [80 altura-140 50 17], 'string', '0.2');
edit3 = uicontrol('parent', w, 'style', 'edit', 'BackgroundColor', [1 1 1],
'position', [80 altura-160 50 17], 'string', '0.001');
```

```
checkbox1 = uicontrol('parent', w, 'style', 'checkbox', 'BackgroundColor', [0.8 0.8
0.8], 'position', [30 altura-200 150 15], 'string', 'Algoritmo RK4 Scilab');
checkbox2 = uicontrol('parent', w, 'style', 'checkbox', 'BackgroundColor', [0.8 0.8
0.8], 'position', [30 altura-220 150 15], 'string', 'Algoritmo RK4 definido');
```



```
checkbox3 = uicontrol('parent', w, 'style', 'checkbox', 'BackgroundColor', [0.8 0.8
0.8], 'position', [30 altura-240 150 15], 'string', 'Algoritmo RK4 FPGA');

button1 = uicontrol('parent', w, 'style', 'pushbutton', 'enable', 'on', 'position',
[40 altura-290 150 20], 'string', 'Lanzar Simulaciones', 'callback', 'tf =
strtod(get(edit2, 'string')); h = strtod(get(edit3, 'string'));
jimport('ethernet.cliente.ClienteUdp'); miClase = ClienteUdp.new();
func_sim(get(checkbox1, 'value'), get(checkbox2, 'value'), get(checkbox3,
'value'), miClase, tf, h)');
```

### 7.3.2. Función para cargar el circuito

La función `func_cargar_circuito()` se ejecuta cuando se pulsa la opción *Circuito* del menú *Cargar* y se selecciona un fichero `.zcos`. Su objetivo es obtener las matrices de estado que representan el circuito RLC.

```
function [mA, mB, mC, mD, mX0, ve] = func_cargar_circuito(circuito)
```

Realiza los siguientes pasos:

#### **Obtención de la información gráfica del circuito**

Se llama a la función `importXcosDiagram()` que devuelve la estructura `scs_m` del circuito contenedora del listado de sus objetos tipo bloque (*Block*) y tipo enlace (*Link*) junto con las correspondientes propiedades gráficas y del modelo. Son objetos tipo bloque los componentes *Resistor*, *Capacitor*, *Inductor*, *ConstantVoltage*, *Ground*, *IMPSPLIT\_f* y *OUTIMPL\_f*.

Con la función `lincos()` de Scilab se puede obtener el modelo del espacio de estados a partir del esquema `scs_m` pero no está permitido para sistemas que contienen elementos eléctricos por lo que se emplea el Análisis Nodal Modificado.

### **Obtención de los nodos del circuito**

Se define el conjunto mínimo de nodos del circuito y la equivalencia con los existentes en el esquema *scs\_m*. En el vector *model\_nodes* se almacena en la posición *k* el nodo del circuito al que equivale.

Si uno de los componentes es *Ground* es el nodo 0.

El esquema puede contener varios componentes *IMPSPILT\_f* y enlaces (*Link*) que conectan tres o más componentes eléctricos y que equivalen a un solo nodo en el Análisis Nodal. La función *func\_nodos()* obtiene de forma recursiva todos los nodos de *scs\_c* que corresponden a un único nodo del circuito modelado.

Del componente *OUTIMPL\_f* se detecta el nodo del voltaje de salida.

### **Generación de las matrices de incidencia**

De la información contenida en la estructura *scs\_m* se obtienen las matrices C, G y L y las matrices de incidencia empleadas en el Análisis Nodal Modificado.

```
pin = scs_m.objs(i).graphics.pin-numBlocks;
pout = scs_m.objs(i).graphics.pout-numBlocks;

if scs_m.objs(i).gui == "Resistor" then

    numR = numR + 1;
    matrixG(numR, numR) = 1/strtod(scs_m.objs(i).graphics.exprs);
```

```

if model_nodes(pin) <> 0 then
    matrixAG(model_nodes(pin), numR) = -1;
end

if model_nodes(pout) <> 0 then
    matrixAG(model_nodes(pout), numR) = 1;
end

elseif scs_m.objs(i).gui == "Capacitor" then

    numC = numC + 1;
    matrixC(numC, numC) = strtod(scs_m.objs(i).graphics.exprs(1));

    if model_nodes(pin) <> 0 then
        matrixAC(model_nodes(pin), numC) = -1;
    end

    if model_nodes(pout) <> 0 then
        matrixAC(model_nodes(pout), numC) = 1;
    end

elseif scs_m.objs(i).gui == "Inductor" then

    numL = numL + 1;
    matrixL(numL, numL) = strtod(scs_m.objs(i).graphics.exprs);

    if model_nodes(pin) <> 0 then
        matrixAL(model_nodes(pin), numL) = -1;
    end
end

```

```
    if model_nodes(pout) <> 0 then
        matrixAL(model_nodes(pout), numL) = 1;
    end

elseif (scs_m.objs(i).gui == "ConstantVoltage") then

    pin = scs_m.objs(i).graphics.pin-numBlocks;

    if model_nodes(pin) <> 0 then
        matrixAV(model_nodes(pin), 1) = 1;
    end

    ve = strtod(scs_m.objs(i).graphics.exprs);
```

### **Obtención de la función de transferencia**

Siguiendo las expresiones del Análisis Nodal Modificado se obtiene la función de transferencia a partir de las matrices de incidencia.

```
matrixEA =
[ matrixAAG * matrixG * matrixAAG' + matrixAAC * s * matrixC * matrixAAC'
matrixAAL matrixAAV;
-matrixAAL' s * matrixL zeros(numL, 1);
matrixAAV' zeros(1, numL) 0];

matrixH = inv(matrixEA) * [zeros(numNodes+numL, 1); 1];
```

### **Cálculo de las matrices de estado**

Con la función `tf2ss()` de Scilab se obtienen las matrices de estado a partir de la función de transferencia. Las matrices mC y mD están limitadas por considerarse la salida sólo para un elemento.

```
sys = tf2ss(matrixH);
```

```
mA = sys(2);
```

```
mB = sys(3);
```

```
mC = sys(4)(indexOutY, :);
```

```
mD = sys(5)(indexOutY);
```

```
mX0 = sys(6);
```

#### **7.3.3. Función de chequeo de comunicación**

La función `func_chequear_com()` se ejecuta cuando se pulsa el submenú *Comprobar comunicación* del menú *FPGA*. Produce el envío de un `acknowledge` a la *FPGA* y la espera de una respuesta.

#### **7.3.4. Función para cargar el modelo**

La función `func_cargar_modelo()` se ejecuta cuando se pulsa el botón *Modelo* del menú *Cargar*. Carga valores aleatorios en las matrices de estado. Se emplea en el proceso de validación.

#### **7.3.5. Función para procesar las simulaciones**

La función `func_sim()` ejecuta el algoritmo para cada uno de los casos y plotea los resultados junto al cálculo de los tiempos de procesado.

### Algoritmo RK4 de Scilab

Se emplea la función `ode()` de Scilab en modo 'rk. La función `func_algoritmo()` devuelve el valor  $dX/dt$  en cada paso.

```
X = ode('rk', X0, t0, t, func_algoritmo);  
y = C * X + D * ve;
```

### Algoritmo RK4 definido

Se llama a la función `func_rk4()`.

```
X = func_rk4(h, t, func_algoritmo);  
y = C * X + D * ve;
```

La función `func_rk4()` viene dada por el siguiente código.

```
function xs = func_rk4(h, t, func_algoritmo)
```

```
    h_div2 = h/2;  
    h_div3 = h/3;  
    h_div6 = h/6;
```

```
    for i = 1 : length(X0)
```

```
        x(i) = X0(i)
```

```
    end
```

```
    for i = 1 : 1 : length(X0)
```

```
        xs(i, 1) = x(i)
```

```
    end
```

```

for s = 2 : 1 : length(t)

    k1 = func_algoritmo(t, x);
    k2 = func_algoritmo(t, x+k1*h_div2);
    k3 = func_algoritmo(t, x+k2*h_div2);
    k4 = func_algoritmo(t, x+k3*h);

    x = x + k1 * h_div6 + k2 * h_div3 + k3 * h_div3 + k4 * h_div6;

    for i = 1 : 1 : length(x0)
        xs(i, s) = x(i)
    end
end

endfunction

```

### **Algoritmo RK4 FPGA**

En este caso se debe preparar primero la trama a enviar en bytes y una vez recibida la trama de respuesta se deben convertir los bytes a valores en coma flotante.

```
trama_solucion = clase.enviarYRecibir(trama_config);
```

### **Ploteado**

Para dibujar las gráficas se emplea la función *plot()* que permite superponer las distintas curvas.

### **Control del tiempo de procesado**

Con las funciones *getdate()* y *etime()* de Scilab se obtienen los tiempos de procesado de los algoritmos.

#### **7.3.6. Función embebida para intercambio de datos con la FPGA**

Las funciones *func\_sim()* y *fun\_chequear\_com()* intercambian información por Ethernet con la FPGA a través de la llamada a código Java embebido en Scilab. Para ejecutarlo se instala el paquete *Java Interaction Mechanism in Scilab (JIMS)* con el *Administrador de Módulos (ATOMS)* de Scilab que va a permitir el acceso de librerías de clases de Java a programas codificados con Scilab.

Primeramente se programa con Java una clase *ClienteUdp* que con funciones de los paquetes *java.net.DatagramPacket* y *java.net.InetAddress* intercambia datagramas con la red Ethernet. La clase contiene el método *enviarYRecibir()* que envía un datagrama de longitud variable desde el puerto 5050 hacia el puerto 5051 de un servidor de IP 192.168.0.1 y a continuación recibe un datagrama del servidor de longitud 800 bytes. La recepción se bloquea como máximo 1 segundo esperando un datagrama.

```
DatagramPacket packetOut = new DatagramPacket(bufferOut, bufferOut.length,  
InetAddress.getByName("192.168.0.1"), 5051);  
socket.send(packetOut);  
  
DatagramPacket packetIn = new DatagramPacket(bufferIn, bufferIn.length);  
socket.setSoTimeout(1000);  
socket.receive(packetIn);  
socket.close();
```



Este código contenido en `PATH_PFC\java\src\ClienteUdp.java` debe compilarse para generar `ClienteUdp.class` en `PATH_PFC\java\class`.

Desde el código Scilab deben estar visibles los ejecutables de Java.

```
javaclasspath(PATH_PFC\java\class');
```

Se importa en Scilab la clase Java que se desea emplear.

```
jimport('ethernet.cliente.ClienteUdp');
```

Se crea un objeto de la clase importada.

```
miClase = ClienteUdp.new();
```

Desde Scilab se llama al método Java de la clase importada que envía y recibe datagramas.

```
trama_salida = clase.enviarYRecibir(trama_entrada);
```

#### **7.4. Verificación de la construcción del modelo**

Es importante verificar que la aplicación obtiene correctamente el modelo matemático de los circuitos. Para ello se generan simulaciones de todos los circuitos de la librería con el simulador de uso libre Qucs y se comparan las gráficas con las obtenidas con Scilab.

En la Figura 7.13 se muestra la ventana principal de la herramienta Qusc.

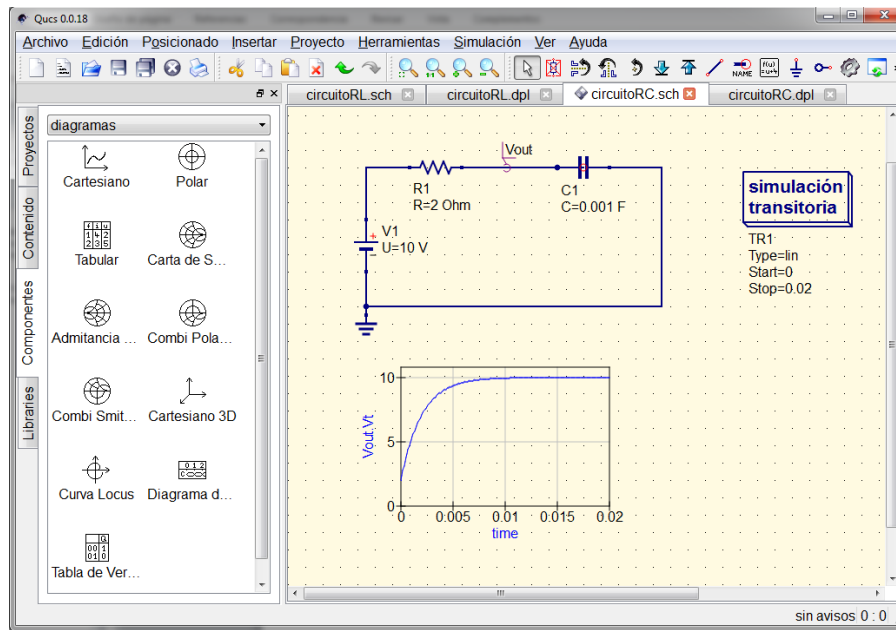


Figura 7.13. Simulación de circuitos con Qucs

## 7.5. Conclusiones

Una aplicación es diseñada para que el usuario final introduzca sus propios circuitos y obtenga los resultados de las diferentes simulaciones. Al mismo tiempo, la aplicación facilita los procesos de verificación y validación de la descripción con VHDL del sistema sobre la FPGA.

## 8. VERIFICACIÓN Y VALIDACIÓN

### 8.1. Introducción

En esta etapa se verifica el comportamiento de los componentes descritos con VHDL que forman la arquitectura. Se realiza de forma incremental y en paralelo con el diseño detallado y la realización física.

Se van a distinguir dos verificaciones, la de la arquitectura del algoritmo y la del controlador Ethernet. Una vez que la realización física y la verificación han sido llevadas a cabo se hacen unas últimas pruebas de validación para comprobar que el comportamiento del sistema final es el correcto.

Cada uno de estos procesos requiere dos fases, la definición de un procedimiento que establece los pasos a realizar y el chequeo del comportamiento.

### 8.2. Verificación del algoritmo

#### 8.2.1. Procedimiento

Para verificar la arquitectura *algorithm\_arch* se crea con VHDL el testbench *algorithm\_tb* que instancia el componente *algorithm* y N memorias ROM de Nx32 bits para sustituir las RAMs que contienen los valores de las matrices A,  $BV_e$  y C. La jerarquía de ficheros se representa en la Figura 8.1.

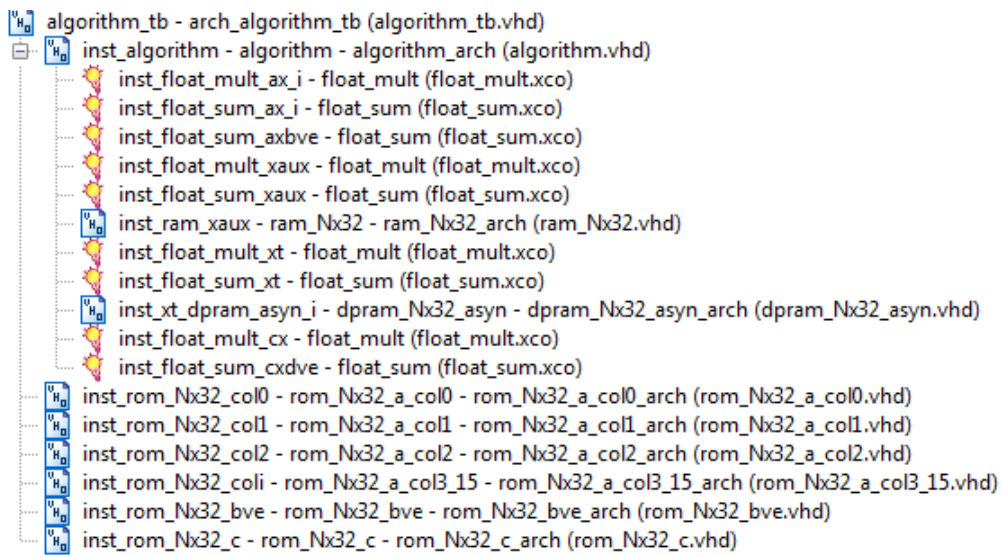


Figura 8.1. Jerarquía de ficheros para el testbench *algorithm\_tb*

El testbench define estímulos para los puertos de entrada de la entidad *algorithm*. Tres procesos definen los estímulos de las señales de entrada del *reset*, del reloj de 100 MHz y de la señal *start\_algorithm*. Además se asignan valores constantes a los puertos H, Hdiv2, Hdiv3, Hdiv6, ORDER y DV<sub>e</sub>. Se activa varias veces la señal *start\_algorithm* para comprobar que se inicializa el procesado correctamente.

```

27     constant h: std_logic_vector(31 downto 0) := x"3851B717";
28     constant h_div_2: std_logic_vector(31 downto 0) := x"37D1B717";
29     constant h_div_3: std_logic_vector(31 downto 0) := x"378BCF65";
30     constant h_div_6: std_logic_vector(31 downto 0) := x"370BCF65";

```

```

181  order <= "0010";
182  dve <= x"40A00000";
183
184  process
185  begin
186      reset <= '0';
187      wait for 15 ns;
188      reset <= '1';
189      wait for 1000 ms;
190  end process;
191
192  process
193  begin
194      clk <= '1';
195      wait for T/2;
196      clk <= '0';
197      wait for T/2;
198  end process;
199
200  process
201  begin
202      start <= '0';
203      wait for T + T/5;
204      start <= '1';
205      wait for T;
206      start <= '0';
207      wait for 900 us;
208  end process;

```

En la aplicación de Scilab se carga el fichero *circuitoRL3.zcos* que proporciona la información gráfica del circuito RLC de orden 3 representado a continuación. Se tiene que  $V_e = 5$  Voltios, las inducciones cumplen  $L = 10,0$  Henrios y las resistencias  $R = 10,0$  ohmios. Interesa obtener  $V_s(t)$  en el nodo de salida 1.

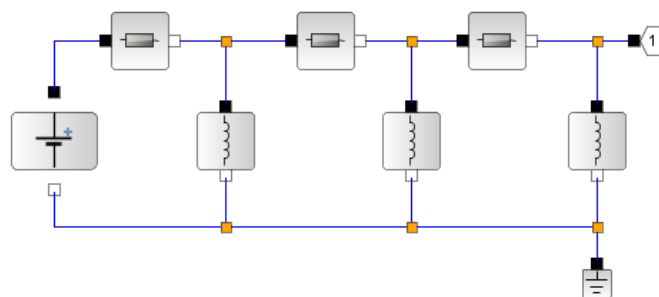
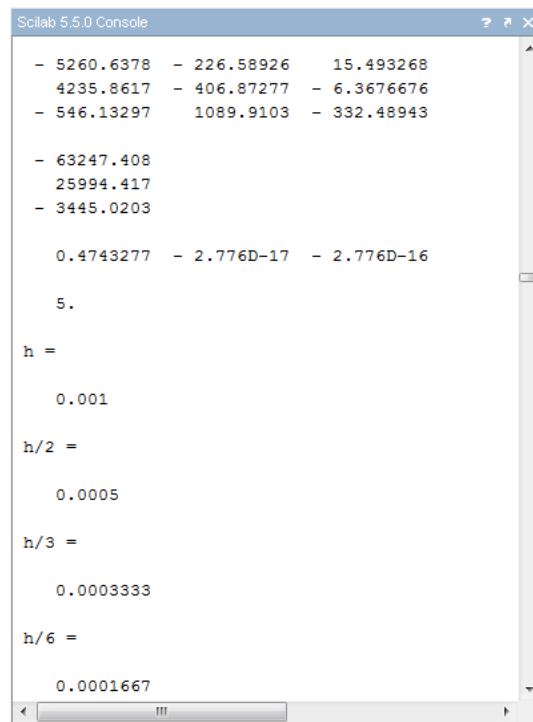


Figura 8.2. Circuito RL de orden 3

Desde la consola de Scilab se obtiene la configuración del algoritmo, es decir, los elementos de las matrices  $A$ ,  $BV_e$ ,  $C$  y los valores de  $DV_e$ ,  $H$ ,  $Hdiv2$ ,  $Hdiv3$  y  $Hdiv6$ .



```
Scilab 5.5.0 Console
- 5260.6378 - 226.58926 15.493268
 4235.8617 - 406.87277 - 6.3676676
- 546.13297 1089.9103 - 332.48943

- 63247.408
25994.417
- 3445.0203

0.4743277 - 2.776D-17 - 2.776D-16

5.

h =

0.001

h/2 =

0.0005

h/3 =

0.0003333

h/6 =

0.0001667
```

Figura 8.3. Valores de configuración extraídos por consola

Son valores decimales que se transforman en hexadecimal con la ayuda del conversor de la página web [www.h-schmidt.net/FloatConverter/IEEE754.html](http://www.h-schmidt.net/FloatConverter/IEEE754.html).

Estos valores se añaden en las memorias ROM y en el testbench. Desde la ventana *Processes* del entorno de desarrollo de Xilinx se chequea la sintaxis del código del testbench y posteriormente se lanza una simulación comportamental (behavioral) con el simulador ISim.

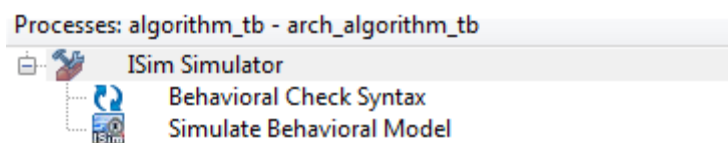


Figura 8.4. Ventana de procesos para el testbench

Este tipo de simulación no requiere que el diseño esté sintetizado por lo que permite una verificación rápida y eficiente del comportamiento del diseño anterior a la realización física. Para esta simulación se ha configurado el tiempo de ejecución a 0 ns.

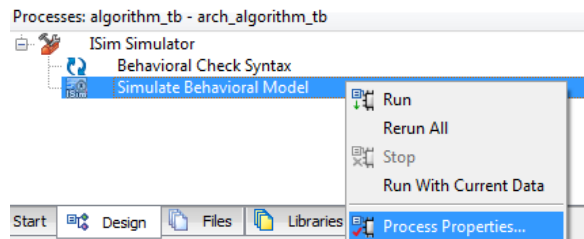


Figura 8.5. Propiedades de la simulación

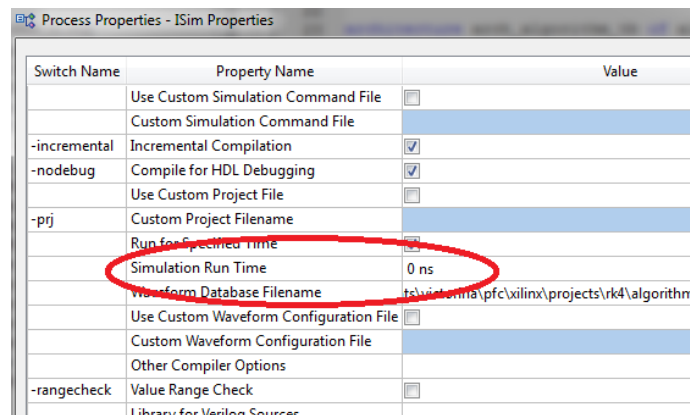


Figura 8.6. Propiedades de la simulación

Una vez dentro del entorno de ISim se borran las señales existentes por defecto en el visualizador y se carga el fichero *algo.tcl* escribiendo en la consola `source algo.tcl`.

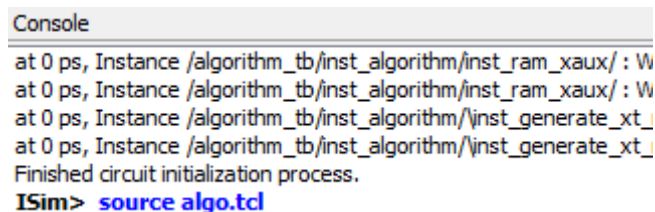


Figura 8.7. Carga del script desde la consola de ISim

El fichero *algo.tcl* es un script que carga las señales de *algorithm* que se quieren visualizar y ejecuta la simulación durante 2.000 us.

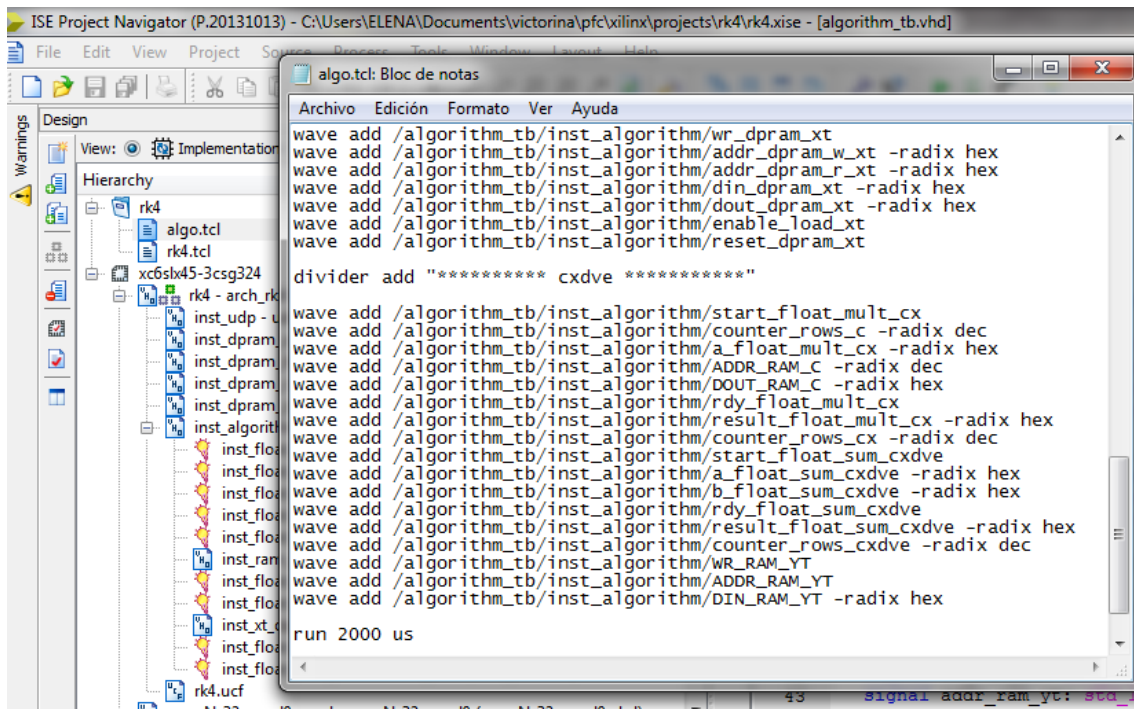


Figura 8.8. Fichero del script *algo.tcl*

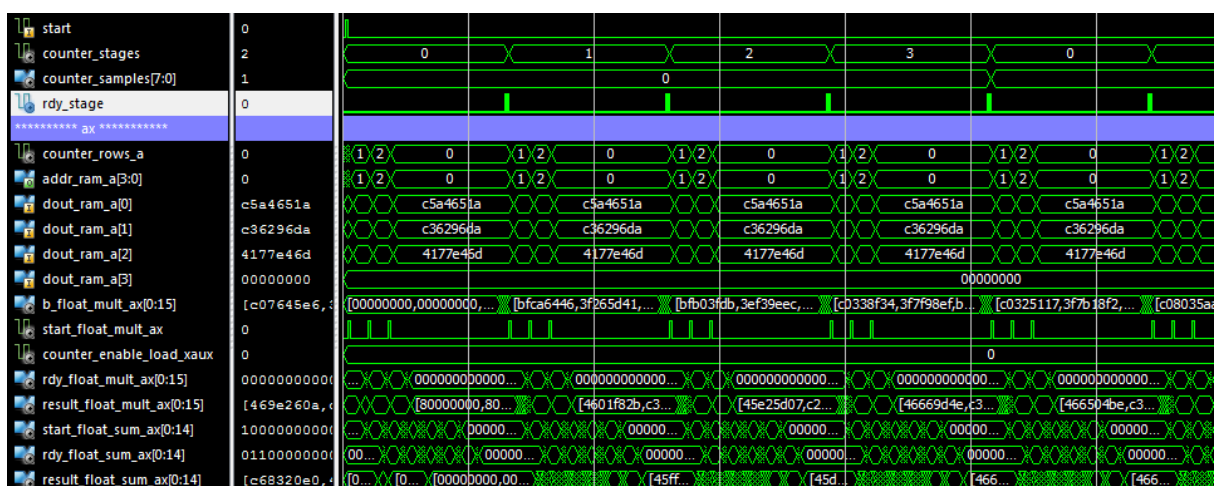


Figura 8.9. Ventana del simulador ISim



El testbench guarda el resultado  $V_s(t)$  escribiendo en un fichero cada muestra del puerto de salida  $din\_ram\_yt$  cuando se activa la señal  $wr\_ram\_yt$ . El proceso para la escritura de los resultados de la simulación toma la siguiente forma.

```

208  process(reset, clk)
209      --
210      file file_out: text;
211      variable str_line: line;
212      --
213  begin
214      --
215      if (reset = '1') then
216          --
217          only_one_reading <= '0';
218          --
219      elsif (clk = '1' and clk'event) then
220          --
221          if ((start = '1') and (only_one_reading = '1')) then
222              --
223              file_open(file_out, "dout.txt", write_mode);
224              --
225          elsif ((rdy = '1') and (only_one_reading = '1')) then
226              --
227              file_close(file_out);
228              only_one_reading <= '0';
229              --
230          elsif (wr_ram_yt = '1') then
231              --
232              write(str_line, din_ram_yt);
233              writeline(file_out, str_line);
234              --
235          end if;
236          --
237      end if;
238      --
239  end process;

```

Después se lanza una simulación tipo *Algoritmo RK4 definido* y se extrae por la consola de Scilab el flujo de datos para cada paso y etapa de las primeras muestras del algoritmo.

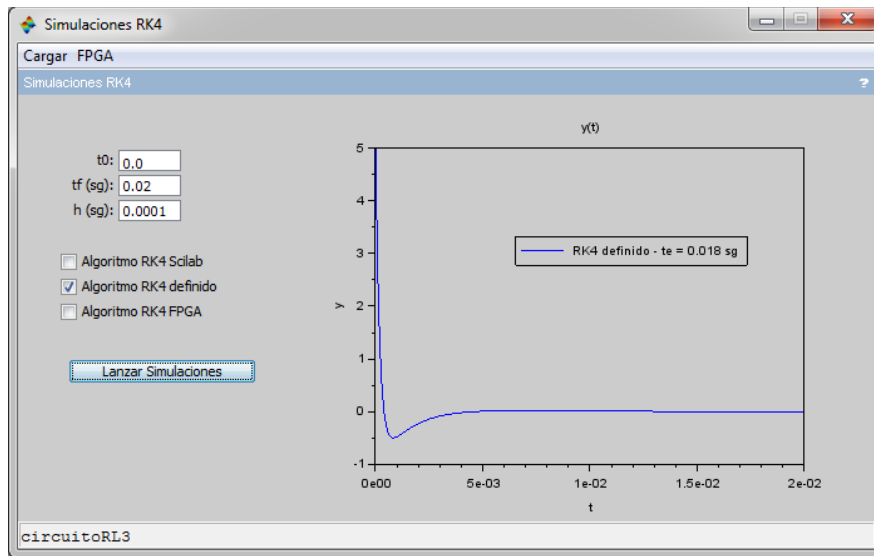


Figura 8.10. Simulación del circuito RL de orden 3 con RK4 definido con código Scilab

Se repite el proceso para circuitos de órdenes 1, 8 y 15.

### 8.2.2. Chequeo

Desde la pantalla de visualización de señales del simulador ISim se comprueba que el comportamiento de las señales de control de la arquitectura *algorithm\_arch* es correcto. Esto incluye:

- Inicialización y avance de contadores.
- Control de lectura y escritura de memorias.
- Activación de multiplicadores y sumadores y señales de habilitación del resultado.

Se comprueba que los buses de datos extraen los valores en coma flotante correctamente en los pasos de cada etapa comparando con los valores extraídos con Scilab para las primeras muestras.

Se comparan todas las muestras obtenidas en el fichero de salida que contiene los valores de  $V_s(t)$  con las obtenidas con Scilab para testear que no se degradan con el tiempo los datos que extrae el algoritmo.

Se repiten las comprobaciones anteriores para los circuitos de órdenes 1, 8 y 15.

### 8.3. Verificación del controlador Ethernet

#### 8.3.1. Procedimiento

Desde la aplicación de usuario de Scilab se carga el circuitoRL3 y se lanza la simulación del algoritmo tipo FPGA.

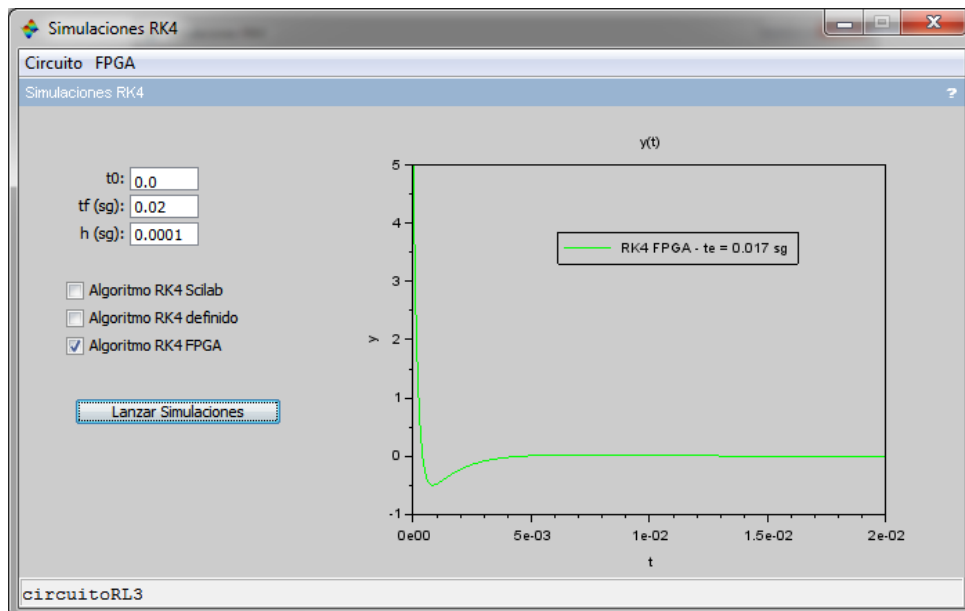


Figura 8.11. Simulación tipo Algoritmo RK4 FPGA

Con el analizador de paquetes de red Wireshark se capturan las tramas ARP y UDP enviadas a la FPGA y se cargan los bytes que las forman en ficheros binarios posteriormente convertidos a texto por Scilab.

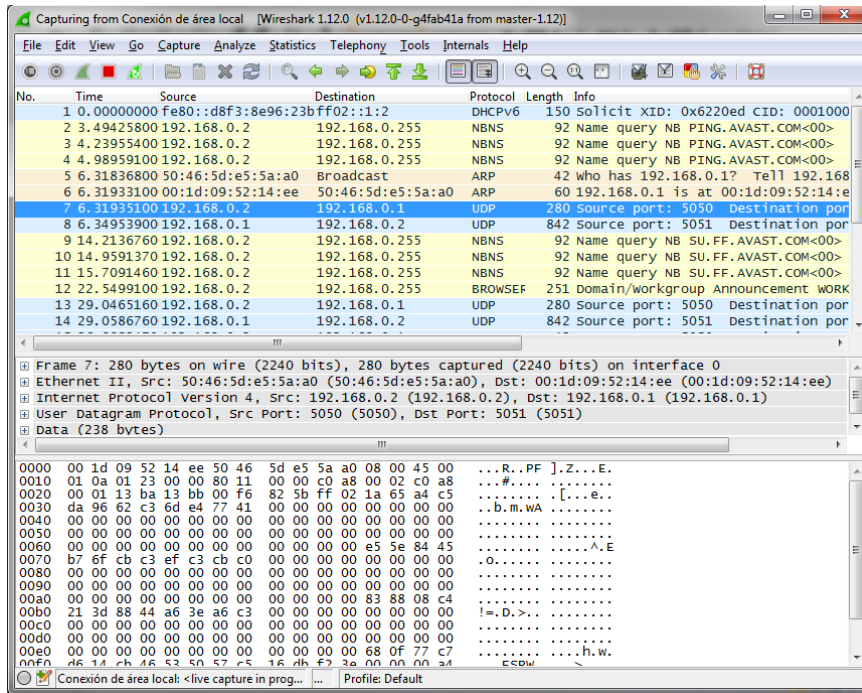


Figura 8.12. Ventana del analizador de paquetes Ethernet Wireshark

Se lanza el testbench *rk4\_tb* con el simulador ISim de Xilinx para chequear la funcionalidad de la arquitectura *arch\_rk4* que instancia tanto la arquitectura del algoritmo como el control del controlador Ethernet. La jerarquía de ficheros se muestra a continuación.



Figura 8.13. Jerarquía de ficheros para el testbench *rk4\_tb*

En el testbench se definen los estímulos de las señales de entrada del *reset*, del reloj de 125 MHz y de *rxdv* en procesos con VHDL. Las tramas introducidas por el puerto de entrada *rxd* de Ethernet se leen de los ficheros de texto obtenidos previamente.

```

106  process
107      --
108      file file_in: text;
109      variable str_line: line;
110      variable rxd_var: integer range 0 to 255;
111      --
112  begin
113      --
114      rxd <= (others => '0');
115      rxdv <= '0';
116      --
117      wait for 15*Trxclk + Trxclk/2;
118      --
119      for i in 1 to 2 loop
120          --
121          rxdv <= '1';
122          --
123          file_open(file_in, "sim/trama_eth.txt", READ_MODE);
124          --
125          for i in 1 to (8 + 1216) loop
126              --
127              readline(file_in, str_line);
128              read(str_line, rxd_var);
129              --
130              rxd <= conv_std_logic_vector(rxd_var, 8);
131              --
132              wait for Trxclk;
133              --
134          end loop;
135          --
136          rxdv <= '0';
137          --
138          file_close(file_in);
139          --
140          wait for 800 us;
141          --
142      end loop;
143      --
144  end process;

```

Una vez dentro del entorno de ISim se carga el script *rk4.tcl* en el que se definen las señales que se quieren visualizar y el tiempo de simulación.

Observar las tramas con el analizador de paquetes Wireshark en modo promiscuo y modo no promiscuo.

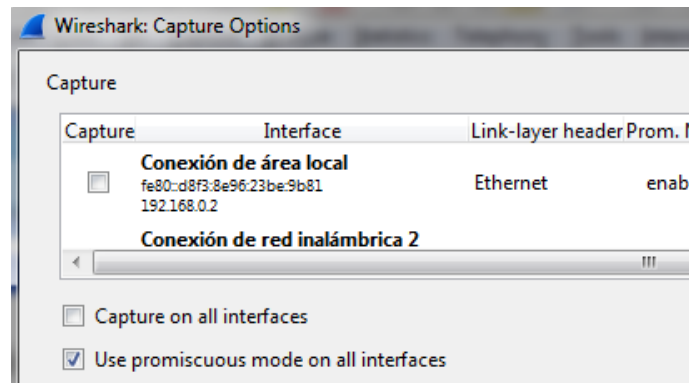


Figura 8.14. Opciones de captura con Wireshark

### 8.3.2. Chequeo

Se debe comprobar que:

- Se recibe correctamente una trama ARP y se envía el mensaje ARP correspondiente.
- Se recibe correctamente una trama UDP.
- El algoritmo se activa correctamente al final de la recepción de la trama UDP.
- El algoritmo se procesa correctamente pues los resultados de  $V_s(t)$  son iguales a los obtenidos en el primer test.
- Se activa la señal de final de procesado del algoritmo.
- Los resultados del algoritmo son correctamente enviados en la trama UDP.
- Se comprueba que los leds cumplen los requisitos descritos en la siguiente tabla. Para apagar los leds e iniciar un nuevo chequeo se debe resetear la FPGA.

LED	Testeado
0	Ha llegado un paquete a la FPGA
1	Ha llegado un paquete ARP a la FPGA
2	Ha llegado un paquete de datos UDP a la FPGA
3	Se inicia la transmisión de un paquete de datos UDP

Tabla 8.1. Comportamiento de los leds

- Con el analizador de paquetes de red Wireshark se testea que se transmiten paquetes ARP y UDP desde la FPGA y llevan los datos correctamente. Para verificar que el CRC se calcula adecuadamente se comprueba que el paquete es detectado por el analizador en modo no promiscuo.

#### 8.4. Validación del sistema

Una vez finalizada la verificación de cada uno de los bloques del diseño, se realiza la validación del sistema, en la que se comprueba que las especificaciones se cumplen en su totalidad y que el comportamiento del sistema, después de una realización física completa, es el correcto. El uso de la aplicación de usuario es fundamental para este proceso.

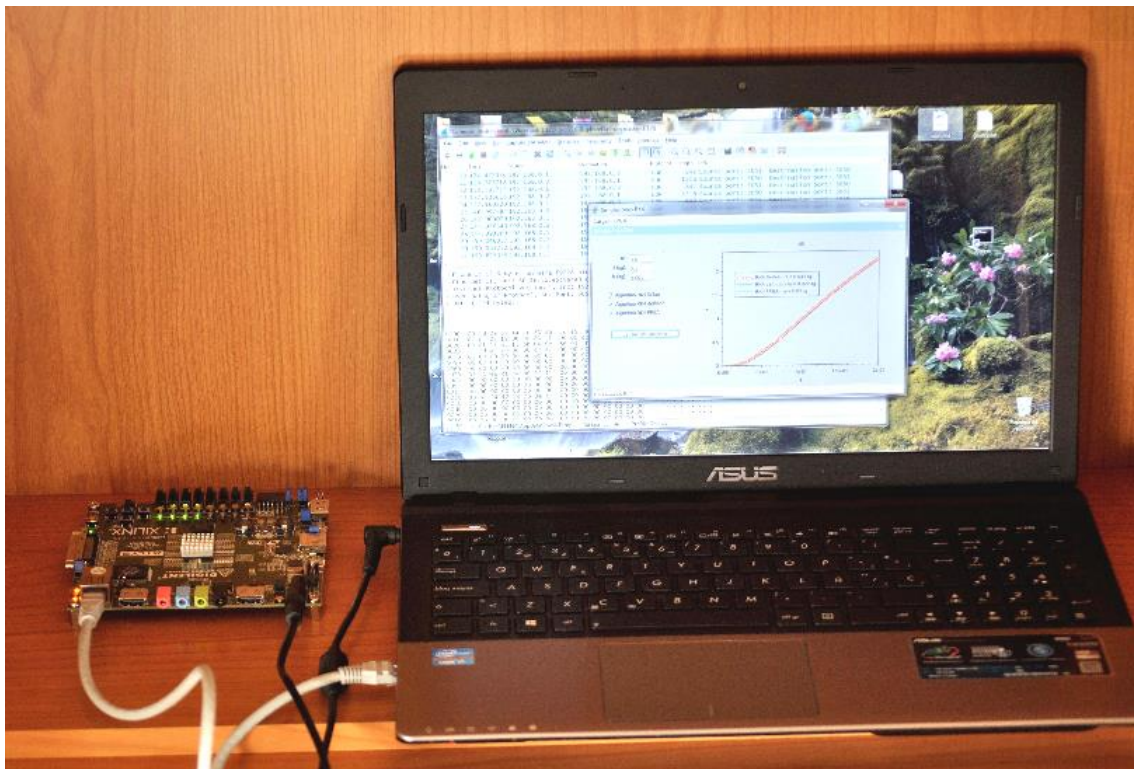


Figura 8.15. Integración del sistema final

### 8.4.1. Procedimiento

Se conecta el PC a la tarjeta con la FPGA. Con la aplicación de usuario se cargan cada de los circuitos almacenados en la librería *PATH\_PFC\scilab\circuits* y se ejecutan para cada uno de ellos los tres tipos de simulaciones.

### 8.4.2. Chequeo

Se comprueba, para todos los circuitos, que las representaciones gráficas con los resultados de las simulaciones son idénticas para los tres tipos de algoritmos.

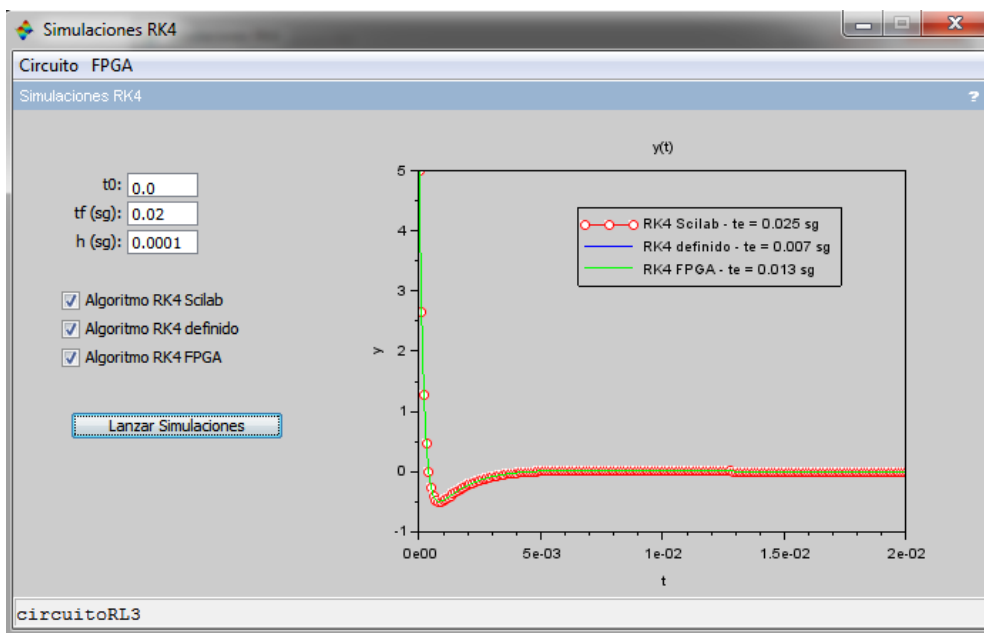


Figura 8.16. Comparación de las simulaciones del circuito RL de orden 3

## 8.5. Conclusiones

Las distintas descripciones que forman la arquitectura del sistema son verificadas empleando testbenches con VHDL. Los resultados de las simulaciones son contrastadas con los datos



obtenidos por consola al ejecutar simulaciones basadas en PC con la aplicación de usuario. Con un analizador de paquetes se comprueba la adecuada recepción y transmisión de tramas de datos.

El sistema final es validado chequeando el comportamiento del algoritmo con distintos circuitos y comparando el resultado con el obtenido por simulaciones basadas en PC.



## 9. ANÁLISIS DE RESULTADOS

### 9.1. Introducción

Una vez construido el sistema para simular circuitos eléctricos lineales sobre una FPGA interesa hacer un análisis de resultados a partir de observaciones realizadas durante su desarrollo y posterior uso.

En primer lugar se evalúa qué factores influyen en el crecimiento de la complejidad y del coste de la metodología. Posteriormente se analiza si este sistema basado en FPGA ofrece una mejora significativa en la velocidad de procesamiento de la simulación frente a otros sistemas basados en PC.

### 9.2. Crecimiento de la complejidad y del coste de la metodología

La dimensión de los circuitos a simular es el factor que complica el desarrollo del sistema. Por lo tanto, el parámetro que determina el aumento de la complejidad y del coste de la metodología es el orden máximo  $N_m$  de los circuitos o tamaño del problema.

Revisando la Sección 4.5 sobre el diseño conceptual, puede recordarse que la arquitectura del algoritmo contiene una combinación de  $N$  multiplicadores y  $N-1$  sumadores, para el cálculo de la operación del Paso 0, que impone que  $N$  sea una potencia de 2. La arquitectura sobre la FPGA está preparada para simulaciones de orden  $N$  mayor que 0 y menor o igual que 16, por lo que el siguiente orden máximo a emplear sería  $N_m = 32$ .

De la Tabla 6.2 del Capítulo 6 se deduce que un orden máximo superior a 64 exigiría una cantidad de lógica y un número de bloques DSP48A superior a lo disponible en la FPGA

seleccionada, lo que supondría el empleo de una FPGA más potente y otra tarjeta de evaluación más costosa.

La arquitectura del algoritmo es fácilmente parametrizable con  $N$  (ver el fichero `algorithm_package.vhd`) pero no ocurre lo mismo con el controlador Ethernet. Adaptar el controlador Ethernet a la recepción de sistemas de orden  $N > 16$  es más complicado, debido a que la información que envía el PC a la FPGA tiene ahora un mayor número de bytes. Las matrices  $A$ ,  $BV_e$  y  $C$  cambian su dimensión y una sola trama Ethernet no basta para transmitirla, puesto que el payload o campo de datos de un paquete Ethernet tiene un tamaño máximo de 1.440 bytes. Por ejemplo, para  $N = 32$ , la configuración contiene  $1 + 32 * 32 * 4 + 32 * 4 + 32 * 4 + 4 + 4 * 4 = 4.373$  bytes enviados en 4 tramas Ethernet.

### **9.3. Tiempos de procesamiento de simulación**

Es importante analizar las ventajas del sistema de simulación basado en FPGA con respecto a otros métodos de simulación basados en PC. La ventaja principal es la mejora de la velocidad de procesamiento de la simulación.

Se va a definir el tiempo de procesamiento de la simulación del circuito como el tiempo que tarda la aplicación Scilab en obtener los resultados de la simulación desde el instante que se solicita. No debe confundirse con el tiempo de simulación del circuito, que representa el tiempo durante el cual se van a recoger muestras del comportamiento de éste.

Para hacer un análisis comparativo de los tiempos de procesamiento de simulación empleados por los dos tipos de métodos de simulación (basado en FPGA y basado en PC), se fija el valor  $h = 0,001$  sg y se consideran tiempos de simulación de 20 ms, 200 ms, 2 sg y 20 sg lo que equivale a tomar 20, 200, 2.000 y 20.000 muestras respectivamente. Se realizan las medidas para modelos de sistemas lineales de órdenes  $N = 4, 8, 16$  y 32.

En la Tabla 9.1 se recogen los diferentes tiempos de procesado, empleando el campo *PC* para un algoritmo basado en PC y el campo *FPGA* para el algoritmo basado en FPGA.

Orden del circuito	Tiempo de simulación del algoritmo (sg)	Número de muestras	Tiempos de procesado del algoritmo (sg)	
			PC	FPGA
4	0,02	20	0,001	0,014
	0,2	200	0,020	<b>0,015</b>
	2	2.000	0,210	0,084
	20,0	20.000	6,010	0,840
8	0,02	20	0,002	0,014
	0,2	200	0,021	<b>0,015</b>
	2	2.000	0,271	0,084
	20,0	20.000	10,678	0,840
16	0,02	20	0,003	0,016
	0,2	200	0,026	<b>0,017</b>
	2,0	2.000	0,396	0,085
	20,0	20.000	20,343	0,850
32	0,02	20	0,004	0,017
	0,2	200	0,040	0,018
	2,0	2.000	0,676	0,087
	20,0	20.000	44,351	0,870

**Tabla 9.1.** Tiempos de procesado de las simulaciones

### **Simulaciones basadas en PC**

Se ejecutan las diferentes simulaciones seleccionando en la aplicación de usuario de Scilab la opción *Algoritmo RK4 definido* como algoritmo basado en PC. Se elige esta opción porque con este algoritmo se obtienen tiempos de simulación menores que con la elección *Algoritmo RK4 Scilab*. La diferencia entre ellas es que *Algoritmo RK4 definido* es de desarrollo propio empleando el mínimo número de operaciones y *Algoritmo RK4 Scilab* (Campbell, 2015: 78-85) utiliza la función de Scilab *ode()* que ejecuta una versión adaptativa del algoritmo de Runge-Kutta de orden 4. El algoritmo RK adaptativo asegura la estabilidad y puede cambiar el paso *h* de la simulación para minimizar el error de cálculo durante el procesado. Una introducción a los algoritmos de Runge-Kutta adaptativos puede obtenerse en (Medina, 2008: 158).

Para el caso  $N = 4$  puede cargarse el circuito *circuitoRC4* y para  $N = 8, 16$  y  $32$  los valores de las matrices de representación de los espacios de estado de estos modelos se generan de forma aleatoria desde la opción *Modelo* del menú *Cargar*.

### **Simulaciones basadas en FPGA**

En este caso el tiempo total del procesamiento de la simulación es la suma del tiempo de entrada de los datos de configuración del algoritmo en la FPGA, más el tiempo de procesamiento del algoritmo sobre la FPGA y más el tiempo de extracción de los resultados hacia el PC.

a) Para 200 muestras y  $N = 4, 8$  ó  $16$

Los tiempos de procesamiento de la simulación con la FPGA se pueden medir sólo para los casos de 200 muestras enviando modelos de sistemas lineales con órdenes  $N = 4, 8$  y  $16$ , situaciones para las que está preparada la arquitectura de la FPGA.

Si se ejecutan simulaciones con *algorithm\_tb* y *rk4\_tb* para sistemas de orden 4, 8 y 16 y se visualizan los resultados con el simulador ISim, se obtiene un tiempo de recepción de datos de 9,8 us (microsegundos) y un tiempo de transmisión de 6,8 us. El tiempo de procesamiento del algoritmo varía con  $N$  y se tabula en la siguiente tabla. En estas medidas no se tiene en cuenta el tiempo de emisión y recepción de una trama ARP en la activación del algoritmo por primera vez.

<b>N</b>	<b>Tiempo de procesamiento del algoritmo (us)</b>
4	584
8	930
16	1.560

**Tabla 9.2.** Tiempos de procesamiento del algoritmo con FPGA

En cambio, con la aplicación de Scilab se observa que el tiempo de procesado de simulación, para un modelo fijo, cambia en cada ejecución, tomando un valor promedio de 15 ms para  $N = 4$  y 8, y 17 ms para  $N = 16$ .

Esta discrepancia en los tiempos se debe a que influyen otros factores en el tiempo de procesado de la simulación, como por ejemplo, el flujo de paquetes de la red, que puede producir colisiones y tiempos de espera, o la prioridad que el sistema operativo da a la aplicación para transmitir los datos de configuración en buffer a la Ethernet y posteriormente para recibir la trama solución hacia el buffer de entrada.

b) Para 20, 2.000 ó 20.000 muestras y  $N = 4, 8$  ó 16

En aquellos casos en los que no puede emplearse el sistema para realizar la simulación puesto que el número de muestras es distinto de 200 deben realizarse estimaciones del tiempo de procesado de simulación. Por ejemplo, 20.000 muestras / (200 muestras / paquete) = 100 paquetes que por 15 ms son 1,5 sg.

Este tiempo estimado es mejorable considerando los siguientes puntos.

- No es necesario enviar la trama de configuración cada vez.
- La arquitectura puede modificarse para permitir el envío de mayor número de muestras resultado de la simulación. Un paquete UDP puede devolver 1.440 bytes como máximo en su campo de datos, lo que permite transmitir hasta 360 muestras (4 bytes en coma flotante) de  $V_s(t)$  por trama. Por ejemplo, 20.000 muestras / (360 muestras / paquete)  $\approx$  56 paquetes que por 15 ms serán 0,840 sg.

- También puede suponerse que cada vez que la arquitectura de la FPGA tiene preparadas 360 muestras envía una trama de resultados mientras continúa en paralelo el procesado del algoritmo.

c) Para 200 muestras y  $N = 32$

Terminar añadiendo que para realizar simulaciones de orden  $N \geq 32$  se requiere un cambio importante en la arquitectura, como ya se ha comentado en la sección anterior, lo que conlleva al envío de varias tramas de configuración adicionales.

### **Resumen**

De los anteriores datos puede concluirse que el tiempo de ejecución del algoritmo sobre PC, en comparación con el del algoritmo sobre FPGA, aumenta de forma significativa cuando se incrementa el número de muestras obtenidas para la simulación y apenas cuando se aumenta el orden del sistema.

Se deduce que usando el algoritmo sobre la FPGA, pero modificando el diseño para que la arquitectura pueda enviar varias tramas de datos UDP y así obtener más de 200 muestras de simulación, el tiempo de procesado es como mínimo un orden de magnitud menor que en el caso del procesado del algoritmo sobre PC.

## **9.4. Conclusiones**

Es construido un sistema basado en FPGA que permite ejecutar un algoritmo de resolución de ecuaciones diferenciales y obtener la simulación de sistemas lineales. El desarrollo del sistema está fuertemente limitado por el orden máximo de los circuitos a simular. En el caso de simulaciones con muchas muestras, el tiempo de procesado mejora en comparación con la ejecución del algoritmo sobre PC.



## 10. PLANIFICACIÓN Y COSTES DEL PROYECTO

### 10.1. Introducción

Seguidamente se describe el proceso de planificación en el que se definen las tareas realizadas en el proyecto, los tiempos de realización de cada tarea y los retrasos según lo planificado. Se incluye un apartado para contabilizar los costes monetarios y de mano de obra del proyecto.

### 10.2. Planificación

A continuación se detallan las diferentes tareas realizadas en el proyecto, identificándolas con el rótulo  $T_i$  donde  $i = 1, \dots, 7$ .

#### ***Tarea T1***

Instalación de Scilab e iniciación. Análisis de métodos numéricos para resolución de ecuaciones diferenciales. Comprensión del algoritmo RK4 e implementación con Scilab, empleando la función `ode()` o con código propio. Ploteado y cálculo de tiempos de procesado de la solución transitoria de sistemas con diferentes órdenes y número de muestras.

#### ***Tarea T2***

Modelado gráfico de circuitos RLC con Xcos. Obtención del modelo matemático de los circuitos a partir del esquema gráfico con el Análisis Nodal Modificado y desarrollo con Scilab.

### **Tarea T3**

Descripción del algoritmo RK4 con VHDL. Se selecciona Xilinx como tecnología a emplear. Creación de un proyecto con la herramienta de diseño ISE de Xilinx. Verificación de la descripción simulando con testbenches VHDL e ISim. Adquisición de la tarjeta de evaluación Atlys de Digilent con la FPGA Spartan-6 XC6SLX45.

### **Tarea T4**

Creación de controlador con Java y JIMS para intercambiar información con Ethernet desde Scilab. Descripción con VHDL y verificación del controlador Ethernet para la comunicación de la FPGA con el PC.

### **Tarea T5**

Diseño, programación y verificación de la aplicación de usuario desarrollada con Scilab. Se crea una interfaz gráfica con menús y diferentes controles para facilitar las distintas operaciones. Se incluyen las funciones de cargar modelo, configuración de simulación, ejecución de simulación y representación de resultados.

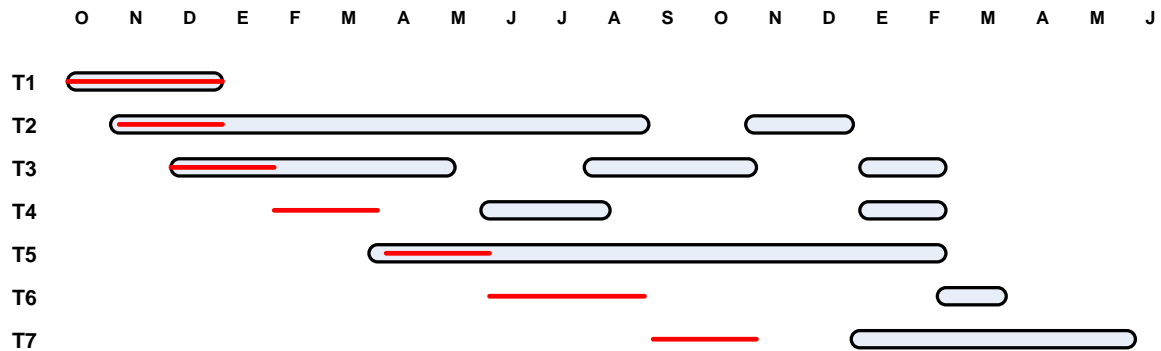
### **Tema T6**

Validación de la representación gráfica de la simulación basada en FPGA comparándola con los resultados obtenidos con Scilab. Análisis de los factores que influyen en el crecimiento de la complejidad y del coste de la metodología. Análisis de la diferencia de los tiempos de procesado empleados por los dos métodos de simulación.

### **Tarea T7**

Elaboración de la memoria.

En el siguiente diagrama de Gantt se representan los tiempos de realización de las distintas tareas y los retrasos según lo planificado. En el eje temporal se encuentran los diferentes meses desde octubre de 2013 hasta junio de 2015. La franja gruesa representa el progreso real de una tarea y la línea roja el progreso planificado.



**Figura 10.1. Diagrama de Gantt de progreso de tareas**

El retraso en la tarea T2 ha sido debido a la dificultad de encontrar una forma de obtener el modelo matemático de los circuitos RLC y plasmarlo con Scilab.

Con respecto a la tarea T3 comentar que el desarrollo del algoritmo con VHDL ha sido dilatado debido a la compleja sincronización de las señales que controlan las diferentes operaciones.

El desarrollo del controlador Ethernet en la tarea T4 se ha complicado porque el PC no comunicaba bien con la tarjeta hasta cambiar a un cable CAT-6. También ha llevado tiempo la descripción correcta del cálculo del campo CRC.

Para la tarea T5, el desarrollo de la aplicación de usuario se ha retrasado inicialmente porque se comenzó a realizar con Java sobre Eclipse y la API Javasci. La idea era abrir desde Java una sesión Scilab y ejecutar código Scilab desde ella. Así se podría importar el circuito y lanzar gráficas. Se descartó la idea por no poderse gestionar adecuadamente distintos hilos y lanzar ploteados.

La tarea T7 para la elaboración de la memoria se ha alargado debido a la falta de tiempo por razones laborales y familiares.

### **10.3. Costes del proyecto**

#### ***Costes monetarios de recursos materiales***

La tarjeta Atlys de Digilent ha costado 450 € y 4€ el cable CAT-4.

#### ***Costes de mano de obra***

La realización del proyecto ha supuesto el trabajo de 450 horas a una persona.

Suponiendo un sueldo de 60 €/hora el trabajo ha requerido 24.500 €.

### **10.4. Conclusiones**

Este proyecto ha conllevado un esfuerzo y tiempo superiores a lo planificado inicialmente debido a las dificultades encontradas en varios puntos entre los que destacan la elección correcta de las herramientas de diseño, la definición matemática del modelo a simular y el cumplimiento de ciertos requisitos de la especificación.

## 11. CONCLUSIONES Y TRABAJOS FUTUROS

### 11.1. Introducción

A continuación se exponen las conclusiones finales analizando si se cumplen los objetivos trazados inicialmente. Se finaliza este capítulo exponiendo los posibles trabajos futuros a realizar como continuación de los desarrollados en el proyecto.

### 11.2. Conclusiones

Se ha elegido el método de Análisis Nodal Modificado para obtener el modelo matemático de un circuito eléctrico RLC de corriente continua a partir de su esquema gráfico. Se ha seleccionado el algoritmo Runge-Kutta de orden 4 para generar la simulación de la respuesta transitoria del circuito partiendo de su modelo matemático.

Ha quedado evaluada la aplicabilidad de la metodología basada en VHDL para la obtención de la simulación dinámica del modelo matemático de un circuito RLC. En esta metodología se ha planteado una especificación de requisitos, se ha realizado el diseño de la arquitectura y su descripción con VHDL; se ha escogido Xilinx como tecnología de lógica programable y la tarjeta de evaluación Atlys de Digilent con la FPGA Spartan-6 XC6SLX45, se han verificado las descripciones por medio de testbenches y se ha implantado físicamente el sistema hardware. Se ha demostrado que son factibles la descripción correcta y eficiente del algoritmo, su síntesis e implementación y el cumplimiento de los requisitos temporales y lógicos exigidos por la arquitectura de la FPGA.

Se ha elegido la herramienta de simulación y modelado Scilab para crear un entorno de usuario con menús y otros diferentes controles que facilitan las siguientes operaciones: extraer el modelo

matemático de un circuito a partir de la representación gráfica realizada con Xcos, configurar y lanzar la simulación y representar gráficamente los resultados.

Se ha posibilitado la comunicación entre el PC y la FPGA creando dos controladores, uno programado con Java y embebido en la aplicación de usuario empleando las librerías JIMS de Scilab, y otro descrito con VHDL incluido en la arquitectura de la FPGA. Así, una vez configurada la FPGA existe la posibilidad de cargar el modelo del circuito en la FPGA y los resultados de la simulación dinámica pueden ser adecuadamente transferidos al PC y representados en la interfaz gráfica.

Ha sido validada la representación gráfica de la simulación basada en FPGA después de ser comparada con las obtenidas con otras simulaciones basadas en PC y desarrolladas con Scilab.

Se han calculado con Scilab y comparado los tiempos de procesado de las simulaciones tanto basadas en FPGA como en PC. Se ha concluido que mejora la velocidad de procesado cuando se emplea la FPGA en aquellos casos en los que hay un alto número de muestras.

Se ha observado que el orden de los circuitos a simular es el parámetro que determina el aumento de la complejidad y del coste de la metodología. El sistema está limitado a circuitos de orden en el rango de 1 a 16 y un aumento de este orden implica cambios importantes en la arquitectura, incluso el uso de otra FPGA con más recursos lógicos.

### **11.3. Trabajos futuros**

Una ampliación del trabajo realizado viene dada por la posibilidad de envío de varias tramas con los resultados de la simulación y la búsqueda de la forma más eficiente para su transmisión. Otras formas de lectura de resultados podrían ser planteadas.

La aplicación de usuario de este sistema puede adaptarse a otros tipos de sistemas físicos cuyas simulaciones pueden resultar interesantes.

Para finalizar, otras FPGAs con más lógica y bloques DSP pueden permitir aumentar el orden del sistema a simular. También se puede estudiar la posibilidad de incrementar la frecuencia del reloj para el algoritmo.





## BIBLIOGRAFÍA Y REFERENCIAS

ATTIA, John Okyere. *Electronics and circuit analysis using Matlab*. CRC Press LLC, Department of Electrical Engineering, Prairie View A&M University, 1999. Capítulos 4 y 5.

CAMPBELL, Stephen L.; CHANCELIER, Jean-Philippe; NIKOUKHAH, Ramine. *Modeling and Simulation in Scilab/Scicos*. Springer, 2006.

CHEN, Hao; SUN, Song; ALIPRANTIS, Dionysios C.; ZAMBRENO, Joseph. *Dynamic simulation of electric machines on FPGA boards*. Department of Electrical and Computer Engineering Iowa State University, 2009.

DIGILENT. *Atlys Board Reference Manual*. 2013.

[https://www.digilentinc.com/Data/Products/ATLYS/Atlys\\_rm\\_V2.pdf](https://www.digilentinc.com/Data/Products/ATLYS/Atlys_rm_V2.pdf)

FLOYD, Thomas L. *Fundamentos de sistemas digitales*. Prentice Hall, 2006.

HALL, Eric A. *Internet Core Protocols: The Definitive Guide*. O'Reilly, 1Ed., 2000.

HANKE, Michael. *An introduction to the Modified Nodal Analysis*. 2006.

<http://www.nada.kth.se/kurser/kth/2D1266/MNA.pdf>

MEDINA, José Díaz. *Ecuaciones diferenciales ordinarias*. Universidad de Valencia, 2008.

[http://www.uv.es/~diazj/cn\\_tema8.pdf](http://www.uv.es/~diazj/cn_tema8.pdf)

NIZZO MC INTOSH, Augusto. *Aproximación de soluciones de ecuaciones diferenciales por métodos iterativos*. 2009.

<http://mna.wdfiles.com/local--files/ode/tp6-13.pdf>

OPENEERING. *How to develop a Graphical User Interface (GUI) in Scilab*. 2012.

[www.openeering.com/sites/default/files/LHY\\_Tutorial\\_Gui.pdf](http://www.openeering.com/sites/default/files/LHY_Tutorial_Gui.pdf)

SPURGEON, Charles E. ; ZIMMERMAN, Joann. *Ethernet: The Definitive Guide*. O'Reilly, 2014.

SUHAG, Anuj. *Transient analysis of electrical circuits using Runge-Kutta method and its application*. School of Mechanical and Building Sciences, V.I.T University, 2013.

TANENBAUM, Andrew S; Wetherall David J. *Computer networks*. Pearson, 5Ed., 2010.

TERÉS, Lluís; TORROJA, Yago; OLCOZ, Serafín; VILLAR, Eugenio. *VHDL, lenguaje estándar de diseño electrónico*. Mc Graw Hill, 1998.

WARREN, Henry S. *Hacker's Delight*. Addison-Wesley, 2Ed., 2012.

WIKILIBROS. *Programación en VHDL*. Arquitectura. 2015.

[http://es.wikibooks.org/wiki/Programaci%C3%B3n\\_en\\_VHDL/Arquitectura](http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_VHDL/Arquitectura)

WIKIPEDIA. *Espacio de estados*. 2015.

[http://es.wikipedia.org/wiki/Espacio\\_de\\_estados](http://es.wikipedia.org/wiki/Espacio_de_estados)

XILINX. *FPGA Design Flow Overview*. 2008.

[http://www.xilinx.com/itp/xilinx10/isehelp/ise\\_c\\_fpga\\_design\\_flow\\_overview.htm](http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_fpga_design_flow_overview.htm)

XILINX. *Spartan-6 Family Overview*. 2011.

[http://www.xilinx.com/support/documentation/data\\_sheets/ds160.pdf](http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf)

XILINX. *Spartan-6 FPGA SelectIO Resources*. 2014.

[http://www.xilinx.com/support/documentation/user\\_guides/ug381.pdf](http://www.xilinx.com/support/documentation/user_guides/ug381.pdf)

XILINX. *Intellectual Property*. 2015a.

<http://www.xilinx.com/products/intellectual-property.html>

XILINX. *What is a FPGA?* 2015b.

<http://www.xilinx.com/fpga>

YILDIZ, Ali Bekir. *Modified Nodal Analysis-Based determination of transfer functions for multi-inputs multi-outputs linear circuits*. 2010.



## **SIGLAS, ABREVIATURAS Y ACRÓNIMOS**

<b>ARP</b>	Address Resolution Protocol
<b>CLB</b>	Configurable Logic Block
<b>CRC</b>	Cyclic Redundancy Check
<b>DSP</b>	Digital Signal Processing
<b>FF</b>	Flip-Flop
<b>FPGA</b>	Field Programmable Gate Array
<b>GMII</b>	Gigabit Media Independent Interface
<b>GUI</b>	Graphical User Interface
<b>IP Address</b>	Internet Protocol Address
<b>IP</b>	Intellectual Property
<b>ISE</b>	Integrated Software Environment
<b>JDK</b>	Java Development Kit
<b>JIMS</b>	Java Interaction Mechanism in Scilab
<b>JRE</b>	Java Runtime Environment
<b>JVM</b>	Java Virtual Machine
<b>LSB</b>	Least Significant Bit
<b>LUT</b>	Look Up Table
<b>MAC</b>	Media Access Control
<b>MSB</b>	Most Significant Bit
<b>PC</b>	Personal Computer
<b>RK4</b>	Runge-Kutta de orden 4
<b>RLC</b>	Resistor, Inductor y Capacitor
<b>RTL</b>	Register-Transfer Level
<b>UDP</b>	User Datagram Protocol
<b>VHDL</b>	Very High Speed Integrated Circuit Hardware Description Language



