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

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Síntesis de alto nivel basada en Xilinx Vivado para aceleradores hardware

Autor: Adrián Domínguez Hernández
Tutor(es): Antonio Núñez Ordóñez
Pedro Pérez Carballo
Fecha: Septiembre 2014

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

Versión: 2.0 – 14 junio 2011

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

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

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Síntesis de alto nivel basada en Xilinx Vivado para aceleradores hardware

HOJA DE FIRMAS

Alumno/a:	Adrián Domínguez Hernández	Fdo.:
Tutor/a:	Antonio Núñez Ordóñez	Fdo.:
Tutor/a:	Pedro Pérez Carballo	Fdo.:

Fecha: Septiembre 2014

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

Versión: 2.0 – 14 junio 2011

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

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

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Síntesis de alto nivel basada en Xilinx Vivado para aceleradores hardware

HOJA DE EVALUACIÓN

Calificación: .....

Presidente : Dr. D. Sebastián López Suárez Fdo.:

Secretario : Dr. D. José María Quinteiro González Fdo.:

Vocal : Dr. D. José ramón Sendra Sendra Fdo.:

Fecha: Septiembre 2014

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

Versión: 2.0 – 14 junio 2011

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Tabla de contenido

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Índice de figuras

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Índice de tablas

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Abstract

This work describes key concepts in the implementation of a real time event processor using high-level synthesis methodology based on Xilinx Vivado HLS. The design flow starts from a SystemC functional model and has been refined using high-level synthesis methodology to RTL microarchitecture. The process is guided with performance measurements (latency, cycle, time, power, resource utilization) with the objective of assuring the quality of the final system.

The results show that Xilinx Vivado HLS provides improvements in the use of resources despite the difficulties to handle certain aspects of system descriptions languages as SystemC.

Finally, some recommendations about high-level synthesis with Xilinx Vivado HLS are given.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Resumen

Este trabajo describe los conceptos clave en la implementación de un procesador de eventos en tiempo real utilizando una metodología de síntesis de alto nivel basado en Xilinx Vivado HLS. El flujo de diseño se inicia a partir de un modelo funcional SystemC y se ha perfeccionado usando una metodología de síntesis de alto nivel hasta obtener una microarquitectura RTL. El proceso es guiado con medidas de rendimiento (latencia, ciclo, el tiempo, la energía, la utilización de recursos) con el objetivo de asegurar la calidad del sistema final.

Los resultados muestran que Xilinx Vivado HLS proporciona mejorasen el uso de los recursos a pesar de las dificultades para manejar ciertos aspectos de lenguajes de descripción de sistemas como SystemC.

Por último, se dan algunas recomendaciones sobre la síntesis de alto nivel con Xilinx Vivado HLS.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Capítulo 1: Introducción

1. Antecedentes

Debido a la complejidad creciente en el diseño de sistemas en chip (SoC) se hace necesario incrementar el nivel de abstracción desde el que se captura su diseño. Tradicionalmente este proceso partía de una descripción hardware a nivel RTL en un lenguaje de descripción de hardware tal como VHDL y/o Verilog. Sin embargo, en la actualidad se tiende a crear descripciones algorítmicas que capturen la funcionalidad del diseño en un lenguaje de alto nivel, tal como C/C++ o en lenguajes especializados como SystemC o SystemVerilog. Estas especificaciones se transforman mediante técnicas de síntesis de alto nivel en la correspondiente descripción RTL.

Las metodologías de diseño ESL separan claramente las interfaces de comunicación del núcleo de procesamiento, permitiendo realizar un refinamiento progresivo de ambos aspectos por separado. Ello da lugar a representaciones alternativas tales como el modelado a nivel de transacciones o TLM.

La utilización de metodologías ESL facilitan el diseño de aceleradores hardware que aprovechen el paralelismo implícito del hardware para acelerar la ejecución de núcleos claves de la aplicación para su comportamiento en tiempo real. La identificación de dichos núcleos de procesamiento, la separación entre los dominios software y hardware y la creación de entornos de verificación completos son tareas se abordan en muy alto nivel. Por otra parte, la utilización de síntesis de alto nivel facilita la migración de metodologías *bottom-up*, donde el resultado final no se define por sus componentes sino que es la aplicación la que define las características del diseño, llevándolo hacia una metodología *top-down.*

Este trabajo fin de máster parte de los conceptos establecidos en los párrafos precedentes y los aplica al diseño de un coprocesador de eventos. El coprocesador de eventos ha sido diseñado en SystemC e implementado usando la metodología de síntesis propuesta por Cadence CtoS y realizado su prototipado en una plataforma basada en FPGA Virtex-5 de Xilinx. Cadence CtoS es un entorno genérico de síntesis de alto nivel, que soporta rutas de implementación hacia FPGA, como se ha mencionado, pero también hacía su implementación en ASIC.

La principal idea que pretende desarrollar este TFM es la evaluación de la metodología de síntesis de alto nivel propuesta por Xilinx Vivado HLS para su comparación en términos metodológicos y de calidad de los resultados obtenidos con Cadence CtoS. Esta comparación permitirá aportar conocimiento y experiencia en cuanto al desarrollo tanto de las interfaces de comunicación entre el coprocesador y el sistema principal y la optimización de su funcionalidad. De igual forma permitirá retroalimentar información relevante en cuando al modelado de IPs en alto nivel que sean utilizables en diferentes entornos de síntesis, analizando la compatibilidad y las técnicas de modelado que mejor se adapten en cada caso.

1.1 Procesamiento de eventos complejos

Se define un sistema en tiempo real (STR) como aquel sistema que interactúa activamente con su entorno con una dinámica conocida entre sus entradas, salidas y restricciones temporales para un correcto funcionamiento, de acuerdo con los conceptos de predictibilidad, estabilidad, controlabilidad y alcanzabilidad [1].

El correcto funcionamiento de estos sistemas depende tanto del resultado lógico producido en sus salidas, como del tiempo consumido en producir dichos resultados. De esta forma, atendiendo al entorno en el que se desarrolle la aplicación, existirá un tiempo límite que el sistema deberá cumplir para que sea exitoso, considerándose todos aquellos resultados

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

producidos después del límite como fallos del sistema aun cuando estos sean los valores esperados. Se puede hablar de dos tipos de STR: *Hard* RT y *Soft* RT. Para el primero de los casos la restricción es completa, con tiempos de respuesta estrictos. Para el segundo, los tiempos de procesamiento pueden estar comprendidos en un determino margen pero no son críticos cuando la salida sigue una determinada cadencia, aunque con cierta latencia inicial.

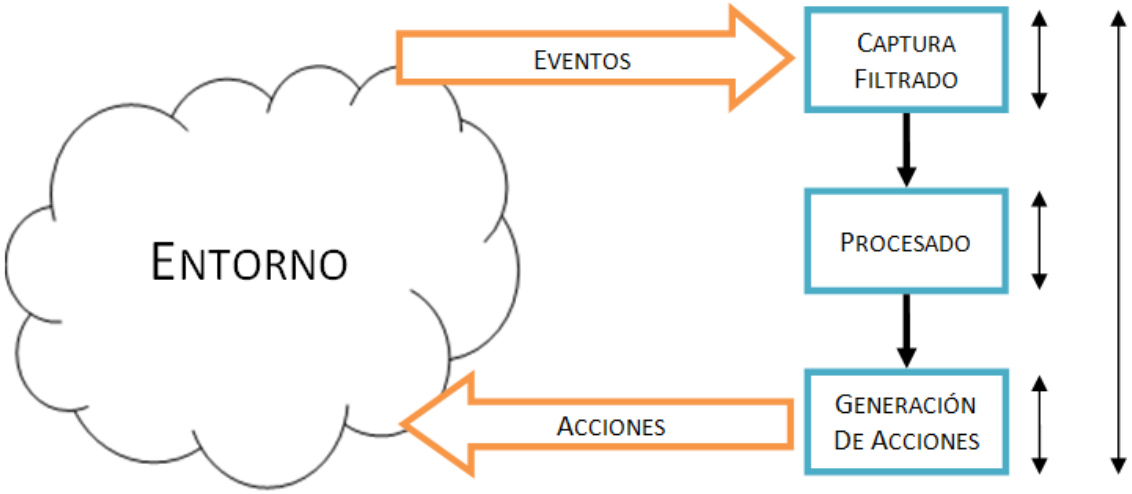


Figura 1. Esquema de un procesador de eventos.

Los sistemas de procesado de eventos son ejemplos típicos de STR. En ellos existe una tasa de llegada de eventos los cuales deben ser procesados en un tiempo límite. En general existirán unas condiciones, que en conjunto con los eventos, serán los que decidan qué acción deberá realizar. El procesado que llevará a cabo el sistema será comprobar si cada evento de entrada cumple, o no, algunas de las condiciones impuestas con anterioridad y decidir, en base a ello, la acción a desempeñar. Un ejemplo son los sistemas automatizados de compra y venta en el mercado de valores. Estos sistemas deben decidir si realizar una acción en función de los cambios en los precios de los productos y de unas estrategias previamente establecidas [2].

El funcionamiento de los sistemas de procesado de eventos se puede dividir en las siguientes etapas: captura y filtrado de eventos, procesado de los eventos y generación de las acciones. En la etapa de captura y filtrado, se requerirá que el sistema recolecte los eventos que se producen en su entorno y filtre aquellos parámetros que sean de utilidad para su posterior procesado, descartando aquella información no relevante. Durante el procesado se realizarán las operaciones y comparaciones que se hayan definido con el fin de comprobar si se cumple una o varias condiciones previamente establecidas. Por último, en la etapa de generación de acciones se enviará información sobre qué acciones se deben ejecutar en función de las condiciones calculadas en el procesador.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

La condición temporal que se debe cumplir, con el fin de no perder ningún evento, es que la suma de los tiempos dedicados a cada una de las etapas no supere el tiempo límite, t_r, que vendrá impuesto por la cadencia de eventos que llegan al sistema.

$$t_{captura} + t_{procesado} + t_{generación\ acción} < t_r$$

Debido a esta condición podemos considerar los sistemas de procesado de eventos como una aplicación de los sistemas en tiempo real.

1.2 Soluciones tecnológicas

Los sistemas de procesado de eventos de referencia para este Trabajo Fin de Máster pueden ser ejecutados sobre diferentes arquitecturas, teniendo cada una de ellas unas características determinadas.

Una primera posibilidad es la ejecución de la aplicación sobre un procesador de propósito general (GPP). Este tipo de dispositivos están preparados para poder realizar cualquier tarea, mediante la implementación de programas en el dominio software, lo que permiten una gran variedad de posibilidades. Su principal ventaja es la flexibilidad. Sin embargo tiene la desventaja de ser un método poco eficiente en términos de prestaciones para cumplir los requisitos de tiempo real.

En cuanto a la programación, para las aplicaciones a ejecutar sobre un GPP se pueden usar lenguajes de alto nivel como Java, Ada, Basic, C++, etc. El código generado será compilado para transformarlo a lenguaje máquina, es decir, un conjunto de instrucciones entendibles por el procesador.

En el lado opuesto se encuentran los circuitos integrados para aplicaciones específicas (ASIC). Al contrario que los GPP, se tratan de soluciones diseñadas de forma optimizada para una aplicación concreta, restringiendo su programabilidad pero permitiendo mejor eficiencia. Otra característica fundamental de este método es la posibilidad de concurrencia real entre tareas, al incluir en el circuito integrado diferentes módulos que realizan distintas operaciones en el mismo instante[3].

En los ASIC, el flujo de desarrollo tiene otra filosofía. Para el diseño de estos sistemas se usan normalmente lenguajes de descripción hardware (HDL) tales como Verilog o VHDL. Los bloques descritos en estos lenguajes son posteriormente sintetizados haciendo uso de las librerías tecnológicas de los fabricantes. En la actualidad se ha incrementado el nivel de abstracción hasta usar lenguajes de descripción de sistemas como puede ser C/C++ o SystemC [4], que tras una síntesis de alto nivel, transforma la descripción algorítmica en una microarquitectura

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

a nivel RTL que luego es optimizada e implementada siguiendo flujos tradicionales de síntesis[5] [6].

Mientras que en el dominio *software*, es decir, aplicaciones para ser ejecutadas sobre un GPP, se puede compilar, modificar y recompilar el código, permitiendo así optimizar el diseño tras varias iteraciones, en el diseño de un ASIC, el coste de la fabricación, tanto temporal como de capital, es elevado. Es necesario que el ASIC funcione correctamente desde el principio. Sin embargo esto requiere de un proceso de verificación complejo, por lo que el uso de FPGAs, facilita las tareas de prototipado, llegando incluso a utilizarse directamente para pequeñas series del sistema. Las FPGAs además poseen recursos de memoria internos, en algunos casos núcleos procesadores y otros bloques funcionales que facilitan la implementación de un sistema electrónico en chip (SOC)[7].

Se puede concluir que las principales ventajas en el uso de FPGAs frente a otras soluciones de diseño electrónico son su bajo coste durante las fases de desarrollo del sistema, lo que permite la producción de un reducido número de unidades (frente al coste de fabricar un ASIC) y su reprogramabilidad. Esta última propiedad permite una optimización post-diseño [8]. Es aquí donde esta tecnología tiene una ventaja competitiva para este tipo de problemas de procesado de eventos.

Los kits de diseño que proporcionan los fabricantes incluyen placas sobre las que vienen interconectadas la FPGA con diversos recursos. Muchos de estos incluyen bloques de interfaz como pueden ser USB, Ethernet, PS2, etc. Además, también incluyen sus propias fuentes de reloj, bloques de memoria e incluso CPUs.

Otras arquitecturas posibles son el uso de GPUs (*Graphics Processing Unit*) gracias a su potencia de cómputo. Los entornos de desarrollo proporcionados por los fabricantes hacen que la GPU sea cada vez más flexible, permitiendo su uso para la ejecución de aplicaciones o *General Purpose Computing on GPU* (GPCGPU). Aunque su programación no es tan flexible como la de las CPUs de propósito general, su gran cantidad de elementos de cómputo, junto a su arquitectura optimizada para la concurrencia mediante SIMD (*Single Instruction Multiple Data*) puede ser muy útil en el procesamiento de eventos, cuando quiere compararse una gran cantidad de ellos en una misma condición.

Por otro lado se encuentran los DSPs (*Digital Signal Processor*), que son procesadores optimizados para el procesado de señales, siendo normalmente especializados en operaciones del tipo MAC (Multiplicación y Acumulación). Este tipo de procesadores son más eficientes que las CPUs o las GPUs, pero mucho menos flexible en su programación.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

En la Figura 2se comparan las diferentes soluciones indicadas teniendo en cuenta su eficiencia frente a su flexibilidad. En este trabajo, como se verá más adelante, se trata de aumentar la eficiencia del sistema, moviendo funciones del dominio *software*, por tanto ejecutadas en un GPP, más flexibles, a su implementación en el dominio *hardware*, más eficientes al aumentar el paralelismo, pero menos flexibles. Con objeto de mantener una cierta flexibilidad en el desarrollo de estas funciones, se emplean técnicas de diseño basadas en la síntesis de alto nivel [9].

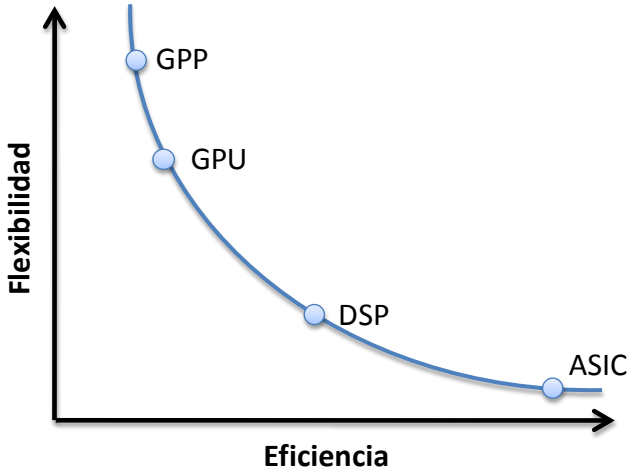


Figura 2. Comparativa de flexibilidad y eficiencia de las arquitecturas.

1.3 Metodologías de particionado HW/SW

La metodología clásica de diseño *hardware* hace del diseño de sistemas integrados una tarea compleja, por lo que con la evolución en la integración de estos sistemas, las metodologías de diseño han evolucionado en concordancia. En la Figura 3, adaptada de [10], se representa un diagrama de flujo del diseño de sistemas integrados. La primera tarea en este flujo consiste en analizar el diseño a realizar, y realizar la partición HW/SW. Esta partición consiste en decidir qué partes del diseño serán ejecutadas por el microprocesador del sistema integrado y qué partes serán ejecutadas mediantes bloques *hardware* específicos. La decisión debe estar basada en el coste de cómputo de cada función, así como de las necesidades de conectividad y dependencia entre ellos.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

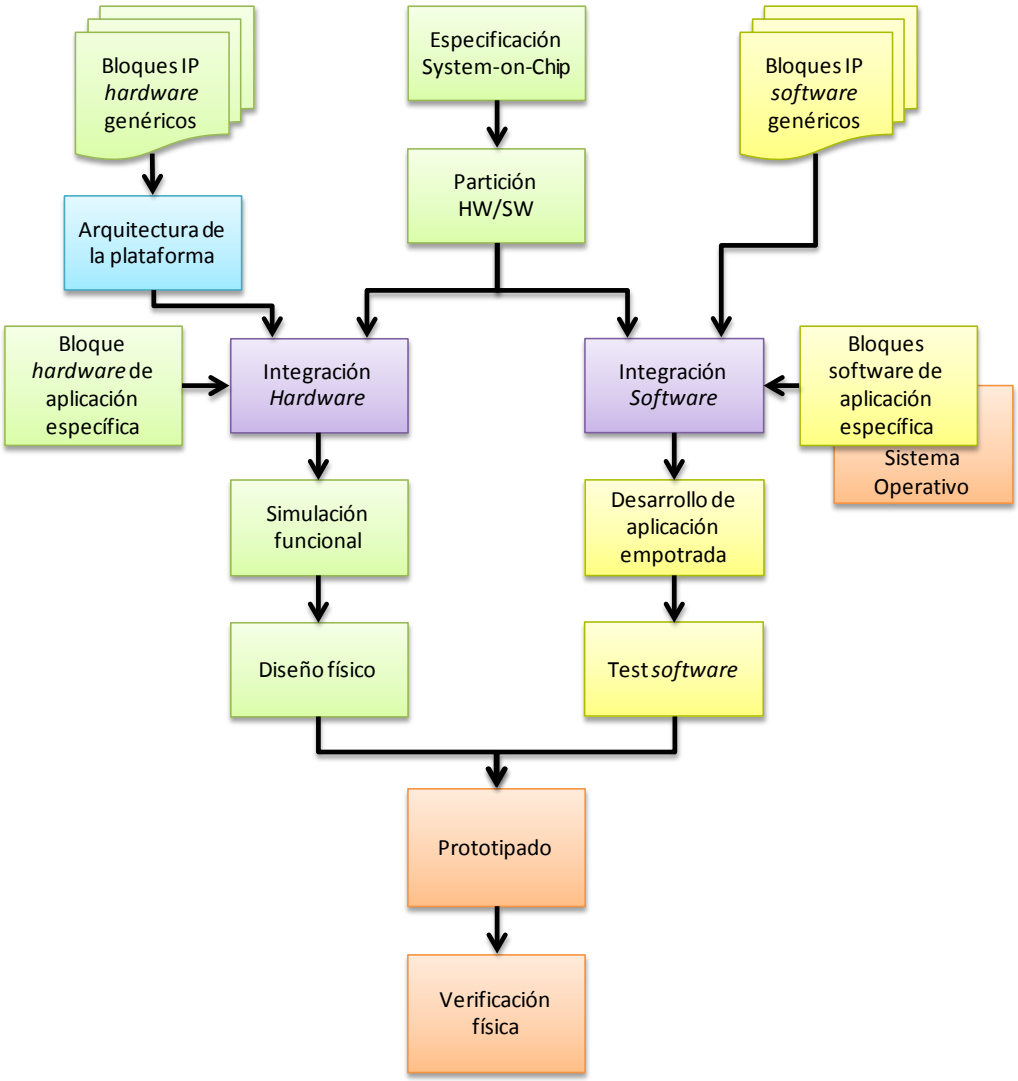


Figura 3. Diagrama de flujo del diseño de sistemas empotrados.

A partir de dicha partición, se realizarán, de forma independiente los diseños de la aplicación *software* empotrada y por otro la descripción *hardware* del resto de bloques. Cada uno de ellos seguirá el flujo clásico de compilación (en el caso *software*) y de síntesis (en el caso *hardware*), para, al final, realizar una integración de ambos.

Esta metodología busca acortar los tiempos de diseño, al ser posible reutilizar una gran parte de bloques *hardware* a conectar al microprocesador y que conforman la plataforma base del sistema integrado, como son controladores de memoria, de red, o de periféricos.

2. Objetivos

El objetivo principal de este TFM es el estudio de la metodología de síntesis de alto nivel propuesta por Xilinx Vivado.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Para esta tarea se define un flujo de diseño partiendo de la descripción en alto nivel en SystemC y realizando las tareas de síntesis necesarias hasta alcanzar una versión sintetizada en FPGA. Los resultados obtenidos en área, tiempo y potencia se compararán con los resultados obtenidos en el flujo de diseño basado en Cadence CtoS y se darán recomendaciones de diseño para compatibilizar ambos flujos de síntesis.

3. Estructura del documento

El presente documento se divide en siete capítulos en los cuales se describe el trabajo realizado, estructurado tal y como a continuación se indica:

- **Capítulo 1: Introducción.** Se detallan el marco general del trabajo, sus objetivos y la estructura del documento.
- **Capítulo 2: Metodología de diseño.** Descripción de las tecnologías, herramientas y lenguajes usados en el TFM, así como el flujo de diseño utilizado especificando secuencialidad de tareas y dependencias de las mismas.
- **Capítulo 3: Dominio de aplicación.** Se describen en este capítulo aspectos arquitecturales y conceptuales específicos del presente trabajo.
- **Capítulo 4: Desarrollo.** Este capítulo recoge los pasos llevados a cabo para, partiendo de un modelo de descripción *hardware* en alto nivel, alcanzar una descripción *netlist* del mismo adaptado al dispositivo de implementación.
- **Capítulo 5: Resultados y conclusiones.** Se presentan en este capítulo los resultados obtenidos y se realiza una comparativa entre los flujos de diseño basados en Xilinx Vivado HLS y el flujo de diseño basado en CtoS. Se exponen las conclusiones generales del Trabajo Fin de Máster.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Capítulo 2: Metodología de diseño

1. Introducción

La metodología de diseño utilizada para el desarrollo del procesador de eventos está basada en la utilización de estrategias de síntesis de alto nivel, tomando como referencia los algoritmos de la aplicación inicial que se desea acelerar. Gran parte del esfuerzo del diseñador se centra en la transformación a nivel algorítmico del código inicial en C/C++ al código encapsulado en SystemC, clases de C++, que soportan los conceptos de abstracción, estructura y tiempo, no disponibles en C++ [4]. Para ello se analiza el código inicial y se identifican aquellas partes críticas en cuanto a complejidad computacional, que serán las candidatas a ser movidas al dominio hardware para su aceleración.

El resto del flujo de diseño utiliza herramientas de síntesis lógica, simulación e implementación. De la misma forma la ruta hacia la implementación se basa en el uso de plataformas que dan soporte a funciones básicas del sistema final (acceso a red, etc.). Sin

embargo, debido a la complejidad del diseño, es necesario utilizar estrategias de implementación de tipo *bottom-up* para poder abordar los tiempos de compilación del diseño completo.

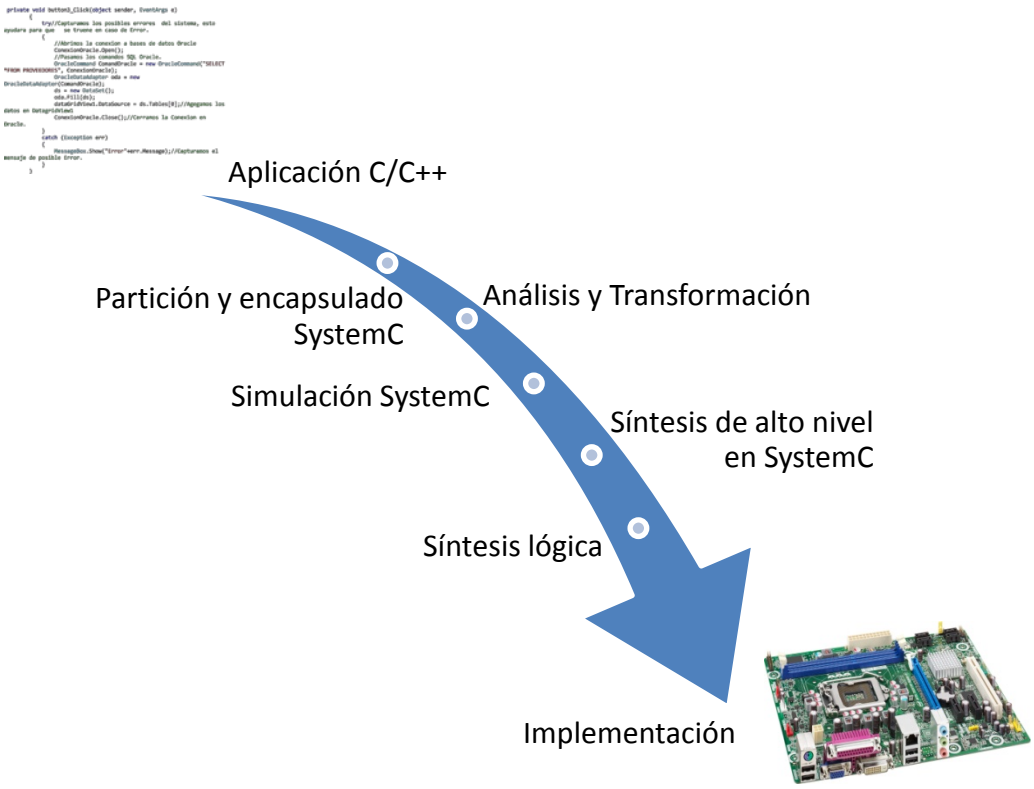


Figura 4. Flujo de diseño

2. Metodología de diseño

En todo diseño, ya sea *hardware*, *software* o de cualquier índole, se pueden enumerar tres fases claves en la metodología de diseño, a saber: especificación, diseño y verificación.

En la primera fase se analizarán las especificaciones del sistema a realizar, ya sean estas funcionales, temporales o de cualquier otra índole que se haya dado. Una vez definidas y acotadas, se procederá al diseño del sistema en el que se decidirá la tecnología, las etapas de desarrollo, la partición del diseño, si la hubiese. Ello produce una primera versión funcional del sistema que se utilizará para su verificación.

Con una primera versión terminada, se verificará que cumple con las especificaciones definidas en primera instancia, para, adoptar las medidas correctoras necesarias en fases

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

tempranas del ciclo de diseño, repitiendo el proceso hasta obtener una versión que cumpla con las especificaciones funcionales.

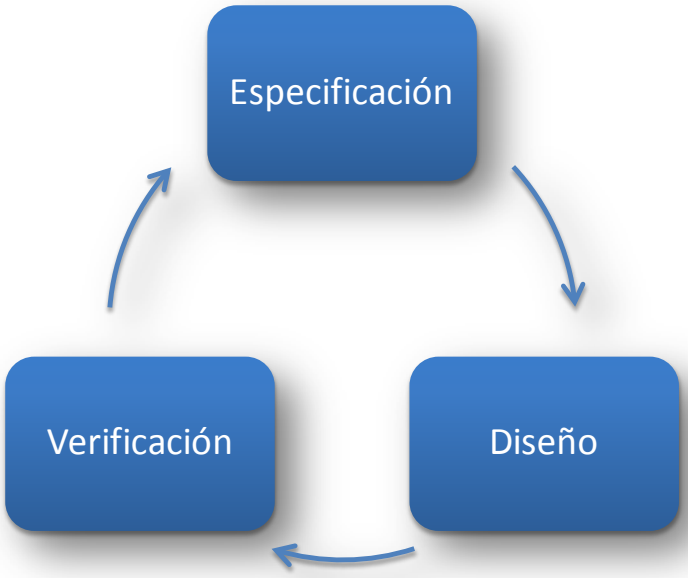


Figura 5. Flujo de diseño genérico.

En este Trabajo Fin de Máster se parte de un proyecto ya especificado, con todas sus fases completadas, verificadas mediante simulación y validado en un prototipo por lo que existe la certeza de que el modelo SystemC es funcional y temporalmente correcto. Las tareas de verificación y diseño serán las que permitan obtener un modelo SystemC sintetizable por la herramienta de síntesis de alto nivel.

El flujo de diseño que se propone es el que se muestra en la Figura 6. Partimos de un código SystemC verificado y funcional y realizamos las adaptaciones necesarias al subconjunto SystemC soportado por la herramienta de síntesis de alto nivel. En cada etapa se realiza una verificación funcional para comprobar que el funcionamiento no se ha visto afectado por las modificaciones realizadas.

2.1 Adaptación a SystemC sintetizable

En esta primera fase se transforma el sistema descrito en SystemC sintetizable en Cadence CtoS a un subconjunto sintetizable por la herramienta Xilinx Vivado y se realizarán las modificaciones necesarias sin que esto afecte a la funcionalidad. Se utilizarán directivas para

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

indicar a la herramienta el comportamiento deseado: desenrollado de bucles, implementación de arrays, *inlining* de funciones, precisión de los tipos de datos, especificación de las interfaces, etc.

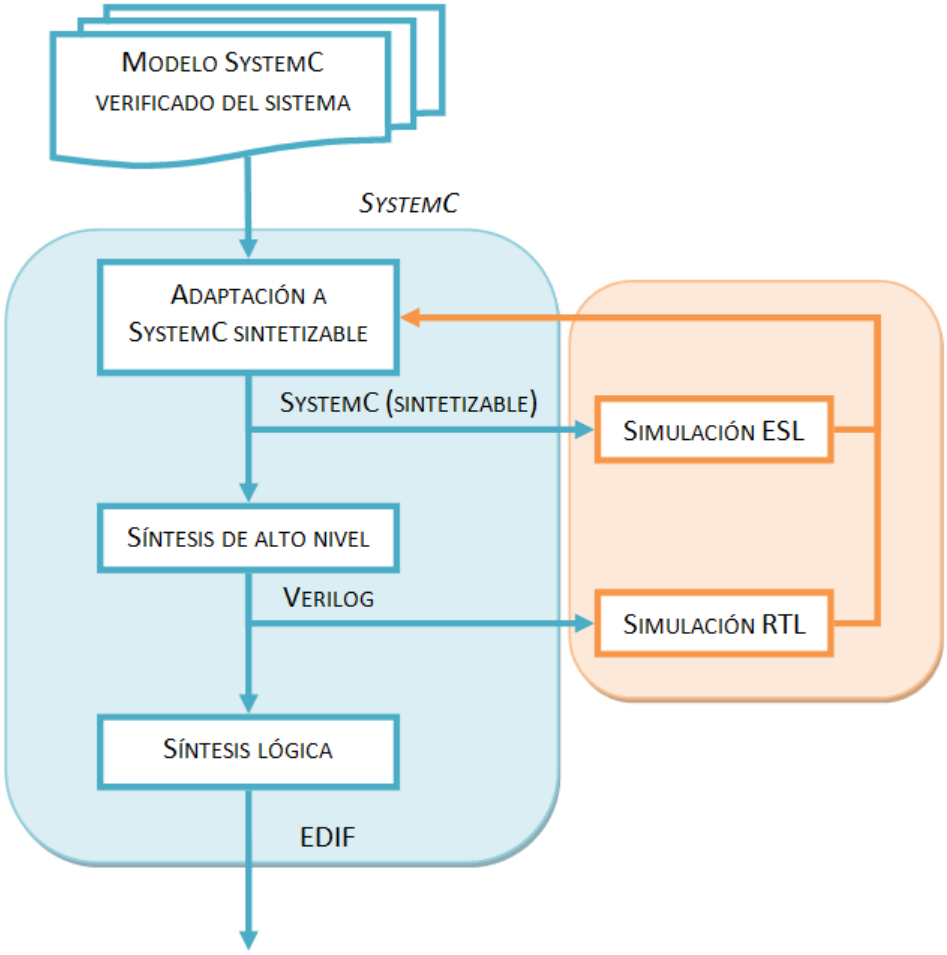


Figura 6. Flujo de diseño propuesto.

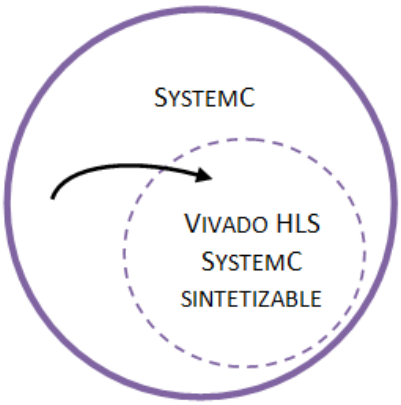


Figura7. Subconjunto sintetizable.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

2.2 Síntesis de alto nivel

Cuando la descripción del sistema haya sido verificada mediante simulación a nivel ESL, comparando su funcionalidad con la aplicación original, se debe realizar la primera fase de síntesis, que traducirá el diseño a una descripción a nivel RTL en VHDL o Verilog. Existen parámetros objeto de optimización a la hora de sintetizar el diseño, entre los cuales se destacan la latencia, el área o la potencia. En muchos casos habrá que establecer una estrategia final ya que no es posible optimizar todos los parámetros de forma simultánea. Es decisión del diseñador, en función de sus requisitos, podrá guiar la síntesis con el fin de obtener la solución deseada.

En todo momento deberá comprobarse que la solución obtenida tras la síntesis inicial mantiene la funcionalidad especificada inicialmente. En caso de disconformidades se modificará la descripción de alto nivel, de forma que la solución obtenida por la herramienta coincida con su diseño de entrada.

2.3 Síntesis lógica

Una vez obtenida una descripción a nivel RTL que cumpla con las especificaciones, se deberá realizar una síntesis lógica que la traduzca a un diseño compuesto por la interconexión de células básicas de la librería de la tecnología de implementación con la que se trabajará. Esta fase tiene por objetivo la obtención de un *netlist* (fichero de instancias de células básicas y sus conexiones), generalmente NGC o EDIF. Los parámetros objeto son por una parte la determinación del tiempo de ciclo y por otra la optimización del área/recursos utilizados.

2.4 Verificación

La fase de verificación se realizará a distintos niveles de abstracción, así como a distintas vistas del diseño. Así, se realiza la verificación funcional del sistema tanto en su descripción ESL como a nivel RTL, verificación física tras su implementación, así como verificación de las restricciones temporales tras un análisis temporal estático del diseño.

3. Tecnologías

Se presentan a continuación las tecnologías de implementación con las que se trabajará en este TFM.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

3.1 Zynq7000

La familia de dispositivos Zynq-7000 (Figura 8) combina la capacidad de programación software de un procesador ARM Cortex A9 con la capacidad de programación hardware de una FPGA, lo que proporciona altos niveles de rendimiento del sistema, flexibilidad y escalabilidad, además de reducción de potencia, menor costo de desarrollo. A diferencia de las soluciones tradicionales de procesamiento de SoC, la lógica programable flexible de los dispositivos Zynq-7000 permite la optimización y la diferenciación, lo que facilita a los diseñadores agregar periféricos y aceleradores para adaptarse a una amplia gama de aplicaciones. Los componentes principales del sistema son los siguientes:

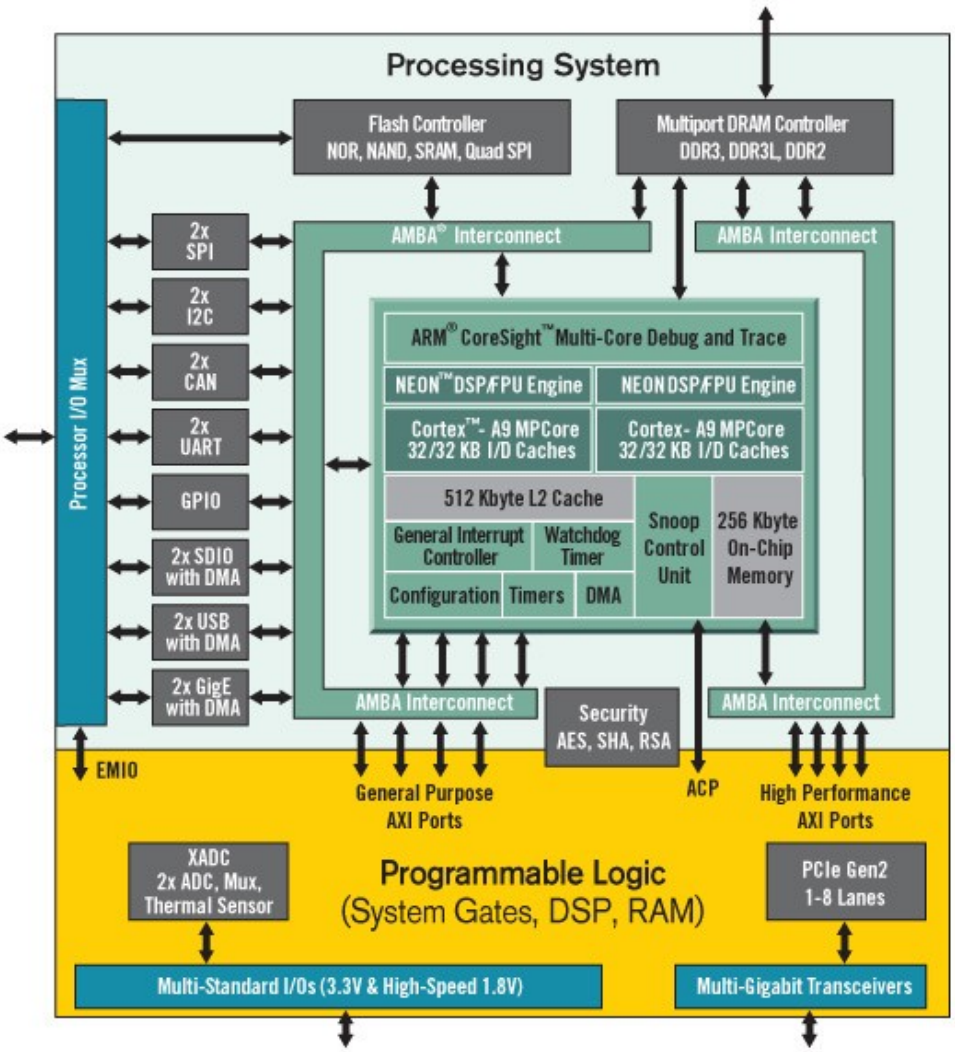


Figura 8. Arquitectura de la plataforma Zynq 7000EPP.

- Configuración:
  - Circuitería de configuración en placa
  - 16MB Quad SPI Flash
  - Interfaz para tarjeta SDIO (boot)

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

- Puertos PC4 y 20 pin JTAG

- Memoria:
  - MemoriaDDR3 1GB
  - Soporte para 32 bits de ancho
  - 16MB Quad SPI Flash
  - IIC - 1 KB EEPROM

- Comunicación y redes:
  - Interfaz Gigabit Ethernet GMII, RGMII and SGMII
  - USB OTG 1 (PS) - Host USB
  - IIC Bus Headers/HUB (PS)
  - 1 CAN con Wake on CAN (PS)
  - USB UART (PS)

- Vídeo/Pantalla:
  - Salida de vídeo HDMI
  - 8X LEDs

- Conectores de expansión:
  - Conectores FMC
  - Expansión IIC HUB
  - Dual Pmod (8 I/O Shared with LED's)
  - Single Pmod (4 I/O Shared with PJTAG)

- Reloj:
  - Oscilador fijo a 200MHz (Diferencial LVDS)
  - Oscilador programable I2C (por defecto a 156.25MHz)(Diferencial LVDS)
  - Oscilador fijo a 33.33MHz (Single-Ended CMOS)

- Control y I/O
  - 3 botones
  - 2 interruptores
  - 8 EDs

3.2 FPGAs Serie-7

3.2.1 Descripción general

El dispositivo Zynq incluye un bloque FPGA de la serie 7 de Xilinx, ya sea Artix-7 en la gama baja o Kintex-7 en la gama alta.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

La serie 7 de las FPGAs de Xilinx comprende tres nuevas familias de FPGA dirigidas a un amplio rango de requisitos, desde bajo coste, tamaño reducido, grandes volúmenes hasta ultra alto ancho de banda, alta capacidad lógica y de procesado de señales para la mayoría de aplicaciones de alto rendimiento. La serie 7 incluye las siguientes familias:

- Artix-7: optimizada para bajo costo y consumo con reducido tamaño para aplicaciones de gran volumen.
- Kintex-7: optimizada para obtener la mejor relación precio-rendimiento con el doble de mejora en comparación con generaciones anteriores, haciendo posible una nueva clase de FPGAs.
- Virtex-7: optimizada para el mejor rendimiento y capacidad. La tecnología de interconexión apilada de silicio (Stacked Silicon Interconnect, SSI) permite dispositivos de muy alta capacidad.

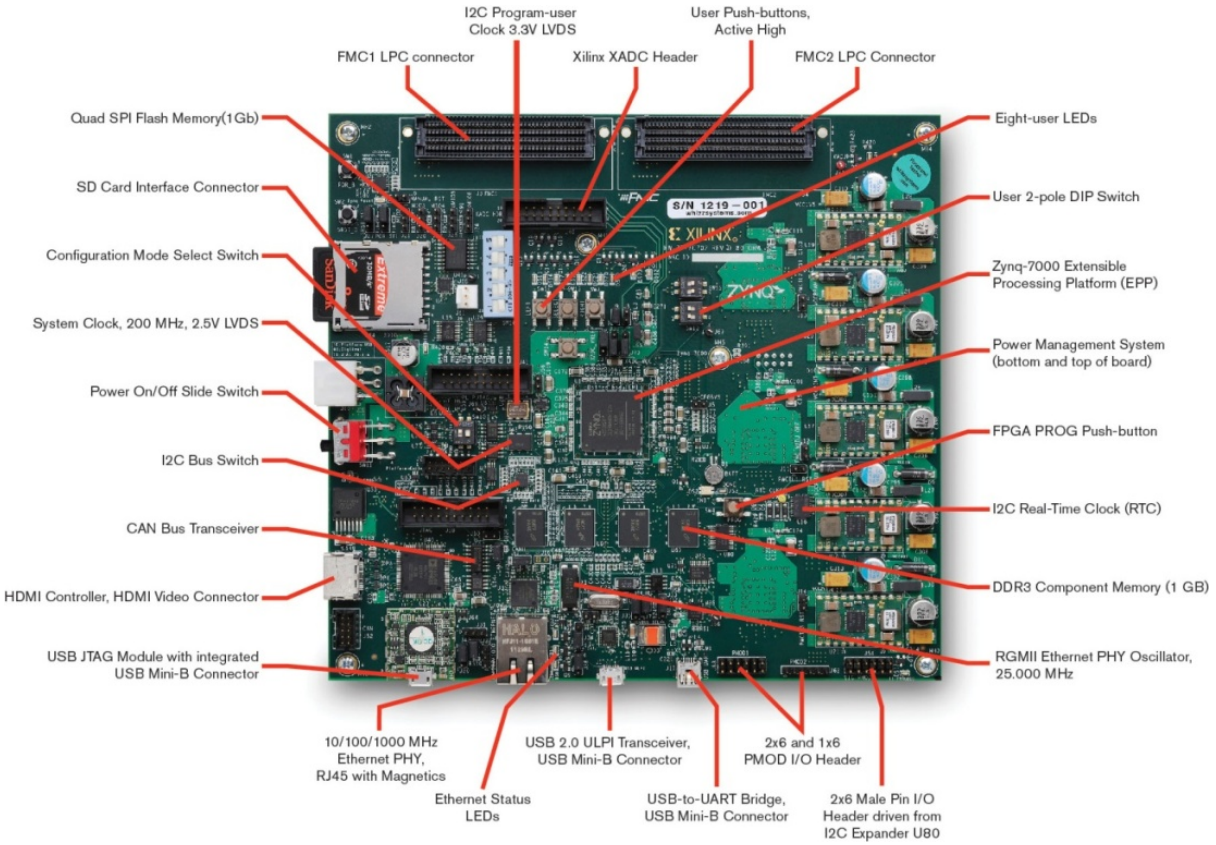


Figura 9. Plataforma de desarrollo ZC702.

Basadas proceso tecnológico High-K Metal Gate (HKMG) de alto rendimiento, bajo consumo (HPL) y CMOS 28nm, la serie 7 permite un aumento en el rendimiento del sistema con 2,9Tb/s de ancho de banda de I/O, capacidad de 2 millones de celdas lógicas y DSP con 5.3TMAC/s, mientras reduce un 50% el consumo en comparación a dispositivos de la generación anterior para ofrecer una alternativa totalmente programable para ASSPs y ASICs.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

3.2.2 Características

- Basada en LUT de 6 entradas que pueden ser configuradas como LUTs, registros de desplazamiento o como memoria distribuida.
- Memoria BRAM de doble puerto de 36 Kb con lógica FIFO integrada para *buffering*.
- Tecnología SelectIO de alto rendimiento con soporte para interfaces DDR3 hasta 1866 Mb/s.
- Conectividad serie de alta velocidad con transceptores integrados multi-gigabit desde 600 Mb/s con tasas máximas de 6.6 Gb/s hasta 28.05 Gb/s, ofreciendo un modo especial de bajo consumo, optimizado para interfaces chip-a-chip.
- Interfaz analógica configurable por el usuario (XADC), incorporando dos convertidores analógico/digital de 12-bit con sensores de corriente y temperatura.
- DSPs con 25 multiplicadores de 18-bit, acumulador de 48-bit y pre-sumador para filtrado de alto rendimiento, incluyendo filtrado de coeficientes simétricos optimizado.
- Bloques de gestión de reloj (CMT), combinando PLL y bloques mixtos de gestión de reloj (MMCM) para alta precisión y bajo *jitter*.
- Bloque integrado para PCI Express (PCIe), para diseños hasta x8 Gen3 Endpoint and Root Port.
- Amplia variedad de opciones de configuración, incluyendo soporte para acomodación de memorias, encriptación AES de 256-bit con autenticación HMAC/SHA-256, y detección y corrección SEU integrada.

4. Lenguajes

Debido a la creciente complejidad de los diseños de sistemas electrónicos, se hace necesaria la aparición de nuevos métodos para realizar el modelado y verificación del diseño. En la actualidad, hay una tendencia cada vez mayor a la descripción del sistema electrónico a nivel de sistemas (ESL) [11], por lo que no han tardado en aparecer lenguajes de descripción hardware con un nivel de abstracción mayor tales como son SystemC o SystemVerilog.

4.1 SystemC

Es el lenguaje usado para realizar la descripción *hardware* del procesador de eventos diseñado. Se detallan a continuación algunos conceptos de interés para las metodologías de diseño *hardware*.

Para profundizar en las características de SystemC se recomienda la lectura de [12], el manual de referencia de la última versión de la librería, así como [13] con guías de diseño y ejemplos.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

4.1.1 Definición

SystemC es una librería de clases de C++ desarrollada conjuntamente con una nueva metodología de diseño y verificación funcional para describir en alto nivel *hardware* a nivel de sistemas.

El uso de SystemC en conjunto con las herramientas estándar de desarrollo para C++ tiene como objeto poder modelar a nivel de sistema el diseño electrónico, y poder verificarlo con los menores costes posibles, además de conseguir un modelo que cumpla las especificaciones con el fin de que la siguientes fases de diseño pueda ser usado como referencia de funcionalidad.

La razón de elegir C++ para el desarrollo de la librería SystemC es que se trata de un lenguaje de alto nivel muy extendido, además de que sus compiladores generan códigos muy eficientes.

SystemC proporciona aquellas características que no están soportadas por C++, necesarias para el desarrollo de sistemas electrónicos:

- **Noción temporal**: C++ no permite conocer los instantes temporales en los que se producen los eventos.
- **Concurrencia**: C++, y en general cualquier lenguaje orientado al diseño de *software* no está preparado para describir sistemas concurrentes, algo que es intrínseco en el diseño de *hardware*.
- **Tipos de datos**: Los tipos de datos que soportan los lenguajes de programación no contemplan factores necesarios en el *hardware* como son el valor de alta impedancia Z en una señal.

La librería de SystemC, a través de la definición de nuevas clases de objetos en C++, da solución a estas necesidades sin redefinir sintácticamente el lenguaje. Además permite realizar una coverificación HW/SW insertando código en C y C++ en su diseño.

4.1.2 Elementos principales

A continuación se detallan algunos elementos de relevancia en el modelado de sistemas con SystemC.

4.1.2.1 Módulos

Es la clase usada por SystemC para modelar la estructura del diseño, dando soporte a conceptos tales como jerarquía, interconexión, etc. Un módulo encapsula un componente del diseño, que puede contener a su vez un conjunto de módulos interconectados a través de canales.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

```
SC_MODULE (adder_reg) {
    …
};
```

4.1.2.2 Puertos

En cada módulo podrán definirse puertos de entrada, de salida y bidireccionales, con el fin de interconectar módulos entre sí, o si fuese el módulo jerárquicamente superior, para representar las entradas y salidas del diseño. Además del sentido del puerto, cada uno podrá tener un tipo de datos, que podrá ser cualquier tipo de datos soportado por C/C++, tipos definidos por el usuario, o tipos de la librería SystemC.

```
sc_in<bool> clk;
sc_in<sc_int<8>> a;
sc_in<sc_int<8>> b;
sc_out<sc_int<9>> c;
```

4.1.2.3 Señales/canales

Al igual que los puertos sirven para comunicar, pero en lugar de módulos la comunicación se realiza entre procesos. Las señales son internas a cada módulo, y no son visibles desde otros módulos externos al contenedor de la señal en cuestión. Como los puertos, los datos pueden ser de tipos de C/C++, de usuario o de SystemC.

```
sc_signal<sc_int<9>> temp;
```

4.1.2.4 Procesos

En ellos se describe las funcionalidad de un módulo, que consistirá en un código escrito en C/C++ haciendo uso tanto de funciones del lenguaje raíz, como de métodos de las clases de la librería SystemC. Existen tres tipos de procesos: SC_METHOD, SC_THREAD, SC_CTHREAD.

SC_THREAD

Son procesos que solo se ejecutan una vez al comienzo de la ejecución del sistema. Sin embargo, permiten suspender el proceso con el fin de ser reanudado en el mismo estado en el que fue suspendido cada vez que se produce un evento en su lista de sensibilidad. Cada vez que la ejecución del proceso encuentre una sentencia *wait();* entrará en suspensión. En general, los procesos de este tipo suelen definirse como un bucle infinito, de forma que cuando termine vuelva a ejecutarse, y se define su latencia como el número de eventos necesario para una

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

ejecución del bucle. Puede usarse para definir la ejecución de un camino de datos segmentados, por ejemplo.

A continuación se muestra un ejemplo de declaración, definición, registro y lista de sensibilidad de un proceso de tipo SC_THREAD. Sin embargo, se recomiendo que la definición del mismo se encuentre separado en un fichero *.cpp, y el resto en un fichero de cabecera *.h [14].

```
SC_MODULE (adder_reg) {
  …
  void reg();
  …
  void adder_reg::reg() {

    while (true) {
      …
      wait();
      …
  }
  }
  …
  SC_THREAD(reg);
  sensitive<< clk.pos();
```

SC_CTHREAD

Se trata de una modificación sobre los procesos SC_THREAD. Su principal diferencia es que en el registro se le añade un evento de reloj como sensibilidad. Un ejemplo es la realización de una suspensión que se reanude un número de eventos más tarde, en lugar de en el siguiente evento..

Estas formas de suspensión pueden también conseguirse con el proceso SC_THREAD como se muestra a continuación, pero requiere que el *kernel* reanude el proceso para volver a ponerlo en reposo en cada evento, haciendo la ejecución más lenta.

A continuación se muestran los modos de suspensión comentados, a la izquierda en su implementación para un SC_THREAD y su equivalente en un SC_CTHREAD a la derecha.

```
for (i=0; i!=N; i++) wait();
do wait() while(!expr);
```

```
wait(N);
wait_until(expr.delay);
```

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

SC_METHOD

Los procesos de tipo SC_METHOD se ejecutan cada vez que se produce un evento en su lista de sensibilidad y no pueden ser suspendidos. En general se usan para simular un comportamiento combinacional ya que se ejecutan en tiempo cero y no se suspende para volver a reanudarse. Al igual que en el tipo SC_THREAD se debe indicar en la lista de sensibilidad los eventos que provocarán la ejecución del método.

```
SC_MODULE (adder_reg) {
   …
   void add();
   …
   void adder_reg::add() {
   …
   }
   …
   SC_METHOD(add);
   sensitive<< a << b;
```

A continuación se detalla un ejemplo de diseño de un sumador con la salida registrada en el que se hace uso de dos procesos en paralelo. Uno de ellos se encargará del proceso combinacional de realizar la suma y otro de registrar la señal [15].

```
#include "systemc.h"

SC_MODULE(adder_reg) {

   sc_in<sc_int<8>> a;
   sc_in<sc_int<8>> b;
   sc_out<sc_int<9>> c;
   sc_in<bool> clk;

   sc_signal<sc_int<9>> temp;

   void add() { temp = a + b; }
   void reg() { while (1) {c = temp; wait();} }

   SC_CTOR (adder_reg) {

      SC_METHOD(add);
      sensitive<< a << b;

      SC_THREAD(reg);
      sensitive<< clk.pos();

   }
};
```

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

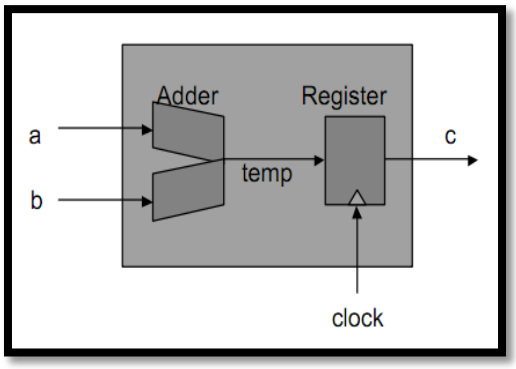


Figura 10. Diseño de ejemplo de uso de procesos en SystemC.

Como puede observarse en el ejemplo anterior, por un lado existe un proceso de tipo SC_METHOD que se ejecuta en tiempo cero cada vez que se produzca un evento en alguno de los puertos de entrada (a y/o b), realizando la suma de ambos y almacenándola en *temp*. Por otro lado, en cada flanco de subida del reloj, el valor almacenado en *temp* se escribe en el puerto de salida *c*. Esta descripción cumple funcionalmente con el diseño propuesto en la Figura 10, incluyendo que en un ciclo hayan varias modificaciones de las señales de entrada, almacenándose en la salida la última antes del flanco de subida del reloj (el proceso SC_METHOD se ejecutará, tantas veces como cambios haya en los puertos de entrada, pero solo se escribirá en el puerto de salida el valor que tuviese en el momento en el que se reanuda el proceso SC_THREAD).

4.1.2.5 Relojes

Los relojes son señales con una notación temporal, que permite dotar al sistema de relaciones temporales en su simulación. Se pueden definir varios relojes, con distintos tiempos de ciclo, ciclos de trabajo, así como fase relativa arbitraria.

```
sc_clock clk("clk", 10, SC_NS, 0.5, 0.0, SC_PS);
```

En el ejemplo anterior se ha declarado un reloj llamado *clk*, con un periodo de 10 nanosegundos, un ciclo de trabajo del 50% y un desfase de 0 picosegundos.

Las variables temporales en SystemC se almacenan como un entero de 64 bits y las unidades en las que se pueden representar son: *SC_FS*, *SC_PS*, *SC_NS*, *SC_US*, *SC_MS*, *SC_SEC*.

4.1.3 Características principales

A continuación se detallan algunas características fundamentales de SystemC que hacen de él una buena alternativa en el diseño hardware de alto nivel.

- **Simulación basada en eventos.** Al contrario que en una simulación por ciclos de reloj, la simulación basada en eventos presenta unos tiempos de simulación mucho

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

más bajos, ya que no precisa de la ejecución de todos los procesos paralelos del sistema en cada ciclo de reloj, sino que pueden haber varios suspendidos en espera de que se cumpla uno de sus eventos.

- **Múltiples niveles de abstracción.** SystemC permite realizar modelos *untimed* con varios niveles de abstracción, desde funcional hasta RTL. Esto permite refinar iterativamente el código hasta llegar a un modelo RTL que cumpla la funcionalidad y usarlo como fuente para pasar a un diseño en VHDL o Verilog y de ahí seguir el flujo estándar de diseño electrónico.
- **Trazado de formas de onda.** Con SystemC es también posible el trazado de formas de ondas en los formatos VCD, WIF e ISDB.

4.1.4 Estructura de capas

La Figura 11 muestra la arquitectura del lenguaje SystemC. Los bloques centrales son el núcleo de SystemC [16].

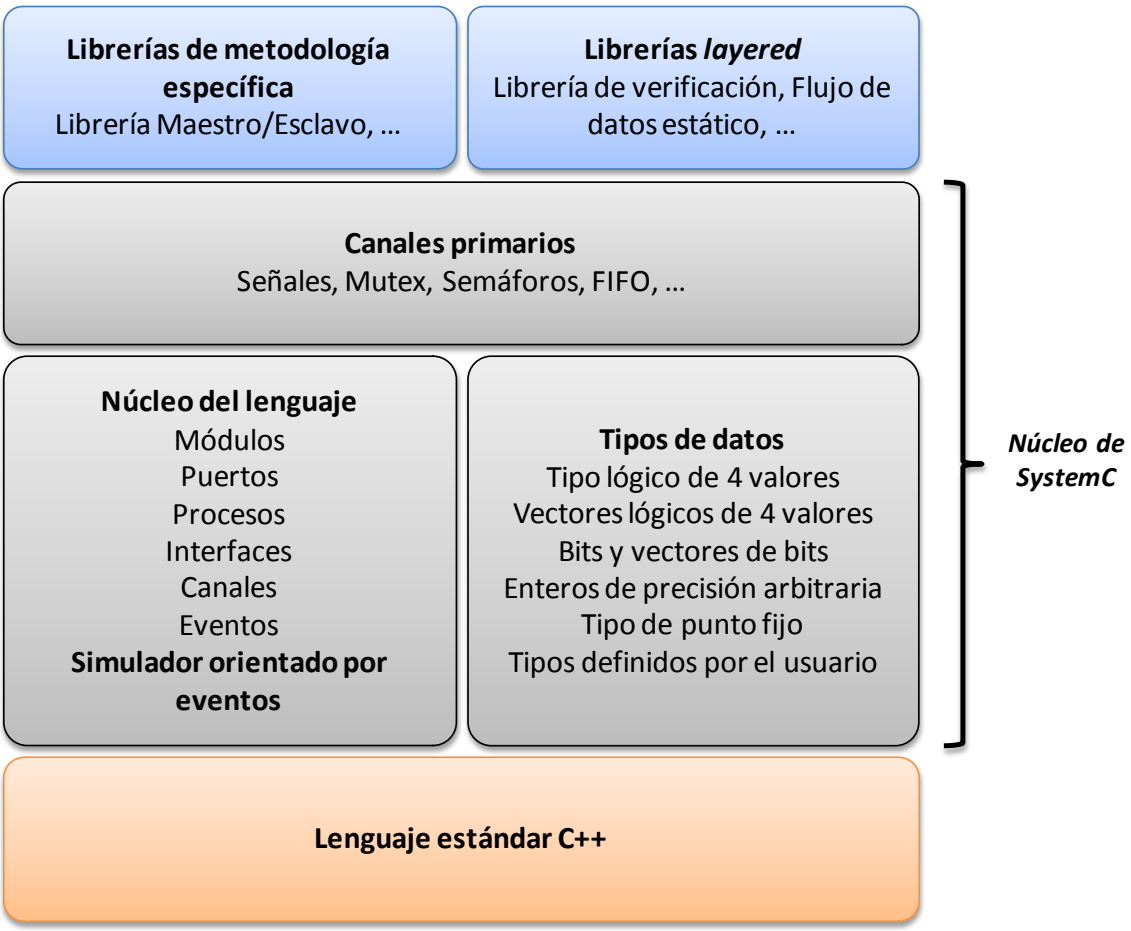


Figura 11. Arquitectura de capas de SystemC.

Las capas superiores, son librerías y estándares de diseño que el usuario puede decidir si usarlos o no. Gracias a esta estructura, se pueden seguir añadiendo librerías sobre el núcleo sin tener que modificar este.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

El núcleo del lenguaje está formado por el simulador orientado por eventos, junto con los elementos comentados anteriormente (puertos, módulos, procesos, etc.). Los tipos de datos son necesarios para el modelado *hardware* y ciertos tipos de desarrollo *software*. Los canales primarios son los canales incorporados que tienen un amplio uso tal como las señales y las FIFOs.

4.1.5 Metodología de diseño

Como ya se indicó anteriormente, SystemC representa, además de una librería de clases, una metodología de diseño que tiene como objetivo acelerar las fases de diseño a nivel funcional así como su verificación.

En un flujo tradicional de diseño electrónico es una práctica habitual realizar un diseño funcional en C/C++ con el fin de crear un modelo de alto nivel que cumpla las especificaciones tras varias iteraciones de diseño y optimización, para luego realizar una traducción a un lenguaje de descripción *hardware* a nivel RTL (VHDL o Verilog). Este tipo de metodología presente inconvenientes conocidos como pueden ser:

- Traducir el código C/C++ a VHDL/Verilog puede producir errores difíciles de depurar. Aún con un modelo en C/C++ que cumpla con las especificaciones, es difícil realizar una traducción sistemática.
- Caducidad del modelo C/C++. Una vez el diseño se realiza a nivel RTL, todas las optimizaciones se realizarán sobre este, lo que llevará a que el modelo en alto nivel quede desfasado.
- Imposibilidad de reutilizar los test de diseño. La naturaleza de los test de una aplicación en C/C++ son muy diferentes a un *testbench* de un sistema *hardware*, lo que obligará a rediseñar los sistemas de test para la versión RTL del diseño.

La metodología que presenta SystemC pretende dar solución a los problemas anteriormente citados y se representa en el diagrama de flujo de la Figura 12.

Las principales ventajas de este flujo modificado pueden resumirse en los siguientes puntos:

- La metodología de refinamiento permite escribir una primera versión del sistema en C/C++ e ir incluyendo características de SystemC al diseño en varias iteraciones.
- El realizar el sistema en SystemC para luego sintetizarlo hace que no sea necesario conocer otros lenguajes de descripción *hardware*.
- El tiempo de desarrollo se reduce, tanto en diseño como, lo que es más importante, en verificación.
- Los *testbenchs* pueden ser reutilizados desde la primera versión hasta la versión más refinada y cercana a RTL.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Una parte importante de la metodología implica la verificación de que la solución obtenida sea equivalente a la especificada. En este caso deberíamos volver a realizar la verificación del sistema, adaptando el *testbench* inicial a los requerimientos de latencia presente en el modelo preciso a nivel de ciclos, obtenidos durante la síntesis de alto nivel. Otra forma de realizar la verificación es utilizar métodos de comparación formal entre las diferentes representaciones.

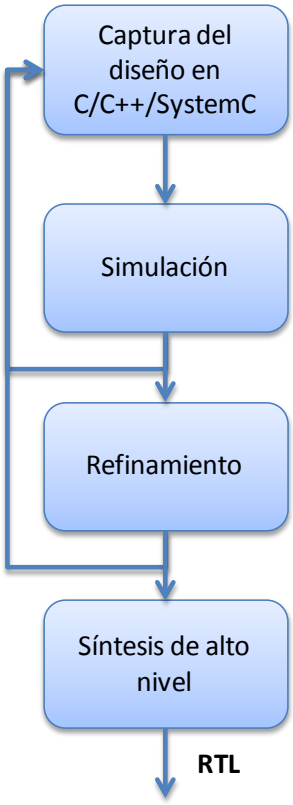


Figura 12. Metodología basada en SystemC.

4.2 Verilog

Verilog es un lenguaje de descripción hardware (HDL) usado para modelar circuitos electrónicos principalmente digitales, aunque existe una versión para el modelado de circuitos analógicos. En el presente TFM se ha utilizado Verilog como lenguaje a nivel RTL. Es por ello que todo el diseño a nivel RTL está descrito en este lenguaje [17].

A continuación se muestran dos ejemplos de descripción RTL en Verilog. En la Figura 13 se muestra la descripción RTL de un flip-flop tipo D, sensible al flanco de subida del reloj. Asimismo, en el siguiente ejemplo se muestra una descripción de comportamiento mediante múltiples procesos concurrentes, el primero de ellos combinacional y el segundo secuencial.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

```
module flipflop (d, clk, q, q_bar);
   input d, clk;
   output q, q_bar;
   reg q, q_bar;

   always @ (posedge clk)
   begin
   q <= #1 d;
   q_bar <= #1 !d;
   end
endmodule
```

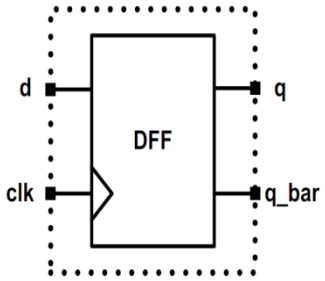


Figura 13. Flip-flop tipo D.

```
module (clk, A, B, C, D);

   input   clk, A, B;
   output  C, D;
   reg     C;

   always@(C)
   begin
     D <= C + 10;
   end

   always@(posedge clk)
   begin
     C <= A + B;
   end

endmodule
```

4.2.1 Señales y puertos

En Verilog existen dos tipos básicos de señales:

- *Nets* (*wire*): representan una conexión directa entre dos puntos del circuito, es decir, es el equivalente a un cable. No almacena información más allá de la conexión.
- *Registers*: almacena información además de conectar puntos. No necesariamente una señal de tipo *reg* se sintetizará en un registro, sino que mantiene el estado de esa señal.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

4.2.2 Puertos

Por otra parte, los puertos definen la interfaz de los módulos. Existen puertos de entrada y de salida y su tipo puede definirse. Las señales de entrada, *input*, no se declaran y serán siempre de tipo *wire*. Las salidas sin embargo, *output*, pueden estar registradas, por lo que habrá que especificar si es te tipo *wire* o *reg*.

4.2.3 Procesos

Un proceso consiste en un conjunto de sentencias que, ejecutándose de forma secuencial, describe el comportamiento de un circuito electrónico. Todos los procesos son ejecutados de forma concurrente entre ellos. Todas las asignaciones realizadas dentro de un proceso se hagan sobre señales de tipo *reg*. Existen dos tipos básicos de procesos en Verilog:

- ***Initial*:** son procesos que se ejecutan una vez al arranque del sistema y una única vez. No son sintetizables, por lo que solo tienen utilidad en el diseño de *testbenchs*.
- ***Always*:** permiten modelar procesos que se ejecutan en un bucle infinito. La condición de arranque del proceso definirse por un evento o por un tiempo determinado.

```
initial
begin
 clk = 0;
 reset = 0;
 enable = 0;
 data = 0;
end
```

```
always @(a or b or sel)
begin
if (sel == 1)
  y = a;
else
  y = b;
end
```

5. Herramientas

A continuación se detallarán las herramientas que se han utilizado durante el desarrollo de este TFM en sus distintas etapas.

5.1 Cadence CtoS

Cadence C-to-Silicon genera automáticamente código HDL a nivel RTL sintetizable a partir de *untimedC/C++/SystemC* reduciendo a un 10% el esfuerzo que se requiere en el uso de métodos manuales. C-to-Silicon ha sido diseñado partiendo de cuatro funciones únicas que ofrecen ventajas revolucionarias a los diseñadores de hardware RTL [7]:

- *Embedded-Logic-Synthesis (ELS):* permite la optimización paralela de lógica de control y ruta de datos, mejorando la calidad de los resultados.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

- *Behavior-Structure-Timing* (BST): una base de datos permite una síntesis gradual y un tiempo de diseño y de verificación mucho más rápido.
- *Constraint-Functionality-Separation (CFS):* permite la reutilización eficiente a través de múltiples aplicaciones y tecnologías de proceso.
- *Fast-Hardware-Models (FHMs):* acelera la verificación y permite el codesarrollo hardware/software.

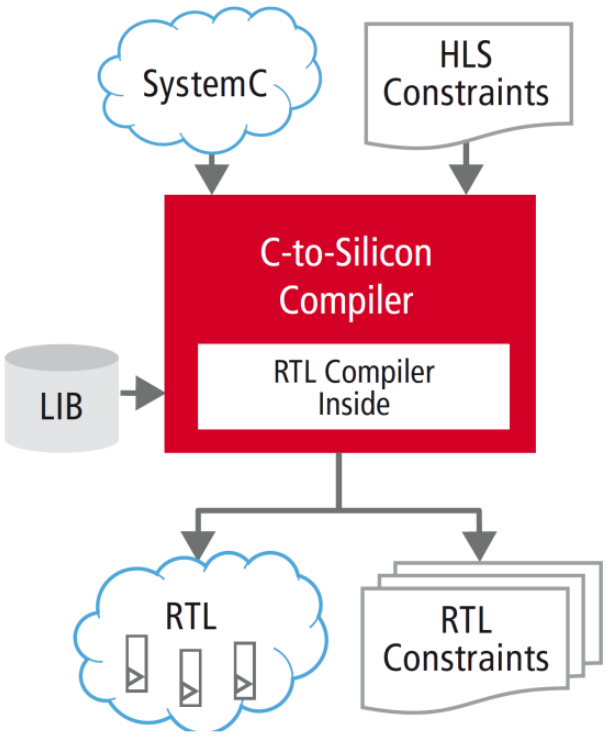


Figura 14. Flujo de diseño de Cadence CtoS

5.2 Xilinx Vivado HLS

Xilinx Vivado Design Suite es la nueva generación de entornos de diseño para FPGAs de Xilinx que ha sido construido desde cero para hacer frente a los cuellos de botella de productividad en la integración e implementación a nivel de sistema. La suite Xilinx Vivado está dirigida a la productividad global, la facilidad de uso y la capacidad de integración a nivel de sistema [18]. Una visión global del flujo de diseño usando Vivado HLS se muestra en la Figura 15.

El proceso de síntesis de alto nivel se realiza en diferentes etapas. Por una parte se realiza la planificación (*scheduling*) de las operaciones incluidas en la descripción algorítmica, su asignación a unidades funcionales genéricas y por último su asignación a unidades funcionales específicas de la FPGA (*binding*) (Figura 16). Además se genera la unidad de control y el secuenciamiento de las entradas y salidas.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

El proceso de síntesis de alto nivel usado en Xilinx Vivado HLS se puede resumir en los siguientes pasos:

- Síntesis de los argumentos de la función de más alto nivel de jerarquía en puertos de entrada salida a nivel RTL
- Síntesis de las funciones en C en una jerarquía de bloques RTL. Si el código C contiene una jerarquía de subfunciones, el código final RTL posee una jerarquía de módulos (Verilog) o entidades (VHDL) que se corresponden con la jerarquía original de funciones en C.
- Por defecto, los bucles en las funciones en C se mantienen enrollados. Esto significa que la síntesis crea lógica para una iteración del bucle y el diseño a nivel RTL ejecuta esta lógica para iteración del bucle en secuencia.
- La síntesis de los *arrays* en el código C se mapean a memoria de tipo BlockRAM en el diseño final de la FPGA. Si el *array* está presente en el nivel más alto de la jerarquía, este se implementa como puertos para acceder a una memoria BlockRAM fuera del diseño.

Algunas características adicionales de Xilinx Vivado HLS son las siguientes:

- Aceleración de la implementación.
    - Implementación del diseño 4 veces más rápida.
    - Densidad de diseño un 20% mejor.
    - Hasta 3 veces mejor rendimiento y un 35% menos de energía.
- Aceleración de la integración
    - Generación de IP basado en C con Xilinx Vivado High Level Synthesis.
    - Diseño integrado de DSP basado en modelos con *SystemGenerator.*
    - Integración IP basada en bloques con *Vivado IP Integrator.*
- Aceleración de la verificación
    - Entorno de diseño integrado para diseño y simulación.
    - Depuración Hardware integral.
    - Verificación con C, C++ o SystemC acelerada más de 100 veces.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

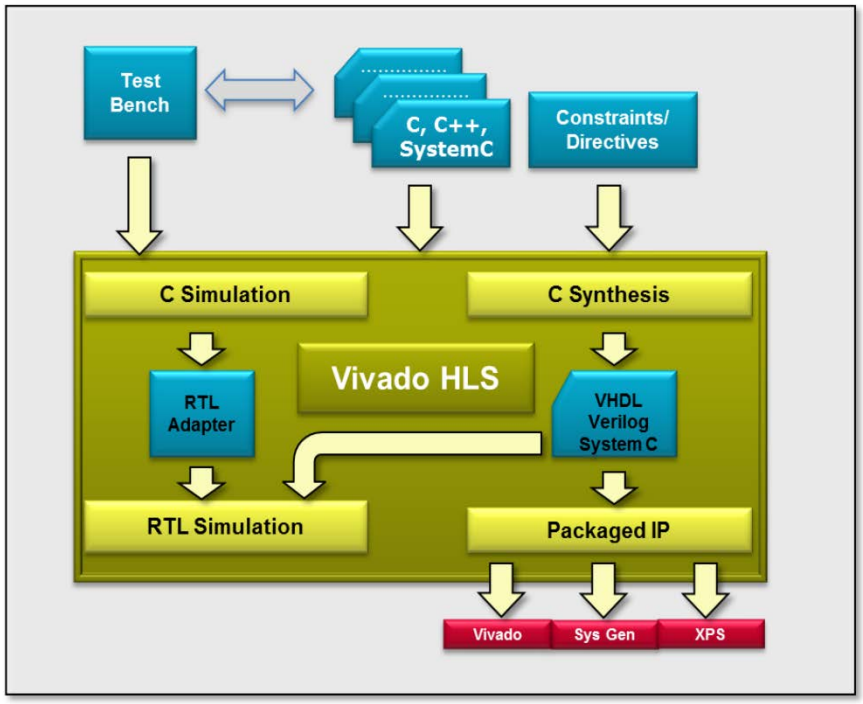


Figura 15. Xilinx Vivado HLS

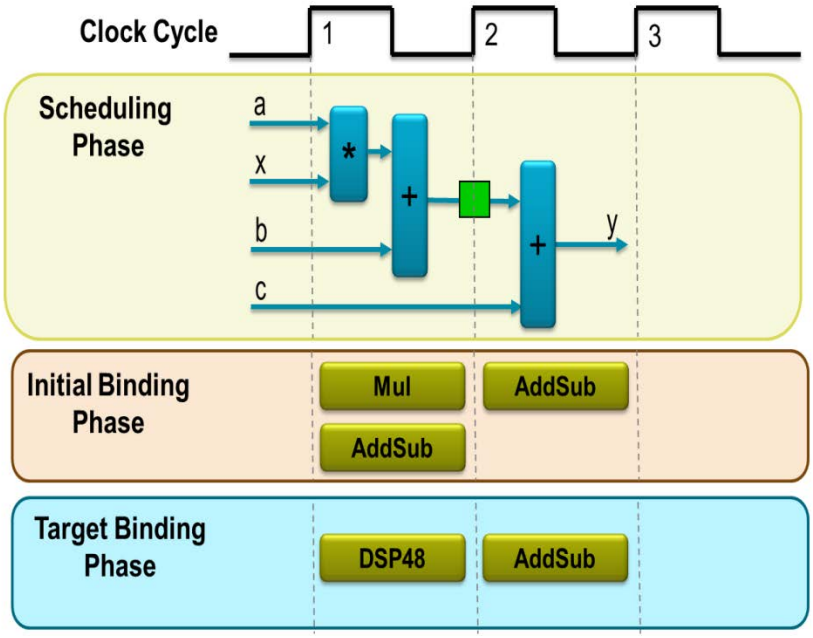


Figura 16. Proceso de síntesis usado en Xilinx Vivado HLS

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

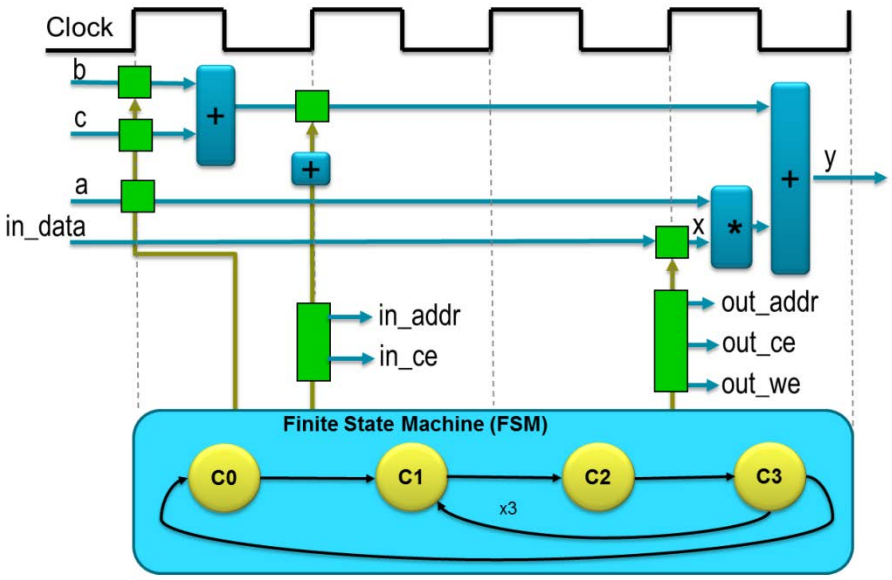


Figura 17. Unidad de control y secuenciamiento de E/S

5.3 Synopsys Synplify Premier Pro

Synopsys Synplify Premier es parte de la familia de soluciones de diseño de FPGA de Synopsys y se presenta como un entorno de implementación y depurado de FPGA. Synopsys Synplify Premier es un paquete de herramientas que realiza la implementación de la FPGA, depuración del diseño, y la automatización de la creación de prototipos basados en FPGA.

La suite de diseño de Synopsys Synplify Premier incluye las siguientes características:

- Automatización de la conversión de diseño ASIC para prototipos basados en FPGA incluyendo funciones de *Netlist Editor, TCL scripting, Synopsys Design Constraints*, conversión de reloj, síntesis y compilación de restricciones *DesignWare IP*.
- Integración con Synopsys DesignWare para validación de ASIC usando prototipos basados en FPGA.
- Síntesis rápida síntesis usando *fast mode* (x4).
- El modo *Continue-on-error* reduce las iteraciones requeridas identificando múltiples errores en una misma síntesis.
- Características de diseño avanzado para alta fiabilidad incluyendo TMR (Triple Modular Redundancy), implementación de tolerancia a fallos FSM e inferencia automática de memorias con soporte a corrección de errores (ECC).
- Depurador *Identify RTL* y visor de formas de onda para establecer *triggers* complejos, depurado sobre placa y verificación de RTL para implementación equivalente.
- Visualización de datos del simulador de resultados VCS y el depurador Idenfity dentro del visualizador de esquemáticos HDL Analyst Schematic Viewer para diagnosis del diseño.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

- La correlación de tiempo precisa mejora los *netlist* existentes para una sincronización más rápida y reducción de la congestión.
- Soporte a la inferencia de bloques para el procesamiento de señales (DSP48, MAC, etc.) .

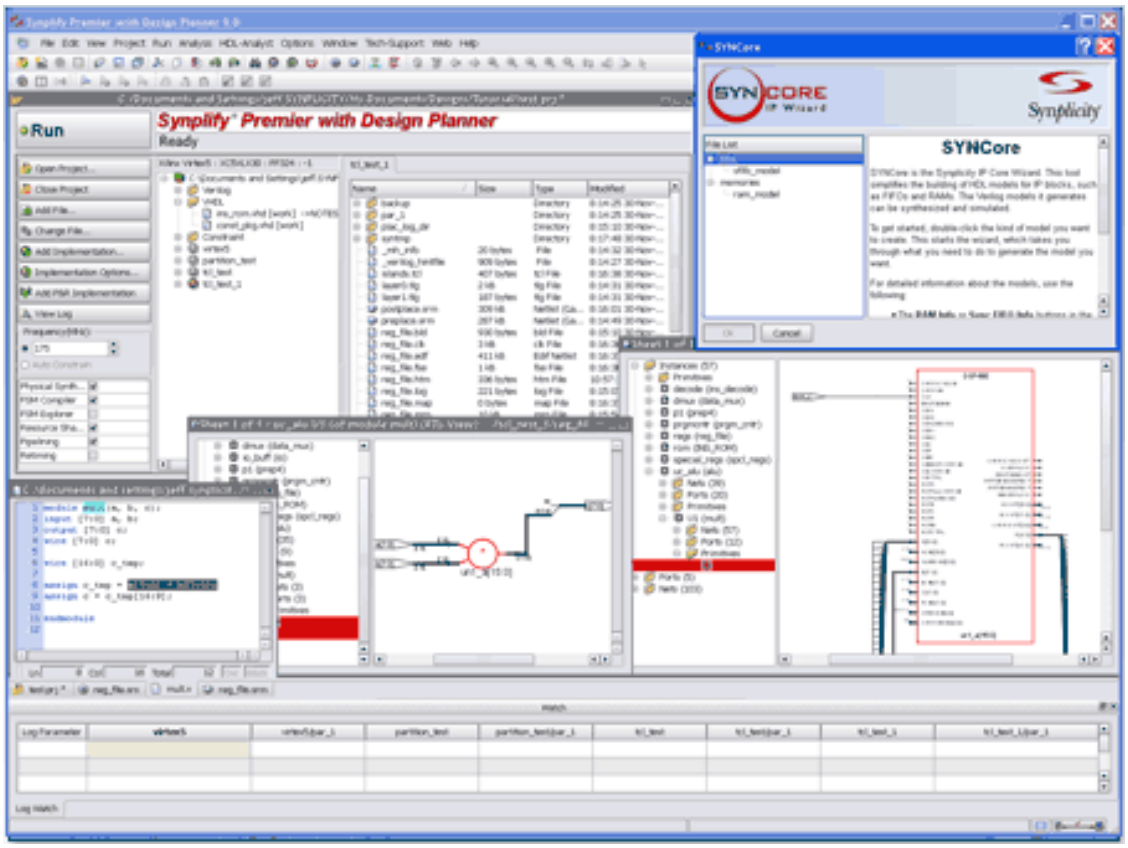


Figura18. Synopsys Symplify Premier

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Capítulo 3: Dominio de aplicación

1. Introducción

En este capítulo se revisarán las características principales de los procesadores de eventos en general, y de las particularidades de la aplicación de referencia usada en este Trabajo Fin de Máster. En concreto se da una visión general de las aplicaciones de procesado de eventos y su arquitectura, continuando con la presentación de los principios de *Algorithmic Trading*. Se presentará la aplicación de referencia y sus requerimientos, explicando las posibles alternativas en cuanto a la solución arquitectural.

2. Procesadores de eventos

Se entiende por procesador de eventos un sistema capaz de encontrar patrones en secuencias de entrada de datos. La naturaleza de los eventos y de los patrones que reconoce el

procesador varía en función de la aplicación para la que se use, no obstante existen propiedades comunes que se expondrán a lo largo de este capítulo.

2.1 Aplicaciones de procesado de eventos

Existen multitud de entornos diferentes que requieren de un procesador de eventos que analice secuencias de entrada y reconozca patrones en ellas. Este procesado puede realizarse en tiempo real, es decir, que el sistema tome como entrada eventos en el mismo instante en que estos son generados; o pueden realizarse tras ser almacenados en una base de datos. La condición de que el sistema realice el procesado en tiempo real o con eventos almacenados dependerá de las necesidades específicas de la aplicación.

Un ejemplo de aplicación sin necesidad de procesado en tiempo real, es el análisis de rentabilidad de un producto para una empresa de ventas *online*. Los eventos de entrada estarán compuestos por las compras del producto, las opiniones de los consumidores, las devoluciones de los clientes, etc. En general, el procesado de dicho tipo de eventos debe realizarse con posterioridad a su generación, y además, no se requiere que las acciones a llevar a cabo, se realicen en tiempo real [19].

Un ejemplo en el que sí se requiere una respuesta en tiempo real es en la seguridad de redes. Una red de ordenadores puede ser el objetivo de ataques informáticos por parte de usuarios maliciosos que quieran acceder a la información almacenada en alguno de los nodos de dicha red. Para evitar estos ataques, un sistema de seguridad consiste en analizar las peticiones Web y DNS, tratando de buscar patrones que se alejen del uso típico del servicio. En este apartado entra el juego el procesador de eventos, donde los eventos son las peticiones Web de entrada, y las acciones pueden ser la interrupción del servicio a una dirección IP determinada, cuando se comprueba un uso malintencionado. Este tipo de respuestas deben generarse en tiempo real, para evitar comprometer los datos almacenados en la red [20].

Otro ejemplo de uso de procesadores de evento en tiempo real, y que se corresponde a la aplicación de referencia usada en este Trabajo Fin de Máster, es el de *Trading automático*. Este término hace alusión a sistemas electrónicos que generan órdenes de compra y/o de venta, automáticamente, en función de los parámetros de algún mercado de valores, sin intervención directa de alguna persona [21].

Este tipo de aplicaciones se encuentra en auge en la actualidad, proporcionando hasta el 80% de las transacciones en algunos mercados de valores [22, 23]. En este tipo de sistemas, se requiere de procesadores de eventos capaces de procesar una gran cantidad de eventos de

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

mercado con una mínima latencia, y proporcionar una orden de compra o de venta (si se cumpliese un cierto patrón predefinido) en el mínimo tiempo posible.

3. Sistema de referencia

En este Trabajo Fin de Máster, se usará como modelo de referencia, un sistema de *Algorithmic Trading* dedicado a la generación de órdenes de mercado en base a estrategias definidas previamente por el inversor.

En el desarrollo del presente Trabajo Fin de Máster, se parte del sistema ya desarrollado y verificado, que servirá como modelo de referencia para la verificación del flujo de diseño basada en Xilinx Vivado, la realización de las medidas de rendimiento y la mejora del diseño planteado.

Se parte de un diseño en alto nivel descrito en SystemC, resultado del estudio de la aplicación original y su posterior partición. Este diseño es fruto del trabajo dentro de un proyecto de investigación del Instituto Universitario de Microelectrónica Aplicada.

Se pueden enumerar a continuación, algunas características que hacen del procesador de eventos un diseño complejo y de gran versatilidad con el que poder manejar todo tipo de estrategias de *Algorithmic Trading*.

- Permite crear estrategias, compuestas por órdenes, cada una con una acción a realizar y un conjunto dinámico de condiciones de disparo.
- Permite definir acciones de activación de nuevas órdenes y estrategias previamente almacenadas en el sistema.
- Pueden definirse condiciones temporales de comienzo, finalización y duración de estrategias.
- Pueden definirse comparaciones entre un gran número de atributos de cada título, comparando atributos diferentes de productos diferentes. Además, las comparaciones pueden ser complejas, no solo de igualdad o de mayor o menor, sino también de valores calculados a partir de atributos, del estilo $K_1 \cdot Atributo + K_2$.

La descripción en alto nivel realizada en SystemC está formada por módulos de procesamiento y módulos controladores de memoria interna. Las memorias internas usarán recursos internos de la FPGA con el fin de almacenar estructuras de datos necesarias para la implementación de la aplicación original. Los bloques de procesamiento pueden dividirse, según su funcionalidad, entre los cuatro bloques conceptuales presentados anteriormente: interfaz, núcleo de procesamiento, ejecutor de resultados y estado del procesador. Para realizar los

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

análisis de la implementación se utilizará esta nomenclatura de cuatro bloques en lugar de la verdadera arquitectura del diseño, con el fin de evitar publicar información protegida por la cláusula de confidencialidad.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Capítulo 4: Desarrollo

1. Introducción

En este capítulo se expone el desarrollo del presente Trabajo Fin de Máster. Se detalla todo el proceso desde la descripción del sistema en lenguaje SystemC hasta su implementación a nivel RTL. Esto implica la transformación del código en SystemC a un subcojunto sintetizable por la herramienta de síntesis Xilinx Vivado HLS. Posteriormente se realiza la síntesis de alto nivel y la síntesis lógica y se obtienen resultados en área, potencia y latencia.

2. Adaptación del sistema para síntesis

La primera etapa del flujo de diseño comprende la adaptación del código SystemC a un subconjunto de éste que esté soportado por la herramienta de síntesis Xilinx Vivado HLS. Indicar que en este proyecto un bloque o módulo está formado por un fichero de cabecera (modulo.h) y

un fichero de funcionalidad (modulo.cpp). En el fichero de cabecera incluye la interfaz de E/S y la declaración de procesos y el de funcionalidad las funciones que implementa el bloque.

2.1 Construcciones SystemC no soportadas en Xilinx Vivado HLS

Se detalla a continuación las restricciones que presenta Xilinx Vivado HLS respecto al lenguaje SystemC desde el punto de vista de la síntesis.

- Un SC_MODULE conectado dentro de otro SC_MODULE.
- Un SC_MODULE derivado de otro SC_MODULE.
- SC_THREAD pero sí SC_CTHREAD.
- Un módulo no puede ser instanciado utilizando el constructor *new.*

```
M1 *t0;
SC_CTOR(TOP) {
   t0 = new M1(t0);
...
```

```
M1 t0;
   SC_CTOR(TOP) : t0("t0") {
   ...
```

- Solamente se pueden utilizar los nombres de los parámetros en la construcción de los módulos. El paso de una variable *temp* de tipo *int* no está permitido.

```
M1 t0;
SC_HAS_PROCESS(dut);
dut(sc_module_name nm, int temp) : sc_module(nm), var(temp) {
...
```

- Funciones virtuales.
- Lectura de puertos de salida *sc_out*.

2.2 Transformación del código

Tras analizar exhaustivamente el código con objeto a determinar la compatibilidad con Xilinx Vivado HLS, se encontraron los siguientes problemas:

- El código de referencia contiene algunos módulos instanciados dentro de otros. En particular existe un módulo de división en punto fijo anidado dentro de algunos

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

módulos que conforman el bloque de ejecutor de acciones. Para resolver este problema se ha eliminado el módulo convirtiéndolo en una llamada de función.

- Todos los procesos están declarados como SC_THREAD activados por el flanco de subida del reloj. Ello hace necesario transformarlos a una versión *clocked*. Las modificaciones a realizar para este caso son simples ya que el comportamiento del código de partida es el de un proceso controlado por los eventos de un reloj.

En ambos utilizamos las directivas preprocesador #ifdef, #else, etc. para mantener la compatibilidad del código y que éste pueda continuar siendo sintetizable por el flujo de diseño original.

```
#ifdef __SYNTHESIS__
   SC_CTHREAD(main, clock.pos());
   reset_signal_is(rst_n, true);
#else
   SC_THREAD(main);
   sensitive<<clock.pos();
#endif
```

De esta manera mantenemos el código funcional tanto para el flujo de síntesis de Xilinx Vivado HLS como para anteriores flujos de diseño. La variable __SYNTHESIS__ está definida en el flujo de diseño de Xilinx Vivado HLS, así de manera automática el compilador de Xilinx Vivado HLS seleccionará aquellas partes de código que son sintetizables por la herramienta.

3. Síntesis de alto nivel con Xilinx Vivado HLS

Una vez realizada la transformación del código se procede a realizar la síntesis del diseño. Debido a las dimensiones del proyecto se realizará una síntesis de alto nivel para cada bloque para disminuir los tiempos de síntesis.

Para realizar la síntesis de alto nivel es necesario crear un nuevo proyecto en Xilinx Vivado HLS. Debemos especificar el dispositivo y seleccionar los ficheros fuente. Este proceso se repetirá para cada uno de los bloques que componen el diseño

3.1 Creación y configuración del proyecto

En primer lugar debemos crear un nuevo proyecto en Xilinx Vivado HLS, le asignamos como nombre del proyecto el nombre del módulo que vamos a sintetizar, establecemos el nombre del módulo top, en este caso es el mismo del proyecto, y añadimos los ficheros *.cpp*. No es necesario añadir los ficheros de cabecera *.h* pero deben estar en un directorio accesible (el mismo directorio o especificado por las opciones *include* del compilador).

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

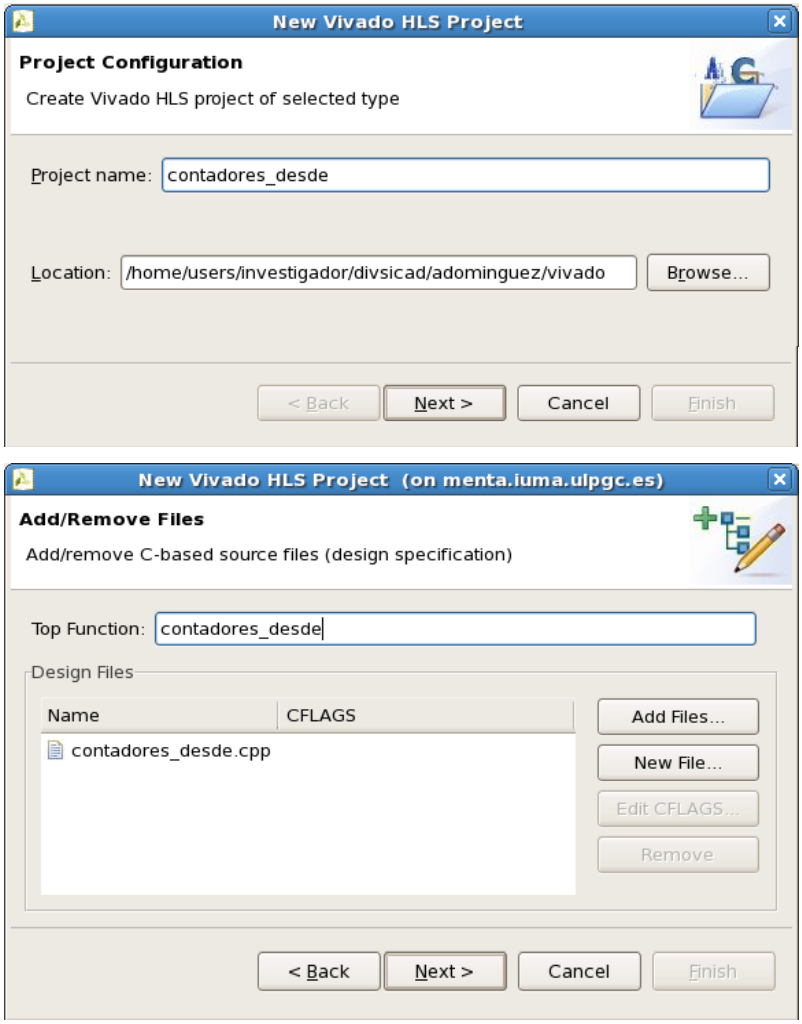


Figura 19. Creación del proyecto en Xilinx Vivado HLS

Además, si lo deseamos, podemos añadir un fichero de *testbench* para realizar una simulación a nivel funcional y tras la síntesis de alto nivel.

Por último elegimos la plataforma de implementación y el periodo de reloj. En este caso es la plataforma de evaluación Zynq ZC702 y un periodo de reloj de 10 *ns*.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

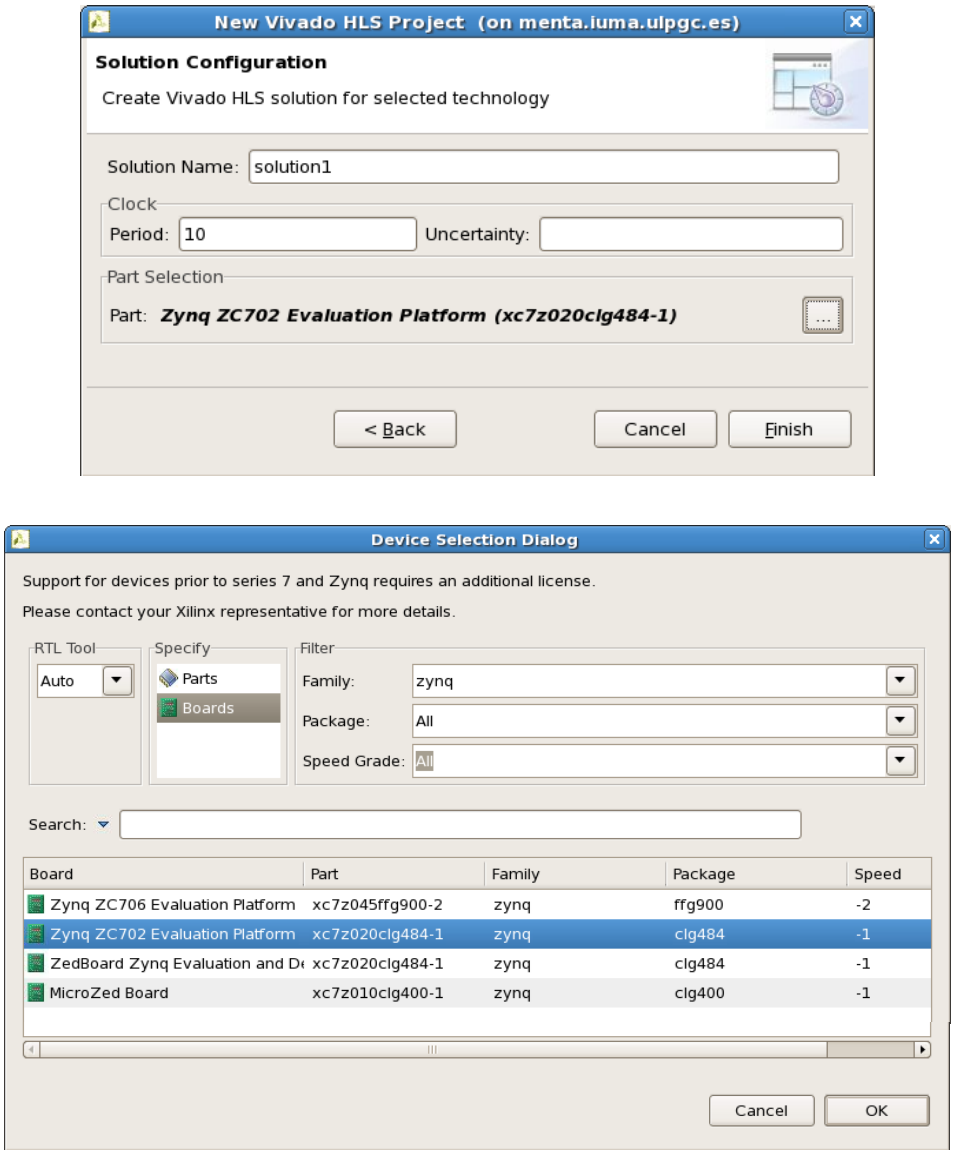


Figura 20. Configuración de la plataforma en Vivado HLS

El análisis de cabeceras (*.h) que realiza Xilinx Vivado HLS muestra la estructura del módulo y los puertos de entrada y salida, como puede verse en la parte derecha, pestaña *Outline*, de la Figura 21.

Además, en la pestaña *Directive*, permite insertar directivas sobre cada variable o función para guiar la síntesis en el uso de recursos.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

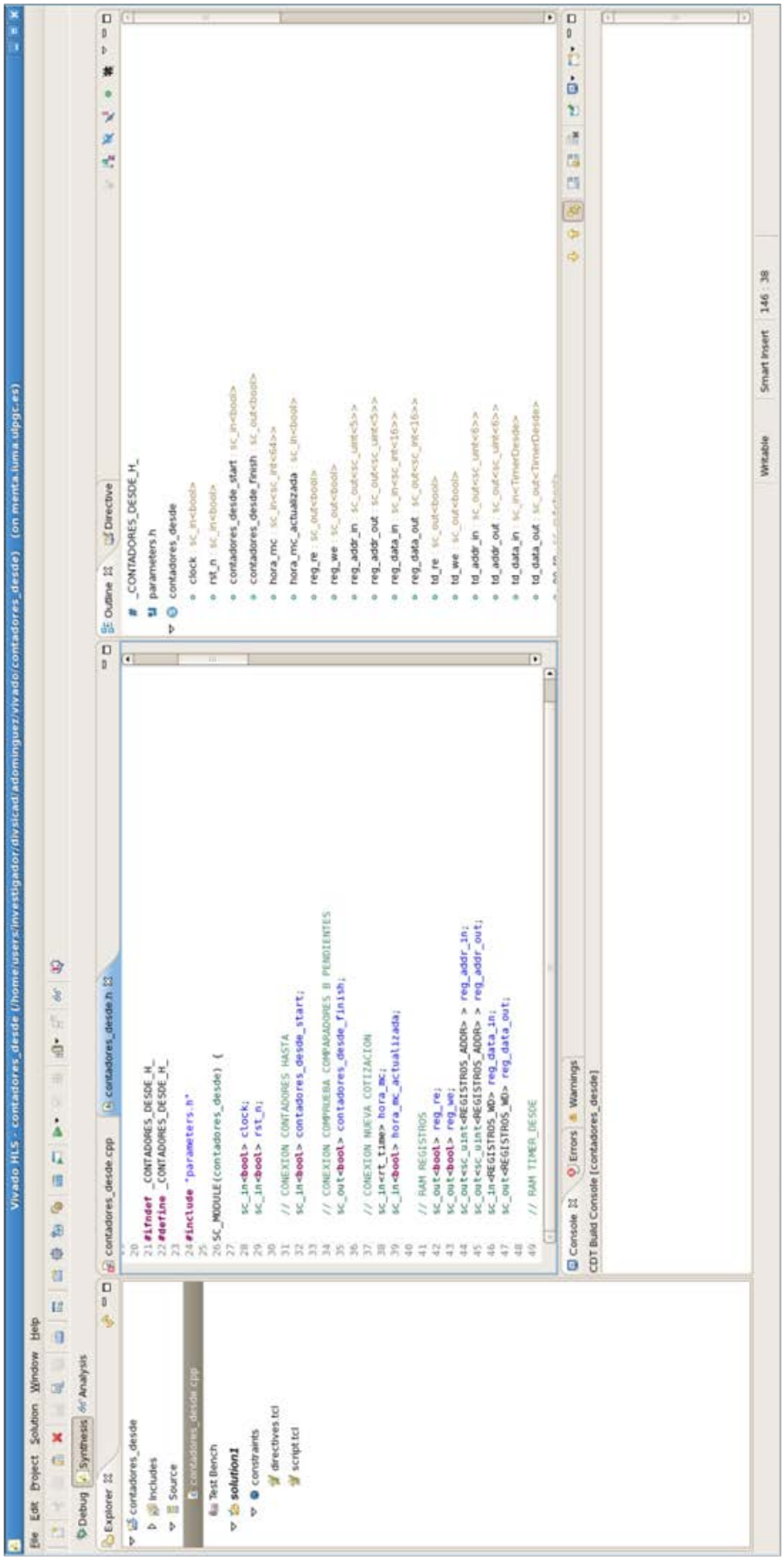


Figura 21. Entorno de trabajo de Xilinx Vivado HLS

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

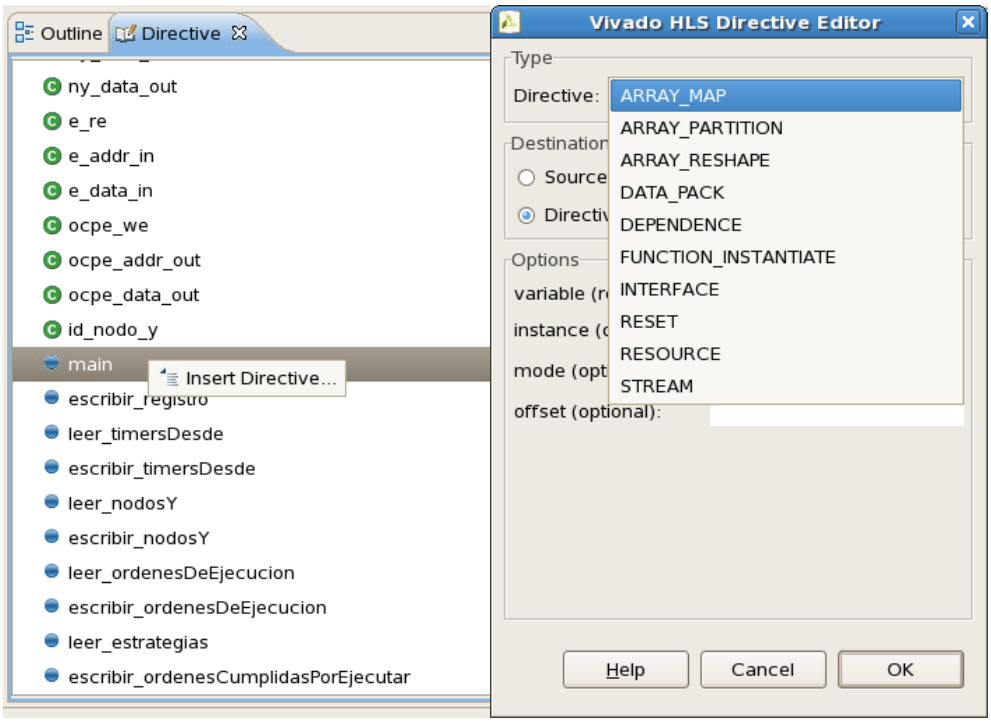


Figura 22. Configuración de directivas en Xilinx Vivado HLS

3.1.1 Directivas de síntesis

Xilinx Vivado HLS permite guiar el proceso de síntesis de alto nivel mediante el uso de directivas que se pueden clasificar en los siguientes grupos:

- Interfaces: directivas aplicadas a los objetos de interfaz de mayor nivel de jerarquía del diseño (puertos de E/S, parámetros de llamadas y salidas de las funciones o variables globales).
- Funciones: directivas aplicadas a todos los objetos dentro del alcance la función. El efecto de la directiva está localizado en la función, excepto para los casos de desenrollado de bucles u opciones recursivas.
- Bucles: directivas aplicadas a los objetos dentro del alcance de los bucles. Por ejemplo si se aplica la directiva LOOP_MERGE a un bucle, esta también afecta a cualquier otro bucle dentro del bucle al que se está aplicando la directiva.
- *Arrays*: directivas que se aplican a los *arrays* en cuanto a su implementación como memorias, registros, partición, etc.
- Regiones: una región es el código incluido entre llaves {}. Las directivas se aplican a todos los objetos incluidos en dicha región.

Las directivas se pueden aplicar mediante un fichero de directivas (directives.tcl) escrito en TCL, por ejemplo:

```
set_directive_stream
```

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

O mediante pragmas embebidos en el código fuente, como por ejemplo:

```
#pragma HLS STREAM depth=8 variable=OutStream
```

Entre las directivas para guiar la síntesis de alto nivel comentaremos las más relacionadas con este Trabajo Fin de Máster y que tienen como objetivo controlar el uso de memorias, *arrays*, estructuras de datos e interfaces en la síntesis.

- **ALLOCATION**: especifica restricciones sobre el mapeado de recursos. Define, y puede limitar, el número de instancias RTL usadas para implementar funciones u operaciones específicas. Ello facilita la compartición de recursos hardware y puede incrementar las latencias. Ello permite disminuir el consumo de recursos del diseño.
- **ARRAY_MAP:** mapea un *array* pequeño en uno mayor. Esto permite agrupar múltiples *arrays* más pequeños para formar uno mayor que puede ser mapeado a una memoria BlockRAM o una FIFO. Esta directiva también está orientada a disminuir el consumo de recursos de la FPGA.
- **ARRAY_PARTITION:** parte un *array* grande en otros más pequeños o incluso en elementos individuales. Esto resultará en múltiples memorias o múltiples registros en vez de una memoria grande. Esto incrementa la cantidad de puertos de lectura y escritura para almacenamiento, eliminando los problemas de cuellos de botella que se producen para acceder a las memorias de tipo BlockRAM en el caso de segmentar el diseño, mejorando el *throughput* del diseño a costa de más memorias o registros.
- **ARRAY_RESHAPE**: recombina la partición de *arrays* con el mapeado de *arrays* vertical, para crear un nuevo *array* con menos elementos pero palabras más anchas, con lo cual se puede implementar en una memoria de tipo BlockRAM, ahorrando el consumo de este tipo de recurso.
- **STREAM:** por defecto, las variables de tipo *array* son implementadas como memorias RAM. Si los datos almacenados son consumidos o producidos de manera secuencial, un mecanismo de comunicación más efectivo es usar un *streaming* de datos, donde se utilizan FIFOs en vez de RAMs.
- **DATA_PACK:** esta directiva empaqueta los campos de un *struct* en un vector con un ancho de palabra equivalente a la suma del ancho de los componentes del *struct*.
- **INTERFACE:** especifica cómo se crean los puertos RTL desde la descripción de la función durante la síntesis de la interfaz. Los puertos en la implementación RTL son derivados de:
    - Cualquier protocolo a nivel de función que se haya especificado.
    - Argumentos de funciones.
    - Variables globales.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

3.1.2 Interfaces soportadas por Xilinx Vivado HLS

Los puertos en un diseño SystemC se especifican en el código fuente. A diferencia de las funciones C y C++, en Xilinx Vivado HLS realiza una síntesis de las interfaces solo sobre interfaces de memoria soportadas para SystemC.

Todos los puertos de la interfaz del *top* del diseño deben ser uno de los siguientes tipos:

- sc_in_clk
- sc_in
- sc_out
- sc_inout
- sc_fifo_in
- sc_fifo_out
- sc_mem_if
- ap_mem_if

Excepto para las interfaces de memoria soportadas, todo el protocolo *handshake* entre el diseño y el *testbench* debe ser modelado explícitamente en la función SystemC. Las interfaces de memoria soportadas son:

- sc_fifo_in
- sc_fifo_out
- ap_mem_if

Xilinx Vivado HLS puede añadir ciclos de reloj adicionales a un diseño SystemC si es necesario para cumplir los tiempos. Por tanto, el número de ciclos después de la síntesis puede ser diferente, los diseños en SystemC deberían controlar mediante protocolo todas las transferencias de datos con el *testbench*.

Xilinx Vivado HLS no soporta el modelado a nivel de transacciones usando TLM 2.0 y el modelado basado en eventos para síntesis.

3.1.2.1 Síntesis de interfaces SystemC

En general, Xilinx Vivado HLS no realiza la síntesis de interfaces en SystemC. Está soportada la síntesis de interfaces para algunas memorias, así como para puertos RAM y FIFO.

Síntesis de interfaces RAM

A diferencia de la síntesis de C y C++, Xilinx Vivado HLS no transforma los puertos de tipo *array* en puertos RAM (RTL). En el siguiente código SystemC, se debe usar las directivas Xilinx Vivado HLS para realizar la partición de los puertos de tipo *array* en elementos individuales.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

De lo contrario, este código de ejemplo no puede ser sintetizado:

```
SC_MODULE(dut)
{
   sc_in<T>in0[N];
   sc_out<T>out0[N];
   …
   SC_CTOR(dut)
   {
      …
   }
};
```

Las directivas para dividir los *arrays* en elementos individuales son:

```
set_directive_array_partition dut in0 -type complete
set_directive_array_partition dut out0 -type complete
```

Si N es un número grande, esto resulta en muchos puertos individuales en la interfaz en RTL. El ejemplo siguiente muestra como una interfaz RAM puede ser modelada en SystemC y completamente sintetizada por Xilinx Vivado HLS. En dicho ejemplo, los *arrays* son reemplazados por tipos *ap_mem_port* que pueden ser sintetizados en puertos RAM.

```
#include systemc.h
#include ap_mem_if.h

SC_MODULE(sc_RAM_port) {
   //Ports
   sc_in <bool>  clock;
   sc_in <bool>  reset;
   sc_in <bool>  start;
   sc_out<bool>  done;
   //sc_out<int>dout[100];
   //sc_in<int>din[100];
   ap_mem_port<int, int, 100, RAM_2P> dout;
   ap_mem_port<int, int, 100, RAM_2P> din;

   //Variables
   int share_mem[100];
   sc_signal<bool> write_done;

   //Process Declaration
   void Prc1();
   void Prc2();

   //Constructor
   SC_CTOR(sc_RAM_port) : dout (dout), din (din)
   {
      //Process Registration
      SC_CTHREAD(Prc1,clock.pos());
      reset_signal_is(reset,true);
      SC_CTHREAD(Prc2,clock.pos());
      reset_signal_is(reset,true);
   }
};
```

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

La sintaxis del tipo *ap_mem_port* es el siguiente:

```
ap_mem_port (<data_type>, < address_type>, <number_of_elements>,
<mem_target>)
```

- ***data_type:*** es el tipo de datos usado para almacenar los elementos.
- ***address_type:*** es el tipo de datos usado para el bus de direcciones. Debe tener el ancho suficiente para manejar todo el rango de direcciones de la memoria.
- ***numer_of_elements:*** especifica el número de elementos del *array*.
- ***mem_target:*** especifica la memoria a la que será conectado y por tanto, determina los puertos de E/S según la Tabla 1.

Tabla 1. Tipos de memorias RAM soportadas por Xilinx Vivado HLS.

Target RAM	Descripción
RAM_1P	Memoria RAM de un puerto
RAM_2P	Memoria RAM de doble puerto
RAM_T2P	Memoria RAM de doble puerto que soporta lectura y escritura simultánea.
ROM_1P	Memoria ROM de un puerto
ROM_2P	Memoria ROM de doble puerto

Síntesis de interfaces FIFO

Los puertos FIFO en la interfaz a nivel del *top* del diseño pueden ser sintetizados directamente del estándar SystemC. En el ejemplo siguiente se utilizan los puertos sc_fifo_in y sc_fifo_out.

```
#include systemc.h
#include tlm.h
using namespace tlm;

SC_MODULE(sc_FIFO_port)
{
  //Ports
  sc_in <bool>  clock;
  sc_in <bool>  reset;
  sc_in <bool>  start;
  sc_out<bool>  done;
  sc_fifo_out<int> dout;
  sc_fifo_in<int> din;

  //Variables
  int share_mem[100];
  bool write_done;

  //Process Declaration
  void Prc1();
  void Prc2();

  //Constructor
```

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

```
    SC_CTOR(sc_FIFO_port)
    {
      //Process Registration
      SC_CTHREAD(Prc1,clock.pos());
      reset_signal_is(reset,true);
      SC_CTHREAD(Prc2,clock.pos());
      reset_signal_is(reset,true);
    }
};
```

Después de la síntesis, cada puerto FIFO tiene un puerto de datos y las señales de control asociadas:

- FIFOs de entrada tienen los puertos de *empty* y *read*.
- FIFOs de salida tienen los puertos de *full* y *write*.

Al usar puertos FIFO, el protocolo *handshake* necesario para sincronizar las transferencias es añadido en el *testbench* RTL.

3.1.3 Protocolos de comunicación soportados

Los tipos de interfaz *ap_crtl_none*, *ap_ctrl_hs* y *ap_ctrl_chain* son usados para especificar si el RTL es implementado con señales de control o no. Las señales de control a nivel de bloque especifican cuando el diseño puede comenzar a realizar su operación estándar y cuando finaliza. Estos tipos de interfaces se especifican en la función o en el retorno de la función. La Figura 23 muestra los puertos RTL resultantes y el comportamiento cuando se especifica un *ap_ctrl_hs* en una función (esto es la operación por defecto). En este ejemplo la función devuelve un valor usando una sentencia *return* y por tanto se crea un puerto de salida *ap_return* en el diseño RTL. Si no hubiera una sentencia *return* este puerto no sería creado.

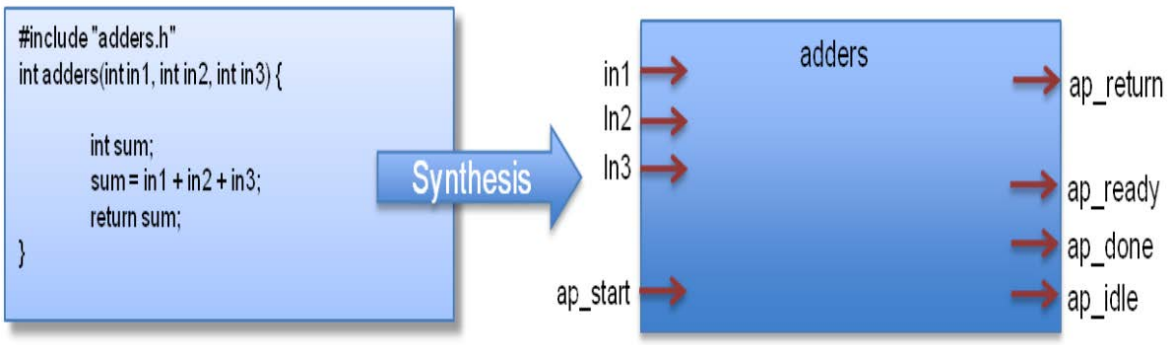


Figura 23. Ejemplo de interfaz ap_ctrl_hs.

La interfaz *ap_ctrl_chain* es similar a *ap_ctrl_hs* pero proporciona una señal de entrada adicional de retorno y es recomendado cuando se encadenan varios bloques. A continuación se exponen brevemente todos los tipos de protocolos soportados.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

ap_none

El tipo de interfaz *ap_*none es el más simple. Ningún puerto de entrada ni salida tienen puertos de control asociados indicando cuando un dato se lee o se escribe. Los únicos puertos en el diseño RTL son aquellos especificados en el código fuente.

ap_ctrl_hs

El comportamiento de las señales de control creadas para el modo de interfaz ap_ctrl_hs se muestran en la Figura 24.

- Después del *reset*, el bloque espera por la señal *ap_start* para comenzar la operación. Cuando pasa a nivel alto se leen las señales del puerto de entrada en el primer ciclo de reloj después de bajar *ap_idle*.
- Cuando el bloque completa todas las operaciones, cualquier valor de retorno se escribe en el puerto *ap_return* y la señal *ap_done* pasa a nivel alto.

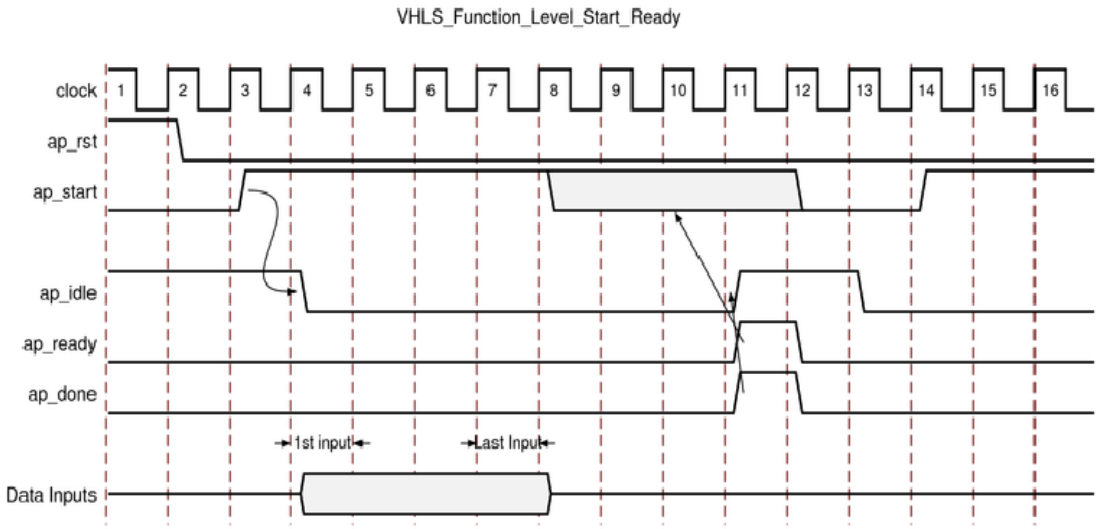


Figura 24. Comportamiento del protocolo ap_ctrl_hs.

ap_ctrl_chain

El protocolo *ap_ctrl_chain* es similar a *ap_ctrl_hs* pero proporciona un puerto de entrada adicional, *ap_continue*. Cuando esta señal pasa a nivel bajo informa que el bloque que consume los datos no está listo para aceptar nuevos datos.

ap_stable

El tipo de interfaz *ap_stable*, como *ap_none*, no añade ningún puerto de control al diseño. Informa a la síntesis que los datos aplicados a este puerto permanecen estables durante la operación normal, pero no es un valor constante que puede ser optimizado, y el puerto no

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

requiere ser registrado. Este tipo de interfaz es usado típicamente para datos de configuración (como datos que cambian durante el *reset* pero permanecen estables durante la operación).

ap_hs

Una interfaz *ap_hs* proporciona tanto una señal de confirmación (ACK) para decir cuando un dato ha sido consumido como una señal de validación para indicar cuando un dato puede ser leído. Esta interfaz es un superconjunto de los tipos *ap_ack*, *ap_vld* y *ap_ovld*.

ap_memory, bram

Los argumentos de tipo *array* se implementan normalmente usando la interfaz *ap_memory*. Este tipo de puerto se usa para comunicar con elementos de memoria (RAMs, ROMs) cuando la implementación requiere acceso aleatorio a localizaciones de memoria.

Si solamente se requiere acceso secuencial al elemento de memoria, la interfaz *ap_fifo* puede reducir la sobrecarga *hardware* ya que no necesita generación de direcciones.

Las interfaces *ap_memory* y *bram* son idénticas, la única diferencia es cuando el bloque es usado dentro de un *IP Integrator* que *ap_memory* se muestra como puertos simples y *bram* como un grupo de puertos.

ap_fifo

Si se requiere acceso a una memoria y el acceso es realizado de forma secuencial, la interfaz *ap_fifo* es más eficiente. Este tipo de interfaz permite al puerto ser conectado a una FIFO, soporta comunicación *full-duplex* y puede ser utilizado para argumentos de tipo puntero, *array* o paso por referencia.

ap_bus

Una interfaz *ap_bus* puede comunicar con un bus. La interfaz no especifica ningún bus determinado pero es lo suficientemente genérico para ser utilizado en un bus que arbitra por turnos con el bus del sistema. El bus debe ser capaz de cachear todas las escrituras en ráfaga.

Las funciones que emplean una interfaz de este tipo usa punteros y puede acceder a la misma variable múltiples veces. Se puede utilizar de dos maneras:

- *Standard Mode:* realiza operaciones de lectura y escritura atómicas, especificando la dirección de cada una.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

- *Burst Mode:* si se utiliza la función *memcpy* de C, se utiliza el modo de ráfaga para las transferencias de datos. En este modo, la dirección base y el tamaño de la transferencia es indicado por la interfaz.

axis, s_axilite, m_axi

Xilinx Vivado HLS permite especificar un protocolo de E/S de tipo AXI Stream, AXI Slave Lite y AXI Master mediante las directivas *axis, s_axilite y m_axi* respectivamente. La descripción completa de la interfaz puede encontrarse en [24].

3.2 Síntesis del alto nivel

Una vez tenemos el código adaptado al subconjunto SystemC sintetizable y el proyecto configurado, ejecutamos la síntesis. Durante la síntesis se ejecutan diferentes etapas:

- Establece las directivas de síntesis.
- Analiza el modelo SystemC.
- Valida las directivas de síntesis.
- Realiza las transformaciones de código como *inlining* de funciones, desenrollado de bucles, introducción de segmentación en ruta de datos (*pipelining*) etc.
- Comprueba la sintetizabilidad del diseño y comienza la síntesis hardware.
- Realiza la planificación de cada proceso, explora su microarquitectura y genera el RTL.
- Realiza la planificación de cada módulo, explora su microarquitectura y genera el RTL.

Finalmente muestra el tiempo total de síntesis y los resultados. A continuación se muestran algunos mensajes de *log* de síntesis.

```
@I [HLS-10] -- Scheduling module 'csdd_main'
@I [HLS-10] -- Exploring micro-architecture for module 'csdd_main'
@I [HLS-10] -- Generating RTL for module 'csdd_main'
@I [HLS-10] -- Scheduling module 'csdd'
@I [HLS-10] -- Exploring micro-architecture for module 'csdd'
@I [HLS-10] -- Generating RTL for module 'csdd'
@I [RTGEN-100] Finished creating RTL model for 'csdd'.
@I [HLS-111] Elapsed time: 1.5 seconds; current memory usage: 144 MB.
@I [HLS-10] Finished generating all RTL models.
@I [WSYSC-301] Generating RTL SystemC for 'csdd'.
@I [WVHDL-304] Generating RTL VHDL for 'csdd'.
@I [WVLOG-307] Generating RTL Verilog for 'csdd'.
@I [HLS-112] Total elapsed time: 267.198 secs; peak memory: 144 MB.
@I [LIC-101] Checked in feature [HLS]
```

Los resultados de consumo de área se muestran en una tabla resumen como la que se puede ver en la Tabla 2 donde figura el consumo de área expresado en BRAMs, DSPs, FFs y LUTs y un porcentaje de utilización respecto al total disponible en el dispositivo seleccionado. Por otra parte, en la Tabla 3 se muestra una estimación de tiempo de reloj mínimo con un cierto margen

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

de error, en este caso, el sistema puede funcionar con un reloj de 3.72 *ns* o lo que es lo mismo, puede trabajar a una frecuencia de 268.9 MHz.

Tabla 2. Resultados de uso de recursos en la síntesis de alto nivel

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	452	293
Memory	-	-	-	-
Multiplexer	-	-	-	416
Register	-	-	532	-
Total	0	0	984	709
Available	280	220	106400	53200
Utilization (%)	0	0	~0	1

Tabla 3. Resultados de tiempo en la síntesis de alto nivel.

Clock	Target	Estimated	Uncertainty
default	10.00	3.72	1.25

Una vez completada la síntesis debemos examinar el fichero RTL para ver si los resultados son coherentes con lo que habíamos planteado en SystemC.

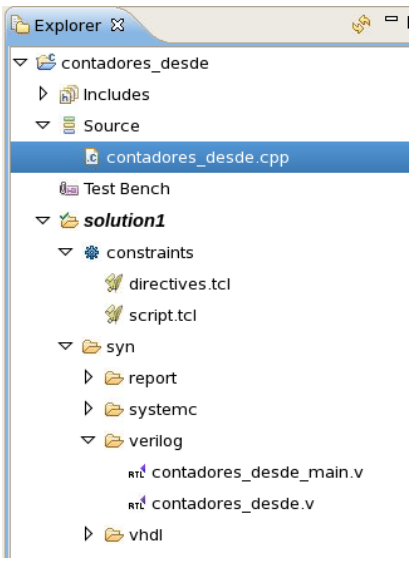


Figura 25. Ventana de exploración del proyecto de Xilinx Vivado HLS

En la ventana de exploración de ficheros *Explorer* tenemos una subcarpeta llamada *syn* que contiene los ficheros RTL tanto en Verilog como en VHDL. Xilinx Vivado HLS genera la solución en tres lenguajes: SystemC, Verilog y VHDL. En la subcarpeta Verilog vemos que existen dos ficheros: *csdd_main.v* y *csdd.v*.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Xilinx Vivado HLS, a diferencia de otros flujos de diseño que generan un único módulo Verilog a nivel RTL por cada módulo SystemC, genera varios ficheros Verilog por cada módulo SystemC en función de la jerarquía de funciones definida. Si examinamos los ficheros Verilog generados a nivel RTL, podemos ver que la herramienta ha generado el módulo *top* original llamado *csdd* pero además ha generado un submódulo *csdd_main*, instanciado dentro del anterior. Este módulo corresponde al proceso (o thread) *main* definido dentro del módulo principal y que Xilinx Vivado HLS lo ha generado como un nuevo submódulo y lo ha interconectado a la interfaz del módulo principal.

Además ha subdividido las interfaces definidas en SystemC cuyo tipo base es una estructura de datos (*struct*) las ha subdividido en sus componentes. Por ejemplo, *td_data_in*, es un puerto de entrada cuyo tipo es una estructura de datos que contiene 6 variables y como vemos en el código se han generado 12 puertos, 6 de entrada y 6 de salida. Para los puertos de salida también subdivide la estructura de datos pero no crea puertos de lectura. Como vimos anteriormente, Xilinx Vivado HLS no soporta la lectura de puertos de salida.

```
modulecsdd (

  clock,
  rst_n,
  ...
  td_data_in_0_i,
  td_data_in_0_o,
  td_data_in_1_i,
  td_data_in_1_o,
  td_data_in_2_i,
  td_data_in_2_o,
  td_data_in_3_i,
  td_data_in_3_o,
  td_data_in_4_i,
  td_data_in_4_o,
  td_data_in_5_i,
  td_data_in_5_o,
  td_data_in_6_i,
  td_data_in_6_o,
  td_data_out_0,
  td_data_out_1,
  td_data_out_2,
  td_data_out_3,
  td_data_out_4,
  td_data_out_5,
  td_data_out_6,
  ...
);

...

csdd_main grp_csdd_main_fu_1550(
  .ap_clk( clock ),
```

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

```
    .ap_rst(rst_n ),
    ...
    .td_data_in_0( grp_csdd_main_fu_1550_td_data_in_0 ),
    .td_data_in_1( grp_csdd_main_fu_1550_td_data_in_1 ),
    .td_data_in_2( grp_csdd_main_fu_1550_td_data_in_2 ),
    .td_data_in_3( grp_csdd_main_fu_1550_td_data_in_3 ),
    .td_data_in_4( grp_csdd_main_fu_1550_td_data_in_4 ),
    .td_data_in_5( grp_csdd_main_fu_1550_td_data_in_5 ),
    .td_data_in_6( grp_csdd_main_fu_1550_td_data_in_6 ),
    .td_data_out_0( grp_csdd_main_fu_1550_td_data_out_0 ),
    .td_data_out_0_ap_vld( grp_csdd_main_fu_1550_td_data_out_0_ap_vld ),
    .td_data_out_1( grp_csdd_main_fu_1550_td_data_out_1 ),
    .td_data_out_1_ap_vld( grp_csdd_main_fu_1550_td_data_out_1_ap_vld ),
    .td_data_out_2( grp_csdd_main_fu_1550_td_data_out_2 ),
    .td_data_out_2_ap_vld( grp_csdd_main_fu_1550_td_data_out_2_ap_vld ),
    .td_data_out_3( grp_csdd_main_fu_1550_td_data_out_3 ),
    .td_data_out_3_ap_vld( grp_csdd_main_fu_1550_td_data_out_3_ap_vld ),
    .td_data_out_4( grp_csdd_main_fu_1550_td_data_out_4 ),
    .td_data_out_4_ap_vld( grp_csdd_main_fu_1550_td_data_out_4_ap_vld ),
    .td_data_out_5( grp_csdd_main_fu_1550_td_data_out_5 ),
    .td_data_out_5_ap_vld( grp_csdd_main_fu_1550_td_data_out_5_ap_vld ),
    .td_data_out_6( grp_csdd_main_fu_1550_td_data_out_6 ),
    .td_data_out_6_ap_vld( grp_csdd_main_fu_1550_td_data_out_6_ap_vld ),
    ...
```

Para las señales de salida desde el módulo *csdd_main* hacia el top, *csdd*, ha generado un proceso de asignación para las señales de salida. Por otra parte, para las señales de entrada, ha generado simplemente una conexión. Como vemos en el siguiente fragmento de código, se comprueba en cada ciclo de reloj si la señal de *valid* es 1 para asignar la salida, en caso contrario asigna un 0.

```
/// td_data_out_1 assign process. ///
always @ (posedge clock)
begin : ap_ret_td_data_out_1
if ((ap_ST_st1_fsm_0 == ap_CS_fsm)) begin
      td_data_out_1 <= ap_const_lv64_0;
end else if ((ap_const_logic_1 ==
grp_csdd_main_fu_1550_td_data_out_1_ap_vld)) begin
      td_data_out_1 <= grp_csdd_main_fu_1550_td_data_out_1;
end
end
```

En Figura 26 se representa la estructura de bloques y adaptadores generados por la herramienta de síntesis. Suponiendo que DATA_IN y DATA_OUT son puertos cuyo tipo es una estructura de datos que contiene 4 variables: 2 *char*, 1 *short* y 1 *integer*, podemos ver que genera al menos dos módulos Verilog, donde se genera un módulo para cada una de las funciones y un módulo top que incluye dichas funciones e implementa la interfaz de entrada/salida. Esta forma de implementar el módulo SystemC proviene de dos conceptos fundamentales en Xilinx Vivado HLS: por una parte una función es equivalente a un módulo RTL y por otra la implementación de la funcionalidad es independiente de la implementación de las interfaces de E/S.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

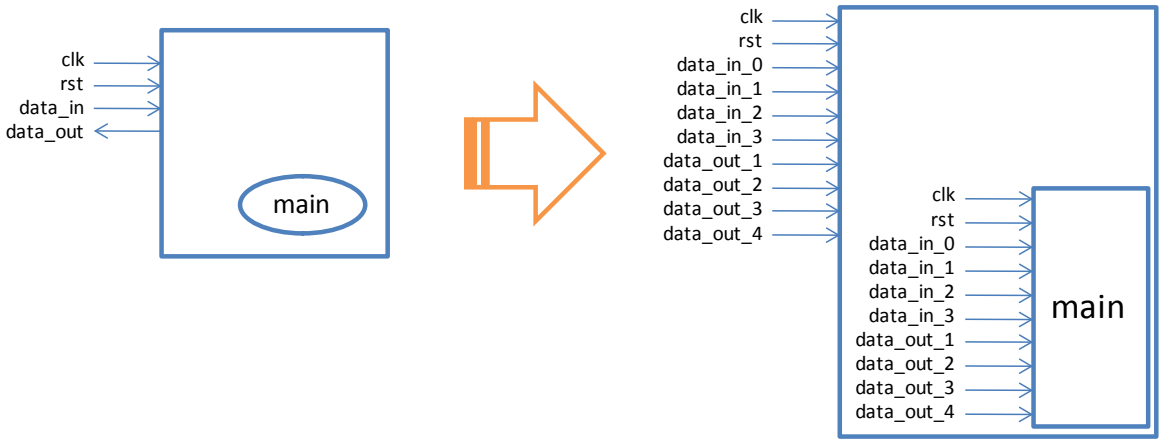


Figura 26. Transformación del diseño tras la síntesis con Xilinx Vivado HLS.

Toda la lógica de interconexión entre el módulo *top* y el submódulo consume recursos adicionales innecesarios. En algunos casos son simples interconexiones pero en otros hay señales de control y registros para manejar el protocolo de asignación de datos desde las salidas.

Los siguientes pasos tratan de resolver estos inconvenientes mediante el uso de directivas para que Xilinx Vivado HLS respete las interfaces definidas en SystemC y no realice el particionado de las estructuras de datos ni cree señales de control adicional ya que se utilizan protocolos de comunicación que no requieren ninguna señalización adicional para asegurar la comunicación entre módulos.

3.2.1 Uso de directivas y ajustes de síntesis

Como hemos visto en el apartado anterior, Xilinx Vivado HLS por defecto implementa su propio conjunto de interfaces de entrada y salida (AXI por ejemplo), por lo que, para diseños, como es este que tratamos en el presente TFM, en que el protocolo de E/S ha sido definido por el diseñador en SystemC, es necesario establecer un conjunto de directivas que evite la introducción de señalización adicional innecesaria. Ello trae como consecuencia la generación de lógica adicional y submódulos que consumen recursos.

En este apartado se explican los pasos realizados para obtener un modelo RTL equivalente a los obtenidos en el flujo de diseño original y que respete las interfaces de entrada y salida. Con ello se pretende compatibilizar las interfaces definidas para los dos flujos de síntesis haciendo que los bloques RTL sean intercambiables ya sean obtenidos mediante Cadence CtoS o mediante Xilinx Vivado HLS.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

3.2.2 Configuración de las interfaces

Mediante el uso de la directiva DATA_PACK podemos empaquetar los datos de los puertos cuya variable es una estructura de datos para agruparlos en una única palabra de ancho igual a la suma de los anchos de las variables que contiene.

Para esto debemos definir en el fichero de directivas la variable o puerto que deseamos empaquetar. También se puede hacer directamente en el código mediante la directiva #pragma.

Para cada caso tenemos:

```
set_directive_data_pack csdd td_data_in
#pragma HLS data_pack variable=td_data_in
```

Desde la interfaz gráfica, añadimos la directiva seleccionando el puerto en la pestaña directivas con el botón derecho en el menú *Insert Directive*. Como vimos antes, podemos insertarla en el fichero de directivas o en el fichero fuente.

Por otra parte, con la opción de *byte_pad* indicamos si queremos que alinee las variables a nivel de variable o a nivel de estructura como podemos ver en la Figura 28.

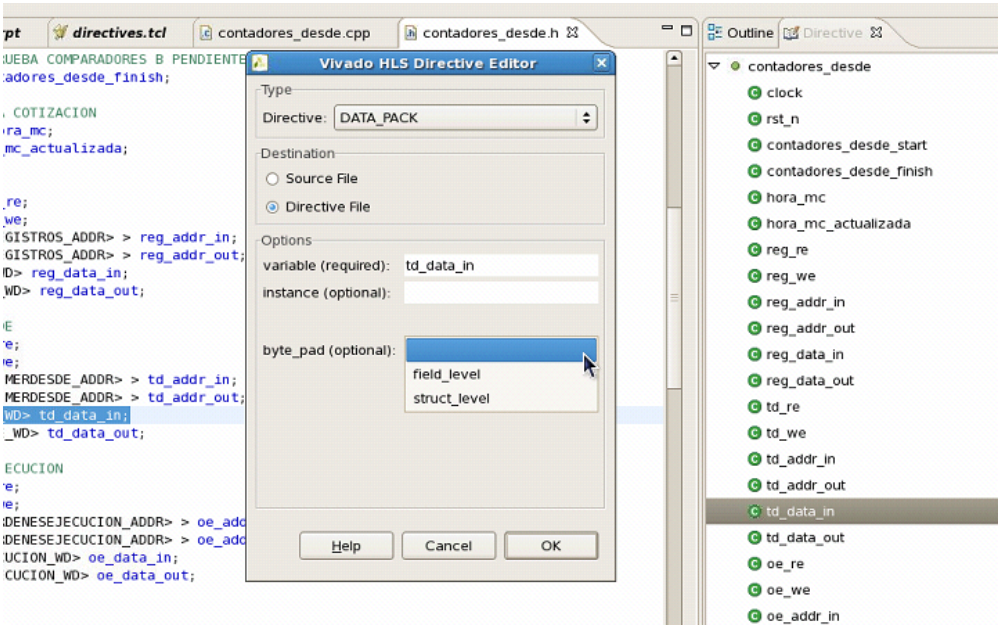


Figura 27. Inserción de directivas en Xilinx Vivado HLS.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

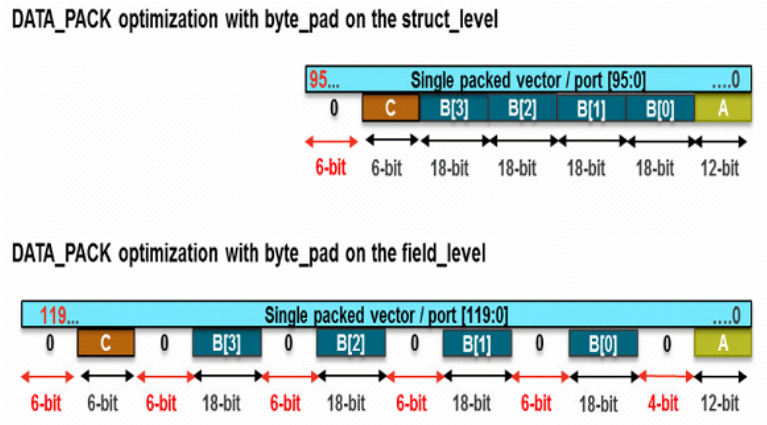


Figura 28. Alineación de los datos en el uso de la directiva DATA_PACK

La aplicación de esta directiva arroja el siguiente error de síntesis:

```
  @I [HLS-10] Checking synthesizability ...
  @E [XFORM-801] Interface read on 'csdd.td_data_in.m_if.Val'
(../Vivado/SystemC/src/csdd.cpp:120) has incompatible types. Possible
cause(s): data pack is only applied on source(port) or
destination(variable).
  @E [HLS-70] Failed building synthesis data model.
  @I [HLS-111] Elapsed time: 105.69 secs; current memory usage: 105 MB.
```

Esto puede deberse a que estamos aplicando la directiva al puerto de entrada y no a la variable de destino por lo que modificamos la directiva para aplicarlo solamente a la variable donde se escribe o lee el puerto:

Figura 29. Fichero de directivas de Xilinx Vivado HLS.

Con esta directiva el diseño sintetiza pero cuando realizamos un análisis del fichero de salida en Verilog vemos que no ha afectado a las interfaces y únicamente ha realizado el empaquetado de los datos a nivel interno.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

```
outputcsdd_orden_ejecucion_predicadoLogicoLiberado;
outputcsdd_orden_ejecucion_predicadoLogicoLiberado_ap_vld;
output  [7:0] csdd_orden_ejecucion_regenerable;
outputcsdd_orden_ejecucion_regenerable_ap_vld;
output  [160:0] csdd_timer_desde;
outputcsdd_timer_desde_ap_vld;
output  [7:0] csdd_nodo_y_id_orden_V;
outputcsdd_nodo_y_id_orden_V_ap_vld;
output  [7:0] csdd_nodo_y_num_timer_total_V;
```

La última opción posible es aplicarlo tanto a los puertos de entrada y salida como a la variable global donde se almacena.

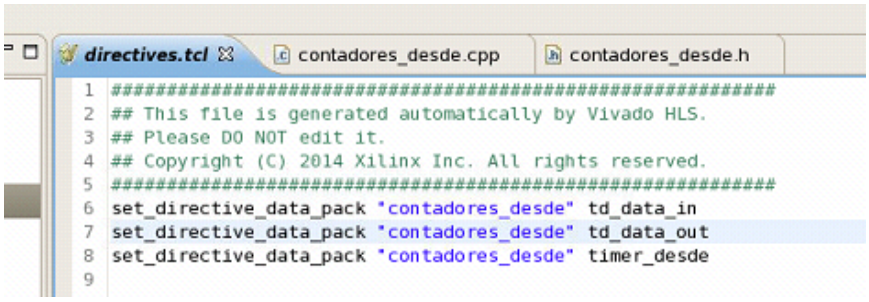


Figura 30. Fichero de directivas en Xilinx Vivado HLS.

Pero con estas directivas el diseño no sintetiza y obtenemos el mismo error que inicialmente. Se han realizado pruebas insertando las directivas en el código fuente pero el resultado es el mismo.

En el manual de referencia de Xilinx Vivado HLS hay un apartado específico dedicado a la directiva *DATA_PACK* y su aplicación a las interfaces de entrada pero en el caso de los ejemplos que se exponen se trata de código en lenguaje C cuya función tiene como parámetro de entrada una estructura de datos. En este caso la síntesis genera correctamente un puerto del ancho total de la estructura de datos sin dividir sus variables.

Así, queda descartada la vía del uso de la directiva DATA_PACK.

Como solución alternativa, con objeto de mantener la compatibilidad a nivel de E/S entre los bloques generados en Xilinx Vivado HLS y Cadence CtoS se realiza la creación de un *wrapper* para el empaquetado de los puertos y para generar las interfaces correctamente. Toda la lógica que se genere adicionalmente será optimizada por la herramienta de síntesis lógica.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

3.2.3 Creación del wrapper

Para la creación del wrapper de forma automática, se ha realizado una aplicación en Java que lea los ficheros de entrada y genere un fichero Verilog con los puertos correctos, la instanciación del módulo y la interconexión. En la Figura 31 se puede ver de manera gráfica.

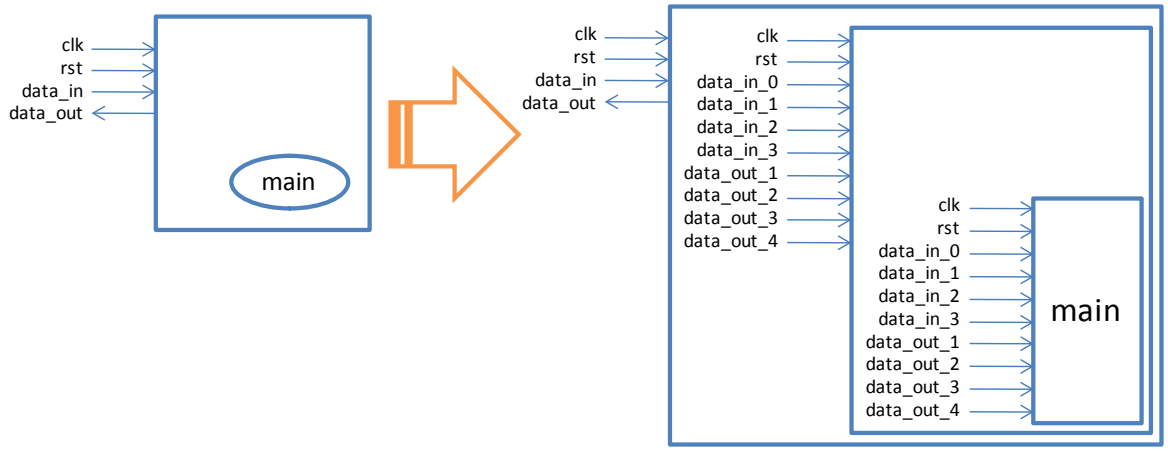


Figura 31. Transformación del diseño tras la utilización de un wrapper.

Los pasos seguidos por el algoritmo para generar los puertos son:

- Leer todos los puertos de entrada.
- Identificar los puertos que son estructuras de datos y calcular su tamaño y posición de cada variable.
- Generar el fichero de salida:
  - Generar el módulo *top* con los puertos reales.
  - Crear señales internas de interconexión con el submódulo.
  - Instanciar el submódulo e interconectarlo.
  - Asignar los puertos de entrada/salida del submódulo con los puertos de entrada/salida del *top* usando las señales definidas previamente.

Utilizando el software obtenemos todos los módulos con las interfaces compatibles y estamos en disposición de pasar a las siguientes fases del flujo de diseño, como por ejemplo la síntesis lógica.

3.2.4 Eliminación de la lógica adicional en puertos de E/S

Mediante el uso de la directiva *INTERFACE* podemos evitar que la herramienta de síntesis genere lógica de control para la asignación de los puertos de E/S:

```
set_directive_interface -mode ap_none "csdd" td_data_in
```

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Como apoyo para la automatización de la síntesis se ha realizado un *script* para ejecutar generar el fichero de directivas de cada puerto. Esta aplicación se ha desarrollado también en Java dentro del TFM, conjuntamente con la herramienta de generación de los *wrappers* anteriormente explicada.

4. Síntesis lógica con Synplify Premier

Una vez completada la síntesis de alto nivel para cada uno de los bloques del procesador de eventos, la siguiente etapa consiste en la síntesis lógica que transforma el diseño de un nivel RTL a un nivel de puertas lógicas, en este caso primitivas de Xilinx interconectadas formando un *netlist* completo en formato EDIF.

4.1 Creación y configuración del proyecto

En primer lugar debemos crear un nuevo proyecto en Synplify Premier desde el menú File / New Project. A continuación añadimos los ficheros Verilog obtenidos de la síntesis RTL del módulo que vamos a sintetizar.

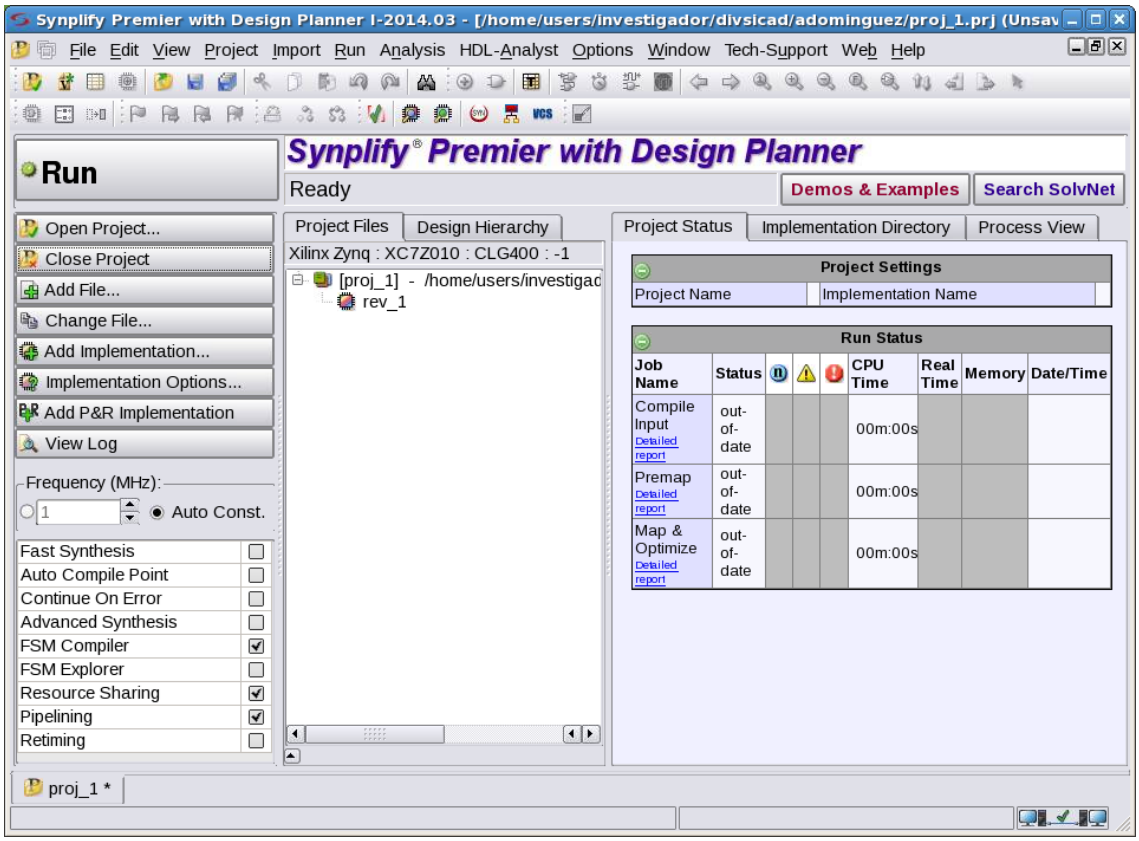


Figura 32. Entorno de trabajo de Xilinx Vivado HLS.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

4.2 Opciones de síntesis

A continuación se describen aquellas opciones a activar en las opciones de síntesis de la herramienta, con el fin de obtener los mejores resultados para el modelo de referencia con el que se trabaja (Figura 33).

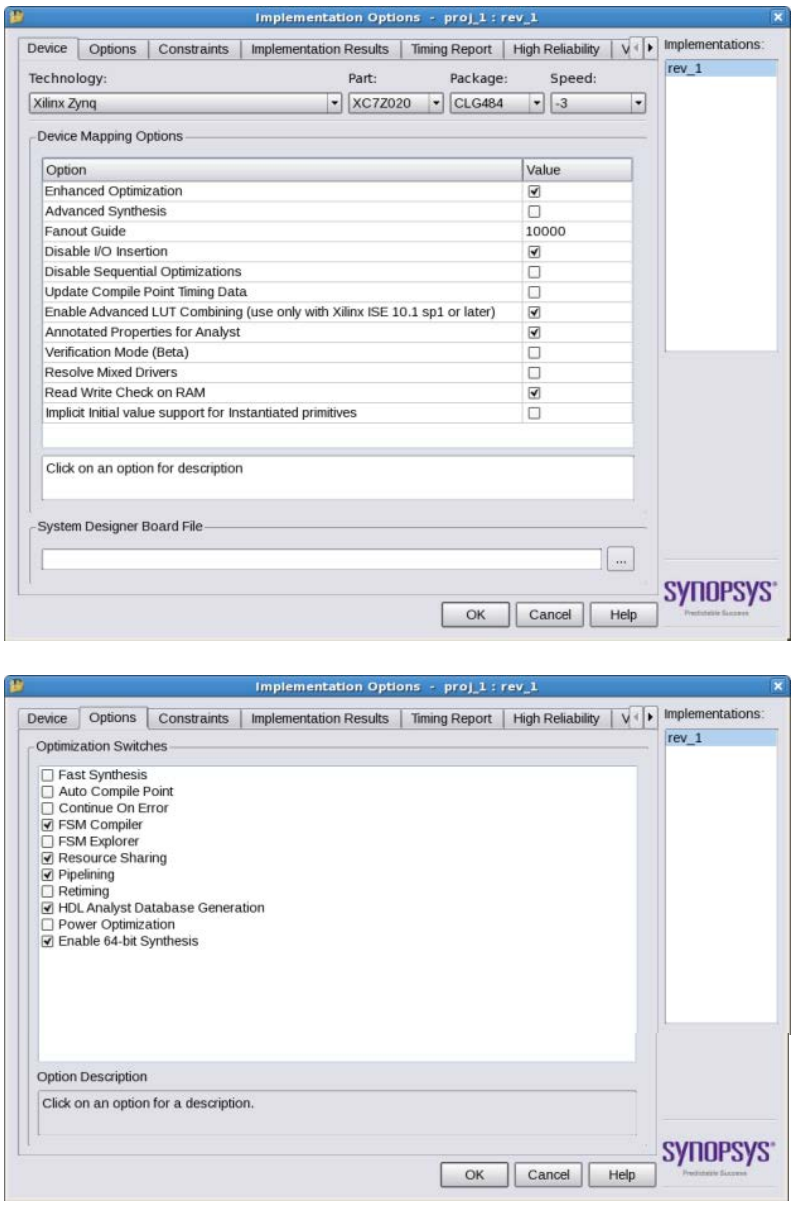


Figura 33. Opciones de síntesis en Synplify Premier

Disable I/O Insertion

Consiste en impedir la inserción de bloques de entrada/salida (IOB) de la FPGA para las señales de entrada y salida del top del diseño. Para el caso que nos ocupa, es importante activarlo ya se trata de sintetizar un bloque IP que será integrado en la plataforma final que se diseña siguiendo otro flujo de diseño diferente. Por tanto las E/S del IP estarán conectadas a otras señales internas de la plataforma.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Por defecto, las herramientas de síntesis dispondrán de IOBs para los puertos del bloque de mayor nivel de jerarquía (top), entendiendo que dichas señales estarán asociadas a pines de la FPGA, con el fin de conectarlas a dispositivos externos.

FSM Compiler

Se trata de un optimizador de máquinas de estado. Synplify ofrece la posibilidad de optimizar la lógica de estado siguiente de las instancias de máquinas de estado del diseño que encuentre, siguiendo una estrategia diferente de codificación de estados en función del número de estos, según la Tabla 4.

Tabla 4. Codificación de estados de FSM Compiler

Número de estados	Tipo de codificación
< 5	Secuencial
5 – 24	One-Hot
> 24	Gray

Resource Sharing

Permite compartir recursos con el fin de reducir el consumo de recursos del dispositivo FPGA, a costa de reducir la frecuencia.

Pipelining

Permite que varias operaciones se realicen a la vez sobre el mismo recurso, partiendo dicha ejecución en etapas, y permitiendo que cada dato se encuentre en una de ellas.

Enable Advanced LUT Combining

Prepara el fichero *netlist* de salida para una posterior combinación de LUTs en los diseños sobre FPGAs de Xilinx.

4.3 Resultados de síntesis lógica

En la Figura 34 se muestra un resumen de los resultados de síntesis tanto en área, expresado en registros, LUTs, BRAMs, como en tiempo, expresado en frecuencia.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Project Settings

Project Name	proj_1	Implementation Name	rev_1
Top Module	[auto]	Pipelining	1
Retiming	0	Resource Sharing	1
Fanout Guide	10000	Disable I/O Insertion	0
Disable Sequential Optimizations	0	Clock Conversion	1

Run Status

Job Name	Status				CPU Time	Real Time	Memory	Date/Time
Compile Input Detailed report	Complete	7	415	0	-	00m:04s	-	19 Aug 2014 20:51:37
Premap Detailed report	Complete	44	2	0	0m:02s	0m:03s	169MB	19 Aug 2014 20:51:44
Map & Optimize Detailed report	Complete	33	31	0	0m:24s	0m:33s	187MB	19 Aug 2014 20:52:20
Multi-srs Generator Detailed report	Complete				00m:02s			19 Aug 2014 20:51:40

Area Summary

I/O ports (io_port)	891	Non I/O Register bits (non_io_reg)	395 (0%)
I/O Register bits (total_io_reg)	0	Block Rams (v_ram)	0 (140)
DSP48s (dsp_used)	0 (220)	LUTs (total_luts)	410 (0%)
Detailed report		Hierarchical Area report	

Timing Summary

Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
despliega_th_rtl|clock	553.3 MHz	470.3 MHz	-0.319
Detailed report		Timing Report View	

Optimizations Summary

Combined Clock Conversion	1 / 0 more

Figura 34. Resultados de síntesis lógica en Synplify Premier.

Podemos obtener información detallada de los recursos utilizados como pueden ser desplazadores, sumadores, buffers, LUTs, etc.

```
Resource Usage Report for despliega_th_rtl

Mapping to part: xc7z020clg484-3
Cell usage:
CARRY4          26 uses
FD              65 uses
FDE             313 uses
FDR             17 uses
GND             3 uses
VCC             3 uses
LUT1            2 uses
LUT2            19 uses
LUT3            26 uses
LUT4            249 uses
LUT5            40 uses
LUT6            103 uses

I/O ports: 891
I/O primitives: 627
IBUF            185 uses
IBUFG           1 use
OBUF            441 uses

BUFG            1 use

I/O Register bits:                  0
Register bits not including I/Os:   395 (0%)

Global Clock Buffers: 1 of 32 (3%)

Total load per clock:
```

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

```
    despliega_th_rtl|clock: 395

  Mapping Summary:
  Total  LUTs: 410 (0%)

  Distribution of All Consumed LUTs = LUT1 + LUT2 + LUT3 + LUT4 + LUT5 +
LUT6- HLUTNM/2
  Distribution of All Consumed Luts 410 = 2 + 19 + 26 + 249 + 40 + 103-
58/2


   Number of unique control sets:             17
```

Por otra parte, también da información detallada del periodo estimado de reloj así como de la ruta crítica:

Tabla 5. Resultados de frecuencia y periodo.

```
Worst slack in design: -0.319

                           Requested     Estimated     Requested     Estimated
Starting Clock             Frequency     Frequency     Period        Period        Slack
-----------------------------------------------------------------------------------------
despliega_th_rtl|clock     553.3 MHz     470.3 MHz     1.807         2.126         -0.319
=========================================================================================
```

4. Síntesis del procesador de eventos

Una vez hemos obtenido resultados de cada bloque y tenemos la certeza de que la síntesis lógica se realiza con éxito, se ha realizado la síntesis completa del procesador de eventos.

Adicionalmente ha sido necesario ejecutar el flujo de diseño con CtoS para los módulos pero con el nuevo dispositivo Xilinx Zynq para obtener resultados y poder comparar ambos flujos eliminando el efecto tecnológico. La comparación de los resultados obtenidos se expone en capítulo siguiente.

5. Conclusiones

En este capítulo se ha descrito el desarrollo del proyecto y las dificultades para realizar la síntesis de alto nivel con SystemC. Ha sido necesario ajustar el código a un subconjunto sintetizable por la herramienta de síntesis y ha sido necesaria la adaptación de las interfaces ya que Xilinx Vivado HLS no respeta las especificaciones descritas en SystemC. Por último se ha realizado una síntesis lógica de los módulos para obtener resultados parciales y una síntesis global del procesador de eventos.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Capítulo 5: Resultados y conclusiones

1. Introducción

En este capítulo se exponen los resultados obtenidos en las diferentes etapas de este Trabajo Fin de Máster, esto es, el uso de recursos de cada bloque que conforma el procesador de eventos en término de bloques de memoria BRAM, procesadores DSP, registros FF y unidades LUT, además de la frecuencia de trabajo para cada módulo. Finalmente se exponen las conclusiones y los posibles trabajos futuros que se deriven.

2. Resultados de síntesis

En este apartado se presentan los resultados de la síntesis realizada en el TFM. Por una parte se presentan los resultados obtenidos durante la fase de síntesis de alto nivel y por otra los datos de la síntesis lógica realizada.

2.1 Resultados de síntesis de alto nivel

En la Tabla 6 se muestran los resultados de la síntesis de alto nivel con Xilinx Vivado HLS. Como podemos ver los módulos de mayor dimensión son *lbrs*, *etos*, *dpca* y *dstx*.

Tabla 6. Consumo de recursos del procesador de eventos desglosado por submódulos.

	BRAM	DSP	FF	LUT	CLOCK(ns)
cca	0	0	514	434	4,74
ccb	0	0	964	695	6,37
ccap	0	0	937	573	2,52
ccbp	0	0	588	339	6,11
csdd	0	0	1.520	1.029	3,72
csdt	0	0	1.201	784	2,71
csht	0	0	1.173	871	3,37
dpca	0	8	3.142	3.497	8,52
dpcb	0	0	702	373	2,00
deoe	0	0	2.321	2.645	3,39
dptd	0	0	818	1.183	4,97
dpth	0	0	1.520	1.321	4,08
dsrx	0	2	1.598	1.628	8,39
dstx	0	8	2.882	3.852	8,52
etos	0	72	7.911	8.926	8,52
elas	0	0	938	1.092	4,74
elcd	0	0	655	640	2,0
iftz	0	0	978	1.597	6,04
lcmd	0	0	3.939	3.085	2,73
lbrs	0	0	8.181	8.962	3,69
ntns	0	0	1.371	3.758	4,78
nvct	0	0	396	295	2,71
rffa	1	0	78	194	6,48
rffc	1	0	99	215	6,48
rffd	0	0	145	237	6,11
rffdi	1	0	87	172	6,23
rffdo	1	0	87	172	6,23
rfflac	0	0	135	392	6,48
rfflc	1	0	93	193	6,35
rfflcap	0	0	114	371	6,48
rffn	1	0	93	193	6,35
tuda	0	0	1.819	1.892	4,08

La frecuencia máxima de trabajo viene determinada por el periodo mínimo de reloj del módulo más lento, en este caso *etos* o *dstx*, con un periodo de reloj de 8,52 ns o lo que es lo

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

mismo, una frecuencia máxima de trabajo de 117,37 MHz. Estos datos pueden sufrir variaciones tras el proceso de síntesis lógica y de las optimizaciones realizadas por el sintetizador como se vio en el capítulo anterior.

En las siguientes gráficas podemos ver una comparativa de los módulos más importantes del diseño y su peso en área dentro del total. Se han obviado los módulos FIFO y RAM ya que no aportan gran información debido a su bajo consumo de recursos respecto al resto de los módulos.

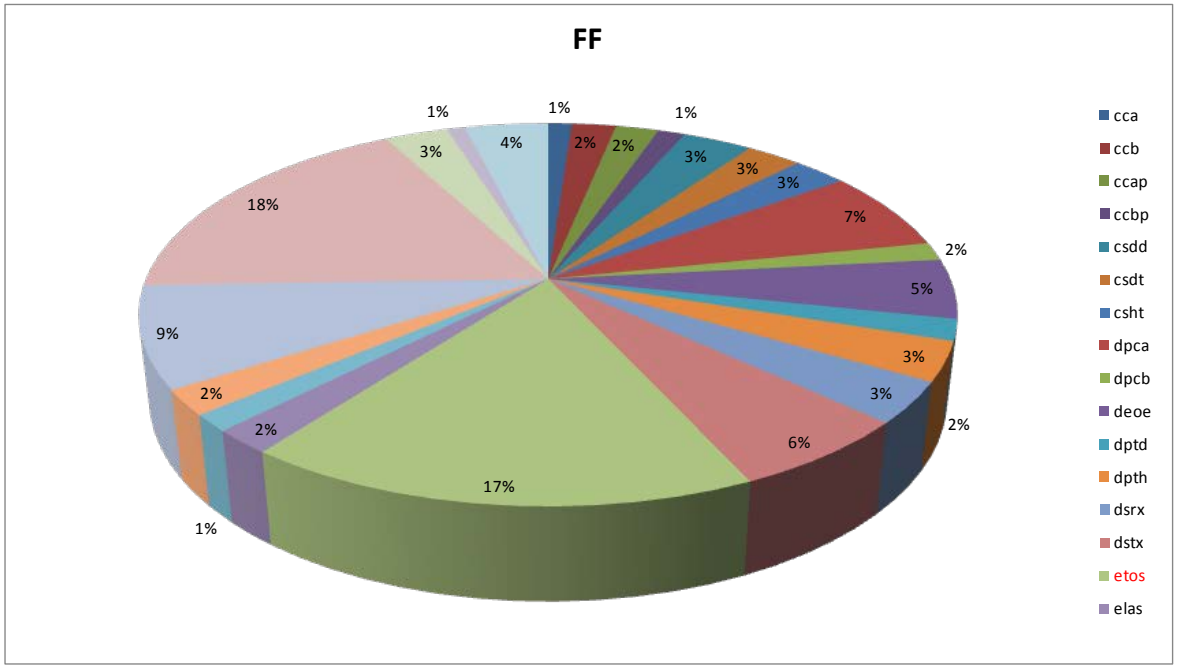


Figura 35. Consumo de registros de los módulos más importantes.

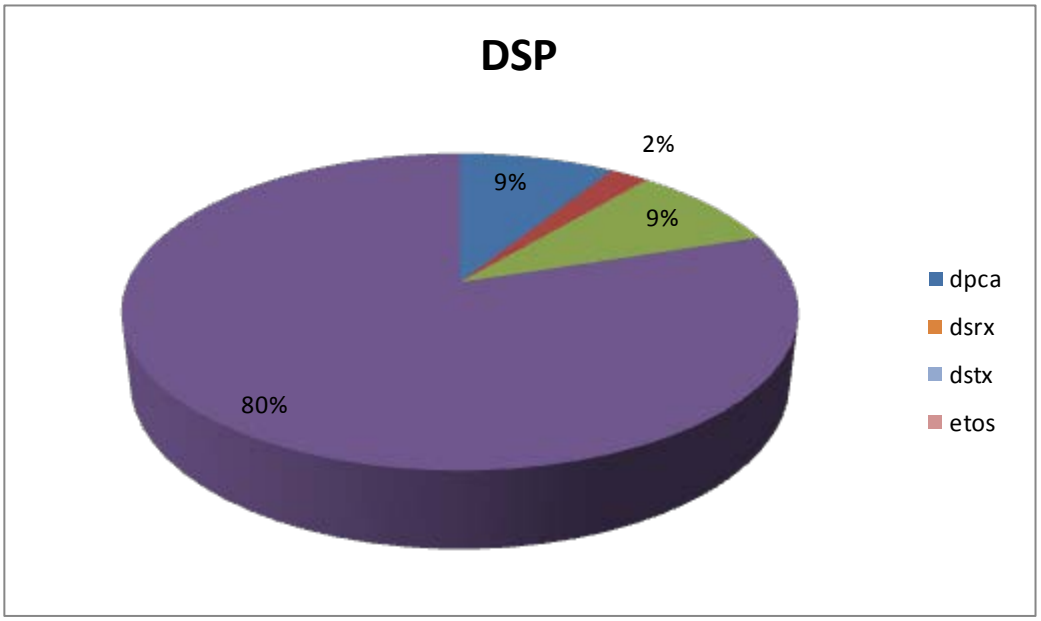


Figura 36. Consumo de DSPs de los módulos más importantes.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

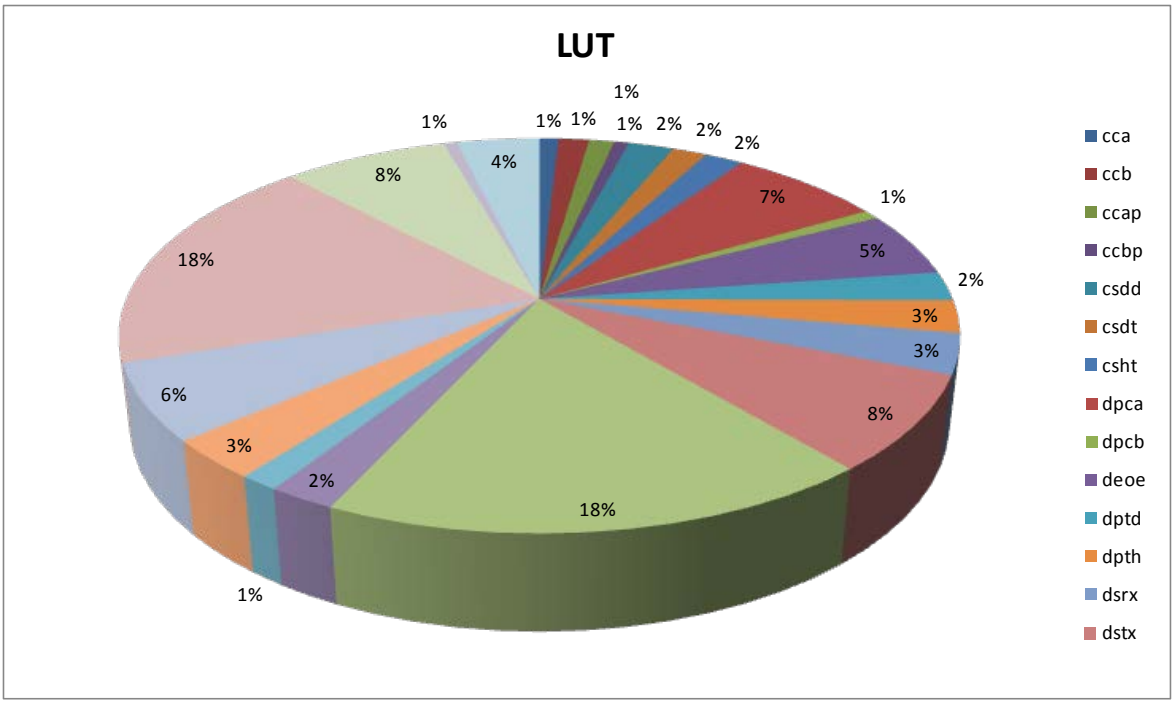


Figura 37. Consumo de LUTs de los módulos más importantes.

2.2 Resultados de síntesis lógica

En la Tabla 7 se exponen los resultados de síntesis lógica con Synopsys Synplify Premier. Los resultados de consumo de recursos y de frecuencia de trabajo han mejorado notablemente con respecto a la previsión hecha por la herramienta de síntesis de alto nivel para la mayoría de los casos. Los resultados de la síntesis de alto nivel indicaban los módulos *etos* y *dstx* como los módulos más lentos. Sin embargo ahora el módulo más lento es *dstx* seguido de *ntns* pero con una notable mejora en el periodo de reloj, pasamos de 8,52 *ns* a 4,40*ns*, o lo que es lo mismo, de 117,37 MHz a 227,27 MHz, lo que supone casi el doble de velocidad.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Tabla 7. Consumo de recursos desglosado por bloques.

	BRAM	DSP	FF	LUT	CLOCK (ns)
cca	0	0	200	170	2,34
ccb	0	0	188	180	2,06
ccap	0	0	138	108	1,25
ccbp	0	0	92	66	1,38
csdd	0	0	214	290	1,88
csdt	0	0	212	154	1,40
csht	0	0	168	149	1,64
dpca	0	0	2.035	4.192	2,09
dpcb	0	0	689	212	1,72
deoe	0	0	724	1242	2,59
dptd	0	0	245	394	2,44
dpth	0	0	395	410	2,12
dsrx	0	0	433	448	2,47
dstx	0	8	2.427	2.082	4,40
etos	0	51	7.144	3.490	3,91
elas	0	0	237	566	2,95
elcd	0	0	149	128	1,72
iftz	0	0	821	856	3,21
lcmd	0	0	1.003	977	2,41
lbrs	0	0	737	752	2,65
ntns	0	0	1.456	2.448	4,39
nvct	0	0	322	154	2,15
rffa	0	0	68	115	1,87
rffc	1	0	69	123	2,13
rffd	0	0	96	156	1,80
rffdi	1	0	63	114	2,13
rffdo	1	0	63	114	2,13
rfflac	0	0	35	58	1,73
rfflc	0	0	109	192	1,84
rfflcap	0	0	96	156	1,80
rffn	0	0	105	206	1,78
tuda	0	0	463	592	2,25

En la Tabla 8 podemos ver un resumen de cómo han mejorado los resultados respecto a la síntesis de alto nivel. Se ha marcado en verde el resultado de mejora y en rojo el caso en que empeora.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Tabla 8. Comparativa de resultados de síntesis de alto nivel y síntesis lógica.

	BRAM	DSP	FF	LUT	CLOCK
cca	0,00%	0,00%	61,09%	60,83%	6,71%
ccb	0,00%	0,00%	80,50%	74,10%	38,15%
ccap	0,00%	0,00%	85,27%	81,15%	-0,20%
ccbp	0,00%	0,00%	84,35%	80,53%	48,36%
csdd	0,00%	0,00%	85,92%	71,82%	-3,17%
csdt	0,00%	0,00%	82,35%	80,36%	-5,02%
csht	0,00%	0,00%	85,68%	82,89%	7,09%
dpca	0,00%	0,00%	35,23%	-19,87%	44,75%
dpcb	0,00%	0,00%	1,85%	43,16%	-65,10%
deoe	0,00%	0,00%	68,81%	53,04%	-41,45%
dptd	0,00%	0,00%	70,05%	66,69%	10,99%
dpth	0,00%	0,00%	74,01%	68,96%	-9,95%
dsrx	0,00%	100,00%	72,90%	72,48%	48,56%
dstx	0,00%	0,00%	15,79%	45,95%	37,95%
etos	0,00%	29,17%	9,70%	60,90%	20,96%
elas	0,00%	0,00%	74,73%	48,17%	23,44%
elcd	0,00%	0,00%	77,25%	80,00%	-68,05%
iftz	0,00%	0,00%	16,05%	46,40%	-14,30%
lcmd	0,00%	0,00%	74,54%	68,33%	-95,90%
lbrs	0,00%	0,00%	90,99%	91,61%	-19,30%
ntns	0,00%	0,00%	-6,20%	34,86%	-45,94%
nvct	0,00%	0,00%	18,69%	47,80%	-74,35%
rffa	100,00%	0,00%	12,82%	40,72%	20,77%
rffc	0,00%	0,00%	30,30%	42,79%	42,61%
rffd	0,00%	0,00%	33,79%	34,18%	50,87%
rffdi	0,00%	0,00%	27,59%	33,72%	49,92%
rffdo	0,00%	0,00%	27,59%	33,72%	49,92%
rfflac	0,00%	0,00%	74,07%	85,20%	26,79%
rfflc	100,00%	0,00%	-17,20%	0,52%	41,43%
rfflcap	0,00%	0,00%	27,59%	33,72%	49,92%
rffn	100,00%	0,00%	-12,90%	-6,74%	43,51%
tuda	0,00%	0,00%	74,55%	68,71%	-4,26%

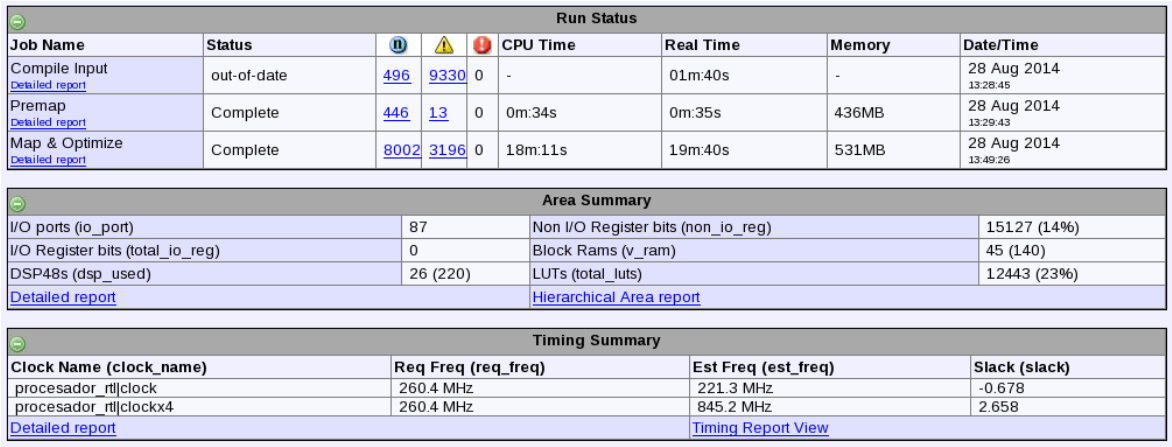
3. Síntesis global del procesador de eventos

Tras la síntesis de cada uno de los módulos para obtener datos parciales y para verificar que se realiza la síntesis correctamente de cada bloque creamos el módulo procesador donde se instancian todos los módulos y se interconectan. Además se incluyen los bloques de memoria BRAM con sus correspondientes controladores.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

En la Figura 38 se muestran los resultados de síntesis, así como en la Tabla 1. Como podemos observar, tenemos un consumo total de 15.127 registros, lo que supone un 14% del total disponible en el dispositivo FPGA; un total de 12.443 LUTs, esto es, un 23% del total disponible, 26 DSPs de los 220 disponible y 45 BRAM de los 140 disponibles.

La frecuencia de trabajo global es de 221,30 MHz, un poco inferior a lo que estimamos a partir del módulo más lento debido a efectos del interconexionado, muy importantes en la FPGAs.



Run Status

Job Name	Status				CPU Time	Real Time	Memory	Date/Time
Compile Input Detailed report	out-of-date	496	9330	0	-	01m:40s	-	28 Aug 2014 13:28:45
Premap Detailed report	Complete	446	13	0	0m:34s	0m:35s	436MB	28 Aug 2014 13:29:43
Map & Optimize Detailed report	Complete	8002	3196	0	18m:11s	19m:40s	531MB	28 Aug 2014 13:49:26

Area Summary

I/O ports (io_port)	87	Non I/O Register bits (non_io_reg)	15127 (14%)
I/O Register bits (total_io_reg)	0	Block Rams (v_ram)	45 (140)
DSP48s (dsp_used)	26 (220)	LUTs (total_luts)	12443 (23%)
Detailed report		Hierarchical Area report	

Timing Summary

Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
procesador_rtl|clock	260.4 MHz	221.3 MHz	-0.678
procesador_rtl|clockx4	260.4 MHz	845.2 MHz	2.658
Detailed report		Timing Report View	

Figura 38. Resultados de síntesis del procesador completo.

Tabla 9. Resultados de síntesis del procesador completo.

PROCESADOR DE EVENTOS	
LUT	12.443
BRAM	45
FF	15.127
Bloques DSP	26
Frecuencia (MHz)	221,30

4. Comparativa del flujo de Xilinx Vivado HLS con Cadence CtoS

Una vez hemos obtenido los resultados finales de síntesis lógica de todos los módulos del procesador de eventos podemos comparar la eficiencia de las herramientas de síntesis de alto nivel sobre la calidad de los resultados finales. Como los resultados se comparan sobre los diseños ya sintetizados a nivel lógico, ha sido necesario eliminar los efectos de las herramientas de síntesis lógica, para lo que se ha utilizado el mismo entorno (Synopsys Synplify Pemier) y el efecto de la tecnología, para lo que se ha utilizado el mismo dispositivo FPGA Zynq. Ello ha conllevado la realización de la síntesis del procesador de eventos para el dispositivo Zynq con el flujo de diseño de Cadence CtoS.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

4.1 Comparativa de síntesis del alto nivel

En las Figura 39, Figura 40 y Figura 41 se muestra una comparativa de los resultados de Xilinx Vivado HLS y Cadence CtoS expresada en términos de recursos consumidos: FFs, LUTs y DSPs respectivamente.

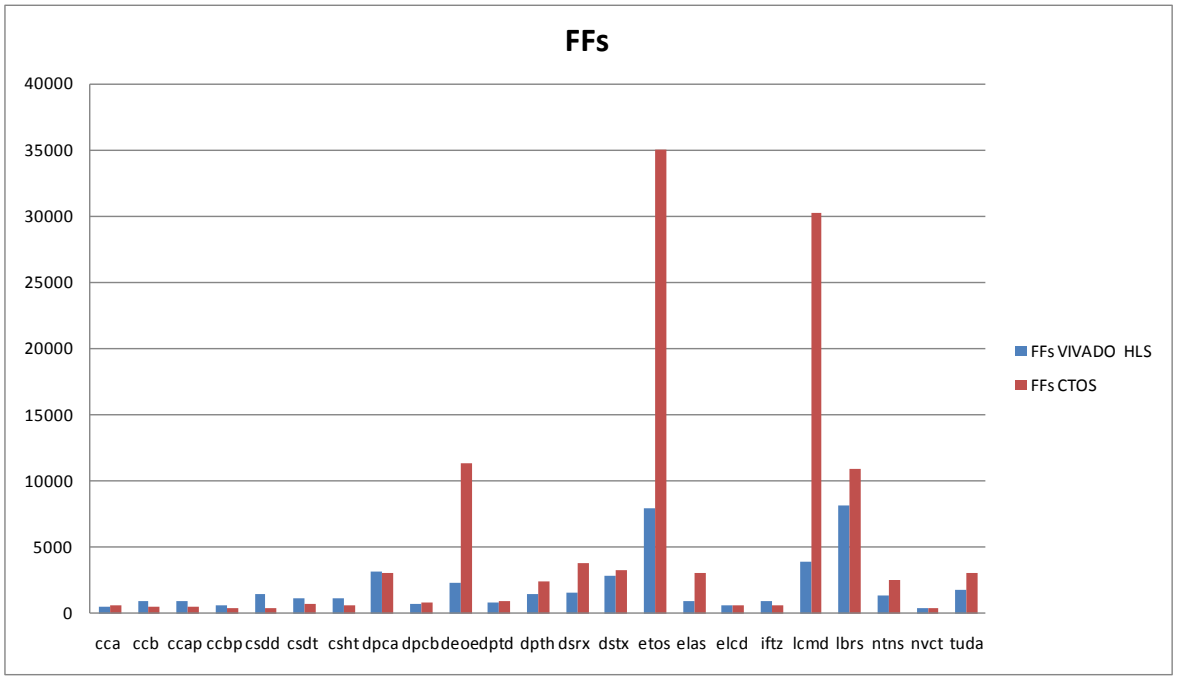


Figura 39. Comparativa de flujos en FFs.

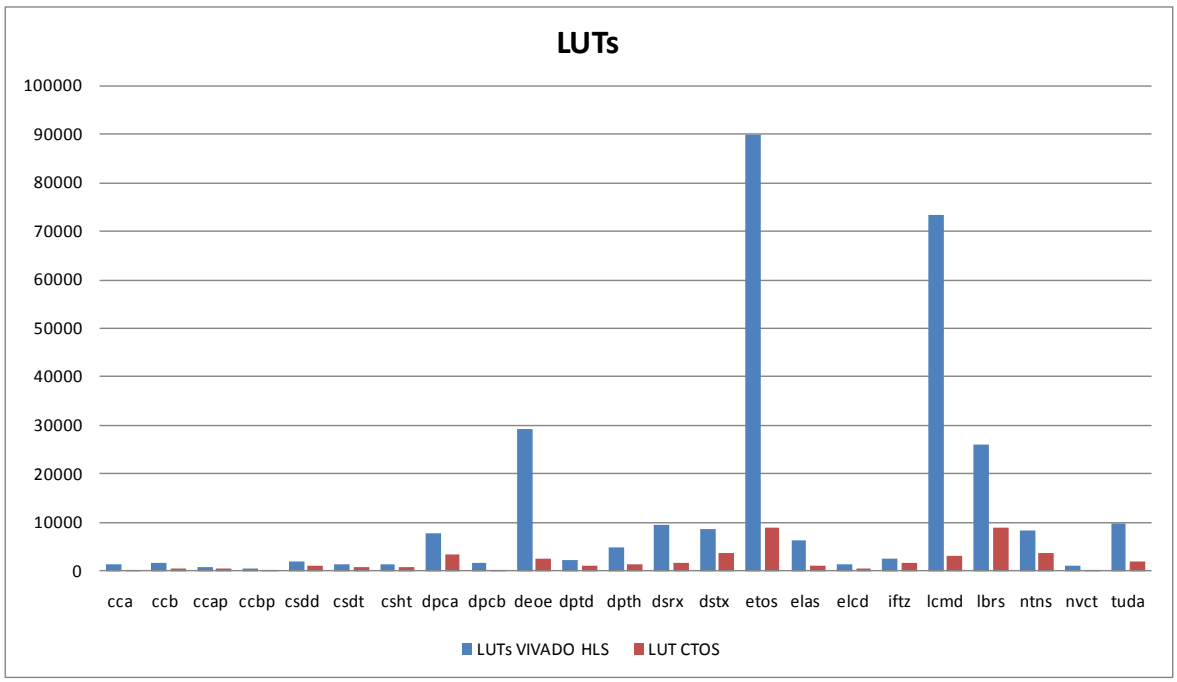


Figura 40. Comparativa de flujos de diseño en LUTs.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

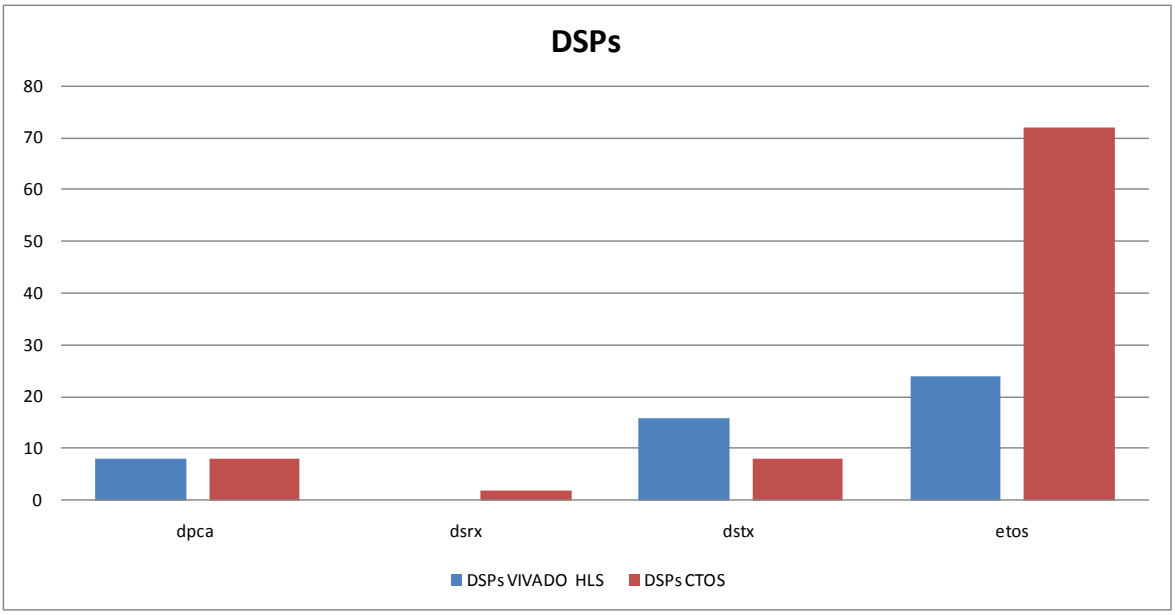


Figura 41. Comparativa de flujos en DSPs.

En cuanto al uso de registros, tal como se aprecia en la Figura 39, para la mayoría de los bloques los resultados han sido mejores en el flujo de Xilinx Vivado HLS que la síntesis de alto nivel realizada con Cadence CtoS.

En cuanto al uso de LUTs, todos los bloques consumen más LUTs cuando se utiliza el flujo de Xilinx Vivado HLS. Un caso especial a analizar es el bloque **etos**, donde parece claro que el entorno Cadence CtoS se ha decantado por utilizar bloques DSP para implementar la funcionalidad del bloque mientras que Xilinx Vivado HLS ha optado por implementar la funcionalidad basada en LUTs, lo que ha tenido la consecuencia directa en la aumento del periodo de reloj.

Por otra parte, los resultados temporales empeoran respecto a Cadence CtoS, en algunos casos la frecuencia de trabajo se reduce hasta la mitad. Esto puede ser debido a opciones de síntesis que priorizan la optimización temporal en detrimento del uso de recursos.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

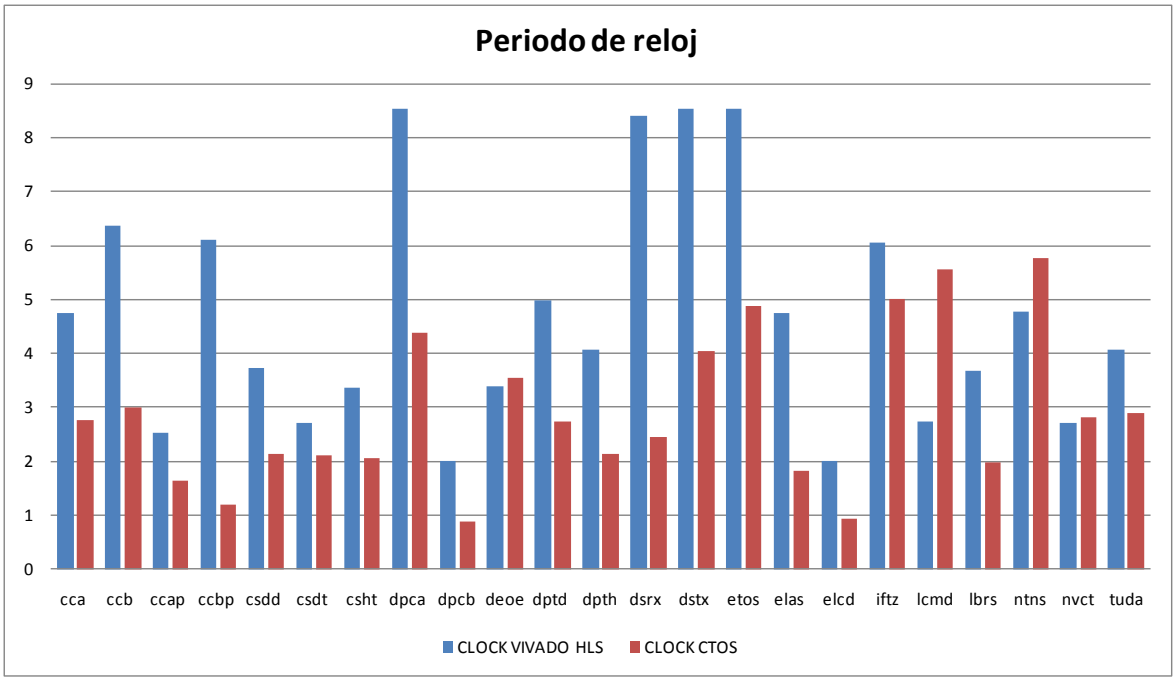


Figura 42. Comparativa de flujos en periodo de reloj.

En la Tabla 10 se muestran los porcentajes de mejora de los resultados de síntesis de Xilinx Vivado HLS con respecto a Cadence CtoS. Como vemos en el uso de recursos hay una notable mejora salvo en algunos módulos pero en las prestaciones temporales el flujo con Cadence CtoS presenta mejores resultados, esto puede ser debido a un mayor paralelismo, lo que implica mayor uso de recursos.

4.2 Comparativa de síntesis lógica

En la Figura 43, Figura 44 y Figura 45 se muestra una comparación del consumo de recursos de la síntesis lógica con Synopsys Synplify Premier para los flujos con Xilinx Vivado HLS y Cadence CtoS expresada en FFs, LUTs y DSPs.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Tabla 10. Comparativa de resultados en la síntesis de alto nivel.

	DSPs	FFs	LUTs	RELOJ
cca	0,00%	14,33%	70,73%	-71,00%
ccb	0,00%	-72,14%	57,34%	-112,05%
ccap	0,00%	-88,91%	30,71%	-54,51%
ccbp	0,00%	-48,11%	51,08%	-410,02%
csdd	0,00%	-281,91%	51,21%	-74,81%
csdt	0,00%	-74,06%	50,00%	-28,07%
csht	0,00%	-73,26%	37,56%	-63,28%
dpca	0,00%	-1,29%	55,02%	-94,30%
dpcb	0,00%	13,33%	78,81%	-122,72%
deoe	0,00%	79,47%	90,99%	4,51%
dptd	0,00%	15,23%	51,01%	-81,32%
dpth	0,00%	37,55%	72,42%	-89,68%
dsrx	100,00%	58,28%	82,73%	-242,73%
dstx	-100,00%	13,66%	56,08%	-110,94%
etos	66,67%	77,38%	90,07%	-74,66%
elas	0,00%	69,47%	83,25%	-158,17%
elcd	0,00%	-7,03%	55,40%	-112,54%
iftz	0,00%	-67,18%	41,46%	-20,87%
lcmd	0,00%	86,97%	95,79%	50,86%
lbrs	0,00%	24,83%	65,60%	-85,61%
ntns	0,00%	46,15%	54,99%	17,14%
nvct	0,00%	15,38%	77,17%	3,42%
tuda	0,00%	40,85%	80,84%	-41,47%

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

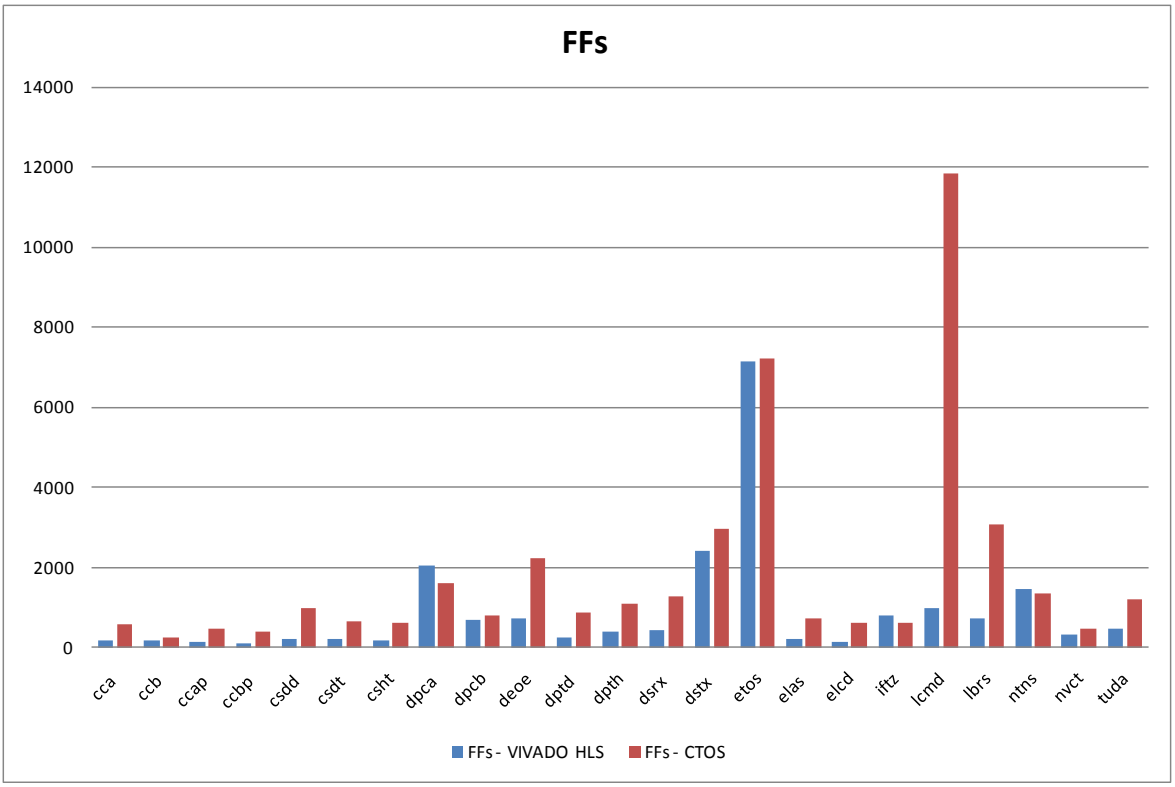


Figura 43. Resultados de síntesis lógica en FFs.

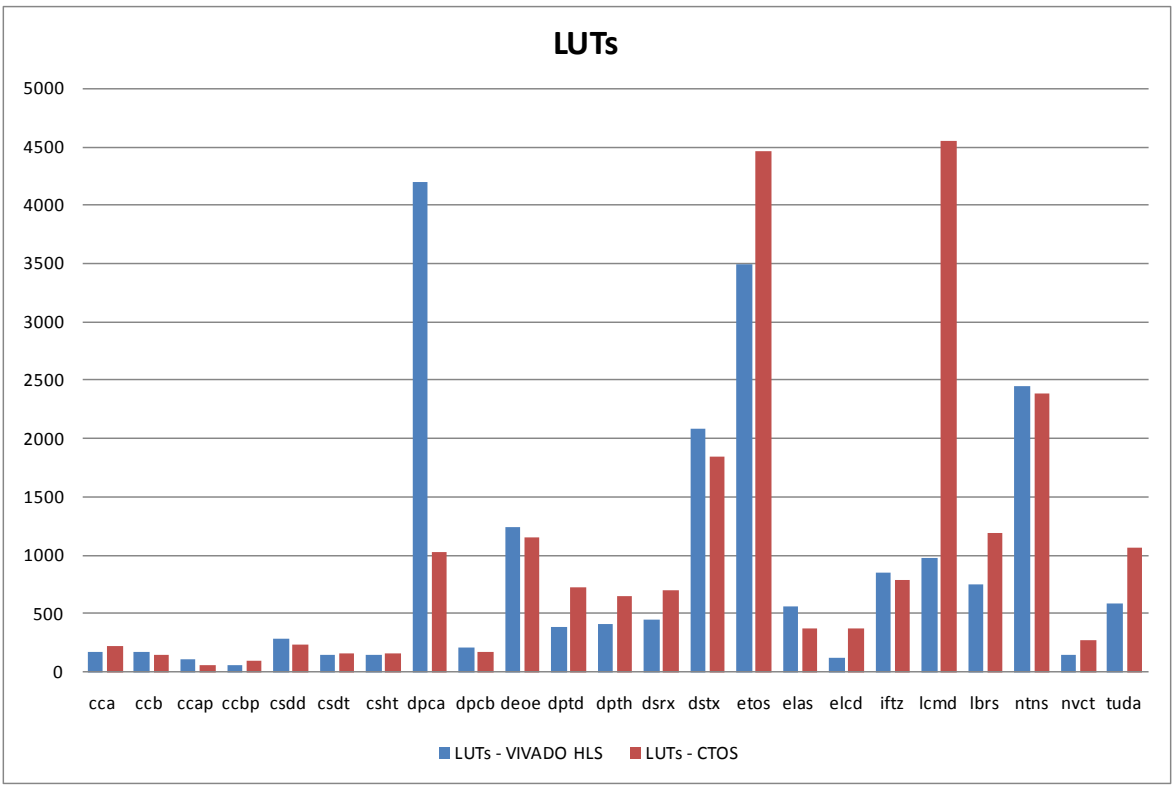


Figura 44. Resultados de síntesis lógica en LUTs.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

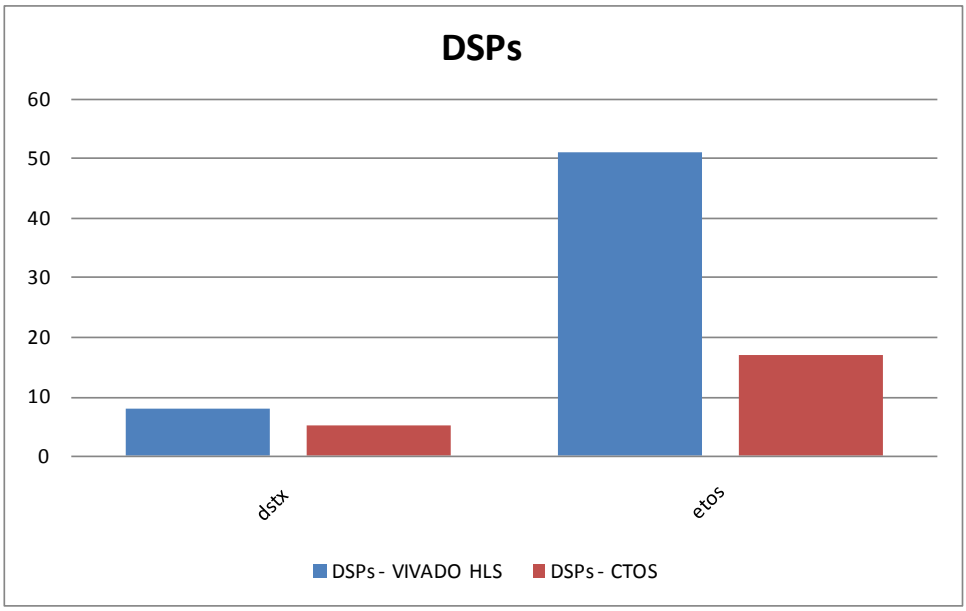


Figura 45. Resultados de síntesis lógica en DSPs.

Por otra parte, los resultados temporales mejoran ahora respecto a Cadence CtoS, en algunos casos la frecuencia de trabajo mejora el doble que con el flujo de diseño con Cadence CtoS.

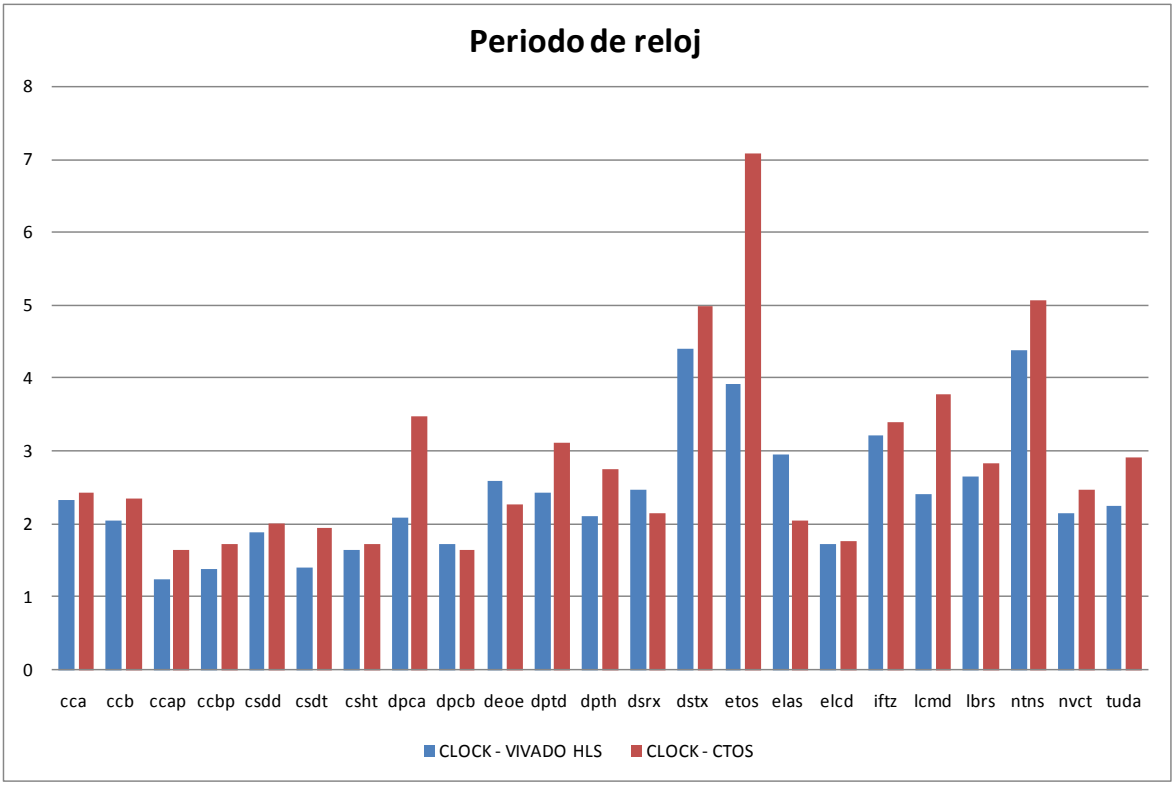


Figura 46. Resultados de síntesis de alto nivel en periodo de reloj.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

En la siguiente tabla se han calculado los porcentajes de mejora de los resultados de síntesis con Synopsys Synplify Premier para los flujos de diseño con Xilinx Vivado HLS con respecto a Cadence CtoS. Como vemos en el uso de recursos hay una notable mejora salvo en algunos módulos tanto en el uso de recursos como en las frecuencias de trabajo.

Tabla 11. Comparativa de resultados en la síntesis lógica.

	DSPs	FFs	LUTs	RELOJ
cca	0,00%	66,50%	23,08%	3,80%
ccb	0,00%	30,37%	-20,81%	12,72%
ccap	0,00%	70,07%	-56,52%	24,28%
ccbp	0,00%	77,06%	31,25%	20,55%
csdd	0,00%	77,98%	-23,40%	6,23%
csdt	0,00%	68,22%	4,35%	27,93%
csht	0,00%	73,33%	8,02%	5,57%
dpca	0,00%	-26,16%	-306,99%	39,89%
dpcb	0,00%	12,90%	-19,77%	-4,46%
deoe	0,00%	67,39%	-8,00%	-14,35%
dptd	0,00%	71,90%	45,73%	21,87%
dpth	0,00%	63,76%	37,12%	22,87%
dsrx	0,00%	65,93%	36,18%	-15,35%
dstx	37,50%	18,15%	-12,91%	11,82%
etos	66,67%	0,83%	21,80%	44,71%
elas	0,00%	66,99%	-48,56%	-43,40%
elcd	0,00%	75,89%	65,50%	2,70%
iftz	0,00%	-33,06%	-7,40%	5,69%
lcmd	0,00%	91,51%	78,52%	36,28%
lbrs	0,00%	76,05%	36,70%	6,30%
ntns	0,00%	-6,51%	-2,51%	13,39%
nvct	0,00%	33,06%	43,17%	13,23%
tuda	0,00%	61,22%	44,47%	23,01%

Finalmente podemos comparar los resultados a nivel del procesador completo. En la Tabla 12 se muestran los resultados del procesador completo. Como vemos la mejora en el consumo de recursos es en algunos casos superior al 60% y más del 40% de mejora en la frecuencia de trabajo.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Tabla 12. Comparativa de flujos para el procesador completo.

	Flujo Xilinx Vivado HLS	Flujo Cadence C-To-Silicon	Mejora
LUT	12.443	25.774	51,72%
BRAM	45	58	22,41%
FF	15.127	44.635	66,11%
DSP	26	38	31,58%
Frecuencia	221,30	131,70	40,49%

5. Conclusiones y líneas futuras

En el presente trabajo se han abarcado los pasos del flujo de diseño para, a partir de una aplicación descrita en un lenguaje SystemC, obtener una descripción hardware lista para implementar.

Los resultados obtenidos con el flujo de diseño basado en Xilinx Vivado HLS mejoran notablemente respecto al flujo con Cadence CtoS a pesar de las dificultades encontradas durante el desarrollo como han podido ser la adaptación de las interfaces, el uso de directivas y las modificaciones necesarias para que el código sea sintetizable.

Por otra parte, gracias al desarrollo de este trabajo, se han alcanzado, además, ciertas conclusiones acerca del desarrollo de sistemas basándose en herramientas de síntesis de alto nivel:

- La necesidad de adaptación del código a un subconjunto sintetizable hace complicada la tarea de resintetizar el código con cada nueva herramienta. Para diseños en los que es necesaria una actualización del diseño y la ejecución de un flujo de diseño completo cada cierto periodo de tiempo, la tarea de síntesis con nuevas y mejores herramientas puede ser compleja.
- Xilinx Vivado HLS no soporta ciertos aspectos de SystemC y lo trata como una clase de C, en vez de un lenguaje de descripción de sistemas TLM. Esto hace que las especificaciones del diseñador en algunos casos no se tengan en cuenta como es el caso de las interfaces de entrada.
- El uso de directivas para guiar la síntesis hacia el resultado deseado en muchas ocasiones no arroja buenos resultados, sobre todo si se utiliza SystemC.
- Xilinx Vivado HLS está orientado a un nivel mayor de abstracción que el que ofrece SystemC y a flujos de diseño para una rápida puesta en el mercado.
- Las herramientas de síntesis de alto nivel están en constante evolución, permitiendo cada vez más abstracción en el diseño a *hardware* a nivel de sistemas. La tendencia de dichas herramientas indican que en un futuro el diseño de sistemas *hardware* será cada vez más abstracto, como por ejemplo, en la etapa de comunicación con las

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

librerías TLM, permitiendo diseñar sistemas mucho más complejos sin la necesidad de plantear las transacciones entre bloques a nivel de ciclos.

- Para obtener resultados comparables, es necesario eliminar o corregir factores tecnológicos que no oculten la verdadera eficiencia de los métodos de diseño.
- La equivalencia de estrategias de síntesis también resulta clave durante la comparación debido al fuerte impacto que tiene sobre las prestaciones del diseño las decisiones realizadas en fases tempranas del diseño (hasta tres órdenes de magnitud), decisiones usuales en un flujo de diseño de síntesis de alto nivel.
- En cualquier caso es necesario un ejercicio de evaluación de la calidad de los resultados antes de adoptar un flujo de diseño determinado.

6. Líneas futuras

A partir de las conclusiones del presente proyecto, se plantean nuevas líneas de trabajo que parten de estas o que usan estas como base en su desarrollo. A continuación se indican algunas posibles:

- Hacer uso de nuevas herramientas de desarrollo de sistemas empotrados orientados a la implementación del sistema en el SoC reconfigurable con el fin de estudiar las mejoras aportadas en la minimización del tiempo de desarrollo y de los recursos humanos necesarios.
- Obtener resultados de síntesis de alto nivel con un mismo algorítmico en su versión en C y SystemC para ejecutar el flujo de diseño y comparar resultados en ambos casos y obtener conclusiones sobre el comportamiento de la herramienta en ambos casos.
- Implementar el diseño sobre la plataforma de prototipado Zynq, adaptando previamente las interfaces al bus AXI para obtener resultados de rendimiento y compararlos al proyecto original.

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

Bibliografía

[1] S. Levi, A. K. Agrawala. *Real-Time System Design.* Universidad de Michigan: McGraw-Hill Pub. Co., 1990.

[2] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, H. Jacobsen, "Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading," en *36th International Conference on very Large Data Bases (VLDB),* 2010.

[3] M. John, S. Smith, *Application-Specific Integrated Circuits.* Addison Wesley Professional,2008.

[4] T. Grötker, *System Design with SystemC.* Boston: Kluwer Academic Publishers, 2002.

[5] M. Fingeroff, *High-Level Synthesis : Blue Book.* S.L.: Xlibris Corporation, 2010.

[6] P. Coussy, A. Morawiec, *High-Level Synthesis from Algorithm to Digital Circuit.*Springer,2008.

[7] C. C.-T.-S. Compiler, «Cadence C-To-Silicon Compiler,» [En línea]. Available: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx. [Último acceso: Marzo 2014].

[8] B. Cico and H. Rexha. "Implementing Codesign in Xilinx Virtex II Pro," presentedo en la cuarta Balkan Conference, 2009.

[9] TMS320DM816x DaVinci Digital Media Processors: Technical Reference Manual , Texas Instruments, Inc., 2011.

[10] P. Bishop. Using ARM Processor-based Flash MCUs as a Platform for Custom Systems-on-Chip. [En línea]. Disponible: http://www.design-reuse.com/articles/13742/using-arm-processor-based-flash-mcus-as-a-platform-for-custom-systems-on-chip.html

[11] «Design + System Drivers Update,» ITRS Public Conference, [En línea]. Available: http://www.itrs.net/Links/2012Winter/1205%20Presentation/DesignSD_12052012.pdf. [Último acceso: Abril 2014].

[12] S. Rigo, R. Azevedo, L. Santos. *Electronic System Level Design: An Open-Source Approach. Springer,*2011.

[13] *IEEE Standard for Standard SystemC® Language Reference Manual*. IEEE Computer Society, 2012.

[14] D. C. Black, J. Donovan, SpringerLink. *SystemC: From the Ground Up.* Springer,2009.

[15] M. Ganguly, *SystemC Overview*, Synopsys, Inc., 2001.

[16] V. de Armas Sosa, *Tutorial Verilog*, 2004.

[17] X. V. D. Suite, «Xilinx Vivado Design Suite,» [En línea]. Available: http://www.xilinx.com/products/design-tools/vivado. [Último acceso: Marzo 2014].

[18] García. (Enero 2010). Procesador Inteligente de Eventos (IEP) con OpenESB. [En línea]. Disponible:http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=intelligent-event-processor-openesb

[19] J. Herrera, J. García, X. Perramon, "Aspectos avanzados de seguridad en redes: Mecanismos para la detección de ataques e intrusiones,".

[20] ActiBVA. (Agosto 2011). ¿Qué es el Algorithmic Trading?. [En línea]. Disponible: http://www.actibva.com/magazine/mercados-financieros/que-es-el-algorithmic-trading

[21] R. Ranjan. (2009). Algorithmic Trading - an Introduction. [En línea]. Disponible: https://sites.google.com/site/rajeevranjansingh/facts-and-figures-need-for-speed

[22] The Economist. (Junio 2007).Algorithmic trading: Ahead of the tape. [En línea]. Disponible: http://www.economist.com/node/9370718

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016

[23] Edosoft Factory S.L. (Julio 2008). Robobroker. [En línea]. Disponible: http://www.edosoftfactory.com/images/stories/dossier_rb.pdf
[24] http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

© Del documento, de los autores. Digitalización realizada por ULPGC. Biblioteca universitaria, 2016