METODOLOGÍA DE SÍNTESIS PARA USO DE BLOQUES DSP CON HDL SOBRE FPGAS

ENRIQUE DE LUCAS CASAMAYOR

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Máster en Ingeniería de Computadores

8 de Septiembre de 2011

Director:

Marcos Sánchez-Élez Martín

Autorización de Difusión

ENRIQUE DE LUCAS CASAMAYOR

8 de Septiembre de 2011

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Metodología de síntesis para uso de bloques DSP con HDL sobre FPGAs", realizado durante el curso académico 2010-2011 bajo la dirección de Marcos Sánchez-Élez Martín en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

En el presente trabajo se propone una metodología para sintetizar código en HDL de tal manera que se haga uso de los bloques DSP48E que aparecen en la familia de FPGAs Virtex 5 de Xilinx. Para conseguirlo se modifica el código HDL original para que la herramienta de síntesis sea capaz de reconocer la parte del código que debe implementarse en los DSPs. En primer lugar se ha intentado conseguir el objetivo empleando construcciones de código HDL con las que XST, herramienta de síntesis de Xilinx, infiere los DSPs. Ante la imposibilidad de obtener ciertas configuraciones específicas para los DSPs se plantea la posibilidad de utilizar la plantilla de macro específica DSP48E, que permite instanciar directamente dichos bloques. Para ello es necesaria una metodología que permite sustituir las operaciones aritmeticológicas más comunes por sus equivalentes mapeadas en un bloque DSP48E. En dicha metodología se proponen transformaciones de código que mantienen la funcionalidad original del diseño y limitan el uso de bloques DSP48E. Los resultados experimentales muestran que los diseños obtenidos con XST al aplicar la metodología utilizan un número de DSPs inferior que el obtenido infiriendo automáticamente los DSP con XST, consiguiéndose además una disminución del área y un aumento de la frecuencia del diseño.

Palabras clave

FPGAs, Compilación Hardware, DSPs, Lenguaje de Descripción Hardware (HDL), Síntesis de Alto Nivel

Abstract

This work proposes a methodology to synthesize HDL code in such a way that makes use of the DSP48E blocks presented in the Xilinx Virtex 5 FPGA family. The original HDL code is modified in order to achieve that the synthesis tool is able to recognize the code that must be implemented in the DSP blocks. First we have tried to achieve the objective using HDL code constructs that would infer DSP blocks, directly with the Xilinx Synthesis Tool (XST) . Since it is unable to obtain certain specific settings for the DSP, raises the possibility of using the DSP48E specific macro template, which allows directly instantiate these blocks. This requires a methodology to replace the most common arithmetic operations to the equivalents in the DSP48. In the methodology proposed the code transformations done maintain the original functionality of the design and limit the use of DSP48E blocks. Experimental results show that the designs obtained by applying the methodology within XST use a lower number of DSPs that those obtained automatically by XST. Moreover, in these designs there is a decrease in the area and an increase in the frequency.

Keywords

FPGAs, Hardware Compilation, DSPs, Hardware Description Language (HDL), High Level Synthesis

Índice de contenidos

Capítulo 1

Introducción

En este capítulo se presenta un breve repaso de las demandas actuales de computación y se realiza una introducción a los dispositivos de hardware reconfigurable denominados FPGA (Field Programmable Gate Array). Se comentan las características típicas de un DSP (Digital Signal Processor) y se comparan con las características de los bloques DSP contenidos en las últimas familias de Virtex, un FPGA de Xilinx. Una vez conocidas las características de las FPGAs y los DSPs se introduce el problema que se quiere solucionar mediante la realización de este trabajo fin de master.

1.1 Demandas Actuales en Computación

Podemos constatar la gran inversión que la industria tecnológica ha realizado en los últimos 10 años en el desarrollo de sistemas cada vez más complejos que se ajusten a las demandas de las nuevas aplicaciones. Y todo ello bajo la presión de un mercado muy competitivo que se mueve también a gran velocidad. Esta marcada tendencia se traduce en que los sistemas deben ofrecer cada vez mayores velocidades de proceso de la información, menores consumos y sin aumentar, o preferiblemente disminuir, su tamaño, precio y peso. Por esta razón han surgido numerosas líneas de investigación en bajo consumo, arquitecturas especializadas, nuevas tecnologías de fabricación... Y es dentro de este contexto donde encontramos también numerosas líneas de investigación relacionadas con el uso de dispositivos de Hardware Reconfigurable y la adición de nuevos módulos complejos a su estructura original para satisfacer las necesidades de las nuevas aplicaciones.

Esta apuesta por el Hardware Reconfigurable está fundamentada en el análisis de la tendencia de los sistemas computacionales actuales, como ha reflejado [1] en su trabajo, en el que expone su convicción de que las FPGAs terminarán siendo las grandes competidoras de las CPUs. En esta línea, [2] hace un análisis del rendimiento pico de FPGAs y CPUs y sostiene que el de las FPGAs puede llegar a ser superiores. Otros autores destacan la importancia del uso de FPGAs como piezas fundamentales integradas en sistemas tradicionales, como por ejemplo [3], para poder así aprovechar las ventajas computacionales de estos dispositivos sin perder las ya de sobra conocidas de las CPUs.

El proceso de diseño con diferentes tecnologías HW es conocido como compilación hardware. El diseño con FPGAs se enmarca dentro de este contexto. Un algoritmo o tarea debe ser descrito en un lenguaje de alto nivel que esté directamente relacionado con elementos HW para su fácil traducción a un circuito.

En los últimos 15 años se ha popularizado y extendido el uso de los llamados lenguajes de descripción HW (HDL, Hardware Description Language), entre los cuales destacan Verilog y VHDL.

Las etapas de diseño con FPGA se pueden resumir en las siguientes cinco:

i) El primer paso para compilar una tarea a HW es describir el algoritmo en alguno de los HDLs disponibles, VHDL en este trabajo.

ii) El siguiente paso es determinar qué tipo de bloques HW son necesarios y cómo están conectados entre sí.

iii) Asignar bloques concretos de la FPGA y rutado de señales entre ellos.

iv) En el cuarto paso del proceso se obtiene el mapa de bits de configuración necesario para que los elementos de la FPGA utilizados en el diseño del circuito realicen cada uno la funcionalidad necesaria.

v) Por último, es necesario escribir dicho mapa de bits en la memoria de configuración del dispositivo.

En la actualidad existen herramientas de compilación HW que traducen de forma automática un algoritmo descrito en VHDL a la arquitectura de FPGA que se esté utilizando, lo que da como resultado un bajo tiempo de compilación HW junto a un aceptable uso de recursos del dispositivo. Estás herramientas están bastante *optimizadas* en la utilización de las celdas

básicas de la FPGA, sin embargo, obtienen peores resultados en la utilización de los bloques específicos que se han añadido a las FPGAs para mejorar su rendimiento de cara al mercado. Es por lo tanto, en este nicho de investigación, donde se encuadra el trabajo de este proyecto de fin de máster.

1.2 Field Programmable Gate Arrays (FPGAs)

Una FPGA es un dispositivo hardware genérico que permite su configuración para obtener un diseño hardware que realice una tarea específica. Éstos se caracterizan por disponer de una serie de elementos básicos configurables, generalmente dispuestos en filas y columnas, que se pueden programar y combinar para realizar funciones específicas con más eficacia que un dispositivo de propósito general. Estos elementos se configuran a través de algún tipo de mecanismo físico cuya disposición determina la función que realizan. Dicha función puede cambiarse de manera sencilla y rápida, tantas veces como sea necesario. Los elementos reconfigurables se conectan entre sí y a los pines de entrada y salida del dispositivo a través de una red de conexiones, también reconfigurables. Tenemos, por tanto, tres tipos de celdas básicas en una FPGA:

- **CLBs**: configurable logic block, bloque de lógica configurable. Junto con otros bloques CLB implementa el diseño hardware.
- **IOBs**: in out block, bloque de entrada salida. Comunican señales lógicas del diseño con pines externos del circuito integrado.
- **Bloques** de interconexión de celdas. Permiten interconectar los CLBs entre sí y con los IOBs.

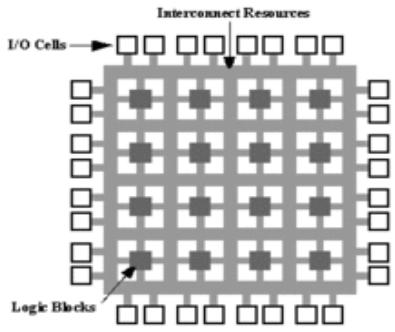


Figura 1-1. Estructura de una FPGA con matriz simétrica

La configuración se basa en distintas tecnologías aunque todas ellas coinciden en que lo que se configura es la matriz de conexiones además del contenido de cada CLB. Podemos hacer una diferenciación entre FPGAs **volátiles** y no **volátiles**. Entre las volátiles tenemos basadas en **SRAM** y entre las no volátiles contamos con FPGAs basadas en tecnología de **fusibles**, **antifusibles** o **EPROM**. En nuestro caso nos vamos a centrar en las FPGAs basadas en SRAM debido a que estas FPGAs suelen ser las más idóneas para investigación y desarrollo ya que pueden reconfigurarse tantas veces como permita la SRAM.

1.2.1 CLB: Configurable Logic Block (Virtex 5)

El CLB varía su estructura y nombre según la familia y la empresa que fabrique la FPGA pero suelen contar con recursos parecidos. Aquí mostramos la arquitectura de los CLBs de la familia Virtex 5 de Xilinx.

El CLB es el bloque principal que nos permite implementar tanto hardware combinacional como secuencial. Como vemos en [4] en cada CLB tenemos dos **slices** que no tienen comunicación entre ellas pero si tienen señales para acarreos de entrada y salida, usado para implementar sumadores y multiplicadores. Cada slice se corresponde con una columna en el CLB y está conectado a una matriz de conmutación para acceder a los bloques de interconexión de celdas (ver figura 1-2).

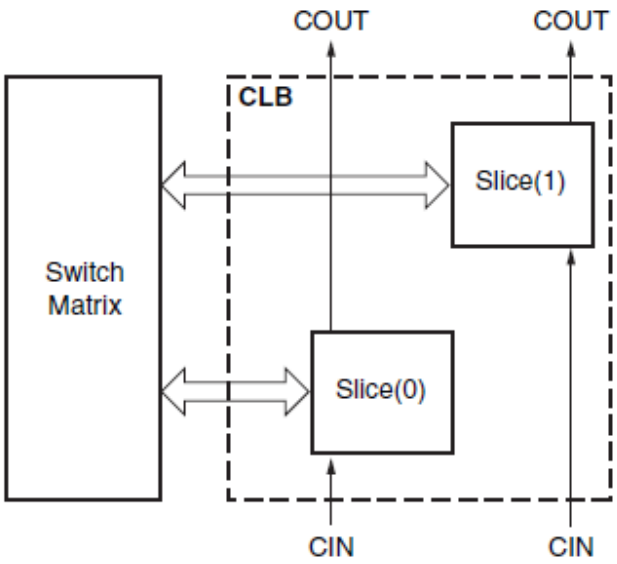


Figura 1-2. Slices dentro de un CLB

En el CLB podemos encontrar dos tipos de slice, **SLICEL** y **SLICEM**, teniendo estas últimas hardware adicional. El **SLICEL** contiene:

- Cuatro LUTs (Look Up Tables)
- Cuatro registros.
- Multiplexores.
- Lógica de acarreo

Todo ello da soporte para implementar funciones lógicas, aritméticas y ROMs (ver figura 1-3).

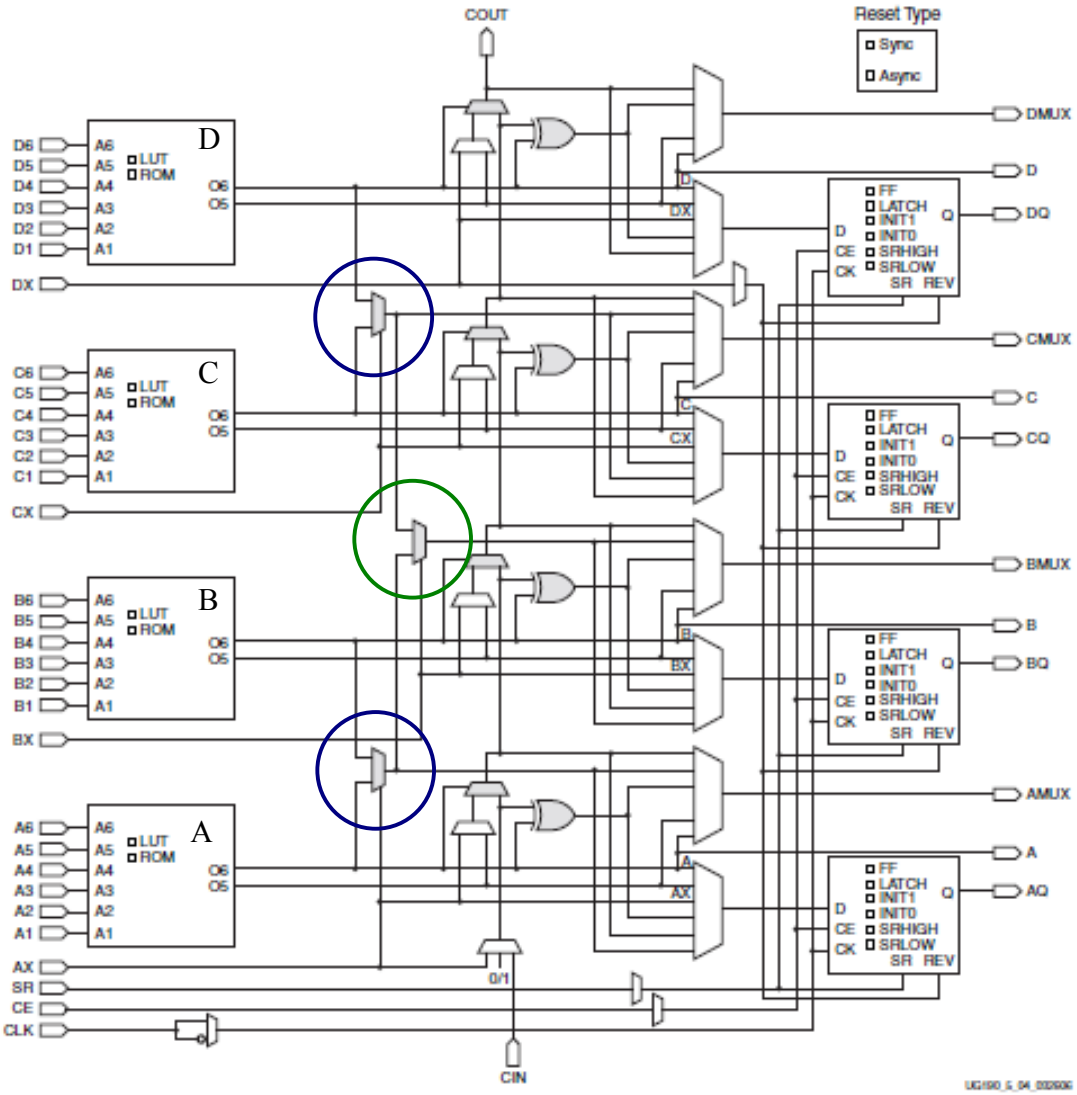


Figura 1-3. Estructura del SLICEL.

El **SLICEM** además de lo anterior incluye recursos para guardar datos mediante ram distribuida y para realizar desplazamientos como un registro de 32 bit (ver figura 1-4).

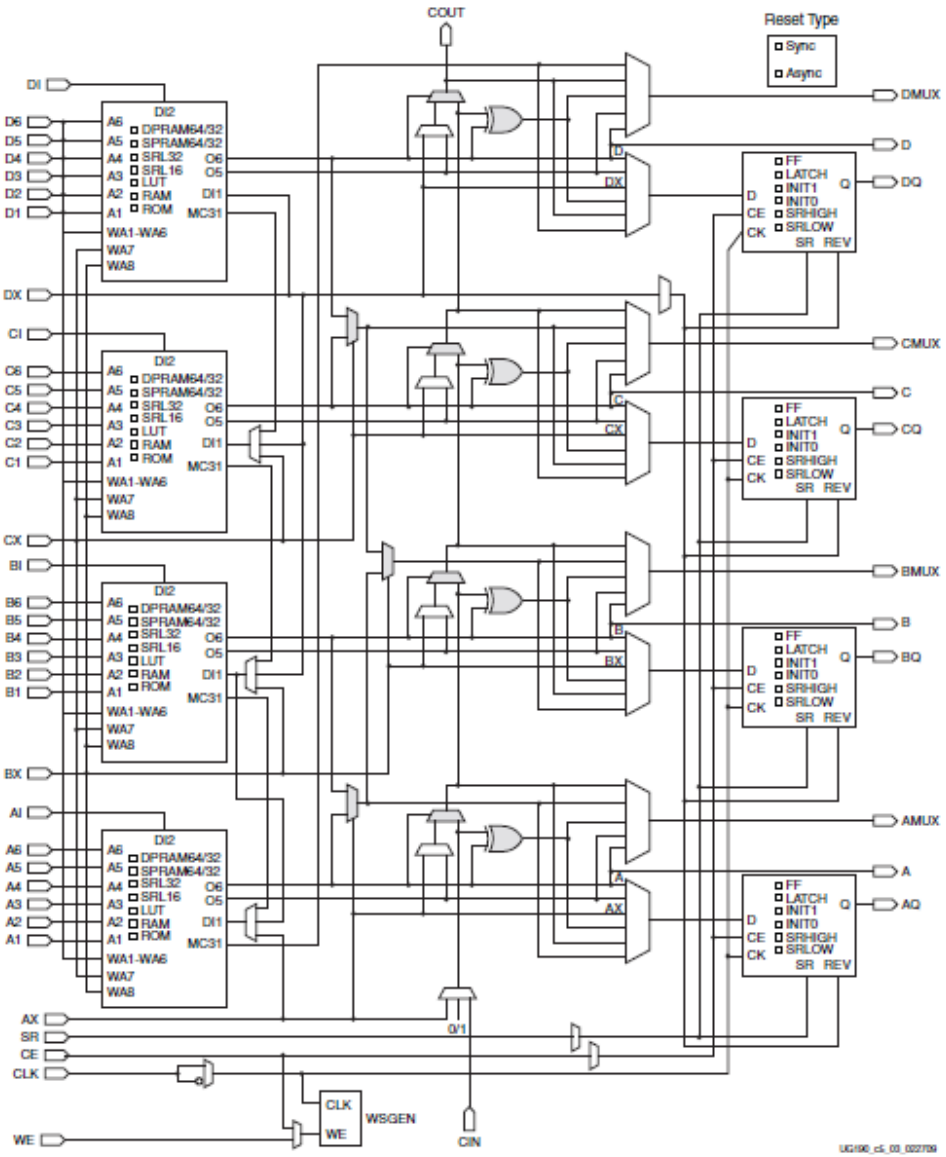


Figura 1-4. Estructura del SLICEM.

1.2.2 LUT (Look Up Table)

Cada LUT es el recurso básico que nos permite implementar una tabla de verdad de seis entradas y una salida o dos tablas de verdad de cinco entradas (siempre que se compartan señales de entrada) y una salida cada una (ver figura 1-5).

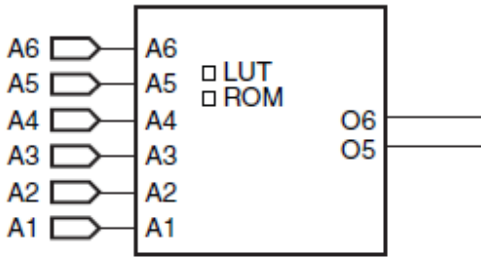


Figura 1-5. Entradas y salidas de una LUT.

Además los multiplexores señalados en azul en la figura 3 nos permiten implementar tablas de verdad de 7 entradas (con LUTs A y B o C y D) y el multiplexor señalado en verde nos permite implementar una tabla de verdad de 8 entradas a partir de la salida de los dos multiplexores anteriores. Si se quieren implementar tablas de verdad con más entradas deberán usarse varios slices.

1.2.3 Elementos de almacenamiento

Pueden ser configurados como un flip-flop D o cómo un latch D y pueden implementarse versiones con o sin CE (Clock Enable) y RS (Reset) .

1.2.4 Ram distribuida (sólo en SLICEM)

Las LUTs de cada slice pueden configurarse para funcionar como elementos de almacenamiento ram síncronos. Estos elementos tienen escritura síncrona y lectura asíncrona aunque puede implementarse una lectura síncrona usando un flip flop. Para la escritura se ha de poner la señal WE (Write Enable) a Uno.

Las LUTs pueden combinarse de diferentes maneras para implementar las siguientes rams:

- Single-Port 32 x 1-bit RAM
- Dual-Port 32 x 1-bit RAM
- Quad-Port 32 x 2-bit RAM
- Simple Dual-Port 32 x 6-bit RAM
- Single-Port 64 x 1-bit RAM
- Dual-Port 64 x 1-bit RAM

- Quad-Port 64 x 1-bit RAM
- Simple Dual-Port 64 x 3-bit RAM
- Single-Port 128 x 1-bit RAM
- Dual-Port 128 x 1-bit RAM
- Single-Port 256 x 1-bit RAM

1.2.5 Registros de desplazamiento (sólo en SLICEM)

El SLICEM puede configurarse para actuar como un registro de desplazamiento de 32 bit sin usar los flip-flop disponibles en el slice. De esta manera puede introducirse un delay en los datos en serie desde 1 a 32 ciclos con un sólo LUT y de 1 a 128 ciclos con un slice consiguiendo el comportamiento de un registro de desplazamiento de 1 a 128 bits.

1.2.6 Bloques reconfigurables complejos

Se pueden usar solo CLBs para implementar un diseño digital siempre que quepa en la FPGA. Sin embargo teniendo en cuenta que lo más eficiente sería tener un ASIC que implementara el mismo diseño es lógico incluir bloques más complejos producidos como un ASIC. Con la intención de mejorar el rendimiento para ciertas tareas aparece el uso de ese tipo de bloques más complejos que realizan dichas tareas más eficientemente en las FPGA que si se hicieran con CLBs. De este tipo de bloques han aparecido básicamente memorias (block ram), multiplicadores y DSPs.

Por ejemplo, cada block ram en la familia Virtex 5 pueden almacenar hasta 36 Kbit de memoria. Las block ram poseen escritura y lectura síncronas y pueden combinarse y utilizarse en diferentes modos formando memorias de 36 Kb, dos de 18 Kb, 16 K x 2, 8 K x 4, 4 K x 9, 2 K x 18 o 1 K x 36. Cada bloque de 18 Kb puede convertirse en memorias 16K x 1, 8K x 2, 4K x 4, 2K x 9 o 1 K x 18. Respecto a los multiplicadores tenemos por ejemplo el bloque Mult18x18 de la familia Virtex 2 que permite realizar multiplicaciones de números de 18 bits y estaba dirigido hacia aplicaciones DSP. Más adelante se ha decidido incluir bloques DSP completos, de uno de los cuales se hablará en profundidad más adelante.

1.3 DSPs: ¿Qué son y por qué aparecen en la FPGAs?

1.3.1 Historia de los DSP

Como vemos en [5] y en [6] un Digital Signal Processor (DSP) es un procesador cuyas instrucciones y arquitectura están desarrolladas específicamente para ejecutar aplicaciones de procesado de señales digitales. Este tipo de aplicaciones suelen trabajar con señales en tiempo real por lo que un DSP debe procesar las muestras de dichas señales a la misma velocidad que le llegan. Sin embargo algunas aplicaciones DSP no requieren una ejecución en tiempo real, por lo que no se requiere un DSP para su ejecución. A mayor frecuencia del DSP podemos hacer más cosas en el mismo tiempo. Por ejemplo, en 1980 aparece DSP-1, de los laboratorios Bell, DSP1 tenía una frecuencia de 5 MHz frente a los 550 MHz que alcanza el DSP programable DSP48E de Xilinx. Esta mejora del rendimiento nos ha permitido:

- Implementar algoritmos más complicados (G.729A frente a G.711 PCM) o añadir más canales a un mismo algoritmo (aplicaciones VoIP con más canales) manteniendo el mismo flujo de datos.
- Implementar algoritmos de complejidad similar con un flujo de datos mayor (MPEG-2 a mayor resolución).

Existen DSPs no programables y DSPs programables por el usuario. La primera generación de DSPs programables podía ejecutar un solo canal del codec ADPCM/DLQ (adaptive differential pulse code modulation/dynamic locking quantizer) a 32-Kbps mientras que un DSP segmentado diseñado para este codec podía ejecutar ocho canales usando la misma tecnología. Esto es así ya que un DSP programable requiere una etapa adicional en la ejecución de una instrucción, la etapa de configuración, lo que cuesta área de silicio y aumenta el consumo. El problema con los DSPs no programables era que se debía diseñar uno para cada aplicación y a medida que el coste de estos diseños fue aumentando el uso de DSPs programables fue ganando fuerza. Además las técnicas Very Large Scale Integration (VLSI) avanzadas permiten a los DSPs programables reducir el coste y el consumo de un algoritmo específico con un flujo de datos determinado. Por ello durante los últimos 30 años el uso de DSPs programables ha ido creciendo hasta ser la opción dominante para implementar aplicaciones DSP.

En 1984 aparece TM320C54XX de Texas Instruments. Se trata e un DSP con un conjunto de instrucciones complejo (CISC) de 16 bits. Para mejorar su rendimiento una opción era incluir nuevas instrucciones, pero contaba con 130 instrucciones e incluir nuevas no era tarea fácil. Además el rápido desarrollo en la escala de integración requería usar los transistores extra para aumentar el rendimiento. Para ello, una opción es aumentar el número de etapas de pipeline, lo que no era factible dada la complejidad de un repertorio de instrucciones CISC. La implementación de VLIW tampoco era una opción, ya que con CISC es difícil identificar instrucciones que puedan ejecutarse en paralelo. Todo esto unido a unos compiladores que no se desarrollaron como debían, ya que no daba tiempo debido al rápido desarrollo de la tecnología, hizo que el C54 quedara obsoleto. Mientras el C54, habiendo sido diseñado en full custom 0.16µm, se ejecutaba a una frecuencia de 160 MHz, el StrongARM RISC alcanzó 600 MHz en 0.18µm. Por ello Texas Instruments sacó el TM320C62XX con una frecuencia de 300 MHz y con arquitectura RISC WLIW de 8 instrucciones en paralelo de 32 bits en la que decidieron no incluir una instrucción MACC. Este tipo de DSP es denominado como RISC-DSP en [5].

Con el desarrollo de los procesadores RISC-DSP algunas de las aplicaciones que en los 80 solo podían ser implementadas en DSPs para alcanzar unos mínimos de rendimiento hoy en día pueden ser implementadas sobre un solo procesador RISC-DSP a 200 MHz. Además el uso de instrucciones que toman los registros fuente y destino de registros de propósito general hace que los compiladores sean más fáciles de desarrollar y tengan más éxito que en anteriores generaciones.

Dentro del grupo de RISC-DSP tenemos Siemens Tricore, Intel MSA (Micro Signal Architecture, desarrollado por Intel y Analog Devices) y Philips Trimedia (VLIW). Además por otro lado los fabricantes de procesadores RISC han tenido en cuenta los requerimientos del procesado digital de señales. ARM Enhanced Extensions [7], Hitachi con SH-DSP e IBM y Motorola con Book E, una extensión de PowerPC.

Según [5] a lo largo del tiempo puede apreciarse como los procesadores RISC y los DSP convergen hacia un modelo RISC-DSP programable. Llevando más allá este modelo de DSP programable existen soluciones basadas en FPGAs que hacen uso de bloques reconfigurables y de bloques DSP programables. Actualmente Altera y Xilinx, proporcionan soluciones DSP reconfigurables que permiten un gran paralelismo de datos y pueden realizar sus operaciones ciclo a ciclo.

1.3.2 Estructura típica de un DSP

Un procesador DSP va a tener como entrada una señal digitalizada por un ADC (Analog Digital Converter), ya sea un ADC del propio sistema u otro, y va a producir otra señal digital que se dirigirá a un DAC (Digital Analog Converter) o será usada tal cual por otro sistema(ver figura 1-6).

Figura 1-6. Sistema DSP típico.

Como se dice en [6] la mayoría de los procesadores DSP están optimizados para ejecutar operaciones del tipo Multiplier Accumulator (MAC). Por ejemplo el filtro FIR (Finite Impulse Filtering) puede ser expresado como

$$y(n) = \sum_{i=0}^{L-1} b_i x(n-i)$$

siendo $b_0,...,b_{L-1}$ coeficientes del filtro, x(n), ... ,x(n-L+1) muestras y L la longitud del filtro. De esta manera será necesario para aplicar el filtro un hardware que vaya calculando productos y los sume, es decir, operaciones de tipo MAC. En concreto un típico procesador DSP contiene las siguientes unidades funcionales y características:

- **Multiplicador acumulador** (MAC)
- **Unidad Aritmético Lógica**: Suma, resta, and, or, not.
- **Desplazadores**: Para escalar los datos antes o después de una operación sobre ellos.
- **Etapas**: Suele aparecer segmentado para disminuir el tiempo de ciclo y aprovechar mejor los recursos del sistema y recibir una nueva instrucción ciclo a ciclo (IPC cercano a 1, Instrucciones Por Ciclo).

- **SIMD**: Suelen tener un modo Single Instruction Multiple Data.
- **Arquitectura Harvard**: Al contrario que la arquitectura Von Neumann que almacena datos e instrucciones en la misma memoria la arquitectura Harvard tiene una memoria de instrucciones y una (o dos) de datos (ver figura 1-7). De esta manera se pueden tener los datos y las instrucciones a ejecutar en el mismo ciclo.

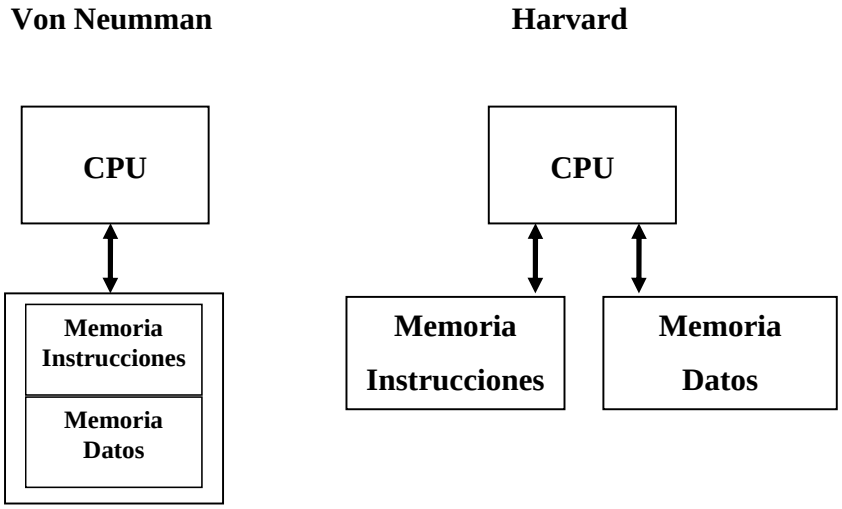


Figura 1-7. Arquitecturas von Neumman y Harvard.

Un DSP por lo tanto ejecutará operaciones típicas de una ALU además de multiplicaciones y desplazamientos con un IPC (instrucciones por ciclo) cercano a 1.

Finalmente para dar una motivación de por qué aparecen los DSPs en las FPGAs es necesaria una breve explicación de su diseño inicial y hacia dónde ha ido desde entonces. Las FPGAs surgieron de la filosofía de implementar hardware a partir de componentes básicos apareciendo en un primer momento las arquitecturas de grano fino (ver figura 1-8).

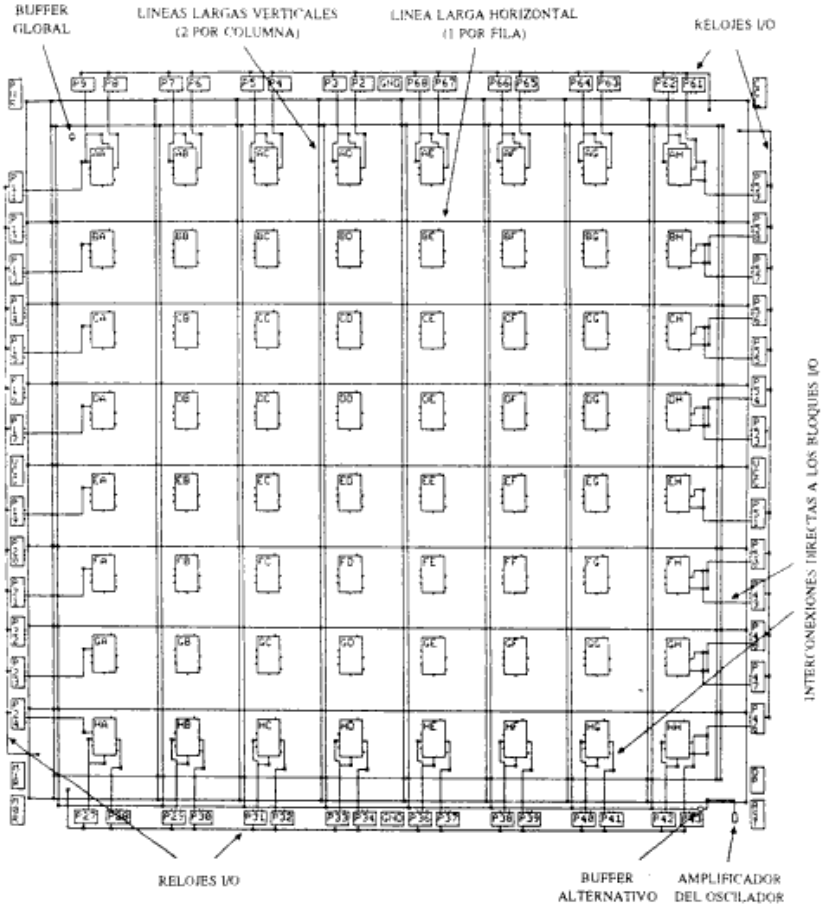


Figura 1-8. Arquitectura XC2064, 1985 .

Sin embargo el diseño inicial de un dispositivo de componentes básicos que implemente todo se ha ido enriquecido por un diseño heterogéneo en menor o mayor medida en el que se incluyen nuevos bloques para realizar tareas específicas (Blockram, Multiplicadores, CLBs modificados para implementar sumadores más eficientes, DSPs). Esto es así dado que un ASIC específico es más compacto y rápido que un diseño con CLBs por lo que paulatinamente se decide ir incluyendo dichos módulos ya optimizados y así realizar las diferentes tareas de manera más eficiente. Los bloques DSP48A, DSP48 y DSP48E que aparecen en las familias Spartan 3A DSP, Virtex 4 y Virtex 5 respectivamente son un ejemplo de dichos módulos. En particular describimos brevemente el bloque DSP48E (ver figura 1-9).

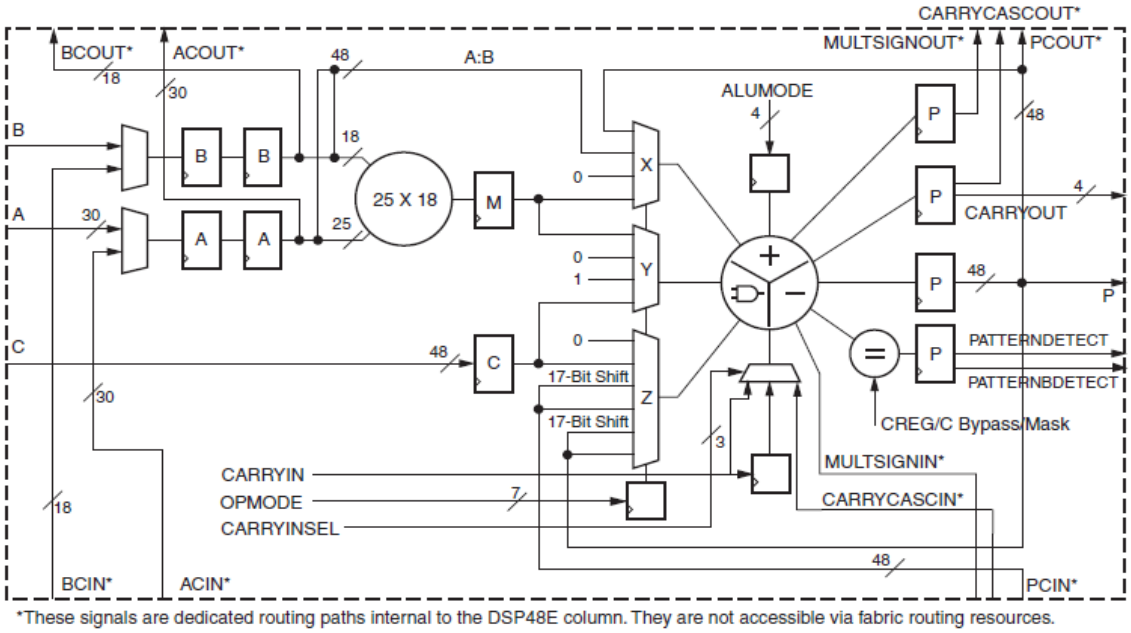


Figura 1-9. Virtex 5 FPGA DSP48E slice.

Como podemos ver en la figura 1-9 un DSP48E cuenta con multiplicador 25x18, unidad aritmético lógica (ALU) de 48 bits, desplazadores, comparador y registros con los que se puede implementar un pipeline. Cuenta con 3 entradas de datos (A de 30 bits, B de 18 y C de 48) pudiendo tomarse A y B como entrada directa o en cascada (procedente de otro DSP48E) y una salida directa (P de 48 bits) que también puede mandarse a la ALU del DSP48E siguiente.

La ALU puede funcionar en modo SIMD TWO24 y FOUR12 (ver figura 1-10), realizándose simultáneamente 2 operaciones de 24 bits o 4 de 12.

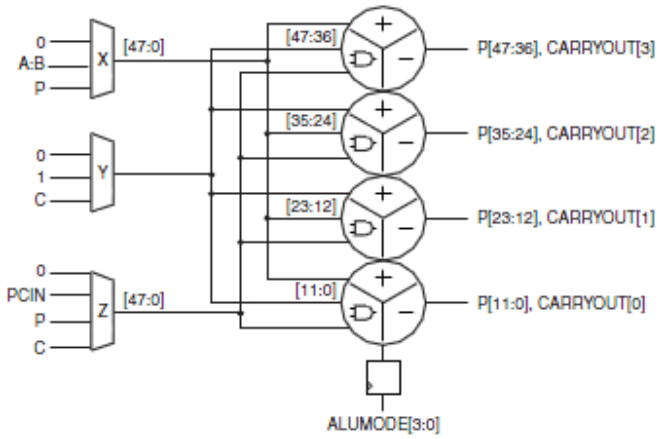


Figura 1-10. Modo SIMD FOUR12.

Dadas las características del DSP48E vemos que se ajusta en gran medida a las características que suele tener un DSP. Respecto a la arquitectura Harvard podemos destacar que si bien el DSP48E no posee una memoria de datos ni de instrucciones en si mismo pero si cumple el requisito de poder recibir ciclo a ciclo los datos y la instrucción a ejecutar. Finalmente en lo relativo a las etapas de pipeline hay que señalar que el DSP48E no es segmentado ya que no es posible lanzar ciclo a ciclo cualquier tipo de instrucción soportada sin esperar a que acabe la ejecución de la anterior. Por ejemplo, no podemos realizar una multiplicación seguida de una resta. Sin embargo las operaciones del mismo tipo si pueden ser lanzadas ciclo a ciclo sin generar conflictos de recursos.

1.4 Objetivos de este trabajo

En un contexto de investigación que tiende a desarrollar metodologías para automatizar el proceso de compilación hardware comentado en la sección 1.1 y por otro lado con la aparición de manera generalizada de nuevos bloques DSP en las FPGAs se hace necesaria una herramienta que pueda el aprovechar automáticamente estos nuevos bloques.

En este caso centramos el estudio en el aprovechamiento de bloques DSP48E a partir de VHDL (behavioral). En un primer momento se analiza el estilo de código VHDL necesario para que la herramienta ISE Design Suite 12.1 (ISE12_1) sintetice los diseños utilizando los DSPs con diferentes configuraciones. Con esta técnica no se ha conseguido siempre la configuración deseada para los bloques DSP en los diseños estudiados por lo que se han modificado dichos diseños instanciando de manera directa los DSP. Esta opción tiene el inconveniente de que ha de realizarse manualmente. He aquí del estudio en el que se basa este trabajo de fin de máster. El objetivo es por lo tanto conseguir una metodología automatizable según la cual se recibe un código VHDL como entrada y se produce como salida un código VHDL que usa los DSPs de la FPGA (ver figura 1-11).

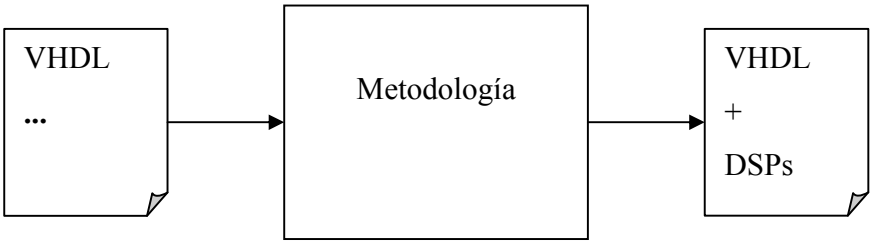


Figura 1-11. Entrada y salida de la metodología propuesta.

Capítulo 2

Estado del Arte

En el presente capítulo se introducen los conceptos de nivel de abstracción y enfoque de diseño de un sistema además de definirse la síntesis de alto nivel y hacer un breve repaso sobre algunas de las herramientas y lenguajes de alto nivel utilizados actualmente en síntesis. Se introduce el lenguaje VHDL, se repasan diferentes opciones para trabajar con DSPs en FPGAs y finalmente se da una motivación del estudio realizado en este trabajo de fin de master.

2.1 Introducción a la Síntesis de Alto Nivel

El rápido desarrollo de la tecnología electrónica ha permitido ir creando circuitos integrados con una escala de integración cada vez mayor, esto es, con un mayor número de transistores por chip (ver figura 2-1).

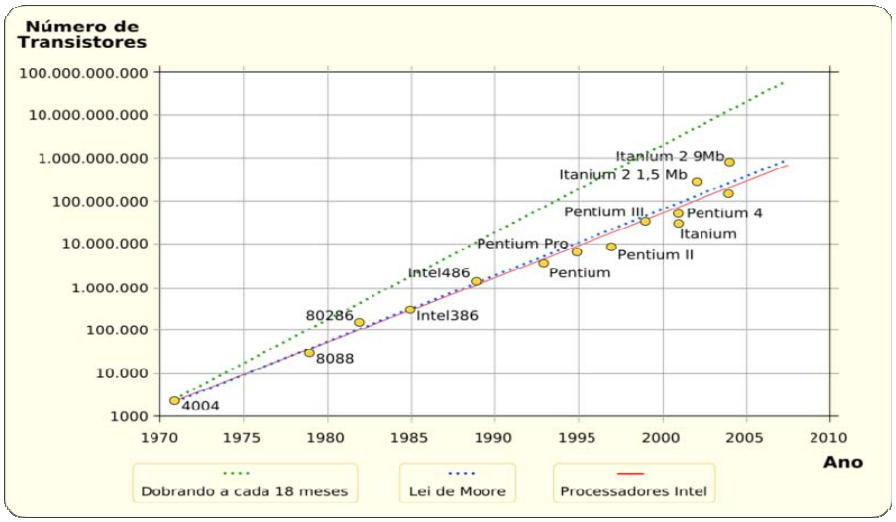


Figura 2-1. Datos reales VS Ley de Moore.

A medida que la escala de integración aumenta también lo hace la complejidad del diseño del chip. Si el procesador 4004 de Intel lanzado en 1971 contaba con 2.300 transistores el procesador Core I5-661 del año 2010 cuenta con 559 millones de transistores, aproximadamente unas 243.000 veces más cantidad. Esto implicaría un aumento tan grande en el número de transistores por ingeniero que haría intratable el diseño de tales chips. Una de las consecuencias es que a medida que ha ido creciendo la complejidad en el diseño han ido desarrollándose diferentes metodologías, diferentes niveles de abstracción y herramientas Computer Aided Design (CAD) que acortan el ciclo de diseño. A finales de los 80 surgen lenguajes de descripción de hardware (HDLs) como Verilog (1986) y VHDL (1987) que también servirán como entrada para las herramientas de síntesis lógica, en los 90 surgieron las primeras herramientas comerciales de síntesis de alto nivel (Synopsys behavioral compiler) y en los 2000 surge el concepto de nivel de sistema.

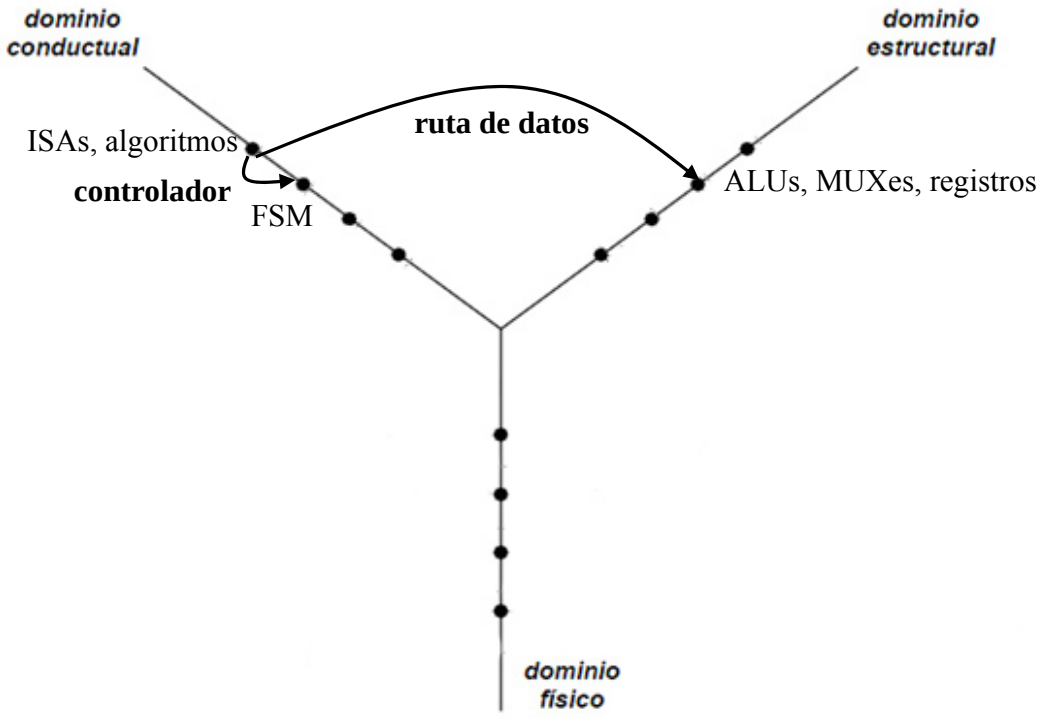


Figura 2-2. Síntesis de alto nivel.

De entre todas las definiciones asociadas al diseño automático de circuitos la más utilizada por las herramientas CAD que trabajan con lenguajes de descripción HW es la **síntesis.** Éste es un proceso que genera una descripción estructural (puertas, elementos HW, …) a partir

de una descripción conductual. Podemos entonces definir la **síntesis de alto nivel (HLS)** como la síntesis que se produce a partir de una serie de restricciones (latencia, tiempo de ciclo, número y tipo de recursos, etc) y una entrada en el nivel algorítmico del dominio conductual con la cual obtenemos una descripción conductual en el nivel de FSM (control) y una descripción estructural a nivel de ALUs, MUXes, registros (ruta de datos). Ver figura 2-2. Por tanto, el principal objetivo de la HLS será conseguir una implementación RTL a partir de una descripción en un lenguaje de alto nivel (HLL) y así reducir el tiempo de diseño y su verificación (ver figura 2-3).

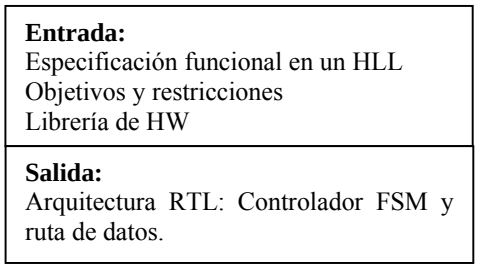


Figura 2-3. Entrada y salida para una herramienta de HLS.

Según [8] una herramienta de síntesis de alto nivel (HLS) usualmente transforma una especificación de alto nivel que no tiene en cuenta, o tiene en cuenta parcialmente, la temporización a una implementación que sí lo tiene en cuenta. Automáticamente o semiautomáticamente generan arquitecturas personalizadas que implementan eficientemente la especificación inicial. Además de la memoria y los protocolos de comunicación la arquitectura generada se describe en RTL y contiene la ruta de datos y el controlador. Toda herramienta de HLS ejecuta las siguientes tareas (ver figura 2-4):

1. **Compilar** la especificación: se obtiene una especificación que contiene el flujo de datos y de control (Control y Data Flow Graph) abstrayéndose del hardware. Se extrae el paralelismo implícito.
2. **Asignar Hardware (HW)**: determina los recursos hardware a utilizar necesarios para la descripción del sistema.
3. **Planificar** las operaciones en ciclos de reloj: se determina en que ciclo o paso de control de ejecuta cada una de las operaciones de la descripción.

4. **Ligar las operaciones** a unidades funcionales (FUs).
5. **Ligar las variables** a elementos de almacenamiento.
6. **Ligar las transferencias** a buses.
7. **Generar** la arquitectura RTL una vez según las decisiones tomadas en las tareas anteriores.

Las tareas de la dos a la seis son interdependientes por lo que idealmente deberían ser optimizadas en conjunto, no obstante debido al alto coste de cómputo estas tareas suelen realizarse secuencialmente, teniendo su orden de ejecución un alto impacto en la calidad del diseño.

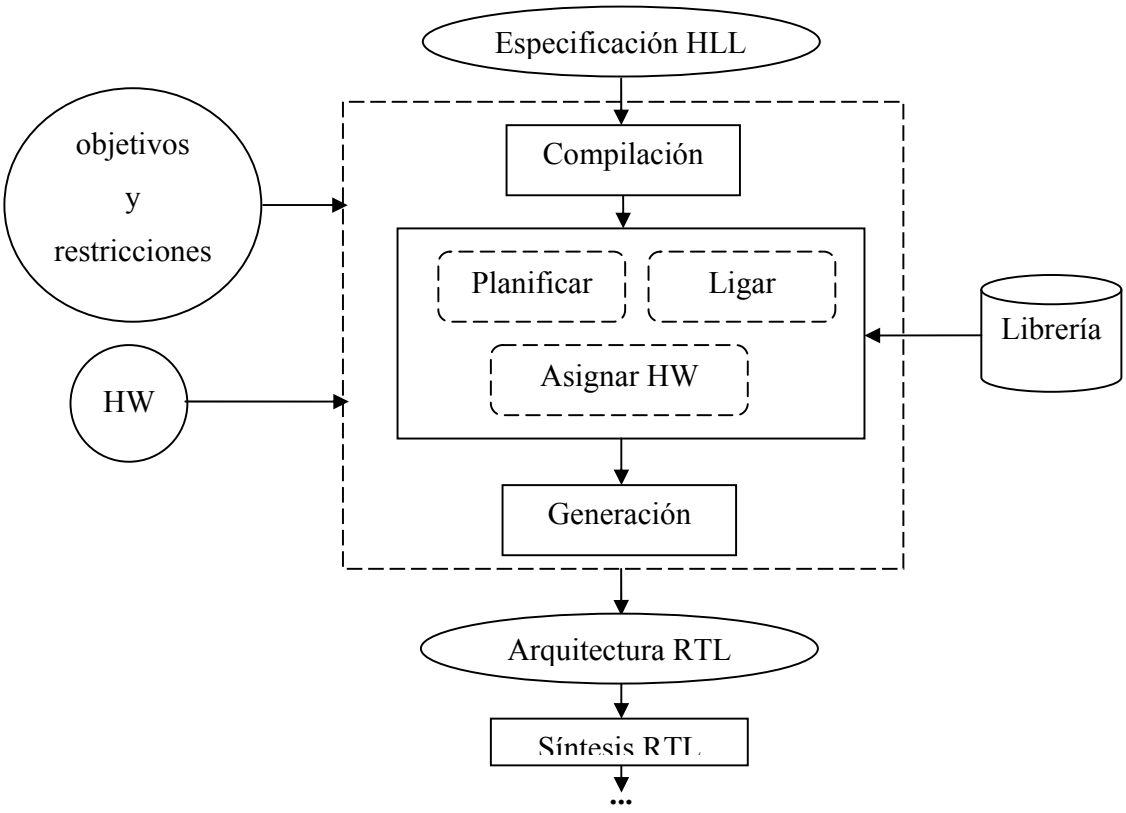


Figura 2-4. Flujo de síntesis de alto nivel.

2.2 Herramientas de Compilación Comerciales

Una herramienta HLS (High Level Synthesis) debe tener como entrada una especificación en un lenguaje de alto nivel y a partir de dicha especificación obtener una especificación a bajo nivel.

Actualmente algunas herramientas de HLS usan ANSI C, C++ o lenguajes basados en ellos (como Handel-C y SystemC). Algunas de estas herramientas son Catapult C (C++, SystemC) de Mentor, Cynthesizer de Forte (SystemC) y DK Design Suite también de Mentor (Handel-C, SystemC). Las herramientas Accel DSP de Xilinx y SimplifyDSP de Synopsys usan el lenguaje Mathwork's Matlab and Simulink. Ambas se basan en IPs para generar la implementación hardware.

A pesar de usar lenguajes de alto nivel la descripción de entrada en estas herramientas debe hacerse pensando en los resultados posibles en hardware para así conseguir mejores resultados en la síntesis. Por ejemplo si queremos tener una implementación eficiente de un bucle do-while con DK Design Suite y Handel-C haremos el contador del bucle en una instrucción paralela al bucle en si, ahorrando un ciclo por cada iteración (ver figura 2-8).

```
static unsigned 4 x = 15;
par
{
     do{
      //do something
     } while(x != 0);
     x--;
}
```

Figura 2-8. Ejemplo de declaración de bucle while en Handel-C.

2.2.1 Catapult C

Según [9] la descripción de entrada se realiza con ANSI C++ (se puede usar un compilador estándar de C++) o SystemC (librería de C++ que añade construcciones hardware) y un conjunto de directivas y se produce una implementación RTL optimizada para la tecnología destino (ver figura 2-9). Automatiza el paso del lenguaje de alto nivel a sistemas totalmente jerarquizados con bloques de control y unidades algorítmicas. Esta automatización elimina los errores típicos que se producen con sistemas manuales.

Además en Catapult C se ha unificado el flujo de modelado, síntesis, verificación de ASICs y FPGAs lo que permite explorar las diferentes opciones de la arquitectura. Tiene además optimizaciones automáticas para reducir el consumo de potencia y se dispone de total visibilidad y control del proceso de síntesis.

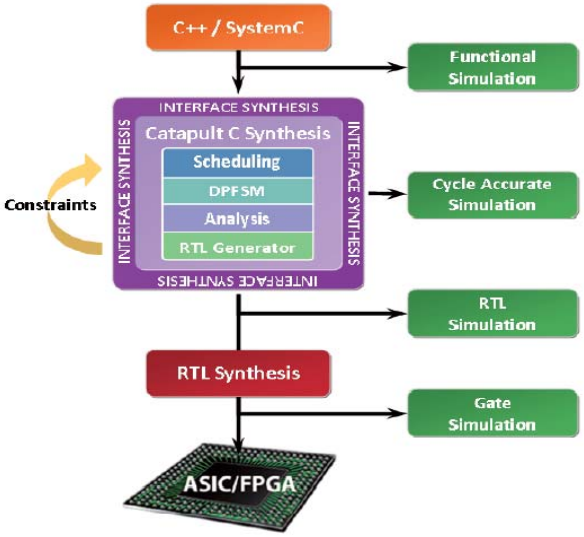


Figura 2-9. Flujo de diseño con Catapult C.

Una funcionalidad interesante es que se pueden usar los test utilizados en la simulación funcional para testear la implementación RTL y además se realiza una comparación entre los resultados de la simulación funcional y la simulación RTL. La síntesis además produce los archivos necesarios (scripts) para realizar una simulación en tiempo sobre el código funcional conservando el comportamiento de la implementación RTL.

2.2.2 Cynthesizer

Según [10] la descripción de entrada se realiza en SystemC, lo que permite utilizar directamente algoritmos escritos en C o C++ sin tener que portarlos. La salida es una especificación RTL en Verilog o VHDL que es usada por las herramientas líderes del mercado en síntesis RTL (ISE, Quartus). Una característica importante que aporta el uso de SystemC en la entrada de diseño es que se puede reutilizar el código anteriormente verificado sin hacer cambios, cosa que con diseños ciclo a ciclo con máquinas de estado en RTL no ocurre generalmente, pues una ligera modificación basta para tener que verificar de nuevo el diseño.

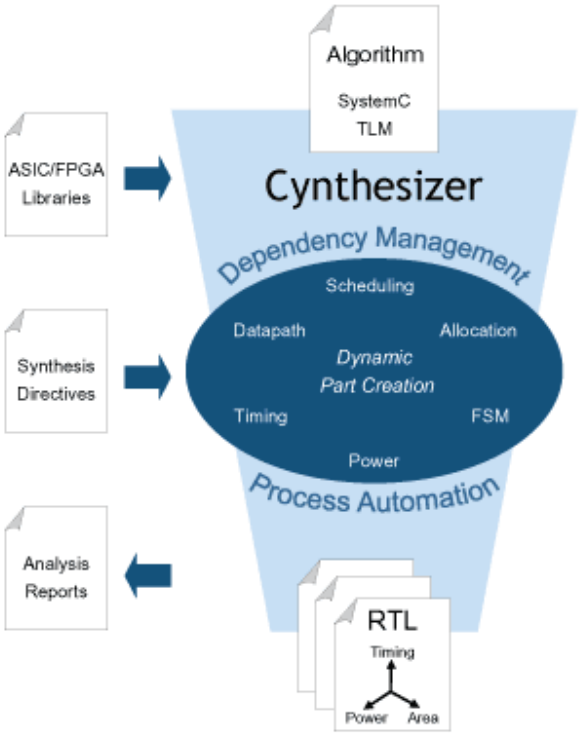


Figura 2-10. Flujo de diseño Cynthesizer.

El proceso de síntesis es dirigido y se hace uso de librerías ASIC o FPGA para producir diferentes implementaciones RTL optimizadas para área, velocidad o consumo. Las construcciones no soportadas para síntesis son emplazamiento dinámico (malloc, free, alloc, new, delete), aritmética de punteros y funciones virtuales.

2.2.3 DK Design Suite

DK Design Suite [11] posee una amplia comunidad de usuarios y más de 14 años de desarrollo comercial y al igual que las dos herramientas anteriores recibe un código en un lenguaje de alto nivel, en este caso Handel-C, y puede producir una salida en VHDL, Verilog o EDIF y directamente utilizar las herramientas de síntesis de Xilinx y Altera para obtener el diseño final.

Respecto a la entrada de diseño es una mala práctica escribir código pensando en términos de máquinas de estados ya que se suele obtener peores resultados que si programamos pensando en alto nivel. Una de las características de Handel-C nos la proporciona con las directivas seq{...} y par{...}. Todo código dentro de una directiva seq se ejecutará

secuencialmente mientras que el código dentro de una directiva par se ejecutará en paralelo (ver figura 2-9).

```
par
{
   seq
   {
   ...// secuencialmente
   }
   seq
   {
   ...// secuencialmente
   }
//Los dos seq se ejecutan
//secuencialmente en paralelo
}
```

Figura 2-9.Directivas seq y par de Handel-C.

Otra característica importante de DK Design Suite es la librería PixelStreams que permite un rápido prototipado de sistemas de vídeo e imagen. Esta librería se usa haciendo llamadas a las funciones de la librería.

Respecto a la verificación del sistema puede realizarse una simulación ciclo a ciclo. Además permite hacer profiling con estimaciones de tiempo y área para optimizar nuestro diseño.

2.3 Herramientas de Compilación Académicas

Antes de pasar a describir las características básicas de algunas herramientas de compilación hardware se hace un breve resumen de las transformaciones y las técnicas básicas de optimización que se efectúan en la compilación.

2.3.1 Transformaciones en las herramientas de compilación

Toda herramienta clásica de compilación realiza transformaciones automáticas de la descripción de entrada. Estas transformaciones normalmente implican un tiempo de ejecución menor pero por ejemplo también pueden ir en la dirección de obtener circuitos que consuman menos recursos, menos potencia u obtengan un área menor. A continuación hacemos un repaso de distintas técnicas de transformación de código mostradas en el capítulo 4 de [12].

Transformaciones a nivel de bit

Se trata de transformaciones utilizadas en compiladores de grano fino.

- **Bit-width narrowing:** El tamaño de las variables que normalmente usamos suele tener más bits de los necesarios lo que no afecta al rendimiento cuando se trata con procesadores o unidades reconfigurables de grano grueso. Sin embargo, en una arquitectura de grano fino eliminar estos bits extra puede suponer una reducción importante del tamaño y/o del delay de la implementación final. Esta técnica tiene especial importancia cuando se programa en lenguajes que no tienen soporte para definir variables de un ancho específico como suele pasar en los lenguajes software típicos.

- **Optimizaciones de bit:** Son aplicables en programas con un uso intensivo de manipulaciones a nivel de bit como son máscaras, desplazamientos constantes, concatenaciones. En estos casos no es necesario implementar puertas and, desplazadores ni nuevos registros respectivamente. Todas esas operaciones pueden realizarse con cables ahorrándonos las puertas y registros correspondientes.

- **Conversión de punto flotante a punto fijo:** El punto flotante es un estándar ampliamente usado para representar números reales en casi todos los ámbitos de la computación. Sin embargo su uso requiere un gasto bastante superior en puertas lógicas y en ciclos de ejecución que la representación en punto fijo. La representación en punto fijo se usa ampliamente en DSP y se implementa realizando desplazamientos en los datos de entrada y/o de salida de cada unidad funcional (de ahí los desplazadores del DSP48E). Sin embargo el proceso de pasar manualmente a punto fijo es laborioso y lleva tiempo. Por ello se han efectuado esfuerzos para automatizar este proceso usándose en muchos casos profiling para obtener información de la precisión requerida en cada caso. Es habitual que cuando la herramienta no puede traducir una variable pida ayuda al diseñador.

  Desde el punto de vista de arquitecturas de grano grueso minimizar el número de desplazamientos (se pueden eliminar desplazamientos cuando coincide el número de bits que representan la parte decimal) que se realizan puede ser un objetivo primario

ya que cada desplazamiento puede implicar un delay en las arquitecturas, sobre todo si se reutiliza la misma unidad funcional (FU) para cálculos consecutivos. Sin embargo desde el punto de vista de las arquitecturas de grano fino estos desplazamientos se implementan como cables de modo que no añaden ningún coste adicional.

- **Punto flotante no estándar:** La utilización de una implementación de punto flotante no estándar que ajuste los bits de exponente y mantisa a lo que se precisa para una aplicación específica permite obtener una solución intermedia entre el punto fijo y el punto flotante estándar. No es necesario traducir toda la aplicación de un formato a otro y puede disminuir el número de ciclos y e hardware necesario para implementarlo. Esto aporta una importante optimización en aplicaciones DSP especialmente para arquitecturas de grano fino.

Transformaciones a nivel de instrucción

Se trata de transformaciones de código que tratan de reducir el uso de recursos hardware.

- **Tree-high Reduction (THR):** Es una técnica que trata de balancear un árbol de operación reordenando sus operaciones para obtener mayor paralelismo y así reducir el número de ciclos en los que se obtiene el resultado.

- **Operation Strength Reduction (OSR):** Es una técnica que trata de sustituir una operación por otra menos costosa desde el punto de vista computacional. Por ejemplo multiplicaciones y divisiones por constantes pueden ser sustituidas por desplazamientos y sumas/restas y dichos desplazamientos son implementados a su vez como simples interconexiones. Además las divisiones y multiplicaciones por potencias de dos son sustituidas igualmente por desplazamientos.

- **Movimiento de código (elevación y hundimiento):** A veces adelantar o retrasar una instrucción dentro del flujo de ejecución. Por ejemplo podemos sacar de un bucle instrucciones que no es necesario que se ejecuten cada iteración como sería la lectura de una variable de memoria.

Transformaciones a nivel de bucle

Se trata de transformaciones que sacan partido del paralelismo de una aplicación haciendo uso de los recursos hardware existentes. Transformaciones clásicas de este tipo son loop-unrrolling, loop tiling y loop merging, las cuales pueden combinarse.

- **Loop unrolling:** Replica x veces el cuerpo del bucle que pasará a realizar en cada iteración x operaciones en paralelo.

- **Loop tiling:** Puede aplicarse sobre bucles internos que hagan uso de diferentes memorias o puertos de una memoria y así crear dos bucles anidados que hagan uso de esos datos en paralelo.

- **Loop merging:** Fusión de bucles que puede aplicarse por ejemplo en distintos bucles que hagan uso de los mismos datos de una memoria.

Transformaciones orientadas a datos

La flexibilidad de las arquitecturas reconfigurables en términos de configuración y organización de estructuras de almacenamiento hace que las transformaciones de datos como la distribución, replicación y el uso de variables escalares sean adecuadas para estas arquitecturas.

- **Distribución de datos:** Se divide un array en varios arrays que pueden accederse de manera independiente. Normalmente cada array será mapeado a una memoria diferente lo que junto con la técnica de loop unrolling nos permite ejecutar los accesos a cada array de manera concurrente.

- **Replicación de datos:** Es una técnica que puede usarse para aumentar el ancho de banda. Teniendo los mismos datos en dos memorias diferentes podemos acceder a cada una de las memorias en paralelo a datos diferentes.

- **Variables escalares:** Si tenemos una serie de variables que se acceden de manera significativa continuamente se pueden señalar como variables escalares. Estas variables se mapearán sobre registros o en memorias más pequeñas de modo que se disminuye el número de ciclos para obtener el valor de dichas variables.

Expansión y creación de macros

Son operaciones complementarias. La **expansión de macros** consiste en sustituir una llamada a una macro por el código de ésta y la **creación de macros** representa el proceso contrario, reconocer partes de código comunes que se encapsulan en una macro y sustituir dichas partes por la llamada a la macro. En hardware en el caso de una expansión de macro esto quiere decir que vamos a generar paralelismo, ya que en cada parte donde se llamaba a la macro ahora vamos a tener diferentes grupos de instrucciones. En el caso de una creación de macros vamos a sustituir cada grupo de código similar con la llamada a la macro lo que provoca que se utilice el mismo hardware para las distintas llamadas a la macro.

2.3.2 Compiladores para sistemas basados en FPGAs

En este apartado se presenta una breve descripción de algunas de las soluciones más actuales que aparecen en [12] las cuales generan únicamente sistemas basados en FPGAs. Además se introduce el concepto de packing y se habla sobre dos frameworks que permiten realizar una exploración rápida del espacio de diseño.

Match Compiler ([13]) acepta descripciones en MATLAB y las traduce a RTL-VHDL behavioral que puede ser procesado por las herramientas comerciales de síntesis y emplazamiento y enrutado. Incluye una interfaz que permite acceder a IP cores descritos en formato EDIF/HDL. Dado que MATLAB es un lenguaje débilmente tipado es necesario realizar un análisis de las variables de la descripción de entrada. Una vez realizado el análisis las operaciones con matrices se expenden en bucles. El paralelismo de dichos bucles puede ser explotado incluso en FPGAs diferentes ya que se incluye comunicación y sincronización entre ellos. El siguiente paso que realiza este compilador consiste en transformar operaciones en punto flotante y operaciones enteras a punto fijo infiriendo el ancho de bits necesario. Entonces el compilador incluye los IP cores y crea una representación VHDL-AST (VHDL abstract syntax tree). Seguidamente se genera un pipeline y se planifican las operaciones de manera que se explota el paralelismo. Los bucles son incluidos en este pipeline y se intentan acomodar los accesos a la misma memoria en etapas diferentes. Finalmente se genera el código VHDL-RTL listo para ser usado por las herramientas comerciales.

Galadriel-Nenya ([14],[15]) se compone de dos aplicaciones. Galadriel recibe como entrada un subconjunto de bytecodes de JAVA y lo traduce a un formato específico que recibe Nenya que a su vez produce una unidad de control y una ruta de datos en VHDL behavioral y estructural respectivamente. Entre las optimizaciones que se incluyen se hace uso del paralelismo a nivel de operación, bloque básico y funcional. Además para diseños que no caben en la FPGA de destino se genera, si es posible, diferentes particiones temporales que realizan las operaciones teniendo cada una su controlador y su ruta de datos. Dado que produce una salida en VHDL este código puede ser usado por herramientas comerciales específicas de emplazamiento y enrutado.

El compilador **Sea Cucumber ([16])** recibe como entrada una descripción en JAVA y produce como salida una especificación EDIF, usando JHDL, la cual es traducida al formato de bitstream de Xilinx. Trata el problema de la extracción de concurrencia haciendo uso del estándar de hilos de JAVA con el que reconoce paralelismo de tareas o grano grueso tal como las diseña el programador dejando las transformaciones de grano fino para cada uno de los hilos.

HP-Machines ([17]) recibe como entrada de diseño una descripción en un subconjunto de C++ con una semántica especial. HP-Machines utiliza un modelo de programación concurrente en el que se especifican máquinas de estados abstractas, a que toda aplicación puede especificarse con un conjunto de máquinas de estado. De esta manera el paralelismo puede representarse explícitamente siendo tarea del programador representar el paralelismo de grano medio ocupándose el compilador del paralelismo a grano fino a nivel de instrucciones. Además se realizan optimizaciones bit-narrowing y optimizaciones de bit. La salida del compilador puede aplicarse a diferentes FPGAs de Xilinx haciendo uso de la aplicación JBits.

El compilador **ROCCC ([18])** recibe como entrada de diseño una descripción en un subconjunto de C y produce una salida en HDL. Este compilador se centra en transformar y optimizar una aplicación sustituyendo partes del software de entrada por recursos hardware dedicados. Optimiza el tiempo de ciclo creando etapas de pipeline y maximiza el uso de datos procedentes de memorias externas reduciendo así el tiempo de acceso a memoria global.

Si se quiere ampliar información sobre estos compiladores pueden seguirse las referencias situadas en [12] donde además tenemos información acerca de otros compiladores orientados a sistemas basados en FPGAs como son Prism I-II, SPC, Compiler from Maruyama and Hoshino, DeepC Silicon, COBRA ABS, DEFACTO, Streams-C, CAMERON y SPARCS. Ahora pasamos a definir el concepto de packing y finalmente comentamos las principales

características de dos frameworks orientados a la exploración del espacio de diseño.

El proceso de mapear los componentes de un circuito en los recursos hardware disponibles en una FPGA se denomina **packing** que tradicionalmente se aplica entre las fases de mapeado tecnológico y emplazamiento aunque actualmente está fuertemente ligado a las fases de emplazamiento, mapeado tecnológico o síntesis. En particular en [p5-ahmed] tratan el packing relativo a la familia de FPGAs Virtex 5. Por un lado describen como el packing basado en emplazamiento puede usarse para utilizar las 6-input LUT de modo dual con el objetivo de disminuir el área y reducir el consumo. Además al aumentar la densidad del diseño se disminuye la longitud de los cables y por lo tanto su capacitancia. Por otro lado también consideran el packing en lo relativo a block RAMs y DSPs y presentan una técnica simple de packing para block RAM que mejora considerablemente la eficiencia en un caso de diseño frecuente en muchos circuitos orientados a DSP (filtros FIR). Dado que en la arquitectura Virtex 5 cada block RAM de 36 Kb está alineada con un par de DSP48E slices se da una alineación de recursos de la que se puede sacar provecho. En las pruebas realizadas se obtiene una mejora media del 11% en la frecuencia del diseño.

Según [19] uno de los principales inconvenientes que tiene la computación reconfigurable (RC) es que las herramientas de diseño RC existentes están en un estado de relativa inmadurez si las comparamos con las herramientas de diseño de software ya que las herramientas RC actuales no permiten una exploración rápida y sencilla del espacio de diseño (DSE). A la hora de diseñar algoritmos RC es común tener que realizar el proceso de DSE para obtener uno que se ajuste a las especificaciones dadas. Por ello nos proponen un framework (CMD) que permite realizar una exploración del espacio de diseño para algoritmos RC a nivel de cores sin necesidad de escribir código. Para realizar dicha exploración nos aportan predicciones de rendimiento del diseño. Según datos experimentales obtenidos con algoritmos DSP en punto flotante con este framework se han producido sistemas en minutos cuyas prestaciones han variado luego respecto a la implementación final un 3%. Esto ayuda en proceso DSE ya que permite rechazar antes de la etapa de diseño las alternativas que no cumplen nuestras expectativas. Aunque se puede crear código automáticamente a partir de los cores en este estudio se han limitado a crear el código a mano para cada uno de los ejemplos mostrados.

En [20] nos encontramos con un enfoque parecido al de [19]. En este caso nos proponen un framework con el que podemos realizar modelado a nivel de kernel (equivalente a core)

orientado a dispositivos y a aplicaciones de procesado de señal. Nos permite desarrollar una librería de modelos especificando la ruta de datos y el algoritmo a un nivel de abstracción adecuado para representar IP cores parametrizables. Esta es una diferencia con el framework presentado en [19], en el que estos parámetros son inferidos a partir de las características del dispositivo. El framework de [20] incluye un estimador a alto nivel del rendimiento en área, frecuencia y energía de los diseños. En este caso se realiza un proceso de DSE jerárquico. En primer lugar se usa automáticamente (mediante una heurística) la herramienta DESERT que realiza una poda del espacio de diseño (compara 73000 diseños en un minuto para un escenario mostrado) y sobre los diseños seleccionados se aplica HiPerE de manera manual teniendo un conjunto de parámetros más extenso que en el primer paso (les llevó unas 10 horas realizar la comparación de 16 diseños).

2.4 Diseñando con DSPs

Como se viene indicando a lo largo de este trabajo actualmente existen FPGAs que contienen bloques DSP que permiten a la FPGA poder implementar algoritmos de cómputo intensivo. Al igual que con los demás sistemas hardware a la hora de diseñar un sistema DSP debemos elegir un nivel de abstracción y un enfoque de diseño. En [21] al tratarse de una herramienta HLS se utilizan los algoritmos desarrollados en C++ como entrada y se obtienen los diseños en RTL como salida. Además se dispone de instrucciones de código (pragmas y templates) que permiten especificar de manera directa ciertas características del hardware y la configuración de los DSP para así mejorar el rendimiento del diseño final.

En [22] se propone utilizar Xilinx System Generator for Matlab para implementar aplicaciones DSP de tiempo real. En este caso la metodología incluye las etapas de conversión AD y DA de entrada y salida típicas de un sistema DSP. Se diseña el sistema de manera gráfica con Simulink (ver figura 2-11) haciendo uso de los bloques de diseño de Simulink para simular la entrada y de los bloques de Xilinx para el resto del diseño (ya que debe poder convertirse a VHDL) La simulación del sistema se realiza igualmente con Simulink.

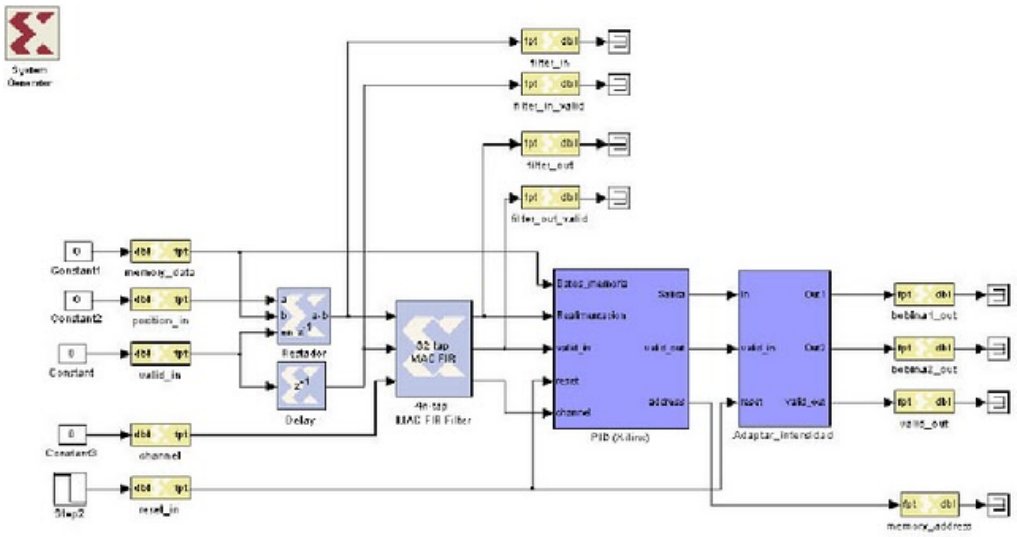


Figura 2-11. Ejemplo de diseño con Xilinx System Generator for Matlab.

Los autores de [23] proponen un generador de números aleatorios Gaussiano multivariable cuyas muestras son generadas como sumas de productos siendo realizada cada una de ellas en un bloque computacional (CB). Para el algoritmo 3 de su estudio esos CB se corresponden con un bloque DSP48.

Se presentan tres implementaciones del Advanced Encryption Standard (AES32, AES128, AES128U) en [24] que hacen uso de los DSP48E de una FPGA Virtex 5. AES32 cuenta con 4 DSP48E conectados en cascada, AES128 está formado por cuatro AES32 y AES128U por diez AES128. Con la implementación AES128U consiguen un throughput de 52.8 Gb/s haciendo uso de 160 bloques DSP48E. Cada DSP48E realiza una función lógica XOR de 32 bits.

En [25] encontramos una implementación de un algoritmo de cópula gaussiana multifactor (MFGC) aplicado a fijación de precios CDO. Esto es, un algoritmo utilizado para simular la fijación de precios de un determinado valor bursátil. Se trata de una implementación que hace uso de bloques DSP48E y que replica varias veces un módulo principal. De esta manera aprovechan el paralelismo y consiguen un speedup igual a 71 con relación a un procesador Intel Xeon 3.4 GHz.

En [26] tenemos una implementación de MUSCL (un método para representar y evaluar ecuaciones diferenciales parciales) sobre FLOPS-2D (un sistema multi-fpga escalable). Los autores crean una versión segmentada de MUSCL a partir de su diagrama de flujo utilizando en

cada nodo computacional IPs de Xilinx Core Generator de punto flotante con doble precisión. Consiguen un tiempo de ejecución de 6 a 23 veces más rápido que la versión software sobre un procesador Intel Core 2 Duo a 2.66 GHz.

Los autores de [27] proponen una nueva técnica para implementar acumuladores en formato punto flotante (PF) de doble precisión. El acumulador, desarrollado en VHDL, resuelve problemas de anteriores técnicas y opera sin añadir un overhead en tiempo respecto a la operación de suma. Dicha operación de suma es implementada sobre 3 DSP48E en cascada.

En [28] se presenta una metodología para generar automáticamente multiplicadores más grandes a partir de los multiplicadores asimétricos contenidos en los bloques DSP48E. Con esta metodología se consiguen multiplicadores que usan menos bloques DSP48E que usando Xilinx Core Generator.

Finalmente en [29] se propone una metodología para explorar el espacio de diseño para diseños FPGA basados en compartición de hardware y la demuestran aplicándola sobre un diseño que implementa una conversión de RGB a YCbCr o calcula un coeficiente DCT en un ciclo. Se hace uso de cuatro bloques DSP48E.

2.5 ¿Donde encaja VHDL?

Hemos definido el concepto de síntesis de alto nivel y hemos hablado acerca de algunos lenguajes y herramientas existentes. Sin embargo no hemos hablado acerca de VHDL [30], el lenguaje con el que desarrollaremos la metodología, ni de los niveles de diseño que abarca. En el capítulo 3 hablaremos de ISE Design Suite 12.1, una herramienta que usa VHDL (entre otras opciones) como entrada de diseño.

VHDL viene de VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. Es un lenguaje de descripción hardware que originalmente fue desarrollado para describir y simular circuitos digitales dirigidos por eventos (de ahí las señales en los process) aunque actualmente también se utiliza para síntesis de circuitos.

VHDL nos permite describir un sistema desde los enfoques **estructural** y de **comportamiento** o de manera mixta. En el modo estructural se describen las conexiones entre los módulos de un circuito, módulos que pueden ser registros, puertas, muxes u otros módulos.

En el modo de comportamiento se hace uso de señales, variables e instrucciones que describen el comportamiento del sistema.

VHDL permite realizar la descripción del sistema en los niveles lógico, RT y behavioral, que se encuentra entre los niveles RT y algorítmico. El nivel behavioral permite realizar una descripción funcional de un componente pero se ha de tener en cuenta el tiempo (el cual no hay que tener en cuenta en el nivel algorítmico).

Como hemos dicho VHDL fue concebido para simulación y luego se ha utilizado para síntesis. En la línea de usar VHDL como HLL (High Level Language) tenemos varios desarrollos [31], [32]. A pesar de ello actualmente no todos los códigos que funcionan en simulación funcionan para síntesis (para más información ver [33]).

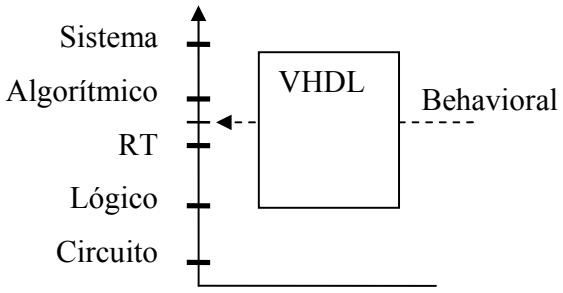


Figura 2-10. Niveles de abstracción cubiertos por VHDL.

Capítulo 3

Bloque DSP48E y su uso con ISE Design Suite 12.1

En este capítulo se realizará una descripción detallada del módulo DSP que ha añadido Xilinx a las FPGAs y que es motivo de estudio en este trabajo. Para poder utilizar este módulo es necesario trabajar con alguna de las herramientas incluidas en el ISE Design Suite 12.1, además se comentará brevemente su finalidad. Seguidamente se describirán las opciones de síntesis que se pueden usar con la herramienta y se explicará brevemente el funcionamiento de Xilinx Synthesis Tool (XST).

3.1 Bloque DSP48E: Arquitectura, etapas y frecuencias.

3.1.1 Arquitectura

Se presenta una breve explicación de la arquitectura del bloque DSP48E (ver figura 3-1), extensión del bloque DSP48, que incluye la familia Virtex 5 de Xilinx y posteriores. Seguidamente enumeramos algunos de los atributos de configuración y las entradas del bloque DSP48E para finalmente hacer una breve mención a sus etapas y frecuencias de trabajo.

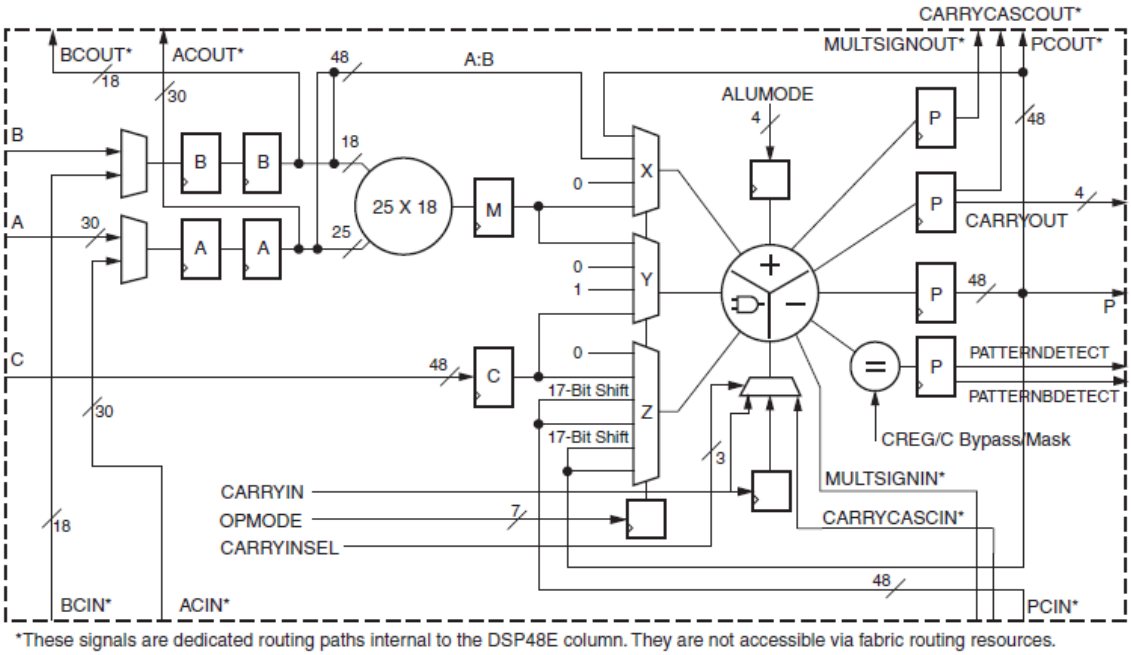


Figura 3-1. Virtex 5 FPGA DSP48E slice.

Cómo vemos en [34] y en la figura 3-1 el bloque DSP48E cuenta con todo lo que debería contener un DSP típico, a saber:

- **Entrada semi-independiente** seleccionable entre entrada directa (A, B) o en cascada (Ain, Cin) y con dos etapas de registros. Nos permitirá crear nuevos diseños a partir de varios bloques DSP.

- **ALU** para suma, resta y operaciones lógicas de 48 bits.
  - AND, NOT, OR, NAND, NOR, XOR, y XNOR
  - La operación es seleccionable dinámicamente mediante ALUMODE.

- **Multiplicador** que permite hacer productos de 25x18 bits. Notar que sus entradas disponen de dos etapas de registros que nos permitirán implementar un pipeline junto con el **registro M**.

- **Detector** de patrones y de patrones complementarios.

- **Registro** para implementar acumuladores (por ejemplo AAC, Adder Accumulator, o MAC, Multiplier Accumulator).

Además disponemos de las siguientes funcionalidades:

- **Modos de operación** seleccionables dinámicamente
  - 7-bit OPMODE controla la selección de los multiplexores X, Y, y Z.
  - ALUMODE seleccionable dinámicamente.

- **Single Instruction Multiple Data (SIMD)**
  - Es estático, no podemos cambiar el modo en ejecución,
  - Modo para adder/subtracter de tres entradas. Excluye el uso del multiplicador en la primera etapa.
  - Dual 24-bit SIMD adder/subtracter/accumulator con dos señales CARRYOUT individuales.
  - Quad 12-bit SIMD adder/subtracter/accumulator con dos señales CARRYOUT individuales.

- **Señales** que permiten combinar diferentes bloques DSP48E para crear.
  - 96-bit accumulators/adders/subtracters

3.1.2 Atributos bloque DSP48E

Los atributos mostrados a continuación se usan durante la síntesis del diseño y controlan el camino de datos que se implementa en el bloque DSP lo que establece parte de su funcionalidad final. Se muestran los atributos que se han considerado más relevantes (ver tabla 3-1) para este trabajo, por lo que si se requiere más información puede encontrarse en [34].

Tabla 3-1. Atributos de control de camino lógico y funcionalidad del DSP48E slice.

Atributo	Valor	Significado
Atributos para control de registros		
AREG	0,1,2	Nº regs. para entrada A
ACASCREG	0,1,2	Junto con AREG, selecciona el número de registros para la salida A en cascada, ACOUT. Debe tener el mismo valor o uno menos que AREG AREG es 0: ACASCREG debe ser 0 AREG es 1: ACASCREG debe ser 1 AREG es 2: ACASCREG debe ser 1 o 2

Tabla 3-1 (continuación). Atributos de control de camino lógico y funcionalidad del DSP48E slice.

BREG	0,1,2	Nº regs para entrada B
BCASCREG	0,1,2	Junto con BREG, selecciona el número de registros para la salida B en cascada, BCOUT. Debe tener el mismo valor o uno menos que BREG BREG es 0: BCASCREG debe ser 0 BREG es 1: BCASCREG debe ser 1 BREG es 2: BCASCREG debe ser 1 o 2
CREG	0,1	Nº regs para entrada C
MREG	0,1	Uso registro resultado parcial multiplicador.
PREG	0,1	Uso reg resultado
Atributos de control de características		
AINPUT	DIRECT, CASCADE	entrada directa o en cascada (desde el DSP48E anterior)
BINPUT	DIRECT, CASCADE	entrada directa o en cascada (desde el DSP48E anterior)
USE_MULT	NONE, MULT, MULT_S (MULT_S)	no usar multiplicador, usarlo sin pipeline (requiere MREG=0), usarlo con pipeline (requiere MREG=1)
USE_SIMD	ONE48, TWO24,FOUR12	hacer una (48 bits), dos (24 bits) o cuatro (12 bits) operaciones en paralelo

3.1.3 Puertos bloque DSP48E

Tabla 3-2. Puertos del bloque DSP48E.

Puerto	Anchura (bits)	Propósito
Entradas		
A	30	Puerto de datos. Puede ser la entrada del multiplicador de 25 bits (A[24:0]) o puede concatenarse con B (A:B) para crear una entrada de 48 bits destinada a la ALU
B	18	Puerto de datos. Puede ser la entrada del multiplicador de 18 bits (A[17:0]) o puede concatenarse con A (A:B) para crear una entrada de 48 bits destinada a la ALU
C	48	Puerto de datos. Puede actuar de entrada de datos para la ALU o de entrada para el detector de patrones
OPMODE	7	Control para los MUXes X, Y y Z
ALUMODE	4	Selecciona la operación realizada en la ALU
CARRYINSEL	3	Permite seleccionar la lógica de acarreo que se adapte a la operación a realizar en el DSP

Tabla 3-2 (continuación). Puertos del bloque DSP48E.

Salidas		
P	48	Resultado de las operaciones del DSP
CARRYOUT	4	Acarreos para las operaciones de suma/resta
PATTERNDETECT	1	Detección del patrón
PATTERNBDETECT	1	Detección del patrón contrario
OVERFLOW, UNDERFLOW	1	Detectan si se ha producido overflow o underflow en la operación del DSP

3.1.4 Etapas y frecuencias

Respecto a etapas del pipeline y frecuencia de trabajo debemos decir que son dos conceptos directamente relacionados. Podemos crear un diseño sin etapas intermedias pero funcionará a una frecuencia menor que un diseño con etapas intermedias.

En general, si se quiere obtener la máxima frecuencia de trabajo el bloque DSP48E deberá tener un pipeline con dos o tres etapas. La máxima frecuencia será 550 Mhz con Speed Grade -3. Para diseños con multiplicador deberán crearse 3 etapas (incluyendo registro M) y para diseños sin multiplicador serán necesarias 2 etapas (ver figura 3-2), apagando el multiplicador para ahorrar energía (USE_MULT none).

Los atributos del DSP48E slice para los dos casos anteriores deberán ser los mostrados en las tablas 3-3 y 3-4.

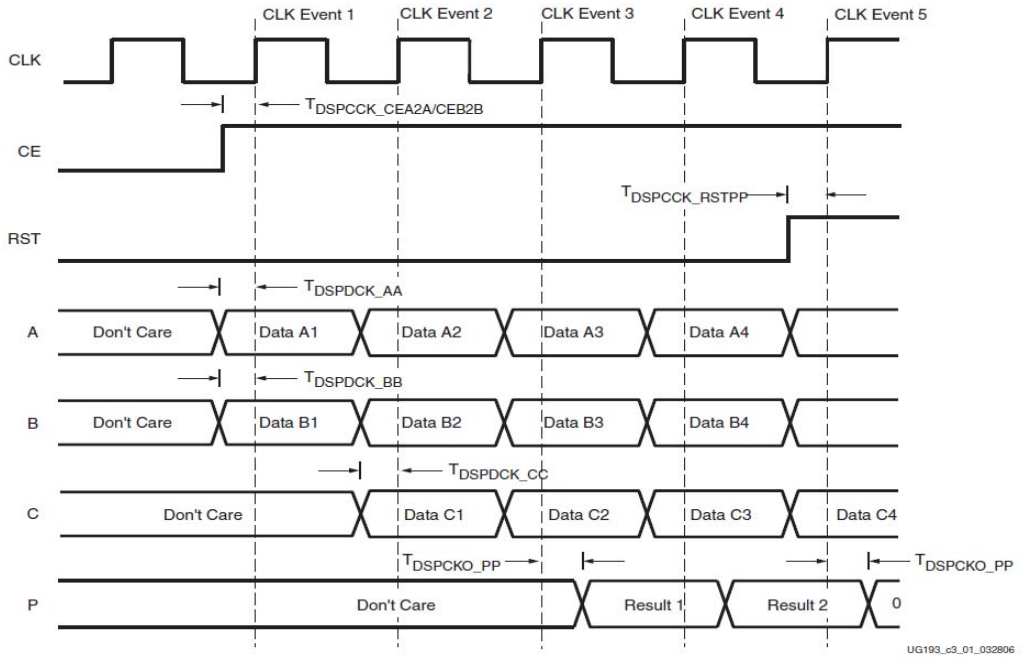


Figura 3-2. diagrama de tiempos para una operación con 2 etapas a las entradas A y B y una etapa para C y P.

Tabla 3-3. Atributos DSP48E slice para diseños con multiplicador segmentado.

Atributo	Valor
AREG	1
BREG	1
CREG	0 (ó 1)
MREG	1
PREG	1
AINPUT	DIRECT
BINPUT	DIRECT
USE_MULT	MULT_S
USE_SIMD	ONE48

Tabla 3-4. Atributos DSP48E slice para diseños sin multiplicador.

Atributo	Valor
AREG	1
BREG	1
CREG	0 (ó 1)
MREG	0
PREG	1
AINPUT	DIRECT
BINPUT	DIRECT
USE_MULT	NONE
USE_SIMD	ONE48

3.2 ¿Qué es ISE?

ISE, desarrollado por la empresa Xilinx, son las siglas de Integrated Software Environment. Según los expuesto en [35] es un software que cubre todo el flujo de diseño de un sistema para posteriormente portarlo a un dispositivo programable de Xilinx.

Se dispone de cuatro versiones distintas del ISE Design Suite de las que se detalla la finalidad de algunas de las aplicaciones que incluyen:

- **Logic Edition:**
  - **Xilinx Synthesis Technology (XST):** Sintetiza VHDL, Verilog, o diseños mixtos.
  - **ISim:** Simulador que nos permite simular funcionalmente y temporalmente los diseños en VHDL, Verilog, o mixtos.
  - **PlanAhead™ software:** Nos permite hace floorplanning avanzado.
  - **CORE Generator™ software:** Librería con múltiples IP cores (Xilinx LogiCORE™ IP) que podemos usar en nuestros diseños.
  - **iMPACT:** Nos permite configurar la FPGA directamente además de

poder crear archivos de programación o hacer readback de la FPGA, verificar datos de configuración del diseño y hacer debug de problemas de configuración entre otras funcionalidades.

- **Reconfiguración parcial:** Permite crear zonas estáticas y regiones reconfigurables dinámicamente en la FPGA.
- **ChipScope™ Pro tool:** Nos permite obtener información de nuestro diseño desde el interior del diseño para así poder verificarlo y depurarlo.

- **Embedded Edition:** Está pensada para ofrecer todo lo necesario en el diseño de sistemas empotrados. Se ofrece todo lo necesario para crear diseños que usen PowerPC® hard processor cores y MicroBlaze™ soft processor cores. Contiene todas las aplicaciones de la Logic Edition además de otras entre las cuales tenemos:
  - **Xilinx Platform Studio (XPS):** Aplicación con la que creamos el hardware de nuestro diseño a partir de procesadores e IPs que pueden ser o no creados por nosotros mismos. Una vez creado el hardware lo exportaremos al SDK.
  - **Base System Builder:** Aplicación que nos permite crear un sistema base para XPS ya configurado.
  - **Software Development Kit (SDK):** Una vez creado el hardware tendremos que crear el software y eso es lo que hacemos con el SDK.

- **DSP Edition:** Incluye todas las aplicaciones contenidas en la Logic Edition. Es un entorno integrado que permite realizar diseños con DSP. Contiene:
  - **System Generator for DSP.**
  - **AccelDSP™ Synthesis Tool.**

- **System Edition:** Contiene todas las aplicaciones y capacidades de las tres ediciones anteriores.

Cabe destacar la interfaz Project Navigator desde la que podemos utilizar varias de las aplicaciones mencionadas anteriormente y con la que podemos crear diseños a partir de código HDL (VHDL, Verilog) o desde esquemáticos.

En nuestro caso nos centramos en la creación y síntesis de diseños en VHDL y para ello utilizamos el Project Navigator como interfaz para la entrada de diseños y desde el Project Navigator hacemos uso de la herramienta XST para realizar la síntesis. Además utilizamos ISim para simular los diseños básicos de este capítulo.

3.3 Herramienta de síntesis XST

El objetivo final de este trabajo de investigación es conseguir sintetizar correctamente código VHDL en los módulos DSP integrados, por lo que una descripción de la herramienta de síntesis que utilizaremos en ISE12_1 es necesaria. Project Navigator nos da la opción de elegir entre XST, Synplify/Synplify Pro o Precision software como herramienta de síntesis. Nosotros nos centraremos en XST y pasaremos a detallar algunas de sus características. Las herramientas de síntesis utilizan el código HDL y generan un tipo de netlist (EDIF o NGC) soportado por la herramienta de Xilinx. Además efectúan en general las tres fases siguientes:

- **Analizar/Chequear la sintáxis:** Chequea la sintáxis del código HDL.
- **Compilar:** Traduce y optimiza el código en una serie de componentes reconocibles por la herramienta.
- **Mapeo:** Traduce los componentes de la fase de compilado a componentes básicos de la tecnología a usar.

3.3.1 Descripción general de XST

Después del diseño en HDL y la simulación del sistema el siguiente paso sería la síntesis, tarea principal de XST. Según [36] XST puede sintetizar VHDL, Verilog o diseños mixtos de VHDL y Verilog. Al contrario que otras opciones del mercado que producen un archivo EDIF y otro NCF, XST produce un fichero denominado NGC que contiene el diseño lógico y las restricciones.

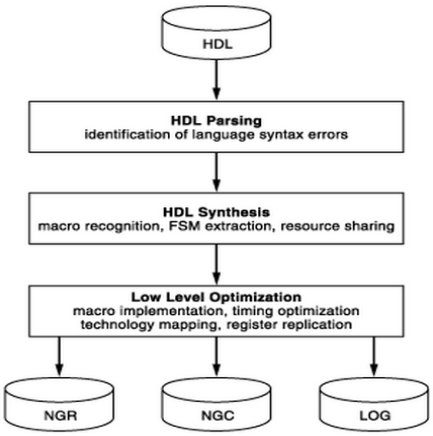


Figura 3-3. Flujo de ejecución de XST.

Respecto al flujo de diseño podemos decir que se divide en tres etapas principales que son parseado, síntesis y optimización a bajo nivel (ver figura 3-3).

Parseado

Se realiza un análisis sintáctico para comprobar que el código HDL cumple con la sintáxis del tipo de HDL especificado en las opciones de diseño del proyecto (ver figura 3-4).

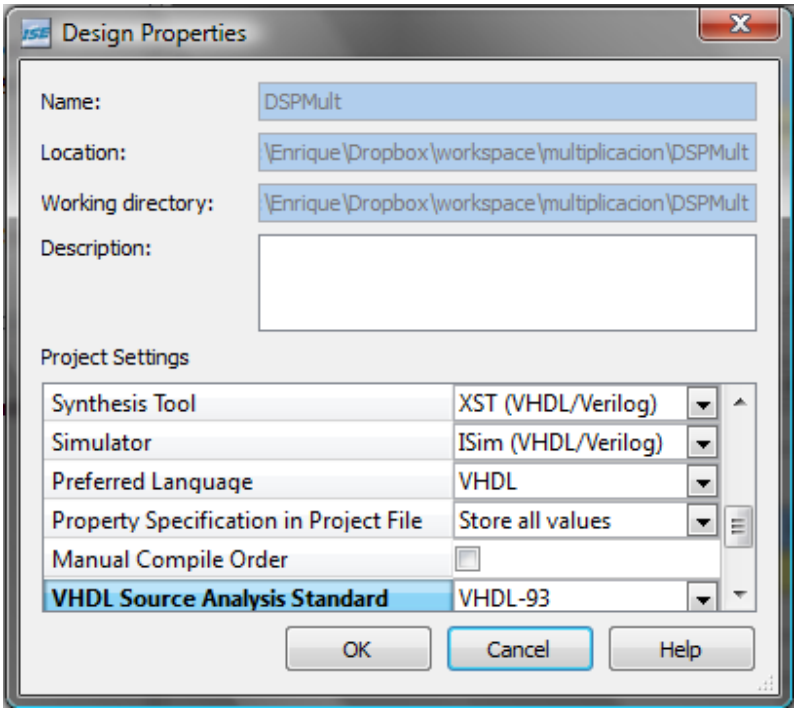


Figura 3-4. Design Properties en Project navigator.

XST soporta VHDL IEEE std 1076-1987, VHDL IEEE std 1076-1993, VHDL IEEE std 1076-2006 (parcialmente implementado), VERILOG IEEE 1364-1995 y VERILOG IEEE 1364-2001. Respecto a VHDL cabe destacar que las construcciones de la versión VHDL87 son aceptadas si no entran en conflicto con las de la versión VHDL93 en cuyo caso se usará la construcción de VHDL93. En casos en los que VHDL87 acepte una construcción pero VHDL93 la considere errónea XST declarará un warning.

Síntesis

Durante la síntesis de HDL XST trata de inferir bloques específicos o macros (MUXes, RAMs, sumadores, restadores, acumuladores, contadores, multiplicadores, registros de desplazamiento, codificadores de prioridad, decodificadores) para los cuales puede crear implementaciones tecnológicamente eficientes. Project Navigator posee una amplia biblioteca con plantillas de macros con las que podemos familiarizarnos con el estilo de codificación requerido para síntesis y además facilitar su inferencia (ver figura 3-5).

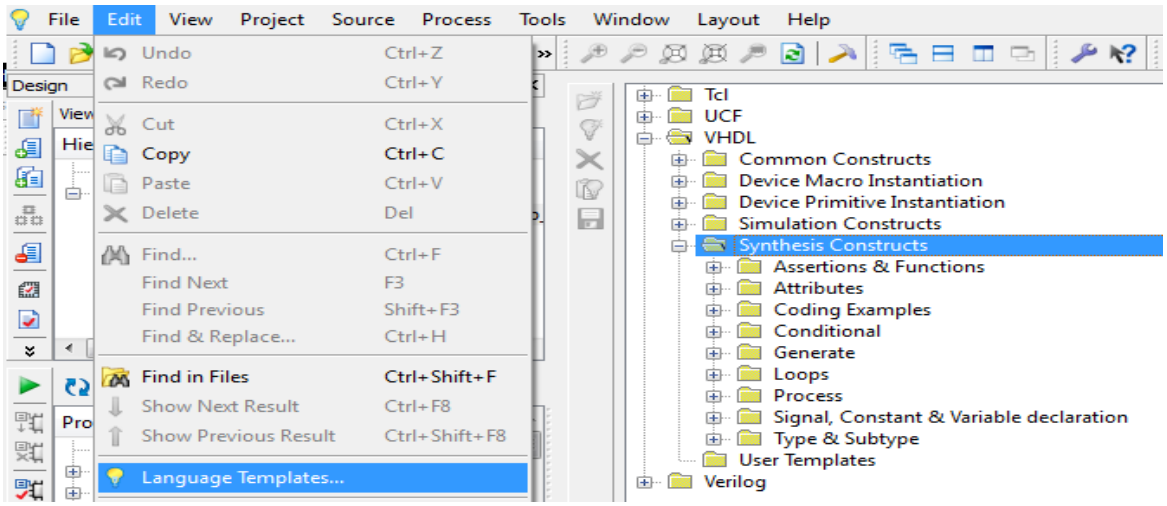


Figura 3-5. Biblioteca de macros de Project navigator.

Además, para reducir el número de bloques inferidos, se realiza un paso denominado *resource sharing*, el cual realiza un análisis de los recursos inicialmente inferidos y de si pueden compartirse o no (ver figura 3-6).

```
if (x=1) then                    Con resource Sharing
        result<=a+b;             se inferiría un
else                             sumador, sin resource
        result<=c+d              sharing tendríamos dos
end if;                          sumadores
```

Figura 3-6. Ejemplo de HDL para resource sharing.

Este paso de *resource sharing* suele llevar a una reducción en el área y un incremento de la frecuencia del diseño.

Durante el paso se síntesis se realizan otras tareas como el reconocimiento de máquinas de estado finitas (Finite State Machine, FSM). XST hace uso del objetivo de optimización específico del diseño (área o velocidad) para elegir el algoritmo de codificación de FSM y así obtener la implementación más eficiente. También podemos elegir el algoritmo de codificación mediante las opciones de proceso de XST (ver figura 3-7).

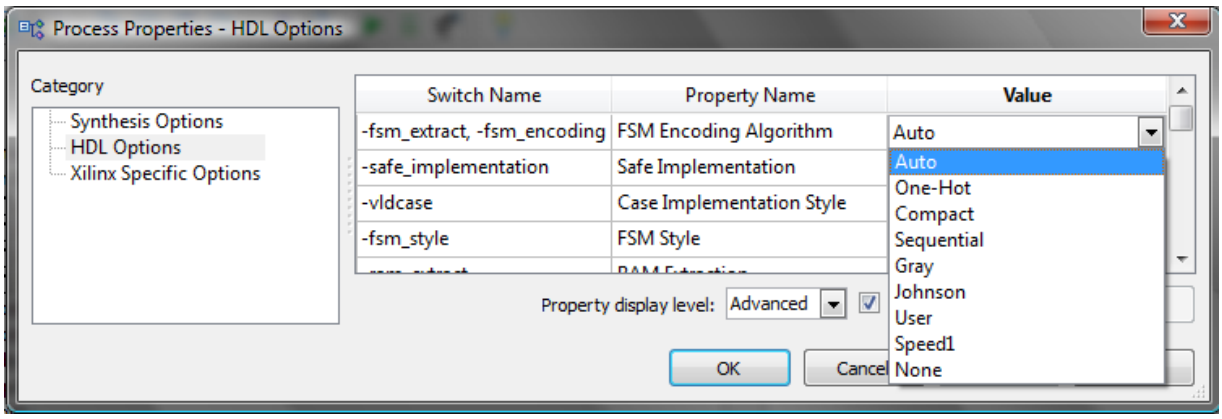


Figura 3-7. Algoritmos de codificación FSM.

El paso de síntesis puede ser dirigido por el usuario mediante el uso de restricciones que pueden ser introducidas en el propio código HDL, haciendo uso de archivo XST Constraint File (XCF) o en las opciones de proceso de Project Navigator.

Optimización a bajo nivel

Durante este paso se transforman las macros inferidas y el resto de la lógica a una implementación específica para la tecnología del dispositivo. Este proceso es timing-driven y puede ser controlado mediante restricciones. Durante esta fase se infieren componentes específicos como:

- Carry logic (MUXCY, XORCY, MULT_AND)
- RAM (block o distribuida)
- Registros de desplazamiento LUTs (SRL16, SRL32)
- Clock Buffers (IBUFG, BUFG, BUFGP, BUFR)
- Multiplexores (MUXF5, MUXF6, MUXF7, MUXF8)
- Funciones aritméticas (DSP48, MULT18x18)

3.3.2 Opciones de XST para FPGA

Varias veces se ha nombrado durante el apartado anterior la existencia de restricciones con las que podemos especificar algunos aspectos de la síntesis del diseño, como puede ser la elección entre diferentes algoritmos o heurísticas para realizar una tarea específica. Sin embargo solamente se ha dado información sobre las opciones de codificación para las FSM y se ha hablado del *resourse sharing*. A continuación se enumeran algunas de las opciones y restricciones que podemos aplicar a nuestros diseños y que nos ayudarán a obtener una mejor implementación para nuestro circuito. Si se requiere más información se pueden consultar los capítulos 4 y 6 de [37].

Modo de optimización (opt_mode)

Es la primera de las opciones que tenemos en *Process Properties->SynthesisOptions* (ver figura 3-8). Se refiere al objetivo principal que tiene la optimización en el proceso de síntesis y puede ser en **area** (intenta reducir el total de elementos lógicos usados en el diseño) o **velocidad** (intenta reducir el número de niveles lógicos y por lo tanto aumentar la frecuencia).

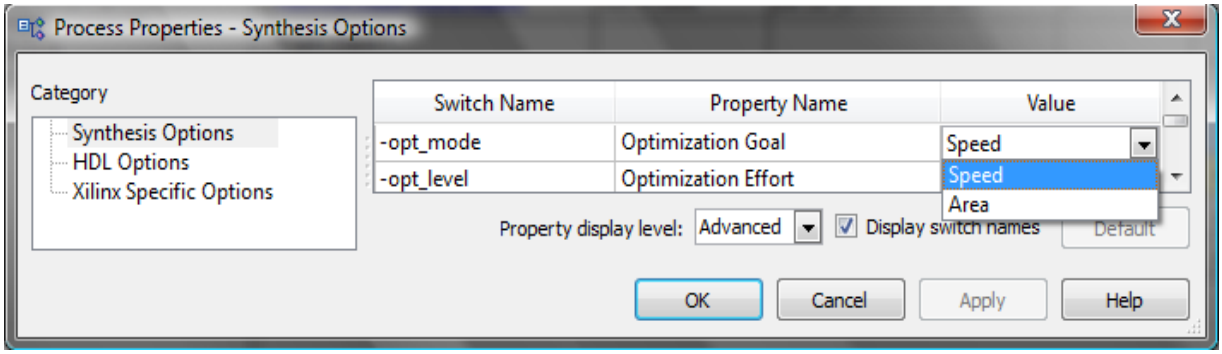


Figura 3-8. Opciones del objetivo de optimización global.

FSM style

Se puede elegir donde implementar las FSM del diseño. Podemos elegir entre **LUT** o **BRAM** (ver figura 3-9). Según [37] se pueden crear grandes FSM más compactas y rápidas usando BRAM en vez de LUTs.

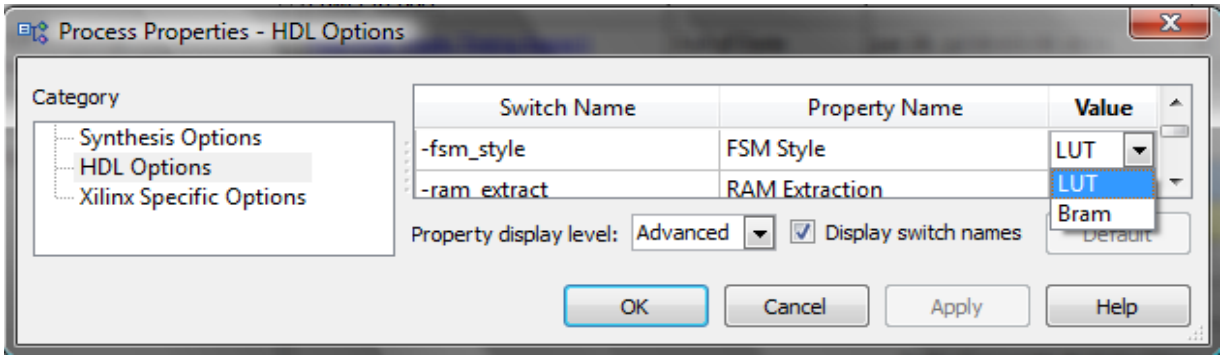


Figura 3-9. Opciones de fsm_style

Flip-flop retiming

Consiste en mover conjuntos de flip-flops y latches por la lógica de la FPGA para reducir el timing y por lo tanto aumentar la frecuencia del diseño. Según [37] existen las versiones forward y backward.

- El *forward retiming* consiste en mover un conjunto de flip-flops que son entrada de una LUT a un solo flip-flop a su salida.
- El backward retiming mueve un flip-flop que sea salida de una LUT a un conjunto de flip-flops que sean entrada de la LUT.

Esta técnica puede incrementar el uso de flip-flops o eliminar alguno pero siempre manteniendo un diseño con el mismo comportamiento.

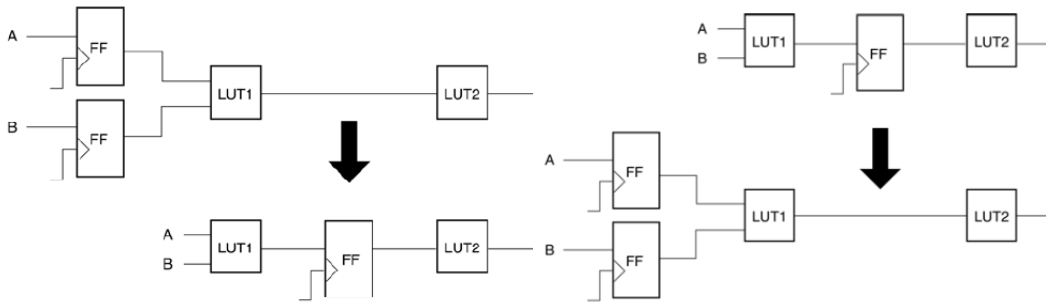


Figura 3-10. Forward y backward retiming.

Podemos controlar el flip-flop remiting con las restricciones *register_balancing*, *move_first_stage* y *move_last_stage*, siendo estas dos últimas parte del *register_balancing*. La técnica de *register_balancing* se corresponde con la definición de retiming (ver figura 3-10) y respecto a las dos otras restricciones cabe destacar que controlan si se aplica (y como se aplica) o no la técnica a la primera y a la última etapa de flip-flops.

- Un flip-flop pertenece a la primera etapa si es el primer flip-flop que aparece en el camino lógico de una señal desde la entrada (ver figura 3-11).
- De manera equivalente un flip-flop pertenece a la última etapa si es el último flip-flop que aparece en el camino lógico de una señal hacia la salida (ver figura 3-11).

El motivo para controlar esta técnica en ambas etapas es que se pueden producir un incremento de los tiempos input-to-clock y clock-to-out al tener que atravesar las señales más lógica combinacional hasta llegar al flip-flop en el caso de la primera etapa, y más lógica desde la salida del flip-flop hasta la salida del diseño en el caso de la última etapa.

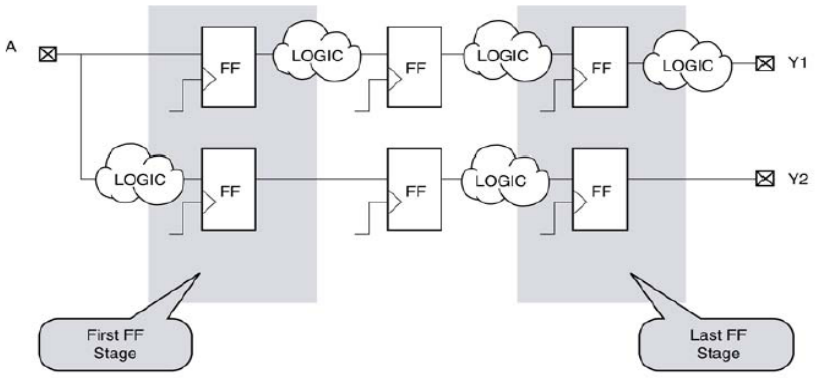


Figura 3-11. Primera y última etapas de flip-flops.

Bloque DSP48E en XST

XST puede implementar automáticamente diferentes macros (suma/resta, acumuladores, multiplicadores, multiplicacion y suma/resta, multiplicador acumulador) para el bloque DSP48E. XST implementará o no algunas de ellas en función del valor de la restricción USE_DSP cuyos valores son **auto**, **yes** y **no**.

En modo **auto** infiere todas las macros menos la suma y la resta, por lo que si se quieren incluir en un bloque DSP48E debe ponerse USE_DSP=yes (ver figura 3-12).

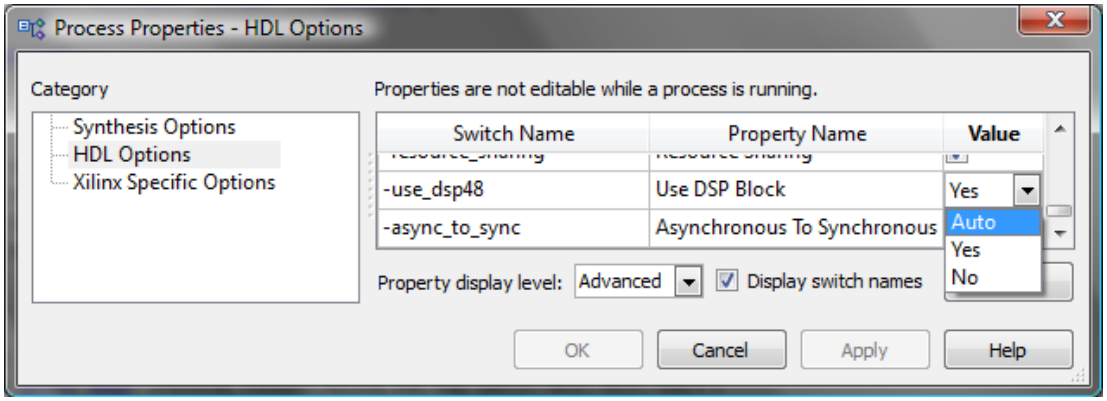


Figura 3-12. Opciones de la restricción USE_DSP.

Además podemos controlar el porcentaje de DSPs disponible para la síntesis mediante la restricción DSP_UTILIZATION_RATIO = x, donde x es un número de 0 a 100. Si se diera el caso de que se infieren más DSP48E de los disponibles para la FPGA XST lanza un warning (ver figura 3-13) y solo usa en la síntesis los DSPs disponibles en la FPGA.

```
Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                2  out of     32     6%
 Number of DSP48Es:                     530  out of     64   828% (*)

WARNING:Xst:1336 -  (*) More than 100% of Device resources are used
```

Figura 3-13. Warning por exceso de recursos utilizados.

Respecto al tipo de macros que se infieren en el DSP cabe destacar que se soporta la versión con registros intermedios de cada una de las diferentes macros citadas más arriba. XST por defecto intentará implementar el máximo de registros dentro del DSP48E. Sin embargo este bloque no soporta registros con set/reset asíncrono por lo que estos no serán incluidos en el DSP, lo cual puede llevar a un diseño con un rendimiento peor. Para evitar esto existe la restricción ASYNC_TO_SYNC que permite sistituir la señales de set/reset asíncronas por señales síncronas en todo el diseño. Será responsabilidad del diseñador comprobar que el comportamiento del diseño no varía cuando se usa esta opción.

3.4 Funcionamiento de XST al inferir DSPs a partir de VHDL

Para usar los bloques DSP48E tenemos varias opciones. Podemos usar DSP IPs de Xilinx, usar System Generator for DSP o podemos crear código VHDL/Verilog. Nos vamos a centrar en esta última opción, ya que en el caso de las dos opciones anteriores se fuerza a XST a generar un DSP particular, mientras que en el tercer caso la herramienta tiene que interpretar el código y decidir que opción de DSP es la que mejor se ajusta para las especificaciones del VHDL, realizando lo que se podría denominar una síntesis de alto nivel.

Vamos a mostrar una serie de ejemplos típicos que hacen uso de la multiplicación y de la suma. Veremos que es necesario escribir el código de una manera específica para conseguir que XST infiera el diseño esperado y en algún caso no se podrá obtener tal cual deseamos. En ese caso sugeriremos el uso de plantillas VHDL para inferir el DSP48E tal cual queremos para nuestro diseño particular.

3.4.1 Multiplicaciones

Mostramos en primer lugar un código VHDL que infiere un multiplicador 25x18 sin ninguna etapa de registros. La configuración de XST usada es la configuración por defecto, en particular, USE_DSP = auto.

```
entity mult is
	generic (
		tamOp1: integer:=25;
		tamOp2: integer:=18;
		tamResul: integer:=43
	);
	port (
		a : in std_logic_vector (tamOp1-1 downto 0);
		b : in std_logic_vector (tamOp2-1 downto 0);
		clock : in std_logic;
		reset : in std_logic;
		p : out std_logic_vector (tamResul-1 downto 0)
	);
end entity;

architecture mult_arch of mult is
begin
process (clock) is
begin
	--Sin Registros en A, B, M ni P
	if clock'event and clock = '1' then
		if reset = '1' then
			p <= (others => '0');
		else
			p <= a*b;
		end if;
	end if;

end process;
end architecture;
```

Una vez sintetizado el código con XST se visualizan el Final Report y el esquemático tecnológico para obtener información de lo que se ha sintetizado.

```
=========================================================================
*                      Advanced HDL Synthesis                           *
=========================================================================

Advanced HDL Synthesis Report
Macro Statistics
# Multipliers                                          : 1
 25x18-bit multiplier                                  : 1


=========================================================================
*                      Low Level Synthesis                              *
=========================================================================
Optimizing unit <mult> ...
Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block mult, actual ratio is 0.
Final Macro Processing ...
```

```
=========================================================================
*                            Final Report                               *
=========================================================================

Final Results
RTL Top Level Output File Name     : mult.ngr
Top Level Output File Name         : mult
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO
Design Statistics
# IOs                              : 88
Cell Usage :
# BELS                             : 2
#      GND                         : 1
#      VCC                         : 1
# IO Buffers                       : 86
#      IBUF                        : 43
#      OBUF                        : 43
# DSPs                             : 1
#      DSP48E                      : 1

=========================================================================

Device utilization summary:
---------------------------
Selected Device : 5vlx110tff1136-3
Slice Logic Utilization:
Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:      0
   Number with an unused Flip Flop:       0  out of      0
   Number with an unused LUT:             0  out of      0
   Number of fully used LUT-FF pairs:     0  out of      0
   Number of unique control sets:         0
IO Utilization:
 Number of IOs:                          88
 Number of bonded IOBs:                  86  out of    640    13%
Specific Feature Utilization:
 Number of DSP48Es:                       1  out of     64     1%
```

Observamos en el Final Report como se ha inferido una macro de un multiplicador 25x18 y que dicha multiplicación se encuentra implementada en un único DSP. Además se ha comprobado, mediante el esquemático tecnológico, que no se infiere ninguna etapa de registros dentro del DSP48E.

Figura 3-14. Diagrama de tiempos multiplicador combinacional con Tc= 2ns (arriba) y Tc=10 ns (abajo).

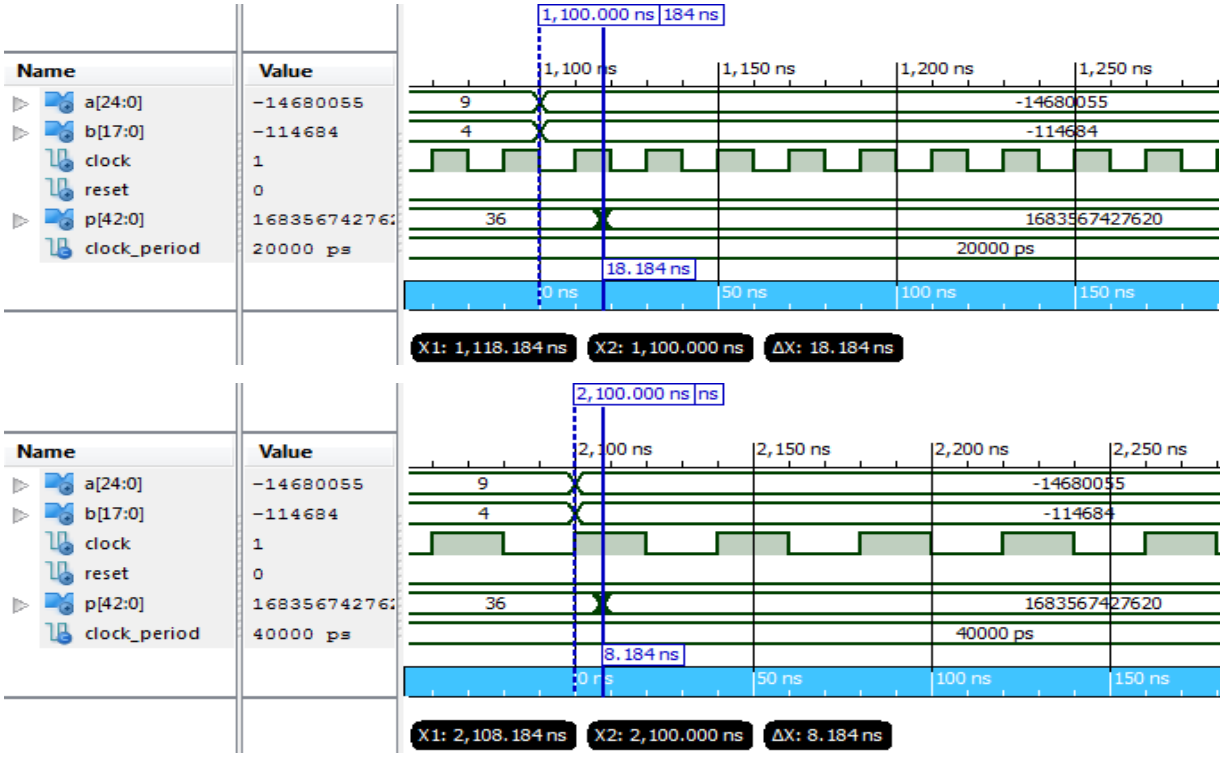


Figura 3-15. Diagrama de tiempos multiplicador combinacional con Tc=20 ns (arriba) y Tc=40 ns (abajo).

Procedemos a realizar una simulación Post Place & Route con ISE Simulator 12.1 (ISIM) con un tiempo de ciclo de 2 ns. Hecho el test con diferentes números de entrada se observa que se obtiene el resultado con un retardo aproximado de 9 ns en los casos de grandes números. Realizamos otra simulación, esta vez con un tiempo de ciclo de 10 ns y obtenemos un tiempo de retardo cercano a 13 ns para el mismo caso (ver figura 3-14). Debido a ello realizamos simulaciones con un tiempo de ciclo de 20 ns y 40. Observando los diferentes diagramas de tiempos se llega a la conclusión de que la simulación tiene en cuenta los nuevos datos a partir del primer flanco de subida. Contando el retardo de esta manera obtenemos un retardo de 8.184 ns en los 4 casos, uniformidad que concuerda con el comportamiento que debería tener un DSP48E en modo combinacional (sin ninguna etapa de registros).

Realizamos ahora unas modificaciones en el código que harán que XST infiera los registros intermedios para el multiplicador. En primer lugar se ha modificado el código de manera que se infieren dos etapas de registros, correspondientes a la entrada y salida de datos, se se ha simulado con 10 ns de tiempo de ciclo (Tc) y se comprueba que se obtiene el resultado tras dos ciclos de reloj (ver figura 3-16). El código necesario para inferir tal multiplicador difiere del código anterior en el proceso principal, cuyo código está incluido a continuación.

```
--Multiplicador con 2 etapas de registros
process (clock) is
begin
      p2 <= a1*b1;
      --Con Registros en A, B y P
      if clock'event and clock = '1' then
            if reset = '1' then
                  a1 <= (others => '0');
                  b1 <= (others => '0');
                  p <= (others => '0');
            else
                  a1 <= a;--------------------------------
                  b1 <= b;--Etapas del pipeline  --
                  p <= p2;--------------------------------

            end if;
      end if;
end process;
```

Por último se ha creado un código que obtiene un multiplicador 25x18 totalmente segmentado, es decir, con 3 etapas de registros (ver figura 3-17a). Un detalle a destacar tanto del código siguiente y el anterior es que los registros deben ser síncronos, incluso el reset, de otra manera no podrán inferirse dentro del bloque DSP48E con las opciones por defecto [Pág. 479].

```
--Multiplicador con 3 etapas de registros
process (clock) is
begin
      p1 <= a1*b1; --Con Registros en A, B y P
      if clock'event and clock = '1' then
            if reset = '1' then
                  a1 <= (others => '0');
                  b1 <= (others => '0');
                  p <= (others => '0');
            else
                  a1 <= a;--------------------------------
                  b1 <= b;--Etapas del pipeline  --
                  p2 <= p1;-------------------------------
                  p <= p2;--------------------------------
            end if;
      end if;
end process;
```

En la simulación a 10 ns de Tc se ha obtenido el resultado con 3 ciclos de delay pero en la simulación a 2 ns de Tc (recordemos que el diseño debería funcionar a 550 Mhz estando totalmente segmentado) el resultado se obtiene con un delay de 7 ciclos (ver figura 3-17b), lo que no debería suceder si el diseño funcionase como indica Xilinx a 550 Mhz.

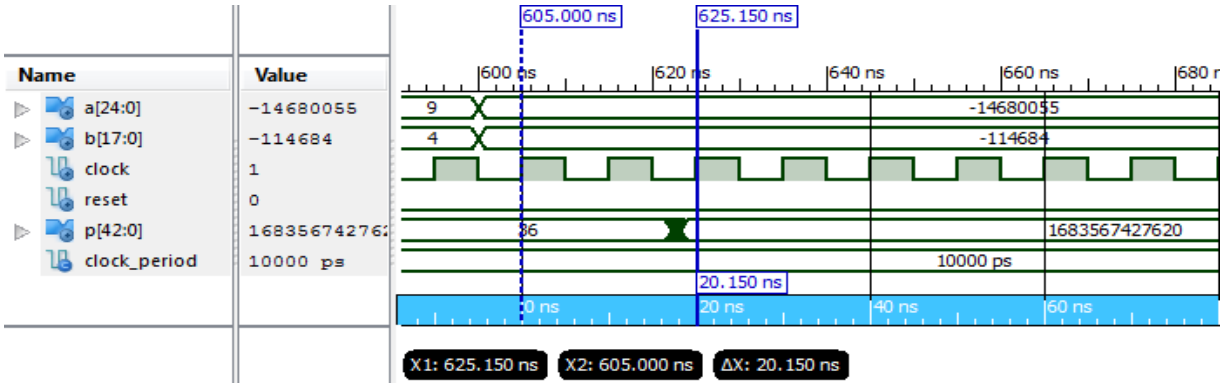


Figura 3-16. Diagrama de tiempos multiplicador de 2 etapas con Tc=10 ns.

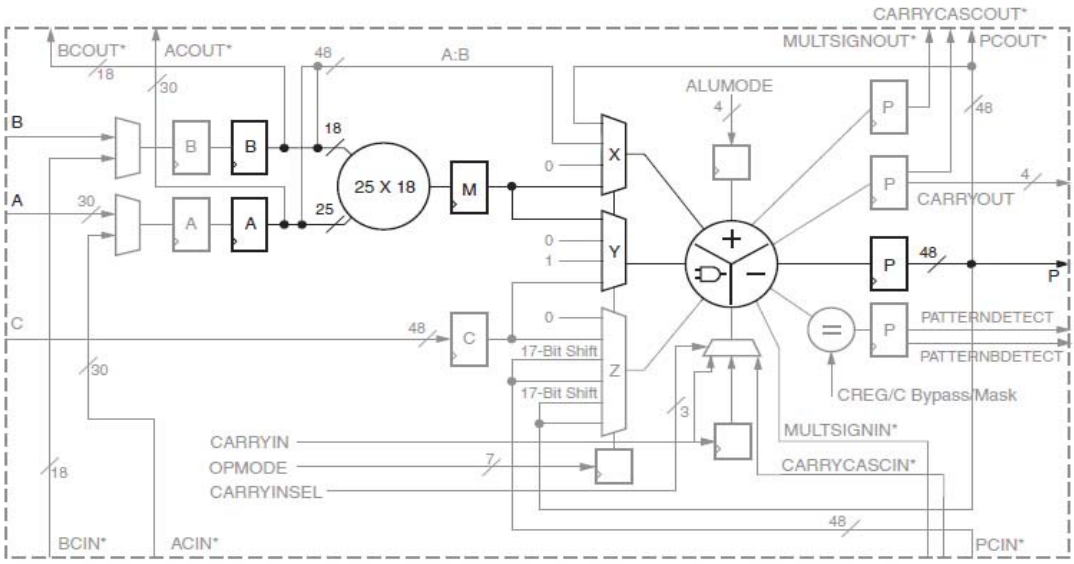


Figura 3-17a. Ruta de datos multiplicación totalmente segmentada.

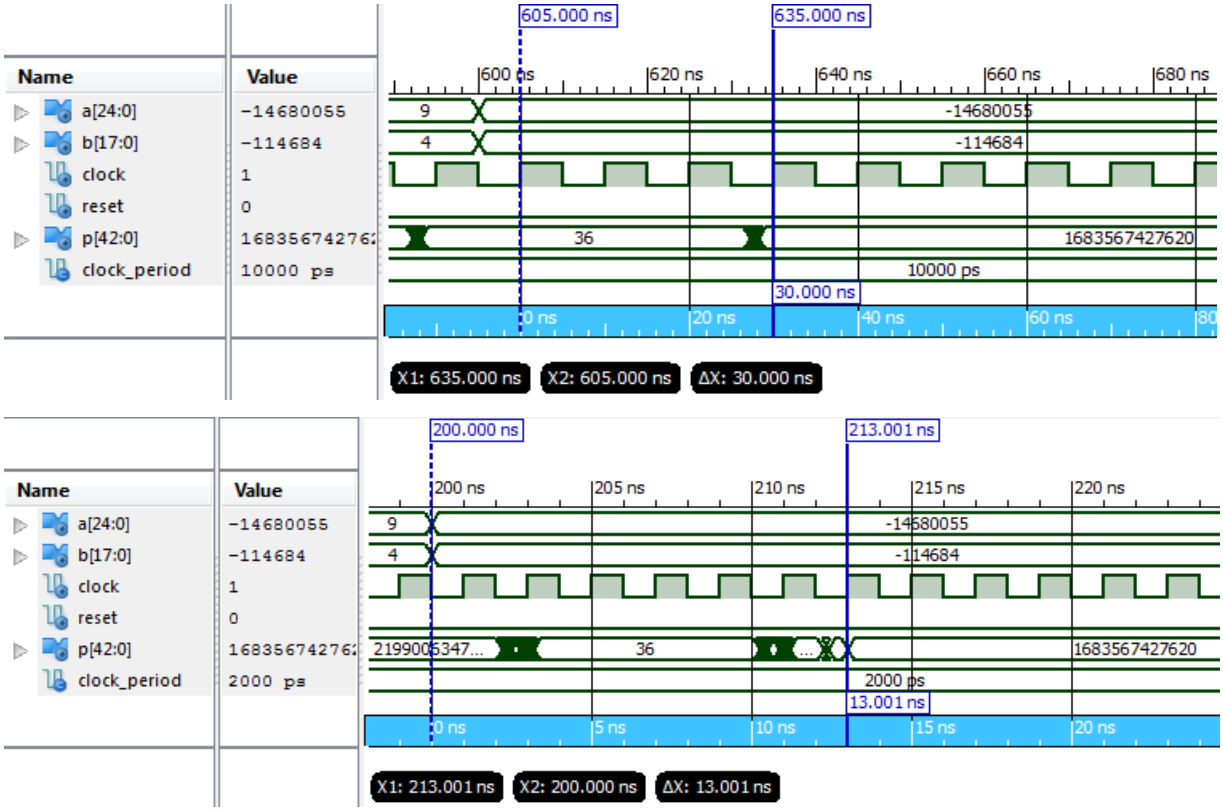


Figura 3-17b. Diagrama de tiempos multiplicador de 3 etapas con Tc=10 ns (arriba) y Tc=2 ns (abajo).

Comprobamos entonces si la síntesis ha sido correcta para el diseño del multiplicador totalmente segmentado. Vemos que se infiere una etapa de registros a la entrada (a1, b1) y dos etapas (p1, p2) a la salida. De esta manera XST se infiere con las 3 etapas del pipeline necesarias (ver figura 3-18) para funcionar a 550 Mhz. Por lo tanto lo más probable es que la simulación no cubra un tiempo de ciclo tan pequeño.

```
Synthesizing (advanced) Unit <mult3reg>.
        Found pipelined multiplier on signal <p1>:
                - 2 pipeline level(s) found in a register connected to the multiplier macro output.
                Pushing register(s) into the multiplier macro.
                - 1 pipeline level(s) found in a register on signal <a1>.
                Pushing register(s) into the multiplier macro.
                - 1 pipeline level(s) found in a register on signal <b1>.
                Pushing register(s) into the multiplier macro.
Unit <mult3reg> synthesized (advanced).
```

Name	Value
SEL_PATTERN	PATTERN
SEL_MASK	MASK
SCAN_IN_SET_P	SET
SCAN_IN_SET_M	SET
SCAN_IN_SETVAL_P	0
SCAN_IN_SETVAL_M	0
ROUNDING_LSB_MASK	0
PREG	1
PATTERN	000000000000
OPMODEREG	0
MULTCARRYINREG	0
MREG	1
MASK	3FFFFFFFFFFF
LFSR_EN_SETVAL	0
LFSR_EN_SET	SET
Instance Name	Mmult_p1
CREG	0
CLOCK_INVERT_P	SAME_EDGE
CLOCK_INVERT_M	SAME_EDGE
CARRYINSELREG	0
CARRYINREG	0
B_INPUT	DIRECT
BREG	1
BCASCREG	1
A_INPUT	DIRECT
AUTORESET_PATTERN_DETECT_OPTINV	MATCH
AUTORESET_PATTERN_DETECT	FALSE
AUTORESET_OVER_UNDER_FLOW	FALSE
AREG	1
ALUMODEREG	0

Figura 3-18. Uso de recursos del bloque DSP48E. USE_MULT = MULT_S, USE_SIMD=ONE48

3.4.2 Multiplicador Sumador/Restador

Como hemos comentado anteriormente el DSP48E es capaz de realizar una operación de multiplicación y suma o resta. Vamos a intentar que se infieran las 3 etapas de registros (entrada, registro M y salida) y haremos uso de las 3 entradas del DSP (ver figura 3-19).

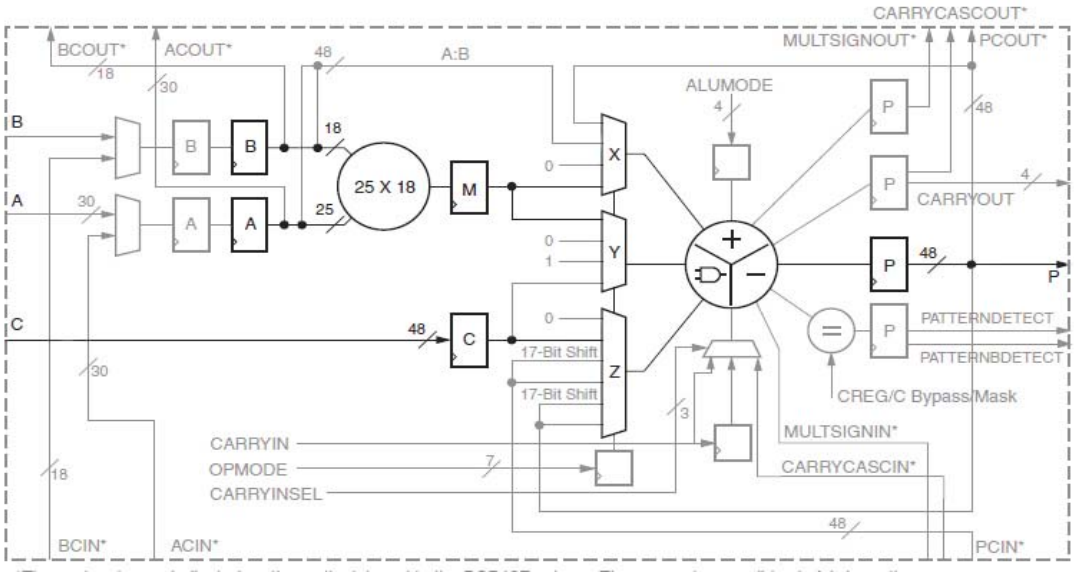


Figura 3-19. Ruta de datos para un multiplicador sumador/restador con 3 etapas.

Se crea el siguiente código para ver si se infiere un diseño tal como queremos y que además se pueda cambiar de suma a resta dinámicamente:

```
entity mult_addsub is
      port (
            a : in std_logic_vector (15 downto 0);
            b : in std_logic_vector (15 downto 0);
            c : in std_logic_vector (31 downto 0);
            clock : in std_logic;
            reset : in std_logic;
            addsub : in std_logic;
            p : out std_logic_vector (31 downto 0)
      );
end entity;

architecture mult_addsub_arch of mult_addsub is

signal p1,p2,p3 : std_logic_vector (31 downto 0);
signal a1 : std_logic_vector (15 downto 0);
signal b1 : std_logic_vector (15 downto 0);
signal c1 : std_logic_vector (31 downto 0);
```

```
      begin

      process (clock) is
      begin
      if(addsub='0')then
            p1 <= p2 + c1;-- p2<=a1*b1 está en el process
      else
            p1 <= p2 - c1;-- p2<=a1*b1 está en el process
      end if;
      if clock'event and clk = '1' then
            if reset = '1' then
                  p <= (others => '0');
                  a1 <= (others => '0');
                  b1 <= (others => '0');
                  c1 <= (others => '0');
            else--Importante, si esto lo sacamos fuera del
                  p2 <= a1*b1;--   ifelse no se infiere el registro M.
                  p <= p1;
                  a1 <= a;
                  b1 <= b;
                  c1 <= c;
            end if;
      end if;
      end process;
      end architecture;
```

Con el código anterior se infieren dos DSPs para realizar la operación de multiplicación más suma/resta. Se modifica el código de manera que solo realice la operación de suma o de resta y se deja para más adelante la comprobación (mediante el uso de plantillas) de si se puede cambiar dinámicamente de suma a resta usando un solo DSP48E. Sin embargo, según la documentación de Xilinx [37] (Pág. 32) sí se puede implementar tal diseño con un bloque DSP48E.

```
=========================================================================
*                      Advanced HDL Synthesis                           *
=========================================================================
Synthesizing (advanced) Unit <mult_addsub>.
      Found pipelined multiplier on signal <p2_mult0000>:
            - 1 pipeline level(s) found in a register connected to the
multiplier macro output.
            Pushing register(s) into the multiplier macro.
            - 1 pipeline level(s) found in a register on signal <a1>.
            Pushing register(s) into the multiplier macro.
            - 1 pipeline level(s) found in a register on signal <b1>.
      Pushing register(s) into the multiplier macro.
            Found registered addsub on signal <p1>:
       - 1 register level(s) found in a register connected to the addsub
```

```
       macro output.
        Pushing register(s) into the addsub macro.
       - 1 register level(s) found in a register on signal <c1>
        Pushing register(s) into the addsub macro.
INFO:Xst:2385 - HDL ADVISOR - You can improve the performance of the
multiplier Mmult_p2_mult0000 by adding 1 register level(s).
Unit <mult_addsub> synthesized (advanced).
=========================================================================
Advanced HDL Synthesis Report
Macro Statistics
# Multipliers                                          : 1
 16x16-bit registered multiplier                       : 1
# Adders/Subtractors                                   : 1
 32-bit registered addsub                              : 1
=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Top Level Output File Name     : mult_addsub.ngr
Top Level Output File Name         : mult_addsub
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO
Design Statistics
# IOs                              : 99
Cell Usage :
# BELS                             : 3
#      GND                         : 1
#      INV                         : 1
#      VCC                         : 1
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 98
#      IBUF                        : 66
#      OBUF                        : 32
# DSPs                             : 2
#      DSP48E                      : 2
=========================================================================
Device utilization summary:
---------------------------
Selected Device : 5vlx110tff1136-3
Slice Logic Utilization:
 Number of Slice LUTs:                    1  out of  69120     0%
    Number used as Logic:                 1  out of  69120     0%
Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:      1
   Number with an unused Flip Flop:       1  out of      1   100%
   Number with an unused LUT:             0  out of      1     0%
   Number of fully used LUT-FF pairs:     0  out of      1     0%
   Number of unique control sets:         0
IO Utilization:
 Number of IOs:                          99
 Number of bonded IOBs:                  99  out of    640    15%
Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     32     3%
 Number of DSP48Es:                       2  out of     64     3%
```

La modificación consiste simplemente en eliminar la entrada "*addsub"* y dejar sólo la instrucción "*p1 <= p2+ c1;"*. En este caso (y en el caso con resta, *p1<=p2-c1*) si se infiere el diseño esperado como puede comprobarse en la figura 3-21. Se obtiene el resultado con 3 ciclos de delay cada vez que cambien los multiplicandos, pero en caso de que se repitan el resultado se obtiene con 2 ciclos de delay (ver figura 3-22).

```
=========================================================================
*                      Advanced HDL Synthesis                           *
=========================================================================
Synthesizing (advanced) Unit <mult_addsub>.
      Multiplier <Mmult_p2_mult0000> in block <mult_addsub> and
adder/subtractor <Madd_p1> in block <mult_addsub> are combined into a
MAC<Maddsub_p2_mult0000>.
      The following registers are also absorbed by the MAC: <a1> in block
<mult_addsub>, <b1> in block <mult_addsub>, <p2> in block <mult_addsub>, <p>
in block <mult_addsub>, <c1> in block <mult_addsub>.
Unit <mult_addsub> synthesized (advanced).
=========================================================================
Advanced HDL Synthesis Report
Macro Statistics
# MACs                                              : 1
 16x16-to-32-bit MAC                                : 1
=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Top Level Output File Name     : mult_addsub.ngr
Top Level Output File Name         : mult_addsub
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO
Design Statistics
# IOs                              : 99
Cell Usage :
# BELS                             : 3
#      GND                         : 1
#      INV                         : 1
#      VCC                         : 1
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 97
#      IBUF                        : 65
#      OBUF                        : 32
# DSPs                             : 1
#      DSP48E                      : 1
Device utilization summary:
---------------------------
Selected Device : 5vlx110tff1136-3
Slice Logic Utilization:
 Number of Slice LUTs:                  1  out of  69120     0%
    Number used as Logic:               1  out of  69120     0%
```

```
Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:      1
   Number with an unused Flip Flop:       1  out of      1   100%
   Number with an unused LUT:             0  out of      1     0%
   Number of fully used LUT-FF pairs:     0  out of      1     0%
   Number of unique control sets:         0
IO Utilization:
 Number of IOs:                          99
 Number of bonded IOBs:                  98  out of    640    15%
Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     32     3%
 Number of DSP48Es:                       1  out of     64     1%
```

Se ha introducido una etapa de registros a la entrada (a1, b1), una intermedia (p2, que es el registro M) y una a la salida (p1). Se ha comprobado que dichas etapas han sido inferidas dentro del bloque DSP48E (ver figura 3-20).

Name	Value
ACASCREG	1
ALUMODEREG	0
AREG	1
AUTORESET_OVER_UNDER_FLOW	FALSE
AUTORESET_PATTERN_DETECT	FALSE
AUTORESET_PATTERN_DETECT_OPTINV	MATCH
A_INPUT	DIRECT
BCASCREG	1
BREG	1
B_INPUT	DIRECT
CARRYINREG	0
CARRYINSELREG	0
CLOCK_INVERT_M	SAME_EDGE
CLOCK_INVERT_P	SAME_EDGE
CREG	1
Instance Name	Maddsub_p2_mult0000
LFSR_EN_SET	SET
LFSR_EN_SETVAL	0
MASK	3FFFFFFFFFFF
MREG	1
MULTCARRYINREG	0
OPMODEREG	0
PATTERN	000000000000
PREG	1
USE_MULT	MULT_S
USE_PATTERN_DETECT	NO_PATDET
USE_SIMD	ONE48

Figura 3-20. Valores inferidos para los atributos del bloque DSP48E.

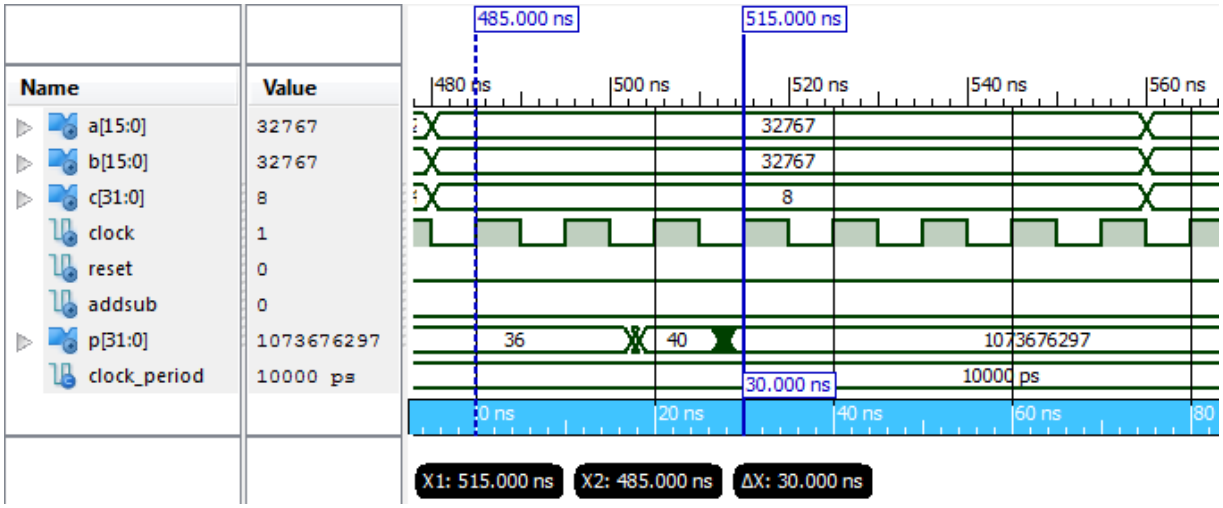


Figura 3-21. Diagrama de tiempos multiplicador sumador. Delay de 3 ciclos.

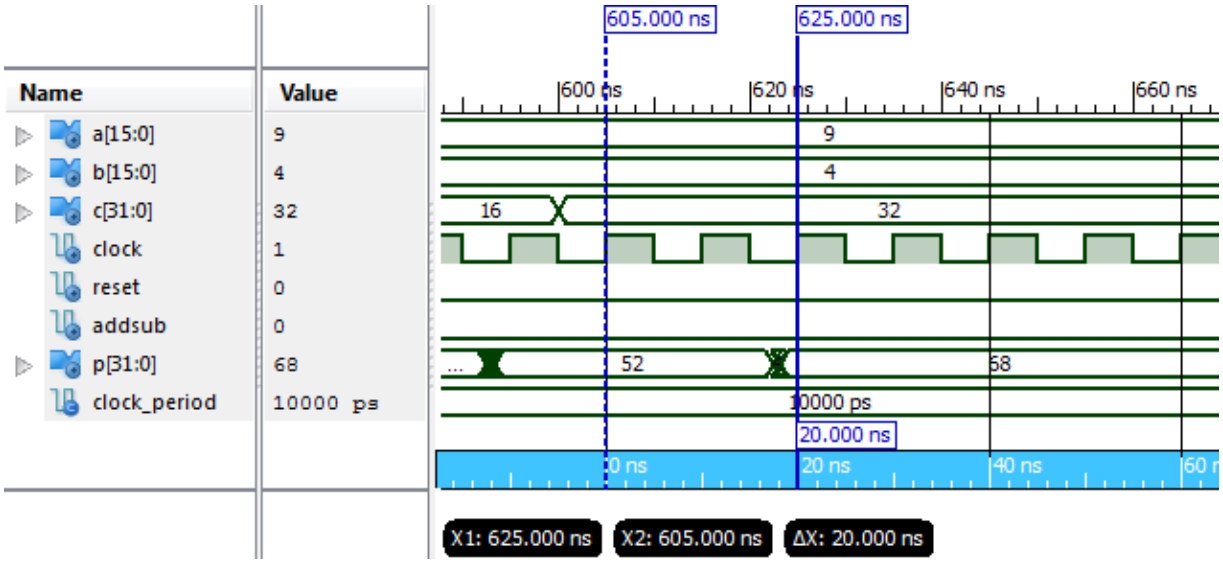


Figura 3-22. Diagrama de tiempos multiplicador sumador. Delay de 2 ciclos.

3.4.3 Suma/resta simple

Para sumar números de 48 bits es necesario hacer uso de las tres entradas de datos que posee el DSP (ver figura 3-23). En este caso vamos a hacer un diseño que utilice registros a la entrada y a la salida de tal manera que el primer operando estará compuesto por los puertos A (30 bit) y B (18 bit) y el segundo operando por el puerto C (48 bits). Será necesario además poner la opción USE_DSP=yes (ver figura 3-24) para que XST infiera la suma en los DSP.

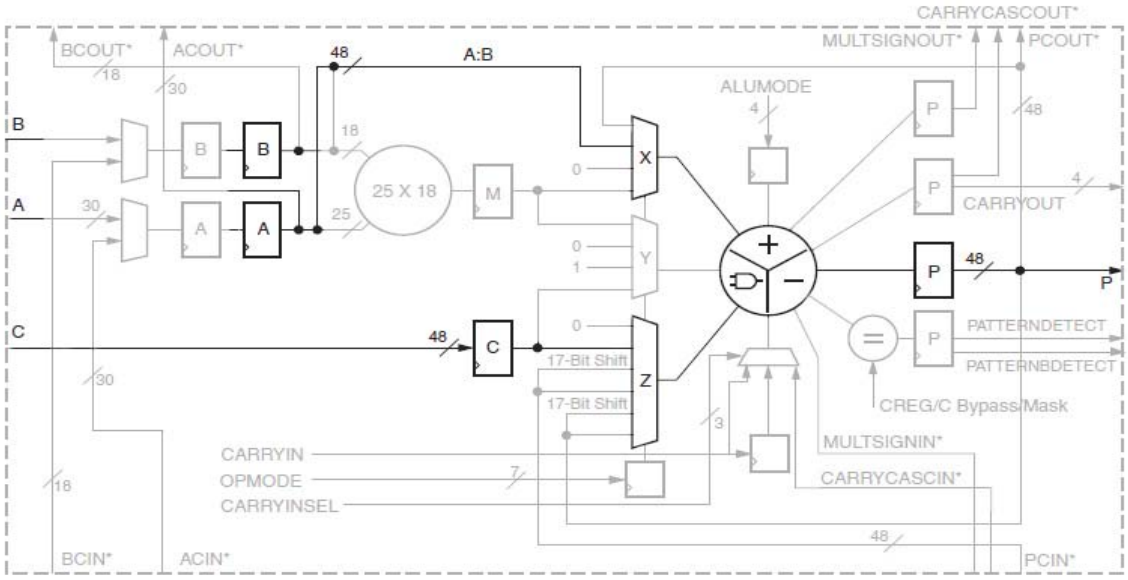


Figura 3-23. Ruta de datos para sumas de 48 bits con 2 etapas de registros.

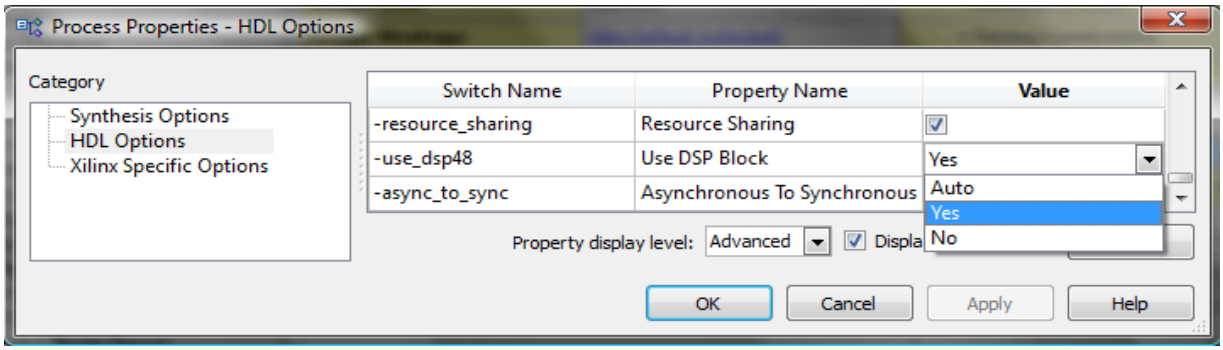


Figura 3-24. Process Properties USE_DSP=yes para inferir suma.

```
entity addsub_2reg is
generic(
      tamOp: integer:=48;
      tamRes: integer:=48
);
port (
      a : in std_logic_vector (tamOp-1 downto 0);
      b : in std_logic_vector (tamOp-1 downto 0);
      clock : in std_logic;
      reset : in std_logic;
      addsub : in std_logic;
      p : out std_logic_vector (tamRes-1 downto 0)
);
end entity;
```

```
      architecture addsub_arch of addsub is

      signal p1 : std_logic_vector (tamRes-1 downto 0);
      signal a1 : std_logic_vector (tamOp-1 downto 0);
      signal b1 : std_logic_vector (tamOp-1 downto 0);

      begin

      process (clock) is
      begin

      if(addsub='1')then
            p1 <= a1+ b1;
      else
            p1 <= a1- b1;
      end if;

      if clock'event and clock = '1' then
            if reset = '1' then
                  p <= (others => '0');
                  a1 <= (others => '0');
                  b1 <= (others => '0');
            else
                  p <= p1;
                  a1 <= a;
                  b1 <= b;
            end if;
      end if;

      end process;
      end architecture;
```

```
=========================================================================
*                     Advanced HDL Synthesis                            *
=========================================================================
Synthesizing (advanced) Unit <addsub>.
      Found registered addsub on signal <p1>:
       - 1 register level(s) found in a register connected to the addsub macro
output.
        Pushing register(s) into the addsub macro.
       - 1 register level(s) found in a register on signal <a1>
        Pushing register(s) into the addsub macro.
       - 1 register level(s) found in a register on signal <b1>
        Pushing register(s) into the addsub macro.
Unit <addsub> synthesized (advanced).
=========================================================================
Advanced HDL Synthesis Report
Macro Statistics
# Adders/Subtractors                                   : 1
 48-bit registered addsub                              : 1
```

```
=========================================================================
*                          Low Level Synthesis                          *
=========================================================================
Optimizing unit <addsub> ...
Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block addsub, actual ratio is 0.
Final Macro Processing ...
=========================================================================
Final Register Report
Found no macro
=========================================================================
=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Top Level Output File Name     : addsub.ngr
Top Level Output File Name         : addsub
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO
Design Statistics
# IOs                              : 147
Cell Usage :
# BELS                             : 3
#      GND                         : 1
#      INV                         : 1
#      VCC                         : 1
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 146
#      IBUF                        : 98
#      OBUF                        : 48
# DSPs                             : 1
#      DSP48E                      : 1
=========================================================================
Device utilization summary:
---------------------------
Selected Device : 5vlx110tff1136-3
Slice Logic Utilization:
 Number of Slice LUTs:                    1  out of  69120     0%
    Number used as Logic:                 1  out of  69120     0%
Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:      1
   Number with an unused Flip Flop:       1  out of      1   100%
   Number with an unused LUT:             0  out of      1     0%
   Number of fully used LUT-FF pairs:     0  out of      1     0%
   Number of unique control sets:         0
IO Utilization:
 Number of IOs:                         147
 Number of bonded IOBs:                 147  out of    640    22%
Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     32     3%
 Number of DSP48Es:                       1  out of     64     1%
```

Se ha inferido un sumador/restador con dos etapas de registros, en un bloque DSP48E (ver figura 3.25b), al que se puede cambiar dinámicamente entre suma y resta cada ciclo (ver figura 3-25a). En el caso de repetir entradas obtenemos el resultado con un ciclo de delay pero en general obtendremos el resultado con dos ciclos de delay.

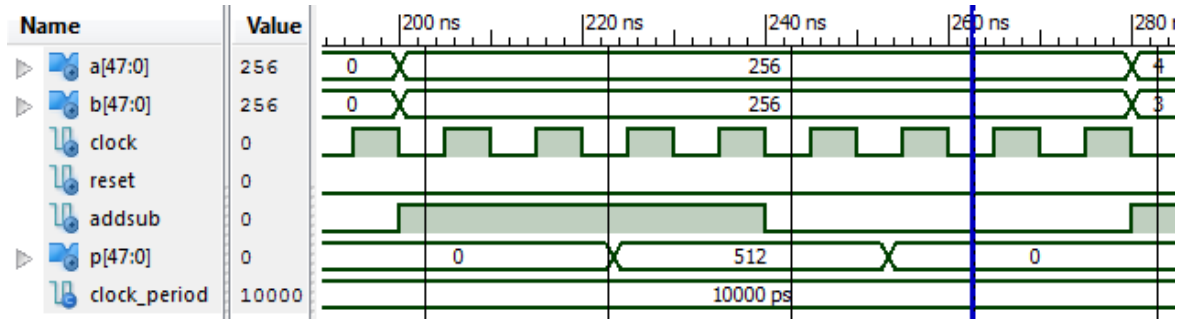


Figura 3-25a. Diagrama de tiempos sumador 48 bits.

Name	Value
AREG	1
AUTORESET_OVER_UNDER_FLOW	FALSE
AUTORESET_PATTERN_DETECT	FALSE
AUTORESET_PATTERN_DETECT_OPTINV	MATCH
A_INPUT	DIRECT
BCASCREG	1
BREG	1
B_INPUT	DIRECT
CARRYINREG	0
CARRYINSELREG	1
CLOCK_INVERT_M	SAME_EDGE
CLOCK_INVERT_P	SAME_EDGE
CREG	1
Instance Name	Madd_p1_add00001
LFSR_EN_SET	SET
LFSR_EN_SETVAL	0
MASK	3FFFFFFFFFFF
MREG	0
MULTCARRYINREG	1
OPMODEREG	0
PATTERN	000000000000
PREG	1

Figura 3-25b. Atributos inferidos para el sumador de números de 48 bits.

3.4.4 Suma doble

La ruta de datos será similar al caso del sumador anterior. Modifico el único proceso del código del sumador de la siguiente manera:

```
process (clock) is
begin

p1 <= a1+ a1 + b1;--Suma doble

if clock'event and clock = '1' then
      if reset = '1' then
            p <= (others => '0');
            a1 <= (others => '0');
            b1 <= (others => '0');
      else
            p <= p1;
            a1 <= a;
            b1 <= b;
      end if;
end if;
```

Como se ha comprobado en el Synthesis Report y en el esquemático tecnológico (se omite el Synthesis Report ya que es similar al del diseño del sumador), con dicho código, XST infiere un diseño con 2 etapas de registros y un único DSP. La configuración interna del bloque DSP implementa la doble suma. Se procede a simular el diseño obteniéndose el comportamiento mostrado en la figura 3-26.

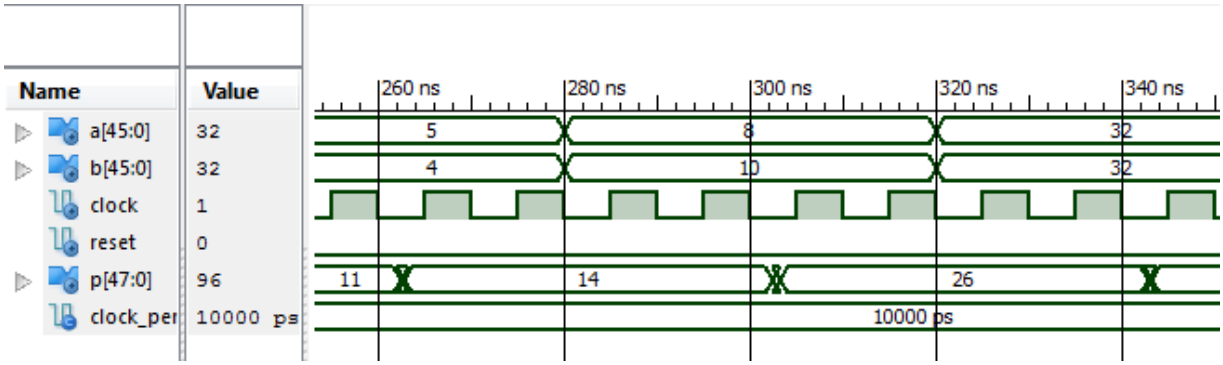


Figura 3-26. Diagrama de tiempos suma doble.

3.4.5 Sumador acumulador

Modifico de nuevo el código del proceso principal del sumador para implementar un sumador acumulador de 2 etapas.

```
process (clock) is
begin

p1 <=a1 + b1 + p1;--Suma doble


if clock'event and clock = '1' then
      if reset = '1' then
            p <= (others => '0');
            a1 <= (others => '0');
            b1 <= (others => '0');
      else
            p <= p1;
            a1 <= a;
            b1 <= b;
      end if;
end if;
```

XST infiere un diseño que hace uso de CLBs (que implementan un registro) y de 2 DSP, uno que actúa de acumulador y otro de sumador.

```
Advanced HDL Synthesis Report
Macro Statistics
# Adders/Subtractors                                   : 2
 48-bit adder                                         : 1
 48-bit registered adder                              : 1
# Registers                                            : 48
       Flip-Flops                                            : 48
```

Se han intentado múltiples variaciones del código anterior para obtener la síntesis del diseño en un DSP48E pero no se ha conseguido. Sin embargo si es posible configurar un DSP para obtener un sumador acumulador (ver [37] Pág. 76). También se ha conseguido eliminar el registro de 48 bits que se infería fuera del DSP48E (eliminando la señal *p* y poniendo como puerto *inout* la señal *p1* del mismo diseño), consiguiendo así un DSP con la ruta de datos mostrada en la figura 3-27.

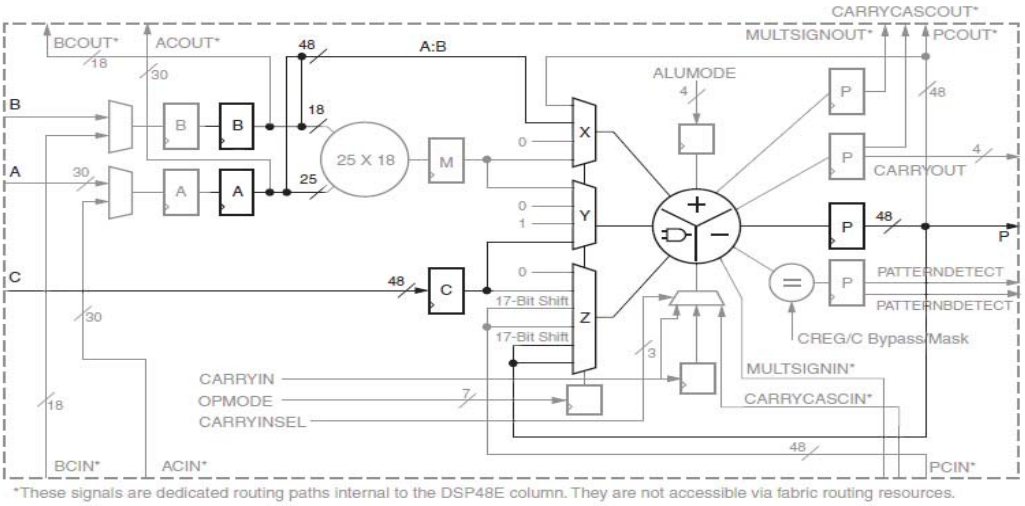


Figura 3-27. Ruta de datos del sumador acumulador.

3.4.6 Multiplicador acumulador (MACC)

Realizamos una modificación del código del multiplicador sumador/restador para implementar un multiplicador acumulador de 3 etapas de registros (ver figura 3-28).

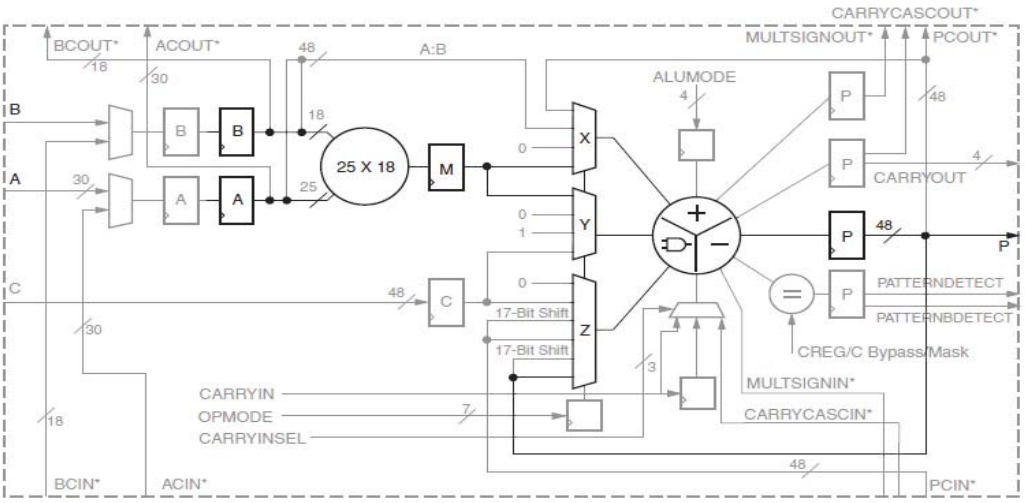


Figura 3-28. Ruta de datos MACC.

Al igual que en el caso anterior hemos hecho uso de un puerto de entrada/salida (inout) para la señal *p*. Eliminando este nivel de registros en el código se consigue inferir el diseño segmentado esperado (ver figura 3-29).

```
entity macc_3reg is
      port (
            a : in std_logic_vector (15 downto 0);
            b : in std_logic_vector (15 downto 0);
            clock : in std_logic;
            reset : in std_logic;
            p : inout std_logic_vector (31 downto 0)
      );
end entity;
architecture macc_3reg_arch of macc_3reg is
signal  p1 : std_logic_vector (31 downto 0);
signal a1 : std_logic_vector (15 downto 0);
signal b1 : std_logic_vector (15 downto 0);
begin
process (clock) is
begin
if clock'event and clock = '1' then
      if reset = '1' then
            p <= (others => '0');
            a1 <= (others => '0');
            b1 <= (others => '0');
      else
            p1 <= a1*b1;--Importante, si esto lo sacamos fuera del
                        --ifelse no se infiere el registro M.
            p <= p1 + p;-- p2<=a1*b1 está en el process
            a1 <= a;
            b1 <= b;
      end if;
end if;
end process;
end architecture;
```

XST infiere un diseño que hace uso 1 DSP y una LUT. Tal como se dice en un comentario del código, si sacamos p1 fuera del bloque de código síncrono XST no infiere el registro M para la multiplicación.

Name	Value
AREG	1
BCASCREG	1
BREG	1
MREG	1
PREG	1
MASK	3FFFFFFFFFFF
A_INPUT	DIRECT
B_INPUT	DIRECT
Type	DSP48E
AUTORESET_OVER_UNDER_FLOW	FALSE
USE_MULT	MULT_S

Figura 3-29. Registros inferidos para el MACC.

Respecto al diagrama de tiempos (ver figura 3-30) destacamos que el resultado en general se obtiene con 3 ciclos de delay toda vez que se han establecido los valores de entrada en los puertos. Sin embargo, en el caso en que no cambian las entradas para la siguiente ejecución se obtiene el resultado con 1 ciclo de delay. Esto es así ya que en ese caso se tiene calculado el producto de la ejecución anterior y solo es necesario realizar la suma de los productos parciales y el registro acumulador.

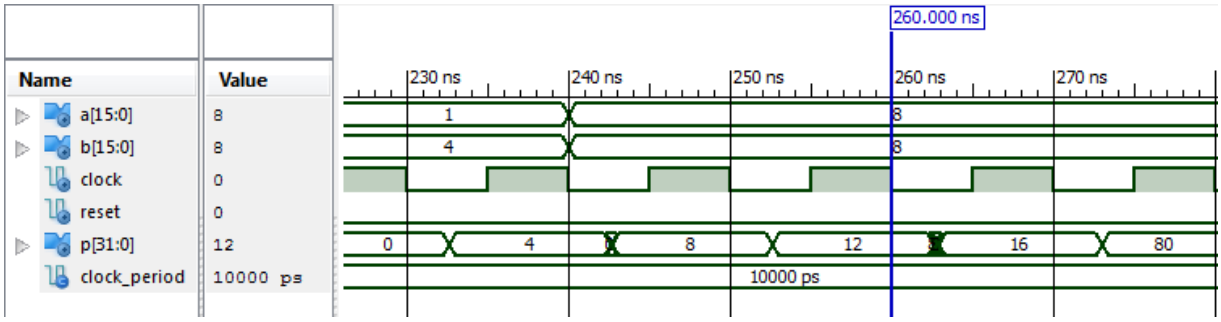


Figura 3-30. Diagrama de tiempos MACC.

3.4.7 Multiplicador Acumulador Cargable

El siguiente código ha sido sacado de [38]. Se trata de un MACC que puede cargarse con un valor inicial.

```
entity load_mult_accum_1reg is
      port (
            a : in std_logic_vector (15 downto 0);
            b : in std_logic_vector (15 downto 0);
            c : in std_logic_vector (31 downto 0);
            p_rst : in std_logic;
            p_ce : in std_logic;
            clk : in std_logic;
            load : in std_logic;
            p : out std_logic_vector (31 downto 0));
end entity;

architecture load_mult_accum_1reg_arch of load_mult_accum_1reg is

signal a1 : signed (15 downto 0);
signal b1 : signed (15 downto 0);
signal p_tmp : signed (31 downto 0);
signal p_reg : signed (31 downto 0);
```

```
begin
with load select p_tmp <= signed(c) when '1' ,
                          p_reg + a1*b1 when others;

process(clk)
begin
if clk'event and clk = '1' then

      if p_rst = '1' then
            p_reg <= (others => '0');
            a1 <= (others => '0');
            b1 <= (others => '0');
      elsif p_ce = '1' then
            p_reg <= p_tmp;
            a1 <= signed(a);
            b1 <= signed(b);
      end if;
end if;
end process;
p <= std_logic_vector(p_reg);
end architecture;
```

En este caso se infieren dos DSP (uno actuando como multiplicador y otro como acumulador) y un registro (registro p para el acumulador) y se obtiene un diseño con una frecuencia de 222.856 Mhz. Podemos ver el funcionamiento en la figura 3-31.

```
Minimum period: 4.487ns (Maximum Frequency: 222.856MHz)
Minimum input arrival time before clock: 3.316ns
Maximum output required time after clock: 2.822ns
Maximum combinational path delay: No path found
```

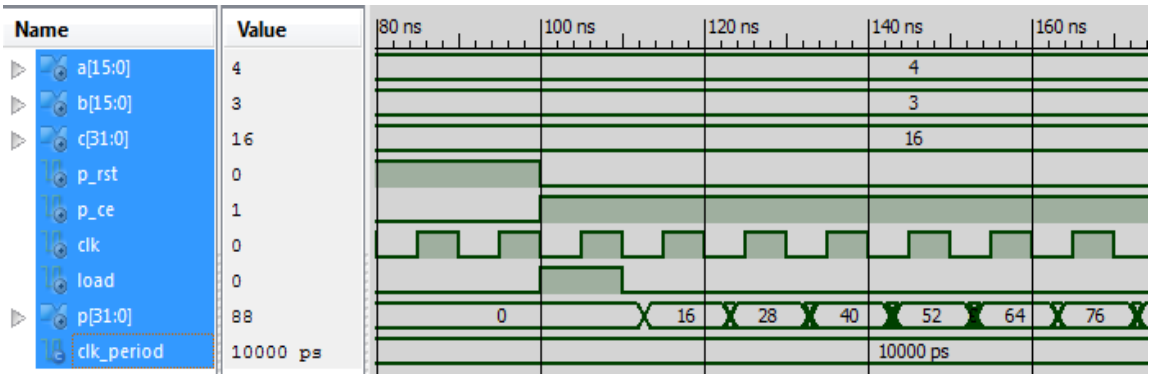


Figura 3-31. Diagrama de tiempos para MACC cargable.

3.5 Plantillas para macro DSP48E

Hemos visto que XST es capaz de inferir los bloques DSP48E automáticamente a partir de código VHDL bajo ciertas condiciones del código. Sin embargo, como veremos en el siguiente capítulo, no se consigue inferir el modo SIMD. Tampoco se consigue obtener una configuración específica para los DSP48E tal como ha sucedido con la multiplicación más suma/resta dinámica. Por lo que si queremos conseguir exactamente un DSP48E configurado acorde a nuestras necesidades tendremos que usar la plantilla de instanciación que ofrece Xilinx en [39] (y la cual podemos ver a continuación de la siguiente figura), cuyas entradas, salidas y atributos se muestran en la figura 3-32.

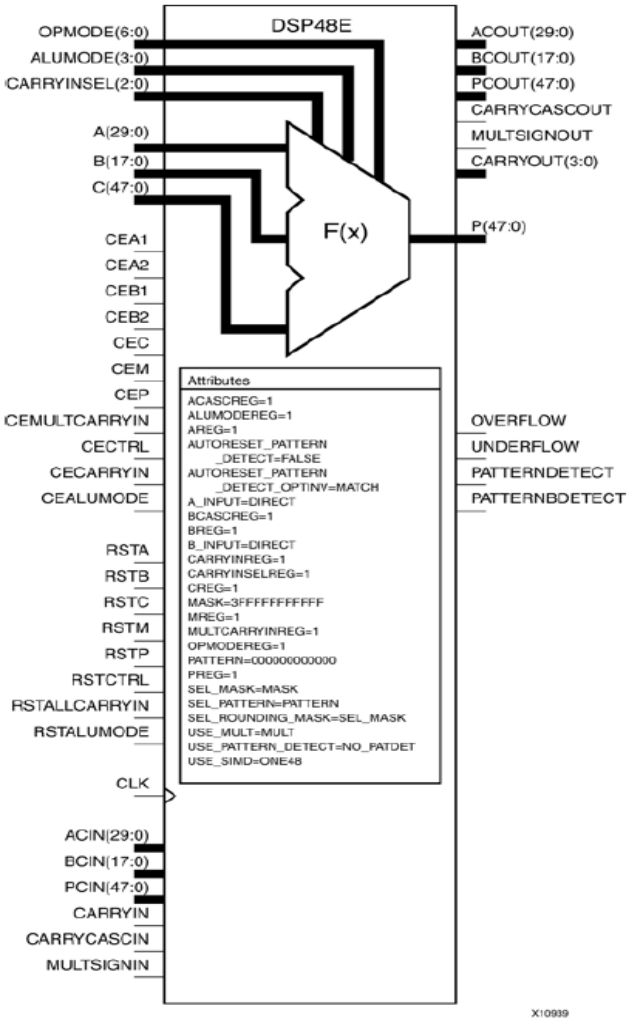


Figura 3-30. Diagrama de entradas, salidas y atributos del bloque DSP48E.

La sección de código correspondiente al *generic map* contiene los atributos de configuración del DSP48E y la sección *port map* contiene los puertos de entrada y/o salida del componente, entre los que se encuentran ALUMODE y OPMODE (ver figura 3-30).

```
-- DSP48E: DSP Function Block
--         Virtex-5
-- Xilinx HDL Libraries Guide, version 12.1

DSP48E_inst : DSP48E
generic map (
      ACASCREG => 1,       -- Number of pipeline registers between
                              -- A/ACIN input and ACOUT output, 0, 1, or 2
      ALUMODEREG => 1,     -- Number of pipeline registers on ALUMODE input,
                              --  0 or 1
      AREG => 1,           -- Number of pipeline registers on the A input, 0,
                              --1 or 2
      AUTORESET_PATTERN_DETECT => FALSE, -- Auto-reset upon pattern detect,
                                    -- TRUE or FALSE
      AUTORESET_PATTERN_DETECT_OPTINV => "MATCH", -- Reset if "MATCH" or
                                                  --"NOMATCH"
      A_INPUT => "DIRECT", -- Selects A input used, "DIRECT" (A port) or
                             -- "CASCADE" (ACIN port)
      BCASCREG => 1,       -- Number of pipeline registers between B/BCIN
                             -- input and BCOUT output, 0, 1, or 2
      BREG => 1,           -- Number of pipeline registers on the B input, 0,
                             -- 1 or 2
      B_INPUT => "DIRECT", -- Selects B input used, "DIRECT" (B port) or
                             -- "CASCADE" (BCIN port)
      CARRYINREG => 1,     -- Number of pipeline registers for the CARRYIN
                             -- input, 0 or 1
      CARRYINSELREG => 1,  -- Number of pipeline registers for the CARRYINSEL
                             -- input, 0 or 1
      CREG => 1,           -- Number of pipeline registers on the C input, 0
                             -- or 1
      MASK => X"3FFFFFFFFFFF", -- 48-bit Mask value for pattern detect
      MREG => 1,           -- Number of multiplier pipeline registers, 0 or 1
      MULTCARRYINREG => 1, -- Number of pipeline registers for multiplier
                             -- carry in bit, 0 or 1
      OPMODEREG => 1,      -- Number of pipeline registers on OPMODE input, 0
                             -- or 1
      PATTERN => X"000000000000", -- 48-bit Pattern match for pattern detect
      PREG => 1,           -- Number of pipeline registers on the P output, 0
                             -- or 1
      SIM_MODE => "SAFE", -- Simulation: "SAFE" vs "FAST", see "Synthesis and
                           -- Simulation Design Guide" for details


      SEL_MASK => "MASK",  -- Select mask value between the "MASK" value or
                             -- the value on the "C" port
      SEL_PATTERN => "PATTERN", -- Select pattern value between the "PATTERN"
                                -- value or the value on the "C" port
      SEL_ROUNDING_MASK => "SEL_MASK", -- "SEL_MASK", "MODE1", "MODE2"
      USE_MULT => "MULT_S", -- Select multiplier usage, "MULT" (MREG => 0),
```

```
                                -- "MULT_S" (MREG => 1), "NONE" (not using
                                -- multiplier)
      USE_PATTERN_DETECT => "NO_PATDET", -- Enable pattern detect, "PATDET",
                                         -- "NO_PATDET"
      USE_SIMD => "ONE48") -- SIMD selection, "ONE48", "TWO24", "FOUR12"

port map (
      ACOUT => ACOUT,  -- 30-bit A port cascade output
      BCOUT => BCOUT,  -- 18-bit B port cascade output
      CARRYCASCOUT => CARRYCASCOUT, -- 1-bit cascade carry output
      CARRYOUT => CARRYOUT, -- 4-bit carry output
      MULTSIGNOUT => MULTSIGNOUT, -- 1-bit multiplier sign cascade output
      OVERFLOW => OVERFLOW, -- 1-bit overflow in add/acc output
      P => P,          -- 48-bit output
      PATTERNBDETECT => PATTERNBDETECT, -- 1-bit active high pattern bar
                                        -- detect output
      PATTERNDETECT => PATTERNDETECT, --  1-bit active high pattern detect
                                      -- output
      PCOUT => PCOUT,  -- 48-bit cascade output
      UNDERFLOW => UNDERFLOW, -- 1-bit active high underflow in add/acc
                              -- output
      A => A,          -- 30-bit A data input
      ACIN => ACIN,    -- 30-bit A cascade data input
      ALUMODE => ALUMODE, -- 4-bit ALU control input
      B => B,          -- 18-bit B data input
      BCIN => BCIN,    -- 18-bit B cascade input
      C => C,          -- 48-bit C data input
      CARRYCASCIN => CARRYCASCIN, -- 1-bit cascade carry input
      CARRYIN => CARRYIN, -- 1-bit carry input signal
      CARRYINSEL => CARRYINSEL, -- 3-bit carry select input
      CEA1 => CEA1,     -- 1-bit active high clock enable input for 1st stage
                        -- A registers
      CEA2 => CEA2,      -- 1-bit active high clock enable input for 2nd
                         -- stage A registers
      CEALUMODE => CEALUMODE, -- 1-bit active high clock enable input for
                              -- ALUMODE registers
      CEB1 => CEB1,   -- 1-bit active high clock enable input for 1st stage B
                      -- registers
      CEB2 => CEB2,   -- 1-bit active high clock enable input for 2nd stage B
                      -- registers
      CEC => CEC,     -- 1-bit active high clock enable input for C
                      -- registers
      CECARRYIN => CECARRYIN, -- 1-bit active high clock enable input for
                              -- CARRYIN register
      CECTRL => CECTRL, -- 1-bit active high clock enable input for OPMODE
                        -- and carry registers
      CEM => CEM,       -- 1-bit active high clock enable input for
                        -- multiplier registers
      CEMULTCARRYIN => CEMULTCARRYIN,  -- 1-bit active high clock enable for
                                       -- multiplier carry in register
      CEP => CEP,       -- 1-bit active high clock enable input for P
                        -- registers
      CLK => CLK,       -- Clock input
      MULTSIGNIN => MULTSIGNIN, -- 1-bit multiplier sign input
      OPMODE => OPMODE, -- 7-bit operation mode input
      PCIN => PCIN,     -- 48-bit P cascade input
```

```
      RSTA => RSTA,     -- 1-bit reset input for A pipeline registers
      RSTALLCARRYIN => RSTALLCARRYIN, -- 1-bit reset input for carry pipeline
                                      --registers
      RSTALUMODE => RSTALUMODE, -- 1-bit reset input for ALUMODE pipeline
                                --registers
      RSTB => RSTB,     -- 1-bit reset input for B pipeline registers
      RSTC => RSTC,     -- 1-bit reset input for C pipeline registers
      RSTCTRL => RSTCTRL, -- 1-bit reset input for OPMODE pipeline
                          -- registers
      RSTM => RSTM, -- 1-bit reset input for multiplier registers
      RSTP => RSTP  -- 1-bit reset input for P pipeline registers
);

-- End of DSP48E_inst instantiation
```

El uso de las plantillas HDL nos permite una mayor versatilidad y especificidad para nuestro diseño debido a que con ellas le indicamos a XST la configuración exacta que queremos para el DSP48E y lo más importante, aunque añadamos nuevas partes a nuestro código la configuración del DSP48E no se verá afectada.

Capítulo 4

Metodología

En este capítulo se presenta la metodología para que a partir de un código VHDL consigamos una especificación sintetizable por la herramienta XST en módulos DSP de la FPGA. Para ilustrar la metodología se presenta un ejemplo de un diseño real sobre el que se realizarán diferentes modificaciones y el cual se sintetizará para comprobar como responde XST. Debido a la dificultad o imposibilidad de obtener ciertas configuraciones para el DSP48E utilizando un determinado estilo de código se planteará el uso de la plantilla de macro DSP48E. Así mismo se propondrá una metodología para insertar las plantillas en nuestro diseño.

4.1 Ejemplo de un benchmark real

Se presenta como ejemplo el diseño b05 perteneciente al conjunto de benchmarks ITC99, el cual podemos encontrar en [40]. A continuación mostramos parcialmente el código original del diseño b05 y vamos a ir explicando los cambios de código que se realizan para intentar que XST infiera los DSP48E tal como queremos.

```
entity b05 is
       port(
            CLOCK: in bit;
            RESET: in bit;
            START: in bit;
            SIGN: out bit;
            DISPMAX1,DISPMAX2,DISPMAX3: out bit_vector (6 downto 0);
            DISPNUM1,DISPNUM2: out bit_vector (6 downto 0)
           );
end b05;
```

```
architecture BEHAV of b05 is

constant st0:integer:=0;
constant st1:integer:=1;
constant st2:integer:=2;
constant st3:integer:=3;
constant st4:integer:=4;

subtype memdim is integer range 31 downto 0;
subtype num9bits is integer range 255 downto -256;
type rom is array (0 to 31) of num9bits;

signal NUM: memdim;
signal MAR: memdim;
signal TEMP: num9bits;
signal MAX: num9bits;
signal FLAG,MAG1,MAG2,MIN1: bit;
signal EN_DISP,RES_DISP: bit;

 constant MEM: rom :=( 50,40,0,229,-10,75,229,181,186,229,186,
                       -11,0,40,50,-29,-18,229,229,151,229,100,
                       125,10,75,-50,0,-22,0,40,50,50 );

begin
     proceso1: process(MAR,TEMP,MAX)
     variable AC1,AC2: num9bits;
     begin
          AC1:= MEM(MAR)-TEMP;


          if AC1<0 then
             MIN1 <= '1';
             MAG1 <= '0';
          else
             if AC1 = 0 then
                MIN1 <= '0';
                MAG1 <= '0';
             else
                MIN1 <= '0';
                MAG1 <= '1';
             end if;
          end if;
          AC2 := MEM(MAR)-MAX;
          if (AC2<0) then
             MAG2 <= '1';
          else
             MAG2 <= '0';
          end if;
     end process;

     proceso2: process (EN_DISP,RES_DISP,NUM,MAX)
     variable TM,TN: num9bits;
     begin
```

```
            if EN_DISP = '1' then
             ...
            else
             if RES_DISP ='0' then
               ...
             else
               TN := NUM;
               if MAX<0 then
                  SIGN <= '1';
                  TM := -MAX mod 2**5;
               else
                  SIGN <= '0';
                  TM := MAX mod 2**5;
               end if;
               if TM> 99 then
                  DISPMAX1 <= "0011000";
                  TM := TM - 100;
               else
                  DISPMAX1 <= "0111111";
               end if;
               if TM > 89 then
                  DISPMAX2 <= "1111110";
                  TM := TM - 90;
               else
                  if TM > 79 then
                     DISPMAX2 <= "1111111";
                     TM := TM - 80;
                  else

                        ...-—Multiples if else anidados con similares
                           --instrucciones
                    end if;
               end if;

               if TM > 8 then
                  DISPMAX3 <= "1111110";
               else
                        ...-—Multiples if else anidados con similares
                           --instrucciones
               end if;

               if TN > 9 then
                  DISPNUM1 <= "0011000";
                  TN := TN - 10;
               else
                  DISPNUM1 <= "0111111";
               end if;

               if TN > 8 then
                  DISPNUM2 <= "1111110";
               else
                        ...-—Multiples if else anidados con similares
                           --instrucciones
               end if;
            end if;
         end process;
```

```
proceso3: process (CLOCK,RESET)
variable STATO: integer range 0 to 4;
variable TMN : bit_vector (5 downto 0);
begin
     if RESET = '1' then
       STATO := st0;
       RES_DISP <= '0';
       EN_DISP <= '0';
       NUM <= 0;
       MAR <= 0;
       TEMP <= 0;
       MAX <= 0;
       FLAG <= '0';

     elsif CLOCK'event and CLOCK='1' then
           case STATO is
               when st0 =>
                RES_DISP <= '0';
                EN_DISP <= '0';
                STATO := st1;
               when st1 =>
                if START = '1' then
                   NUM <= 0;
                   MAR <= 0;
                   FLAG <= '0';
                   EN_DISP <= '1';


                   RES_DISP <= '1';
                   STATO := st2;
                else
                   STATO := st1;
                end if;
               when st2 =>
                MAX <= MEM(MAR);
                TEMP <= MEM(MAR);
                STATO := st3;
               when st3 =>
                if MIN1 = '1' then
                   if FLAG ='1' then
                      FLAG <= '0';
                      NUM <= NUM+1;--Contador
                   end if;
                else
                   if MAG1 = '1' then
                      if MAG2 = '1' then
                         MAX <= MEM(MAR);
                      end if;
                      FLAG <= '1';
                   end if;
                end if;
                TEMP <= MEM(MAR);
                STATO := st4;
```

```
                        when st4 =>
                           if MAR = 31 then
                              if START = '1' then
                                 STATO := st4;
                              else
                                 STATO := st1;
                              end if;
                              EN_DISP <= '0';
                           else
                              MAR <= MAR+1; --Contador
                              STATO := st3;
                           end if;
                  end case;
                  end if;
            end process;
      end BEHAV;
```

Como podemos ver en el código anterior se realizan 14 operaciones de suma/resta y el tamaño de las variables implicadas en estas operaciones es de 9 bits en 11 de ellas y de 5 bits en las 3 operaciones restantes.

El diseño se compone de 3 procesos. Respecto a las operaciones de suma proceso1 es el encargado de calcular el valor de las variables AC1 y AC2, proceso2 calcula el valor de TM y TN y proceso3 implementa los contadores NUM y MAR. Proceso1 y proceso2 son procesos asíncronos y proceso3 es síncrono. Se sintetiza el diseño con la opción USE_DSP=auto (por lo que no se mapearán sumas en los DSP48E) obteniendo una frecuencia de 262.557 MHz y un área de 185 pares LUT-Flip Flop (LUT-FF). En caso de cambiar la opción a USE_DSP=yes obtenemos un diseño con una frecuencia de 220.197 MHz y un área de 182 pares LUT-FF y 9 DSP48E.

Como hemos explicado anteriormente para que se infieran registros de nuestro código dentro del DSP48E estos registros deben ser síncronos, por lo que realizamos las siguientes modificaciones en el código para convertir las restas del proceso1 en restas síncronas.

1. Dividimos el proceso1 en los procesos proceso1a y proceso1b. El proceso1a implementa las 2 restas y el proceso1b implementa el resto de la lógica de proceso1.
2. AC1 y AC2 pasan a ser señales. En este punto se realiza una síntesis y se obtiene una frecuencia de 291.754MHz y un área de 169 pares LUT-FF sin DSPs y una frecuencia de 228.623MHz y un área de 163 pares y 7 DSP48E Con DSPs.

3. Hacemos la modificación de la figura 4-1 para AC1 (y una equivalente para AC2):

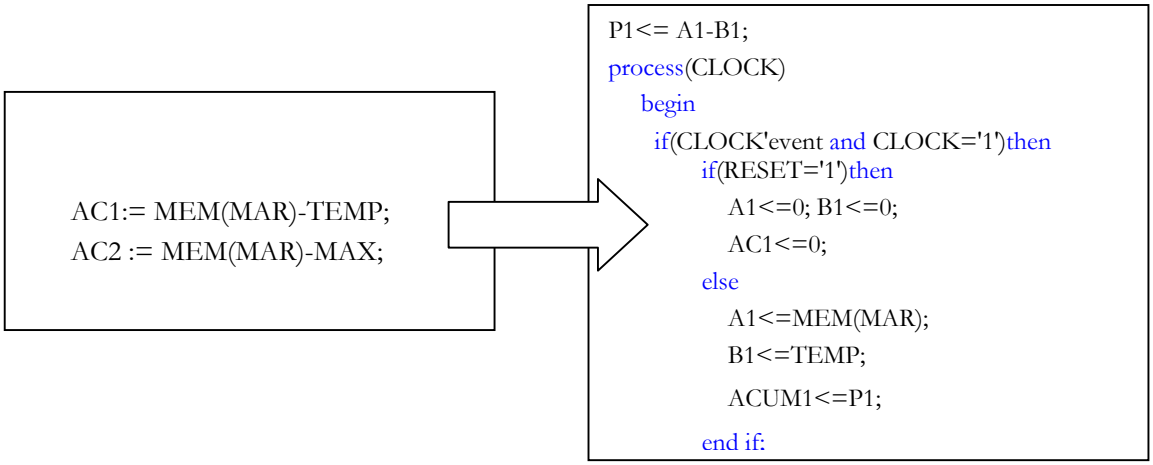


Figura 4-1. Paso de asíncrono a síncrono.

Con dicha modificación obtenemos un diseño con una frecuencia de 432.395 MHz y un área de 185 pares LUT-FF sin DSPs y una frecuencia de 382.029 MHz y un área de 191 pares LUT-FF y 7 DSP48E con DSPs. Los DSPs inferidos para realizar las restas (AC1 y AC2) tienen la configuración mostrada en la tabla 4-1.

Tabla 4-1. Configuración de los DSP48E para las restas AC1 y AC2.

Atributos	Valores AC1	Valores AC1
AREG	1	1
BREG	0	0
CREG	0	0
MREG	0	0
PREG	0	1
AINPUT	DIRECT,	DIRECT,
BINPUT	DIRECT	DIRECT
USE_MULT	NONE	NONE
USE_SIMD	ONE48	ONE48

En ambos casos se infiere el registro A y en ninguna se infiere el registro B. Observamos que para un código idéntico se infieren distintas configuraciones del DSP, ya que para AC1 no se infiere el registro P del DSP, sino que se infiere un FlipFlop externo a pesar de ser tanto AC1 como AC2 señales síncronas.

Vamos a realizar ahora una modificación del proceso proceso1a que compacte en una señal cada par de señales homólogas involucradas en las restas AC1 y AC2. Esto se hace para comprobar si con esta construcción XST infiere los DSP48E en modo SIMD. La modificación deja el proceso1a tal como sigue:

```
P(17 downto 9)<= A(17 downto 9)-B(17 downto 9);

P(8 downto 0) <= A(8 downto 0)-B(8 downto 0);

proceso1a: process(CLOCK)
     begin
            if (CLOCK'event and CLOCK='1')then
                  if(RESET='1')then
                        A<=(others=>'0');
                        B<=(others=>'0');
                     ACUM<=(others=>'0');
                  else
                    A<=std_logic_vector(to_unsigned(MEM(MAR),9))&
                       std_logic_vector(to_unsigned(MEM(MAR),9));
                    B<=std_logic_vector(to_unsigned(MAX,9)) &
                       std_logic_vector(to_unsigned(TEMP,9));
                    AC1(17 downto 9)<=conv_integer(P(17 downto 9));
                    AC2(8 downto 0)<=conv_integer(P(8 downto 0));
                  end if;
             end if;
end process;
```

Con este código se obtienen resultados similares en frecuencia pero no se han inferido las 2 restas en un solo DSP48E, sino que se han inferido de manera similar a la anterior modificación. Dado que no hemos podido obtener una configuración idéntica para los DSP48E, aún partiendo del mismo código, y tampoco somos capaces de obtener un DSP48E que funcione en modo SIMD utilizamos en el apartado 4.5 las plantillas de macro DSP48E para así obtener el diseño que buscamos.

4.2 Consideraciones iniciales

A la hora de definir una metodología hay múltiples opciones y decisiones de diseño que tomar y es necesario encontrar un equilibrio entre versatilidad y sencillez. Además, en nuestro caso, es obligatorio que el código obtenido después de aplicar la metodología siga siendo sintetizable. Para este trabajo solo vamos a considerar operaciones de suma/resta y multiplicación y entre las decisiones principales a tomar están:

- ¿Usar SIMD?
- ¿Mezclar diferentes tipos de operación en un DSP?
- ¿Compartir DSPs? ¿En qué condiciones?
- Numero de etapas del pipeline del DSP48E. De 0 (combinacional, delay de 1 ciclo) a 1, 2 o 3 (solo para multiplicaciones).

Respondiendo a la primera pregunta, en general es bueno para nuestro diseño hacer uso del modo SIMD ya que ahorramos área.

Respecto a la mezcla de sumas y restas en un mismo DSP48E hay que decir que existen determinadas condiciones que permiten hacerlo (gracias a ALUMODE), no obstante en general no es posible sin hacer uso de lógica programable extra, por lo que sería más lógico dejar fuera del DSP la operación que requiera esa lógica extra dejando así libre el slot del DSP. Sin embargo, en el caso de sumas y restas de constantes tiene el mismo coste A+k que A-k.

Si se comparten o no DSPs y la manera en la que se comparten es una cuestión que requiere un estudio en profundidad para obtener una metodología clara. En nuestro caso haremos una primera aproximación mediante la introducción de los conceptos de **slot**, **instancia de slot** y **ámbito de una instancia de slot**.

Es responsabilidad del programador el establecer el número de etapas de cada operación y diseñar el sistema teniendo en cuenta dicho número a la hora de obtener el resultado de cada operación. En nuestro caso, dado que realizamos el estudio con diseños que no implementamos se realiza el estudio con DSPs sin etapas de pipeline para no cambiar, o limitar el cambio en, la funcionalidad del diseño original. De igual modo las transformaciones hechas en los códigos al aplicar la metodología están dirigidas a mantener el comportamiento original del benchmark.

4.3 Definiciones

Antes de pasar a explicar la metodología y de mostrar su aplicación sobre el benchmark b05 se hace necesario en este punto introducir los siguientes conceptos, los cuales forman una jerarquía mostrada en la figura 4-2.

Clase de operación (CO): multiconjunto de **conjuntos instancia de clase (CIC)** con características similares. Dada una clase de operación sus **CICs** pueden ser unitarios o no representando cada uno de ellos una o varias **instancias de clase**. El tipo de una clase de operación se corresponde con una plantilla específica y se incluye como un componente en el código VHDL. Define el tipo de entradas y salidas de la operación.

Ejemplo: La clase de operación de suma de 2 números de 48 bits contiene elementos que se corresponden con la plantilla VHDL que implementa dicha operación.

```
component dsp_adder48Comb IS
PORT (
    AIN1 : IN std_logic_vector(47 DOWNTO 0);
    BIN1 : IN std_logic_vector(47 DOWNTO 0);
    CLK  : IN std_logic;
    RST  : IN std_logic;
    OUT1 : OUT std_logic_vector(47 DOWNTO 0));
END component;
```

Conjunto instancia de clase (CIC): Conjunto de operaciones dentro de una clase de operación. Este conjunto se dividirá en uno o varios subconjuntos denominados **instancias de clase.** Toda instancia de clase especifica los registros de entrada y salida que deben ser del tipo que especifica la clase. En código VHDL se corresponde cada instancia de clase con un *port map* y sus señales de reloj y reset asociadas.

Ejemplo: Una instancia de la clase de operación de suma de 2 números de 48 bits se corresponde con el *port map* del componente que implementa dicha operación y sus conexiones de reloj y reset.

```
dsp_adder48_1: dsp_adder30Comb port map (AIN1=>A4,
                                         BIN1=>B4,
                                         CLK=>CLOCK,
                                         RST=>RESET,
                                         OUT1=>P4);
```

Slot: hueco en una instancia de clase para realizar una operación. Un slot viene definido por su(s) entrada(s), su salida y la instancia de clase a la que pertenece. Se considera que el slot está desocupado, y por lo tanto puede usarse, si no hay una operación que lo use en el ámbito actual.

Ejemplo: La instancia anterior tiene un solo slot que viene definido por A4, B4 y P4. A4 y B4 se suman y se guarda el resultado en P4 (A4, B4->P4). Cuando usamos SIMD TWO24 tenemos 2 slots de 24 bits y 4 slots de 12 bits en el caso de FOUR12.

Instancia de un slot: Al igual que un port map es una instancia de clase**,** las conexiones de los registros de entrada y de salida de un slot representan una instancia de un slot. Una instancia de slot es equivalente a una operación.

Ejemplo: Siguiendo el ejemplo de la instancia de clase anterior, en este ejemplo la instancia del slot está formada por las tres instrucciones siguientes:

```
A4<=Signal1; B4<=Signal2; Result<=P4;--Conexiones
```

Ámbito: el ámbito de una instancia de slot u operación define cuando está libre el slot instanciado para poder ser usado por otra operación. Podrá ser un ámbito global, esto es, una vez ocupado el slot no se podrá usar por otra instancia, o podrá ser un ámbito local, esto es, el slot se considera libre si no ha sido ocupado en la sección de código actual, en una sección de código más general que englobe la actual o si no ha sido ocupado en otro process.

El estudio en profundidad del ámbito, y por lo tanto de la formación de los conjuntos de una clase, es una cuestión que se deja para estudios posteriores. En el estudio actual consideramos que diferentes process y el código combinacional comparten ámbito entre si para evitar conflictos de recursos al igual que diferentes if-else y case.

Ejemplo: En este ejemplo tenemos dos operaciones que pertenecen al mismo conjunto de la misma clase. Dado que se llaman dentro de un if-else tienen un ámbito distinto, por lo que la instancia podrá ser ocupada por una u otra operación en sus respectivos ámbitos si así se quiere.

```
...
if TM > 89 then
          DISPMAX2 <= "1111110";
          TM := TM - 90;
else if TM > 79 then
             DISPMAX2 <= "1111111";
             TM := TM - 80;
     else
...
```

Cabe destacar que los términos **conjunto de instancia de clase** e **instancia de clase** no son términos equivalentes. Podemos decir que un conjunto de instancia de clase se puede dividir en subconjuntos llamados instancias de clase que se corresponden con las llamadas a plantillas de macro DSP48E en el código VHDL.

Creación de CICs

Como hemos dicho, el estudio en profundidad del **ámbito** y por lo tanto la formación de **CICs**, se deja para estudios posteriores. A falta de una heurística completamente definida, en este trabajo se han seguido las siguientes recomendaciones a la hora de crear **CICs**:

- Hacemos diferenciación de **COs** (+, - y *).
- Dentro de una misma **CO**:
    - Operaciones con **mismo ámbito:** En **general** van en **CICs diferentes.**
        - Pero operaciones con **mismo ámbito** y mismas variables de entrada se asignan a un mismo slot**.** El slot ya no usará otras variables de entrada.
    - Operaciones con **distinto ámbito:** Pueden ir en un mismo CIC o no, según las características del diseño. Se ha realizado la asignación a CICs teniendo en cuenta en primer lugar la afinidad entre operaciones respecto a sus registros de entrada y salida y en segundo lugar respecto al número de operaciones no afines que comparten el slot. Operaciones afines van en un mismo CIC, operaciones no afines pueden compartir o no CIC. Cuantas más operaciones no afines compartan un slot menor frecuencia se suele obtener en el diseño final.

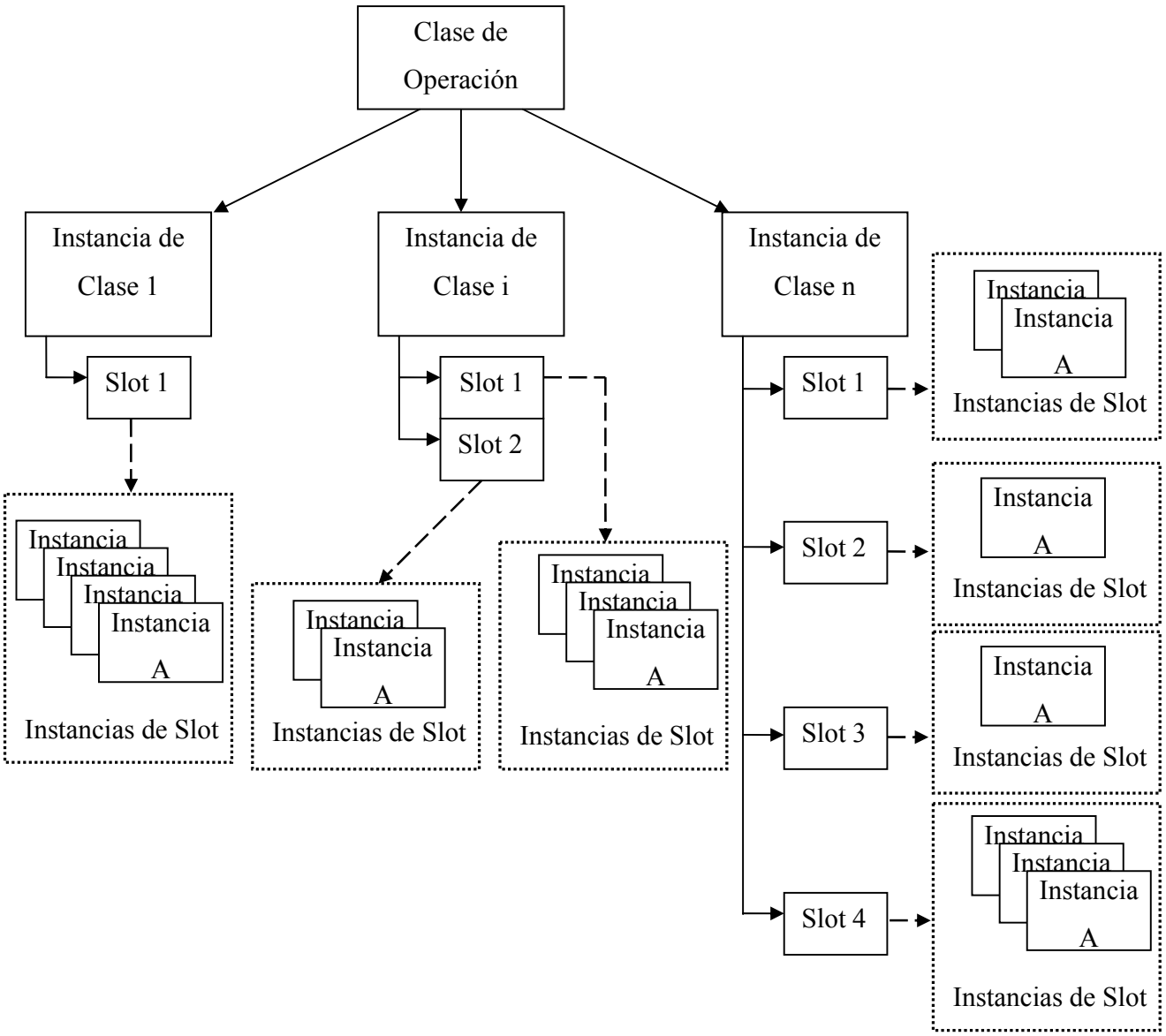


Figura 4-2. Ejemplo de jerarquía para una clase de operación.

4.4 Clases de operación

Vamos a distinguir clases de operación de suma/resta, multiplicación y multiplicación más suma/resta de 1, 2 y 3 entradas con 1 salida (ver tabla 4-2). En la presente memoria no se incluye el código que implementa estas clases. Sin embargo se dispone de tal código, que ha sido utilizado para realizar las pruebas, y si se precisa puede pedirse por correo electrónico al autor.

Tabla 4-2. Clases de operación contempladas.

Entradas y salida	**+**	**-**	*****	***,+**	***,-**
A,P->P	AACC	SACC	-	-	-
A,B,P->P	AACC2	SACC2	-	MACC	MSACC
A,A,B->P	DADD	-	-	-	-
A,B(up to 48 bits)->P	ADDER2	SUB2	MULT	-	-
A,B,C(up to 47 bits)->P	ADDER3	SUB3	-	MULT-ADDER	MULT-SUB
P, k->P	kUP-COUNT	kDOWN-COUNT		-	-

4.5 Metodología propuesta

1. Reconocer operaciones susceptibles de ejecutarse en el DSP48E (+, -, *).
2. Teniendo el tipo de la operación, los registros de entrada y salida y su anchura
    i. Asignamos cada operación a un conjunto instancia de clase dentro de su de clase de operación correspondiente y según su ámbito (Heurística)
3. Para cada clase de operación
    a. Inserto el código VHDL correspondiente a la clase
    b. Para cada conjunto de instancia de clase
        i. Inserto el código correspondiente a la instancia de clase
        ii. Para cada operación:
            1. Eliminamos la operación del conjunto instancia de clase
            2. Si no existe al menos un slot libre en la instancia de clase actual
                a. Añadimos una nueva instancia de clase al código
            3. Creamos una instancia de slot en el código
                a. Creamos las señales de un slot de la instancia de clase (señales de entrada y salida) para el DSP48E
                b. Conectamos las señales de la instancia de slot (operación) con las de la instancia de clase

- En el paso 1 buscamos y marcamos todas las operaciones, en este caso solo tenemos operaciones de suma y de resta (separamos con un salto de línea operaciones que comparten ámbito):

```
AC1:= MEM(MAR)-TEMP;

AC2 := MEM(MAR)-MAX;

TM := -MAX mod 2**5;

TM := TM - 100;

TM := TM - 90;
TM := TM - 80;
```

```
        TM := TM - 70;
        TM := TM - 60;
        TM := TM - 50;
        TM := TM - 40;
        TM := TM - 30;
        TM := TM - 20;
        TM := TM - 10;

        TN := TN – 10;

        NUM <= NUM+1;
        MAR <= MAR+1;
```

Tenemos únicamente 2 clases de operaciones, la clase SUB2 y la clase ADD2. Se usa la clase ADD2 en vez de la clase kUP-COUNT ya que ésta última se reserva para diseños que precisen contadores que necesiten la máxima velocidad que permite la FPGA. Tampoco se usa en este caso la clase SACC ya que TM está en una sección de código asíncrona en el benchmark b05, y SACC es síncrona ya que contiene el registro acumulador. En general para los test con diseños reales se van a usar clases generales de suma, resta y multiplicación, ya que son las clases más versátiles y que no requieren código específico para su inicialización. Sin embargo, a la hora de diseñar un sistema para luego aplicarle esta metodología si podremos crear el código adecuado para sacar el mayor provecho de clases más específicas.

Se ha decidido crear plantillas con entradas de 30 bits de anchura ya que en muchos diseños de los estudiados se usa el tipo INTEGER, el cual se implementa con 30 bits. Respecto a las operaciones de desplazamiento, éstas se aplican sobre la señal de salida de la instancia de slot y en la instrucción de conexión con el resto del diseño.

- En el paso 2 asignamos las operaciones anteriores a las 2 clases reconocidas estando operaciones que comparten ámbito en conjuntos de instancia de clase distintos:

SUB2= { {(TN,10->TN)}, {MEM(MAR),TEMP->AC1}, MEM(MAR),MAX->AC2},
{(0,MAX)[4..0]->TM)},{(TM,100->TM)},
{(TM,90->TM),(TM,80->TM), (TM,70->TM),
(TM,60->TM), (TM,50->TM), (TM,40->TM),
(TM,30->TM), (TM,20->TM), (TM,10->TM)} }

ADD2= { {(MAR,1->MAR)}, {(NUM,1->NUM)} }

- Finalmente llevamos a cabo el paso 3.
    - Insertamos el código de ambas clases de operación:

```
component dsp_adder30Comb IS --ADDER2
   PORT (
      AIN1 : IN std_logic_vector(29 DOWNTO 0);
      BIN1 : IN std_logic_vector(29 DOWNTO 0);
      CLK                        : IN std_logic;
      RST                        : IN std_logic;
      OUT1 : OUT std_logic_vector(30 DOWNTO 0));
END component;

component dsp_sub30Comb IS --SUB2
   PORT (
      AIN1 : IN std_logic_vector(29 DOWNTO 0);
      BIN1 : IN std_logic_vector(29 DOWNTO 0);
      CLK                        : IN std_logic;
      RST                        : IN std_logic;
      OUT1 : OUT std_logic_vector(30 DOWNTO 0));
END component;
```

    - Para el conjunto instancia de clase creo todas las instancias de clase y las señales necesarias, incluyendo las señales de las instancias de slot.

```
--señales para DSPs
signal A1, A2, A3, A4, B1, B2, B3, B4,
       A3b, B3b, A6, B6, A3c, B3c: std_logic_vector(29 downto 0);
signal P1, P2, P3, P4, P3b, P3c, P6: std_logic_vector(30 downto 0);
signal AC1,AC2: num9bits;

constant MEM: rom :=(50,...,-22,0,40,50,50 );

begin

      dsp_sub30_1: dsp_sub30Comb port map (AIN1=>A1,
                                           BIN1=>B1,
                                           CLK=>CLOCK,
                                           RST=>RESET,
                                           OUT1=>P1);

      A1<=std_logic_vector(to_signed(MEM(MAR),30));
      B1<=std_logic_vector(to_signed(TEMP,30));

      dsp_sub30_2: dsp_sub30Comb port map (AIN1=>A2,
                                           BIN1=>B2,
                                           CLK=>CLOCK,
                                           RST=>RESET,
                                           OUT1=>P2);
```

```
      A2<=std_logic_vector(to_signed(MEM(MAR),30));
      B2<=std_logic_vector(to_signed(MAX,30));

      dsp_sub30_3: dsp_sub30Comb port map (AIN1=>A3,
                                           BIN1=>B3,
                                           CLK=>CLOCK,
                                           RST=>RESET,
                                           OUT1=>P3);

      dsp_sub30_3b: dsp_sub30Comb port map (AIN1=>A3b,
                                           BIN1=>B3b,
                                           CLK=>CLOCK,
                                           RST=>RESET,
                                           OUT1=>P3b);
      A3b<=std_logic_vector(to_signed(0,30));
      B3b<=std_logic_vector(to_signed(MAX,30));

      dsp_sub30_5: dsp_sub30Comb port map (AIN1=>A6,
                                           BIN1=>B6,
                                           CLK=>CLOCK,
                                           RST=>RESET,
                                           OUT1=>P6);

      B6<=std_logic_vector(to_signed(10,30));

      dsp_sub30_6: dsp_sub30Comb port map (AIN1=>A3c,
                                           BIN1=>B3c,
                                           CLK=>CLOCK,
                                           RST=>RESET,
                                           OUT1=>P3c);
      B3c<=std_logic_vector(to_signed(100,30));

      dsp_adder30_1: dsp_adder30Comb port map (AIN1=>A4,
                                               BIN1=>B4,
                                               CLK=>CLOCK,
                                               RST=>RESET,
                                               OUT1=>P4);
      B4<=std_logic_vector(to_unsigned(1,30));

      process(MAR,TEMP,MAX)
      begin
            --AC1:= MEM(MAR)-TEMP;--Operacion DSP
            AC1:=conv_integer(P1);
            if AC1<0 then
                  MIN1 <= '1';
                  MAG1 <= '0';
            else
                  if AC1 = 0 then
                      MIN1 <= '0';
                      MAG1 <= '0';

                  else
                      MIN1 <= '0';
                      MAG1 <= '1';
                  end if;
            end if;
```

```
            --AC2 := MEM(MAR)-MAX;--Operacion DSP
            AC2:=conv_integer(P2);
            if (AC2<0) then
                  MAG2 <= '1';
            else
                  MAG2 <= '0';
            end if;
      end process;

      process (EN_DISP,RES_DISP,NUM,MAX)
      variable TM,TN: num9bits;
      begin
            A3<=std_logic_vector(to_signed(TM,30));
            A6<=std_logic_vector(to_signed(TN,30));
            A3c<=std_logic_vector(to_signed(TM,30));

            if EN_DISP = '1' then
                 DISPMAX1 <= "0000000";
                 DISPMAX2 <= "0000000";
                 DISPMAX3 <= "0000000";
                 DISPNUM1 <= "0000000";
                 DISPNUM2 <= "0000000";
                 SIGN <= '0';
            else
                 if RES_DISP ='0' then
                   DISPMAX1 <= "1000000";
                   DISPMAX2 <= "1000000";
                   DISPMAX3 <= "1000000";
                   DISPNUM1 <= "1000000";
                   DISPNUM2 <= "1000000";
                   SIGN <= '1';
                 else
                   TN := NUM;
                   if MAX<0 then
                      SIGN <= '1';
                      --TM := -MAX mod 2**5;--Operacion DSP
                      TM:=conv_integer(P3(4 downto 0));
                   else
                      SIGN <= '0';
                      TM := MAX mod 2**5;
                   end if;
                   if TM> 99 then
                      DISPMAX1 <= "0011000";
                      --TM := TM - 100;--Operacion DSP
                      TM:=conv_integer(P3c);
                   else

                        DISPMAX1 <= "0111111";
                   end if;

                  if TM > 89 then
                      DISPMAX2 <= "1111110";
                      --TM := TM - 90;--Operacion DSP
                      B3<=std_logic_vector(to_signed(90,30));
                      TM:=conv_integer(P3);
                   else
```

```
                    if TM > 79 then
                       DISPMAX2 <= "1111111";
                       --TM := TM - 80;--Operacion DSP
                       B3<=std_logic_vector(to_signed(80,30));
                       TM:=conv_integer(P3);
                    else
                       if TM > 69 then
                          DISPMAX2 <= "0011100";
                          --TM := TM - 70;--Operacion DSP
                          B3<=std_logic_vector(to_signed(70,30));
                          TM:=conv_integer(P3);
                       else
                          if TM > 59 then
                             DISPMAX2 <= "1110111";
                             --TM := TM - 60;--Operacion DSP
                             B3<=std_logic_vector(to_signed(60,30));
                             TM:=conv_integer(P3);
                          else
                             if TM > 49 then
                                DISPMAX2 <= "1110110";
                                --TM := TM - 50;--Operacion DSP
                                B3<=std_logic_vector(to_signed(50,30));
                                TM:=conv_integer(P3);
                             else
                                if TM > 39 then
                                   DISPMAX2 <= "1011010";
                                   --TM := TM - 40;--Operacion DSP
                                   B3<=std_logic_vector(to_signed(40,30));
                                   TM:=conv_integer(P3);
                                else
                                  if TM > 29 then
                                    DISPMAX2 <= "1111001";
                                    --TM := TM - 30;--Operacion DSP
                                    B3<=std_logic_vector(to_signed(30,30));
                                    TM:=conv_integer(P3);
                                   else
                                     if TM > 19 then
                                       DISPMAX2 <= "1101100";
                                       --TM := TM - 20;--Operacion DSP
                                 B3<=std_logic_vector(to_signed(20,30));
                                       TM:=conv_integer(P3);
                                       else
                                         if TM > 9 then
                                         DISPMAX2 <= "0011000";
                                         --TM := TM - 10;--Operacion DSP


                                B3<=std_logic_vector(to_signed(10,30));
                                TM:=conv_integer(P3);

                                         else
                                              DISPMAX2 <= "0111111";
                                         end if;
                                       end if;
                                   end if;
                                end if;
```

```
                              end if;
                           end if;
                        end if;
                     end if;
                  end if;
                  if TM > 8 then
                     DISPMAX3 <= "1111110";
                  else
                     if TM > 7 then
                        DISPMAX3 <= "1111111";
                     else
                        --Multiples if-else anidados
                        ...
                     end if;
                  end if;
                  if TN > 9 then
                     DISPNUM1 <= "0011000";
                     A6<=std_logic_vector(to_signed(TN,30));
                     TN := conv_integer(P6);
                  else
                     DISPNUM1 <= "0111111";
                  end if;

                  if TN > 8 then
                     DISPNUM2 <= "1111110";
                  else
                        --Multiples if-else anidados
                  ...
                  end if;
         end process;

         process (CLOCK,RESET)
         variable STATO: integer range 0 to 4;
         variable TMN : bit_vector (5 downto 0);
         begin
    --añado la descripción de casos para no introducir un ciclo de retraso
          if(RESET='1')then
                A4<=std_logic_vector(to_unsigned(0,30));
          else
          case STATO is
            when st0 =>
            when st1 =>
            when st2 =>
            when st3 =>
                if MIN1 = '1' then
                  if FLAG ='1' then
                    --NUM <= NUM+1;--Operacion DSP
                    A4<=std_logic_vector(to_unsigned(NUM,30));
                  end if;
                end if;
            when st4 =>
                if MAR = 31 then
                else
                  --MAR <= MAR+1;--Operacion DSP
                  A4<=std_logic_vector(to_unsigned(MAR,30));
                end if;
```

```
            when others=>
                A4<=std_logic_vector(to_unsigned(0,30));
        end case;
    end if;


    if RESET = '1' then
                STATO := st0;
                RES_DISP <= '0';
                EN_DISP <= '0';
                NUM <= 0;
                MAR <= 0;
                TEMP <= 0;
                MAX <= 0;
                FLAG <= '0';
     elsif CLOCK'event and CLOCK='1' then

                        case STATO is
                          when st0 =>
                           RES_DISP <= '0';
                           EN_DISP <= '0';
                           STATO := st1;

                          when st1 =>
                           if START = '1' then
                              NUM <= 0;
                              MAR <= 0;
                              FLAG <= '0';
                              EN_DISP <= '1';
                              RES_DISP <= '1';
                              STATO := st2;
                           else
                              STATO := st1;
                           end if;
                          when st2 =>
                           MAX <= MEM(MAR);
                           TEMP <= MEM(MAR);
                           STATO := st3;
                          when st3 =>
                           if MIN1 = '1' then
                              if FLAG ='1' then
                                 FLAG <= '0';
                                 --NUM <= NUM+1;--Operacion DSP
                                 NUM<=conv_integer(P4);
                              end if;
                           else
                              if MAG1 = '1' then
                                 if MAG2 = '1' then
                                    MAX <= MEM(MAR);
                                 end if;
                                 FLAG <= '1';
                              end if;
                           end if;

                           TEMP <= MEM(MAR);
                           STATO := st4;
```

```
                        when st4 =>
                         if MAR = 31 then
                            if START = '1' then
                               STATO := st4;
                            else
                               STATO := st1;
                            end if;
                            EN_DISP <= '0';
                         else--MAR <= MAR+1;--Operacion DSP
                            MAR<=conv_integer(P5);
                            STATO := st3;
                         end if;
                end case;
                end if;
        end process;
    end BEHAV;
```

Una vez sintetizado se obtiene un diseño con una frecuencia de 224.147 MHz y un área de 162 LUT-FF y 8 DSP48E.

Ya que en el diseño b05 los datos tienen una anchura máxima de 9 bits y las clases de operación son SUB2 y ADD2 podemos usar el modo SIMD FOUR12. En este caso nos vale con usar tres DSP48E de la siguiente manera:

```
    dsp_sub12_1: dsp_sub12Comb port map (AIN1=>A1,
                                         AIN2=>A2,
                                         AIN3=>"000000000000",
                                         AIN4=>"000000000000",
                                         BIN1=>B1,
                                         BIN2=>B2,
                                         BIN3=>"000000000000",
                                         BIN4=>"000000000000",
                                         CLK=>CLOCK,
                                         RST=>RESET,
                                         OUT1=>P1,
                                         OUT2=>P2,
                                         OUT3=>open,
                                         OUT4=>open);

    dsp_adder12_1: dsp_adder12Comb port map (...);

    dsp_sub12_2: dsp_sub12Comb port map (AIN1=>A3,
                                         AIN2=>A3b,
                                         AIN3=>A3c,
                                         AIN4=>A6,
                                         BIN1=>B3,
                                         BIN2=>B3b,
                                         BIN3=>B3c,
                                         BIN4=>B6,
                                         CLK=>CLOCK,
```

```
                                        RST=>RESET,
                                        OUT1=>P3,
                                        OUT2=>P3b,
                                        OUT3=>P3c,
                                        OUT4=>P6);
```

De esta manera obtenemos un diseño con una frecuencia de 226.819 MHz y un área de 162 LUT-FF y 3 DSPs. Esta última modificación nos sirve para ilustrar el problema de qué tratamiento se debe dar al ámbito de un slot. Las operaciones sí pueden encapsularse dentro de un mismo DSP48E aunque compartan ámbito. Sin embargo en este caso las operaciones han sido asignadas a un DSP diferente por cada process obteniéndose así un mejor resultado. En este caso la reutilización de slots hace que con 3 instancias de clase y 8 slots utilizados podamos implementar 16 operaciones (ver tabla 4-3).

Tabla 4-3. Instancias de clase, slots e instancias de slot.

#Instancias de Clase	#Slots	#Instancias de Slot
3	7	16

Finalmente cabe decir que los cambios hechos siguiendo esta metodología sobre un código ya escrito deben mantener la funcionalidad original del diseño. Para ello es necesario que el resultado de las operaciones se siga obteniendo en el mismo momento. Para obtener esto lo más sencillo es definir las entradas de un DSP de manera estática de tal modo que ciclo a ciclo se va a tener el resultado esperado. Sin embargo cuando se comparten DSPs con entradas diferentes no es posible definir la entrada de manera estática. En este caso si el diseño es asíncrono bastará con definir la entrada en el mismo sitio que se define en el código original, pero en el caso de diseños síncronos esto no es posible ya que obtendríamos el resultado al ciclo siguiente al esperado. Tenemos dos opciones, por un lado podemos definir una estructura de decisiones similar a la del diseño pero asíncrona, de manera que implementará multiplexores, o podemos adelantar la asignación de entradas un ciclo. La primera opción disminuye considerablemente la frecuencia del diseño y no siempre se obtiene un código sintetizable. La segunda opción es por lo tanto la utilizada en los casos en que no se definen estáticamente las entradas de un DSP.

Capítulo 5

Resultados y comparativa

Una vez propuesta la metodología de síntesis para usar los bloques DSP48E y mostrada su aplicación sobre un ejemplo real la aplicamos sobre una serie de diseños en VHDL. Estos diseños se corresponden con parte de la batería de benchmarks ITC'99, concretamente con los benchmarks I99T, desarrollados por el grupo CAD en el Politecnico di Torino. Además de los I99T se han usado 6 benchmarks adicionales que hacen uso de sumas y multiplicaciones. Para todos los diseños anteriores se presentan los valores de frecuencia y área para la versión original sin hacer uso de los bloques DSP48E, haciendo uso de dichos bloques automáticamente con XST y para las dos versiones obtenidas al aplicar la metodología propuesta en este trabajo.

5.1 Benchmarks I99T

Según podemos ver en [41] este grupo de benchmarks se compone de 22 diseños en VHDL a nivel de transferencia entre registros (RTL) cuyos tamaños van desde 45 puertas y 5 Flip Flop (FF) hasta 98.000 puertas y 6.600 FF. Algunos de los diseños más grandes se componen de varios módulos de diseños más pequeños y todos tienen un solo reloj y están libres de señales inout, búferes triestado y memorias. Además pueden sintetizarse sin hacer uso de librerías extra.

La tabla 5-1 lista los benchmarks del 1 al 22 y su funcionalidad original. Dicha funcionalidad puede no cumplirse tal como avisan en su página web [41], no obstante son sintácticamente correctos.

Tabla 5-1. Benchmarks I99T, funcionalidad original y si contienen operaciones de suma/resta.

Nombre	Funcionalidad Original	Operaciones +, -
b01	FSM that compares serial flows	No
b02	FSM that recognizes BCD numbers	No
b03	Resource arbiter	No
b04	Compute min and max	Sí
b05	Elaborate the contents of a memory	Sí
b06	Interrupt handler	No
b07	Count points on a straight line	Sí
b08	Find inclusions in sequences of numbers	Sí
b09	Serial to serial converter	No
b10	Voting system	No
b11	Scramble string with variable cipher	Sí
b12	1-player game (guess a sequence)	Sí
b13	Interface to meteo sensors	Sí
b14	Viper processor (subset)	Sí
b15	80386 processor (subset)	Sí
b16	Hard to initialize circuit (parametric)	-
b17	Three copies of b15	Sí
b18	Two copies of b14 and two of b17	Sí
b19	Two copies of b18	Sí
b20	A copy of b14 and a modified version of b14	Sí
b21	Two copies of b14	Sí
b22	A copy of b14 and two modified versions of b14	Sí

Como es lógico los diseños que no contienen operaciones de suma/resta o multiplicación no han sido usados en este estudio, así como el diseño b16, que no se proporciona, ni tampoco los diseños b20 ni b22, ya que se han considerado redundantes de cara al estudio actual.

5.2 Benchmarks Grupo 2

Se trata de una serie de 6 benchmarks (ver tabla 5-2) escritos en código VHDL estructural que han sido creados por Alberto del Barrio y utilizados en su tesis doctoral [42]. Todos los diseños hacen uso de operaciones de suma y multiplicación.

Tabla 5-2. Funcionalidad original grupo 2.

Nombre	Implementa
2EWF	filtro elíptico 2º orden
DCT	transformada discreta del coseno
DiffEq	desconocido
IDCT	inversa DCT
Lattice	filtro Lattice
LMS	filtro LMS

Estos benchmarks tienen las sumas, restas y multiplicaciones diseñadas a partir de celdas básicas por lo que no tenemos forma directa de reconocer que se trata con dichas operaciones. Debido a ello se ha procedido a modificarlos sustituyendo los diseños con celdas por llamadas directas a las operaciones de + y * en cada código VHDL.

5.3 Resultados experimentales

Se procede ahora a sintetizar cada uno de los diseños de los dos grupos de benchmarks anteriores. Se sintetiza el diseño original sin hacer uso de bloques DSP48E (USE_DSP=none) y haciendo uso (USE_DSP=yes) para así tener una referencia con la que comparar los resultados de los diseños modificados siguiendo la metodología propuesta (USE_DSP=auto). Para cada diseño se presentan los valores de frecuencia y área.

Ninguno de los diseños de la tabla 5-3 contiene multiplicaciones. La tabla 5-5 contiene los resultados para los benchmarks originales del grupo dos, los cuales contienen multiplicaciones en todos los casos. Como hemos dicho, el código VHDL de los benchmarks del grupo dos se modifica obteniendo versiones que hacen uso directo de las operaciones +, - y *. Los resultados que se obtienen con los diseños modificados del grupo 2 se muestran en la tabla 5-6.

Respecto al área se muestra el número de pares LUT-FF y el número de bloques DSP48E utilizadas. No se ha hecho distinción entre los casos en que se ocupa solo la LUT, el FF o ambos. En las tablas 5-3, 5-5 y 5-6 el área se presenta como a+b, siendo a el número de pares LUT-FF y b el número de DSPs. En el caso de la primera columna de área no se utilizan DSPs, por lo que solo aparece un número que representa el número de pares LUT-FF. La frecuencia se presenta en MHz.

A la hora de presentar los datos gráficamente la frecuencia se presenta igualmente en MHz y el área en pares LUT-FF. Para representar fielmente el área de un DSP frente a los CLBs en la arquitectura Virtex 5 se ha ponderado dicha área teniendo en cuenta la ocupación real del DSP en la FPGA. Sabemos que en la arquitectura Virtex 5:

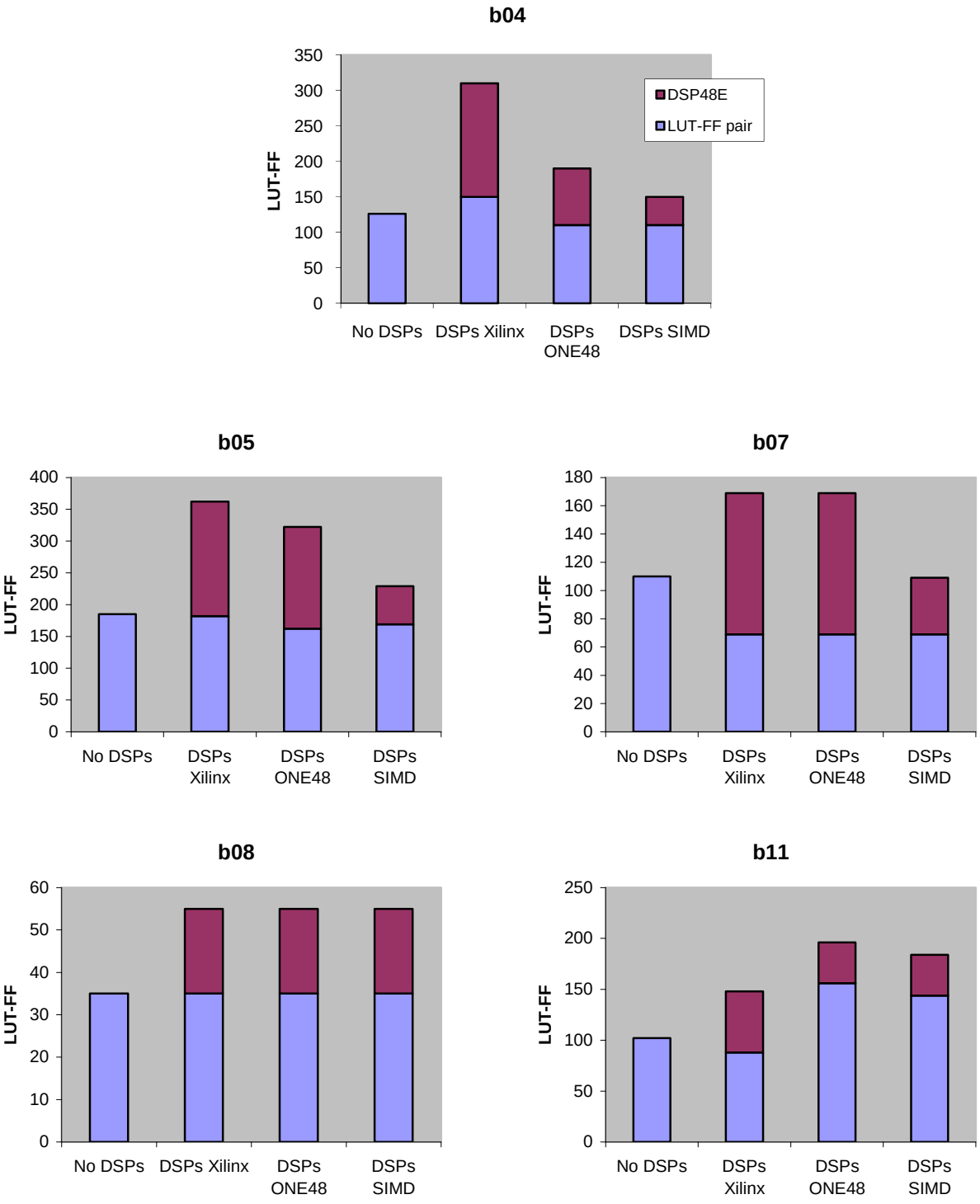
- 2 DSP-slices ocupan lo mismo que 5 CLBs
- 1 CLB tiene 2 slices
- 1 slice tiene 4 pares LUT-FF

Por tanto cada DSP equivale a 20 pares LUT-FF.

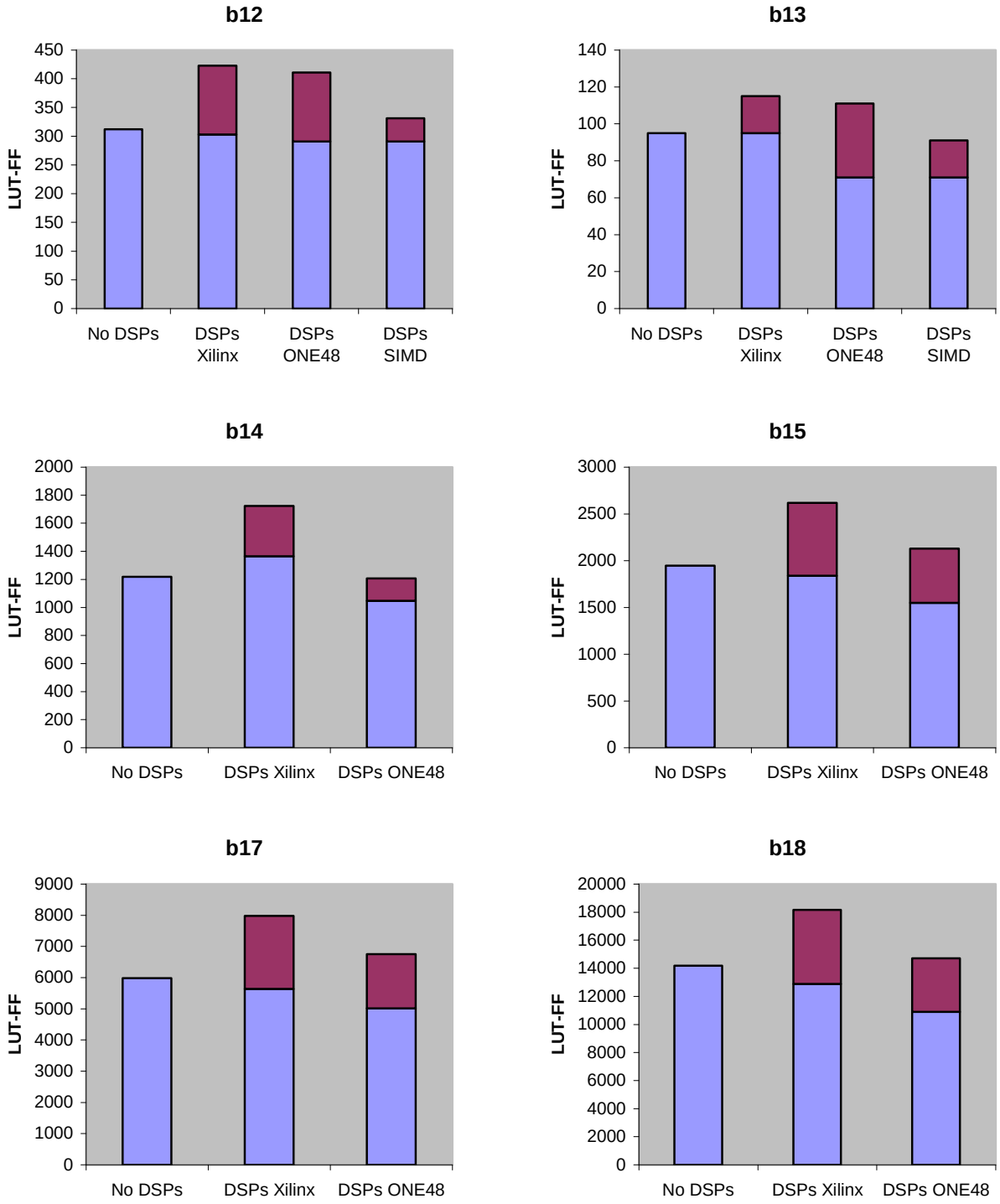
Tabla 5-3. Área y frecuencia benchmarks I99T.

Testbench	Sin DSPs		DSPs Xilinx		DSPs ONE48		DSPs SIMD	
	Area	Frec (Mhz)	Area	Frec (Mhz)	Area	Frec (Mhz)	Area	Frec.(Mhz)
b1	No contiene operaciones +, - o *							
b2	No contiene operaciones +, - o *							
b3	No contiene operaciones +, - o *							
b4	126	392.172	150+8	103.382	110+4	363.391	110+2	355.366
b5	185	262.557	182+9	220.197	162+8	224.147	162+3	226.819
b6	-	-	-	-	-	-	-	-
b7	110	376.407	69+5	365.167	69+5	362.674	69+2	362.674
b8	35	361.860	35+1	361.860	35+ 1	361.860	35 + 1	361.860
b9	No contiene operaciones +, - o *							
b10	No contiene operaciones +, - o *							
b11	102	327.156	88+3	265.625	156+2	304.869	144+2	311.313
b12	312	315.457	303+6	313.687	291+6	338.983	291+2	338.983
b13	95	391.053	95+1	353.995	71+2	342.830	71+1	342.830
b14	1218	126.567	1364+18	77.361	1047+8	195.505	-	-
b15	1946	167.195	1838+39	105.27	1548+29	127.636	-	-
b16	No lo proporcionan							
b17	5984	163.81	5640+117	105.27	5010+87	129.543	-	-
b18	14190	93.189	12893+263	77.362	10900+190	129.057	-	-
b19	27950	90.475	29853+530	77.341	22410+380	129.064	-	-
b21	2609	126.938	2807+39	77.433	2184+16	195.505	-	-

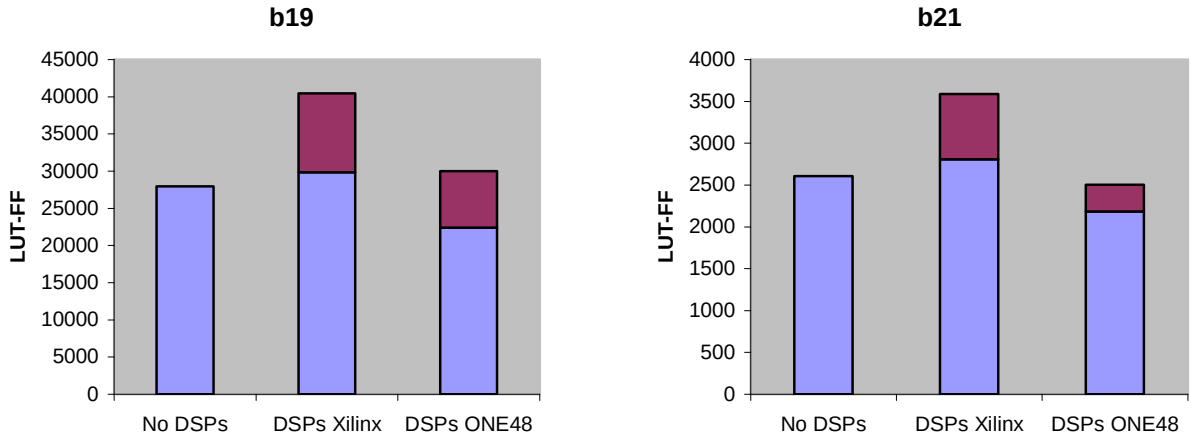
Gráfica 5-1. Área Benchmarks I99T



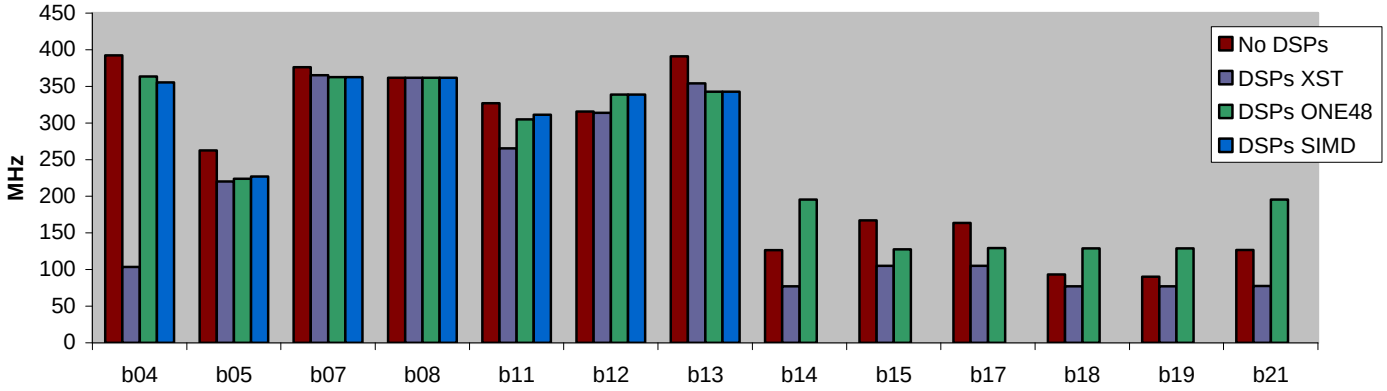
Gráfica 5-1 (continuación). Área Benchmarks I99T



Gráfica 5-1 (continuación). Área Benchmarks I99T



Gráfica 5-2. Frecuencia Benchmarks I99T.



Como vemos en la gráfica 5-1 el área de los diseños que usan plantillas en modo ONE48 (diseños DSP ONE48) es inferior al área de los diseños con síntesis automática de DSPs en XST salvo en b07 y b08, en los que se obtiene un área igual, y en b11, que se obtiene un área mayor. En b11 se aplica una transformación que adelanta las entradas de los DSP un ciclo, por lo que cambia la estructura original de la maquina de estados del diseño. La disminución en general del área respecto a los diseños DSP XST se debe principalmente a que en la mayoría de benchmarks (salvo en b13) se usan menos bloques DSP48E (ver tabla 5-4), lo que parece a su vez disminuir el número de pares LUT-FF. Además para cada benchmark el área de los diseños que hacen uso de SIMD es inferior a la de los diseños ONE48 menos en b08 (que tiene igual área) ya que usan

aún menos DSP48E. Respecto a los diseños DSP XST los diseños DSP ONE48 presentan una disminución de área media del 15 % y los diseños DSP SIMD una disminución del 21.5 %.

Teniendo en cuenta que el área es igual o inferior para los diseños obtenidos con la metodología pasamos ahora a ver la frecuencia. Hay que recordar (ver tabla 5-1) que el diseño b21 contiene dos copias del diseño b14, b17 contiene tres copias de b15, b18 dos de b14 y dos de b17 y b19 dos de b18, por lo que es lógico que tengan resultados similares entre sí. Como vemos en la gráfica 5-2 exceptuando los diseños b07 y b13 la frecuencia obtenida es inferior para las versiones DSP XST teniéndose una frecuencia media un 28% superior con nuestra metodología. Cabe destacar que la frecuencia de los diseños DSP ONE48 y DSP SIMD se acerca bastante a la frecuencia obtenida con los diseños que no usan DSPs.

Tabla 5-4. Número de DSPs utilizados.

Benchmark	DSPs Xilinx	DSPs ONE48	DSPs SIMD
b04	8	4	2
b05	9	8	3
b07	5	4	2
b08	1	1	1
b11	3	2	2
b12	6	6	2
b13	1	2	1
b14	18	8	-
b15	39	29	-
b17	117	87	-
b18	263	190	-
b19	530	380	-
b21	39	16	-

Tabla 5-5. Área y frecuencia benchmarks originales grupo 2.

Testbench	Sin DSPs	
	Area(FF Lut)	Frec.(Mhz)
EWF_8	576	70.898
DCT_8	671	70.750
DiffEq_8	205	120.844
IDCT_8	581	110.498
Lattice	192	117.294
Lms_filter_8	275	105.274

Gráfica 5-3. Área y frecuencia Benchmarks grupo 2.

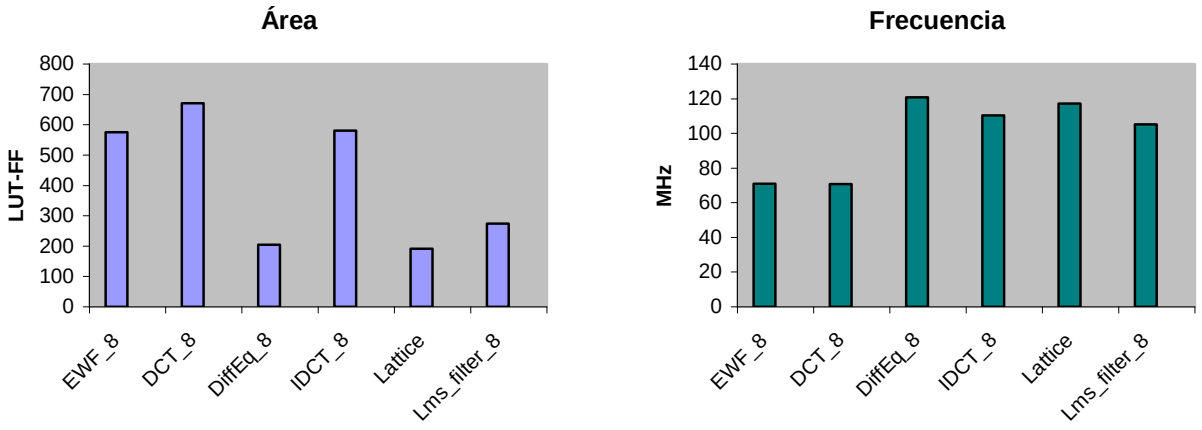
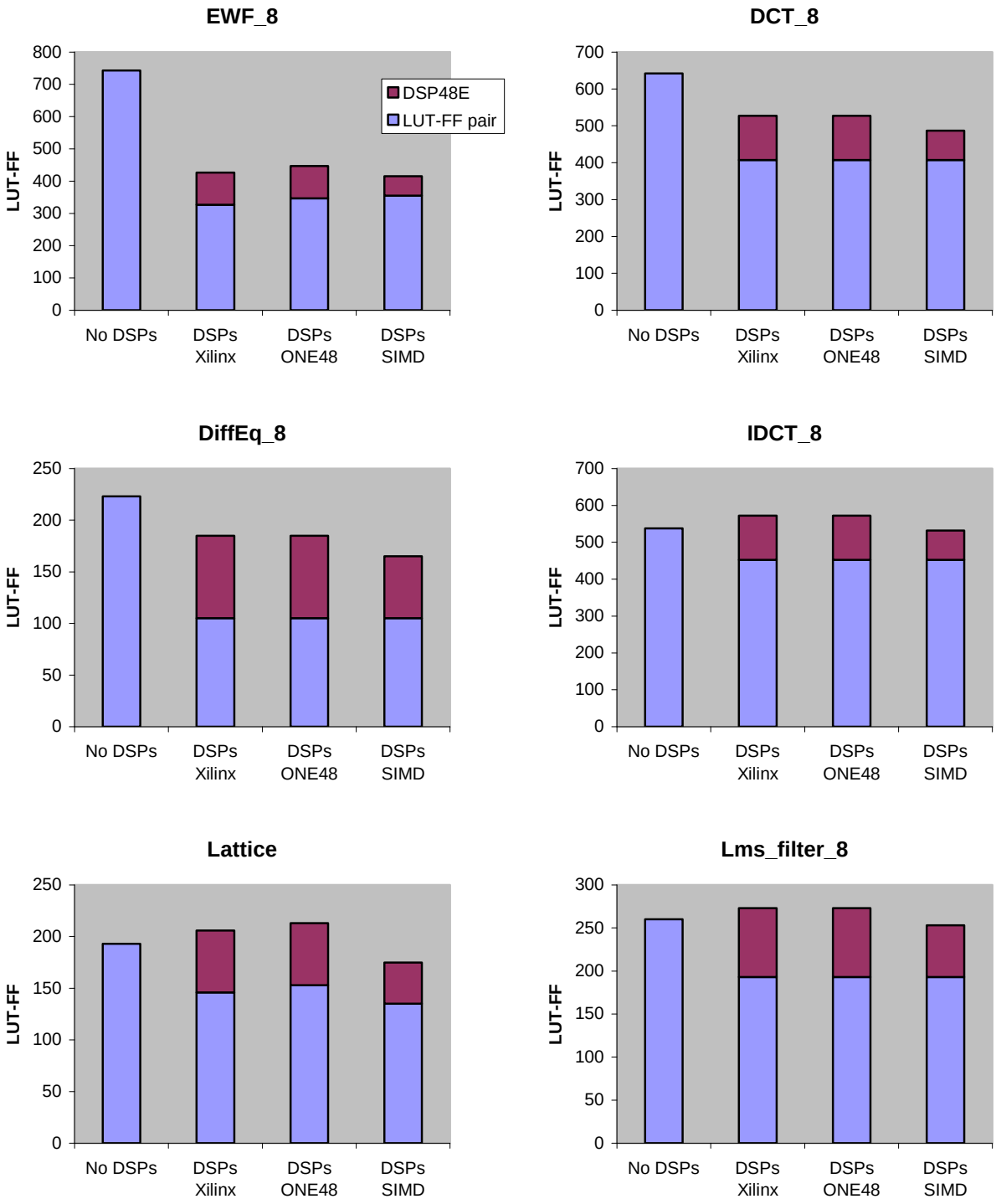


Tabla 5-6. Área y frecuencia benchmarks modificados grupo 2.

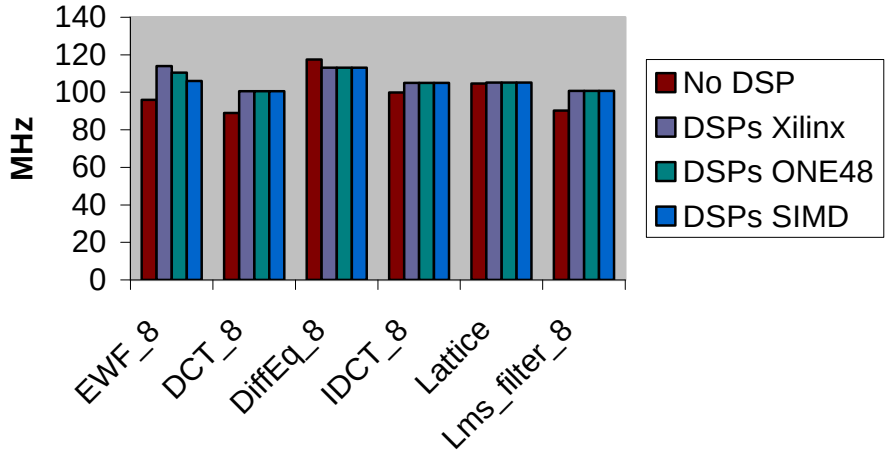
Testbench	Sin DSPs		DSPs Xilinx		DSPs ONE48		DSPs SIMD	
	Area (FF Lut)	Frec.(Mhz)	Area	Frec.(Mhz)	Area (FF Lut)	Frec.(Mhz)	Area (FF Lut)	Frec.(Mhz)
EWF_8	743	95.986	327+5	114.007	347+5	110.579	355+3	106.086
DCT_8	642	89	407+6	100.563	407+6	100.563	407+4	100.563
DiffEq_8	223	117.407	105+4	113.233	105+4	113.233	105+3	113.233
IDCT_8	538	99.843	452+6	105.096	452+6	105.096	452+4	105.096
Lattice	193	104.778	146+3	105.212	153+3	105.212	135+2	105.212

Lms_filter_8	260	90.229	193+4	100.698	193+4	100.698	193+3	100.698

Gráfica 5-4. Área Benchmarks grupo 2 modificados.



Gráfica 5-5. Frecuencia Benchmarks grupo 2 modificados.



En primer lugar se presentan los datos de los benchmarks originales del grupo dos (ver tabla 5-5 y gráfica 5-3) que pueden compararse con los datos de sus versiones modificadas (tabla 5-6 y gráfica 5-4). Se observa que no hay una tendencia general en los datos ni respecto a área ni a frecuencia.

Pasamos ahora a estudiar los resultados de DSP ONE48 y DSP SIMD con las versiones modificadas del grupo dos (DSP Xilinx). Respecto al área era de esperar que al tratarse de un diseño estructural no difirieran significativamente los resultados. De hecho solo en los benchmarks EWF_8 y Lattice se tienen diferencias en el número de LUT-FF utilizados. Las versiones DSP SIMD utilizan entre uno y dos DSPs menos que las versiones DSP Xilinx y DSP ONE48, las cuales emplean el mismo número de DSPs entre benchmarks.

De igual forma era de esperar que la frecuencia fuese similar. Salvo en EWF_8 y Lattice en el resto de diseños se obtienen resultados iguales frecuencia para los diseños DSP XST, DSP ONE48 y DSP SIMD.

Respecto a los diseños que no hacen uso de DSPs el área disminuye en todos los benchmarks para los diseños DSP SIMD mientras que la frecuencia es un 5.9% superior.

Capítulo 6

Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones generales que sacamos del presente estudio. Se presentan los objetivos iniciales y se resume la metodología propuesta y los resultados más destacados. Finalmente se comenta brevemente la heurística a seguir para seleccionar los códigos sobre los que aplicar la metodología actual y se presentan las posibilidades de automatizarla.

6.1 Conclusiones

Con el presente trabajo se ha estudiado la posibilidad de aprovechar los bloques DSP48E contenidos en la familia Virtex 5 de Xilinx y posteriores. En primer lugar se intentó hacer uso de estos bloques de manera automática con XST empleando determinadas construcciones de código VHDL behavioral.

Ante la imposibilidad de obtener ciertas configuraciones específicas como por ejemplo el uso de los modos SIMD TWO24 o SIMD FOUR12 se plantea la posibilidad de hacer un uso más directo de los DSP. Para ello se utiliza la plantilla de macro específica para el bloque DSP48E que permite instanciar dichos bloques. Con la plantilla podemos configurar diferentes aspectos del DSP48E, como son las etapas de pipeline, las operaciones a realizar o el uso de diferentes modos SIMD.

Se plantea entonces una metodología que permite sustituir las operaciones de +, - y * en un código VHDL por sus respectivas operaciones mapeadas en un DSP48E. En dicha metodología se proponen transformaciones de código para mantener la funcionalidad original del diseño y limitar el uso de bloques DSP48E de manera rápida y sencilla.

Con los resultados actuales puede decirse que la aplicación de la metodología consigue disminuir el número de DSPs utilizados por el diseño tanto en VHDL behavioral (benchmarks I99T) como VHDL estructural (grupo dos) siempre que el ancho de las operaciones permita hacer uso de los modos SIMD TWO24 o FOUR12. Además para los diseños en VHDL behavioral se consigue una mejora en área de un 15% (sin SIMD) y un 21.5% (con SIMD) y una frecuencia media un 28% superior por lo que la aplicación de la metodología resulta beneficiosa.

6.2 Trabajo futuro

El presente trabajo puede desarrollarse en diferentes modos. Por ejemplo la metodología propuesta puede ser totalmente automatizada desarrollando una heurística que tome las decisiones a la hora de formar CICs (Conjuntos Instancia de Clase), único punto no automático actualmente, liberando así de dicha tarea al diseñador. En esta línea, en algunos casos el adelanto de un ciclo en las entradas del DSP nos permite reutilizar slots sin variar la funcionalidad del diseño, pero debe estudiarse de manera más extensa cuando vale la pena y cuando no realizar este adelanto de entradas. En el presente estudio se proponen unas recomendaciones (sección 4.3) para la creación de CICs pero se cree necesario un estudio más profundo y extenso del alcance y los resultados de tales recomendaciones antes de poder afirmar que la metodología es totalmente automática. Una vez definida la heurística para crear CICs la automatización de la metodología es relativamente sencilla y puede implementarse como un traductor de código VHDL.

Otro aspecto que puede desarrollarse es la inclusión de DSPs con varias etapas de pipeline, que aunque han sido implementados varios diseños, estos no se han probado finalmente ni se han aplicado con la metodología. Esto es así ya que al aplicar DSPs con varias etapas según la metodología propuesta se cambia el ciclo en que se obtiene el resultado y como se ha dicho, en el presente estudio se quería mantener la funcionalidad original de los diseños. Sin embargo el uso de estas etapas puede incrementar el throughput de un diseño por lo que merece la pena realizar un trabajo en la dirección de estudiar como añadir estos DSP al flujo de ejecución de un código VHDL.

Referencias

[1] D. S. Poznanovic. “The Emergence of Non-von Neumann Processors”.
In Reconfigurable Computing: Architectures and Applications, pp. 243-254. SpringerLink, 2006.

[2] K. Underwood. “FPGAs vs CPUs: Trends and Peak Floating Performance”.
In Proc. of the Twelfth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 171-180, 2004.

[3] H. Simmler, L. Levinson, R. Manner. “Multitasking on FPGA Coprocessors”. In Proceedings of FPL'00, pp. 121-130, 2000.

[4] Xilinx. Virtex 5 FPGA User Guide, pp. 173-216, 2010.

[5] W. P., Hays. “DSPs: Back to the Future”. Magazine Queue 2, 1, pp. 42-51, ACM, Mar. 2004.

[6] Sen M. Kuo, Woon-Seng Gan. “Digital Signal Processors: Architectures, Implementations, and Applications”. Pearson Prentice Hall, pp. 1-39, 2005.

[7] H. Francis. “ARM DSP-Enhanced Extensions”. White paper, ARM Ltd, 2001

[8] P. Coussy, D.D. Gajski, M. Meredith, A. Takach. "An Introduction to High-Level Synthesis". Design & Test of Computers, IEEE, vol.26, no.4, pp. 8-17, Julio-Agosto 2009

[9] Catapul DS. Recurso Web, http://www.mentor.com/esl/catapult/upload/Catapult_DS.pdf

[10] Cynthesizer. Recurso Web,
http://www.forteds.com/products/cynthesizer_datasheet_2008.pdf

[11] DK. Recurso Web, http://www.mentor.com/products/fpga/handel-c/dk-design-suite/upload/dk-ds.pdf

[12] M. P. Cardoso, P. C. Diniz, M. Weinhardt. “Compiling for reconfigurable computing: A survey”. *ACM Comput. Surv.* 42, 4, Article 13, 65 páginas, Junio 2010

[13] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. JONES, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, D. Zaretsky. “A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems”. In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’00). IEEE, Los Alamitos, CA, pp.39–48, 2000

[14] CARDOSO, J. M. P. AND WEINHARDT, M. 2002. XPP-VC: “A C compiler with temporal partitioning for the PACT-XPP architecture”. In Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL’02). Lecture Notes in Computer Science, Springer, Berlin, 864–874.

[15] CARDOSO, J. M. P. AND NETO, H. C. 1999. “Macro-based hardware compilation of Java bytecodes into a dynamic reconfigurable computing system”. In Proceedings of the IEEE 7th Symposium on Field-Programmable Custom Computing Machines (FCCM’99). IEEE, Los Alamitos, CA, 2–11.

[16] TRIPP, J. L., JACKSON, P. A., AND HUTCHINGS, B. 2002. “Sea Cucumber: A synthesizing compiler for FPGAs”. In Proceeedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL’02). Lecture Notes in Computer Science, vol. 2438, Springer, Berlin, 875–885.

[17] SNIDER, G., SHACKLEFORD, B., AND CARTER, R. J. 2001. “Attacking the semantic gap between application programming languages and configurable hardware”. In Proceedings of the ACM 9th International Symposium on Field-Programmable Gate Arrays (FPGA’01). ACM, New York, 115–124.

[18] GUO, Z. AND NAJJAR, W. 2006. “A compiler intermediate representation for reconfigurable fabrics”. In Proceedings of the 16th International Conference on Field Programmable Logic and Applications
(FPL’2006). IEEE, Los Alamitos, CA, 741–744.

[19] G. Wang, G. Stitt, H. Lam, A. D. George. “A framework for core-level modeling and design of reconfigurable computing algorithms”. In Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA '09). ACM, New York, NY, USA, pp. 29-38, 2009

[20] S. Mohanty, V. K. Prasanna. “A model-based extensible framework for efficient application design using FPGA”. *ACM Trans. Des. Autom. Electron. Syst.* 12, 2, Articulo 13, Abril 2007

[21] “Designing High-Performance DSP Hardware Using Catapult C Synthesis and the Altera Accelerated Libraries”, White paper, Altera & Mentor Graphics, 2007

[22] M. Ownby, W.H. Mahmoud. "A design methodology for implementing DSP with Xilinx® System Generator for Matlab®". Proceedings of the 35th Southeastern Symposium on System Theory, pp. 404- 408, 16-18 March 2003

[23] C. Saiprasert, Christos-S. Bouganis, G. A. Constantinides. “An Optimized Hardware Architecture of a Multivariate Gaussian Random Number Generator”. *ACM Trans. Reconfigurable Technol. Syst.* 4, 1, Articulo 2, 21 pp., Diciembre 2010

[24] S. Drimer, T. Güneysu, C. Paar. “DSPs, BRAMs, and a Pinch of Logic: Extended Recipes for AES on FPGAs”. *ACM Trans. Reconfigurable Technol. Syst.* 3, 1, Articulo 3, 27 pp., Enero 2010

[25] A. Kaganov, A. Lakhany, P. Chow. “FPGA Acceleration of MultiFactor CDO Pricing”. *ACM Trans. Reconfigurable Technol. Syst.* 4, 2, Articulo 20, 17 pages, Mayo 2011

[26] H. Morisita, K. Inakagata, Y. Osana, N. Fujita, H. Amano. “Implementation and evaluation of an arithmetic pipeline on FLOPS-2D: multi-FPGA system”. *ACM SIGARCH Comput. Archit. News* 38, 4, pp. 8-13, Enero 2011

[27] K. K. Nagar, Y. Zhang, J. D. Bakos. “An integrated reduction technique for a double precision accumulator”. In *Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications* (HPRCTA '09), ACM, 2009

[28] S. Srinath, K. Compton. “Automatic generation of high-performance multipliers for FPGAs with asymmetric multiplier blocks”. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 51-58. ACM, 2010

[29] S. Lopez, R. Sarmiento, P.G. Potter, W. Luk, P.Y.K Cheung. "Exploration of hardware sharing for image encoders," Design, Automation & Test in Europe Conference & Exhibition, pp.1737-1742, 8-12 Marzo 2010

[30] "IEEE Standard VHDL Language Reference Manual", IEEE Std 1076-1987, 1988

[31] J.S. Lis, D.D. Gajski. "Synthesis from VHDL”. Proceedings of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp.378-381, 3-5 Oct 1988

[32] R. Camposano, L.F. Saunders, R.M. Tabet. "VHDL as input for high-level synthesis" Design & Test of Computers, IEEE , vol.8, no.1, pp.43-49, Marzo 1991

[33] J. Pick, "VHDL synthesis techniques and recommendations". ASIC Conference and Exhibit, 1995, Proceedings of the Eighth Annual IEEE International, pp.389-394, 18-22 Septiembre 1995

[34] Xilinx, “Virtex-5 FPGA XtremeDSP Design Considerations”, 2010

[35] Xilinx, “ISE 12.1 In-depth Tutorial”, 2010
[36] Xilinx, “XST Synthesis Overview”. Recurso Web
http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ise_c_using_xst_for_synthesis.htm

[37] Xilinx, “XST user guide v. 11.3”, 2009

[38] Xilinx, “Synthesis and Verification Design Guide 7”, 2004

[39] Xilinx, “Virtex-5 Libraries Guide for HDL Designs”, pp. 95, 2010

[40] Benchmarks ITC199. Recurso web. Benchmarks.
http://www.cerc.utexas.edu/itc99-benchmarks/bench.html

[41] Benchmarks ITC99. Recurso Web. Información.
http://www.cad.polito.it/downloads/tools/itc99.html

[42] A. del Barrio. Tesis Doctoral: “Utilización de unidades especulativas en síntesis de alto nivel”, Universidad Complutense de Madrid, 2011