Diseño y Generación Automática de Filtros Digitales Orientados a FPGAs

Jorge Arroyuelo
Mónica Arroyuelo
Alejandro Grosso
Departamento de Informática Universidad Nacional de San Luis
República Argentina
{bjarroyu,mdarroyu,agrosso}@unsl.edu.ar

Resumen

Los filtros digitales se han convertido en una de las herramientas más utilizadas dentro del Procesamiento Digital de Señales. En los últimos tiempos se han apoyado para su implementación en dispositivos lógicos programables tales como las FPGAs, debido a las ventajas que estos dispositivos ofrecen. En el presente artículo, se propone un método de construcción de filtros digitales orientados a estos dispositivos, con el objetivo de reducir la cantidad de celdas lógicas utilizadas en los mismos. Además se presenta una herramienta de software, la cual permite obtener especificaciones en VHDL automáticamente a partir de características introducidas por el usuario, y que luego pueden ser sintetizadas de manera directa mediante las herramientas de síntesis actuales.

Palabras claves: Filtros Digitales, Filtros FIR, Filtros IIR, FPGA, VHDL.

Abstract

Digitals filters have became one of the most frequently used in the Digital Signal Processing; in the last years the programmable logic devices such as FPGAs have been used for their implementation, due to the advantages this devices offers. In this articule, we propose a construction method of digital filters addressed to this devices, which aim is to produce a high reduction in the logic cells utilization. Besides, we present a software tool, which allow to obtain automatic VHDL specifications from particular characterisitcs desired, and then can be sintetized in a straight way through the actuals sintesis tools.

Keywords: Digital Filters , FIR-Filters, IIR-Filters, FPGA, VHDL.

1. INTRODUCCIÓN

En los últimos tiempos los filtros digitales se han convertido en una de las herramientas más utilizadas dentro del Procesamiento Digital de Señales (DSP) para dar solución a un porcentaje considerable de problemas de ingeniería. Estos son muy útiles en casos como la eliminación o reducción de ruido e interferencias y la transformación de la respuesta espectral de señales, entre otros. Su utilidad repercute en aplicaciones tales como la compresión de información para la transmisión de datos y el procesamiento de audio y video.

El diseño de filtros se ha apoyado en los dispositivos lógicos programables, los cuales han jugado un papel muy importante en el montaje de los mismos, puesto que gracias a ellos se ha logrado un adecuado funcionamiento en tiempo real. Las FPGAs (Field Programmable Gate Array) son unos de estos dispositivos, los cuales poseen la cualidad principal de ser re-configurables, lo que permite realizar cambios en la arquitectura sin necesidad de producir variaciones en el montaje o en el software que se está operando. También es posible implementar filtros en otros dispositivos programables tales como los CPLDs (Complex Programmable Logic Device). Gracias a la facilidad que se presenta al montar estos diseños en los sistemas modernos y al poder ser diseñados e implementados en circuitos lógicos programables como FPGAs y CPLDs, los filtros digitales gozan hoy en día de una gran popularidad y un extendido uso.

Un filtro es por lo general implementado mediante el uso de circuitos multiplicadores, los cuales son costosos en términos de espacio en una FPGA. Por esto, varias técnicas son actualmente utilizadas para reducir al mínimo el hardware necesario para la implementación. Una técnica que es ampliamente utilizada es la sustitución de las arquitecturas aritméticas paralelas por arquitecturas aritméticas series a nivel de bits. Las arquitecturas serie procesan la entrada de a un único bit por vez. La ventaja es que una única lógica es usada para computar todos los bits de la entrada, pudiendo reducir de manera considerable el hardware necesario. Por otro lado, esto produce un aumento en el tiempo de cálculo, ya que se necesitan n ciclos de reloj para procesar una entrada de n bits.

Una arquitectura para implementar filtros digitales utilizando aritmética serie se presentó en [1]. En este artículo se presenta como construir filtros digitales de orden superior orientados a FPGAs mediante el uso de estas arquitecturas serie, lo cual logra reducir considerablemente la cantidad de celdas lógicas utilizadas, permitiendo construir filtros de más alto orden o tener otras aplicaciones corriendo simultáneamente sobre la FPGA. Otras arquitecturas pueden verse en [2],[3],[4],[5] y [6], las cuales poseen buen desempeño pero no reducen significativamente la cantidad de celdas lógicas utilizadas al ser implementadas. Además en este articulo se muestra la construcción de una herramienta de software que permite la generación automática de filtros en VHDL (VHSIC hardware description language), a partir de la introducción de ciertas características, la cual puede ser sintetizada de manera inmediata mediante el uso de herramientas de síntesis.

2. ARQUITECTURA DE LOS FILTROS

De manera general, un filtro es caracterizado por la siguiente ecuación:

$$y[n] = a_0 x[n] + \cdots + a_p x[n-p] + b_1 y[n-1] + \cdots + b_p y[n-p] \tag{1}$$

donde p es el orden del filtro, los a_p's y b_p's los coeficientes, $x[n]$ la entrada del filtro en el tiempo n e $y[n]$ la salida del filtro en el tiempo n. En [7] se muestra como los operandos en complemento a dos (2'C) $x = (x_{(0)}.x_{(-1)} \ldots x_{(-l)})_2$ e $y = (y_{(0)}.y_{(-1)} \ldots y_{(-l)})_2$ de la ecuación 1 pueden ser expandidos en

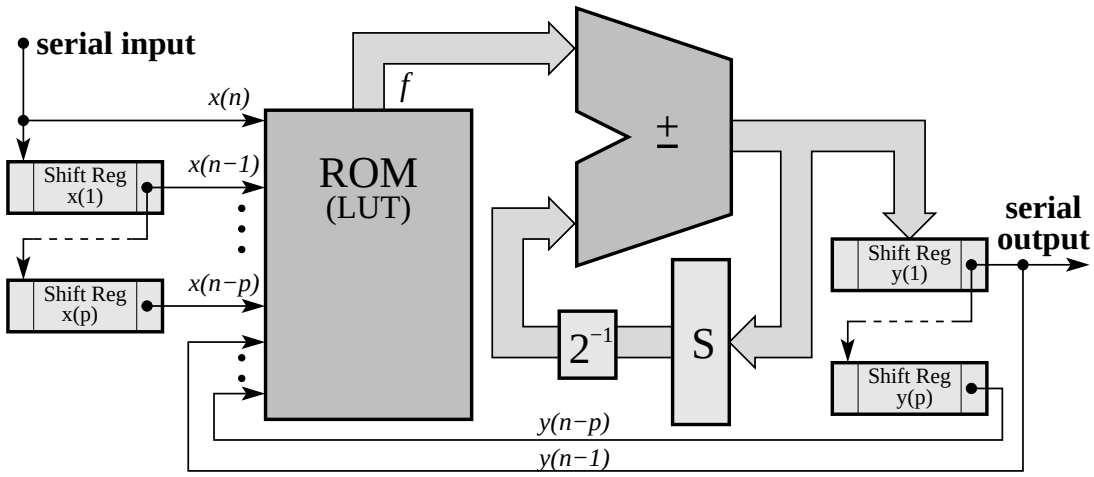


Figura 1: Arquitectura del filtro digital utilizando aritmética serie.

términos de sus bits individuales, obteniendo:

$$y[n] = a_0\left(-x_{(0)}^{[n]} + \sum_{j=-l}^{-1} 2^j x_{(j)}^{[n]}\right) + \cdots + a_p\left(-x_{(0)}^{[n-p]} + \sum_{j=-l}^{-1} 2^j x_{(j)}^{[n-p]}\right) + b_1\left(-y_{(0)}^{[n-1]} + \sum_{j=-l}^{-1} 2^j y_{(j)}^{[n-1]}\right) + \cdots + b_p\left(-x_{(0)}^{[n-p]} + \sum_{j=-l}^{-1} 2^j y_{(j)}^{[n-p]}\right) \quad (2)$$

Ahora, definimos una función f de la siguiente forma:

$$f(s,\ldots,u,v,\ldots,w) = a_0 s + \cdots + a_p u + b_1 v + \cdots + b_p w \quad (3)$$

donde $s,\ldots,u,v,\ldots,w$ son variables de un único bit y $a_0,\ldots,a_p,b_1,\ldots,b_p$ coeficientes constantes. Usando esta función, podemos re-escribir la expresión para $y[n]$ de la ecuación 1 como:

$$y[n] = \left(\sum_{j=-l}^{-1} 2^j f(x_{(j)}^{[n]},\ldots,x_{(j)}^{[n-p]},y_{(j)}^{[n-1]},\ldots,y_{(j)}^{[n-p]})\right) - f(x_{(0)}^{[n]},\ldots,x_{(0)}^{[n-p]},y_{(0)}^{[n-1]},\ldots,y_{(0)}^{[n-p]}) \quad (4)$$

Lo cual da origen a la arquitectura mostrada en la figura 1, la cual se presento y analizo detalladamente en [1]. Aquí, la función f es representada como una tabla pre-computada (LUT) de $((2p+1)\times(m+\lceil\log_2(2p+1)\rceil))$ bits, donde cada entrada corresponde a una combinación lineal que indica cuales coeficientes deben considerarse en cada momento del tiempo de acuerdo a los bits de direccionamiento, por lo tanto, en dicha entrada se almacena la suma de los coeficientes indicados.

Esta arquitectura posee una entrada y una salida serial. La memoria ROM es direccionada por medio del bit menos significativo de los registros que contienen los operandos x's e y's y su salida, junto con el contenido del registro S, es procesada en la unidad aritmética. Este resultado es acumulado nuevamente en el registro S. Luego de l ciclos el valor que obtenemos es la salida del filtro, la cual es almacenada en el registro $y(1)$ para cálculos futuros. Luego, el registro S es inicializado nuevamente en 0 y un nuevo cálculo comienza.

Existen principalmente 2 tipos de filtros digitales: IIR y FIR. Los filtros IIR calculan la salida a partir de un conjunto de muestras de entrada de una señal y un conjunto de salidas previas, por lo que su arquitectura es idéntica a la mostrada en la figura 1. En un filtro tipo FIR la salida depende solo de la entrada actual y de un conjunto de entradas previas, por lo que la arquitectura es como la mostrada anteriormente pero con una pequeña diferencia, ésta no posee retroalimentación y sólo posee un único registro de desplazamiento y, el cual contendrá el valor de salida del filtro.

Usando estas arquitecturas es posible construir filtros de cualquier orden, ya sea IIR o FIR, pero el tamaño requerido para la LUT que contiene los coeficientes pre-computados crecerá exponencialmente dependiendo del número de coeficientes del filtro. Por esta rabón, A continuación veremos como construir filtros de orden superior a partir de un conjunto de filtros de bajo orden.

2.1. Filtros IIR

Para construir filtros IIR de orden superior se hace uso de las propiedades de *conmutatividad* y *asociatividad* de los sistemas LTI (Linear Tine-Invariant). La propiedad de asociatividad establece que podemos analizar un sistema LTI "dividiéndolo" en subsistemas mas simples mientras que la propiedad de conmutatividad postula que si ubicamos en serie o cascada un grupo de subsistemas, éstos pueden ser colocados en cualquier orden sin afectar el desempeño final. Luego, interconectando subfiltros de bajo orden adecuadamente, podemos construir filtros de cualquier orden deseado. Esta técnica permite una reducción considerable en el espacio usado de la FPGA, ya que reemplazamos una gran LUT por un conjunto de pequeñas LUTs. En la figura 2 se muestra el esquema de interconexión empleado para este tipo de filtros, donde la salida del subfiltro i se conecta directamente a la entrada del subfiltro $i+1$:

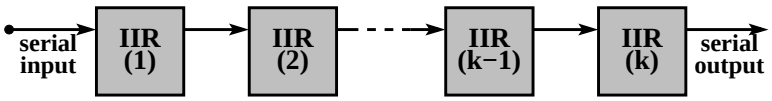


Figura 2: Esquema de interconexión para la construcción de filtros IIR de orden superior.

2.2. Filtros FIR

Debido a que comúnmente se requieren filtros FIR con una gran cantidad de coeficientes para obtener buenos comportamientos, es necesario hacer un análisis más detallado con el objetivo de reducir la lógica necesaria para la construcción de los mismos, permitiendo sintetizar filtros de mayor orden en una FPGA.

Como los coeficientes de un filtro FIR son simétricos, podemos sacar ventaja de esta simetría agrupando los mismos de manera tal de generar una LUT de menor tamaño. Esto se debe a que en la LUT se tendrán entradas repetidas y las herramientas de síntesis detectan esta situación y realizan las optimizaciones correspondientes.

Por lo tanto, dada la siguiente ecuación general de un filtro FIR:

$$y[n] = a_0x[n] + a_1x[n-1] + \cdots + a_{p-1}x[n-(p-1)] + a_px[n-p] \tag{5}$$

podemos reagrupar la ecuación, por ejemplo de a cuatro términos, de la siguiente forma:

$$\begin{aligned} y[n] = &\Big(a_0x[n] + a_1x[n-1] + a_{p-1}x[n-(p-1)] + a_px[n-p]\Big) + \\ &\Big(a_2x[n-2] + a_3x[n-3] + a_{p-3}x[n-(p-3)] + a_{p-2}x[n-(p-2)]\Big) + \cdots \end{aligned} \tag{6}$$

donde los términos entre paréntesis formarán cada uno de los subfiltros. El primer subfiltro utilizará los coeficientes a_0, a_1, a_{p-1} y a_p en su LUT, la cual será direccionada por los bits menos significativos de los registros $x[n]$, $x[n-1]$, $x[n-(p-1)]$ y $x[n-p]$, El segundo subfiltro utilizará los coeficientes a_2, a_3, a_{p-3} y a_{p-2} en su LUT, la cual será direccionada por los bits menos significativos de los registros $x[n-2]$, $x[n-3]$, $x[n-(p-3)]$ y $x[n-(p-2)]$, y así en adelante. El esquema de interconexión es mostrado en la figura 3.

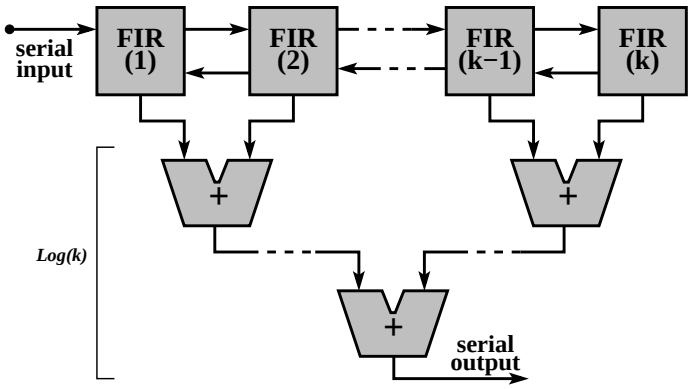


Figura 3: Esquema de interconexión para la construcción de filtros FIR de orden superior.

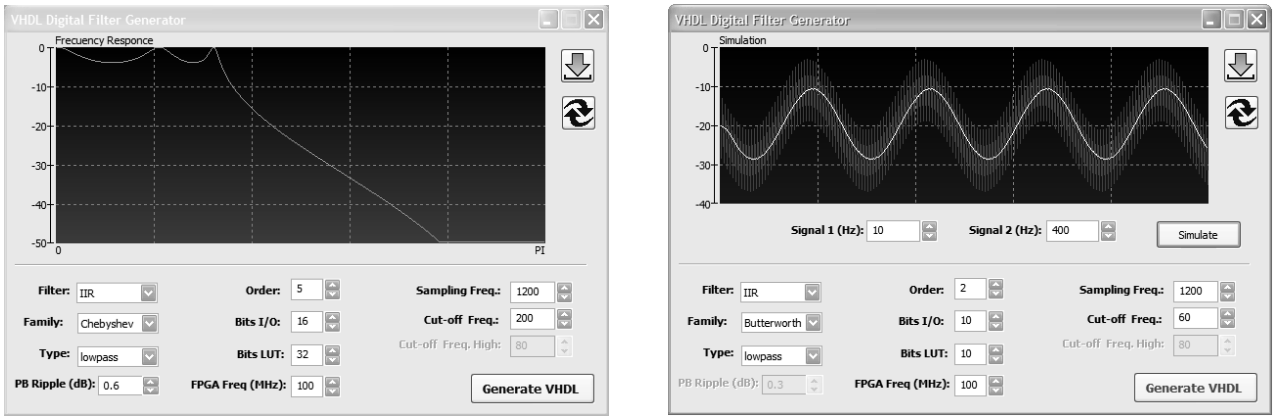
Como se observa en la figura, se deben interconectar los subfiltros de manera tal que la entrada pase a través de ellos serialmente llegando hasta el subfiltro k y retornando hacia el subfiltro 1, a fin de que cada subfiltro direccione su LUT con las entradas correspondiente de acuerdo a los coeficientes que utilice. Por último, las salidas producidas por los subfiltros son sumadas mediante sumadores serie para producir la salida del filtro de orden superior.

Un Filtro FIR que hace uso de k subfiltros, producirá un resultado cada $l + \lceil log_2(k) \rceil$ ciclos, donde l es la cantidad de bits de la entrada y $\lceil log_2(k) \rceil$ es la profundidad del árbol de sumadores.

3. HERRAMIENTA DE SOFTWARE

La arquitecturas para implementar filtros digitales como las que se mostraron en la sección anterior son muy simples de especificar en lenguajes de descripción de hardware tales como VHDL. Además, también es muy simple realizar especificaciones que utilicen un conjunto de estos filtros para construir filtros de orden superior. Por otro lado ya son conocidos los algoritmos que permiten encontrar los coeficientes de un filtro, ya sea para tipo IIR o FIR. Por estos motivos, se construyo una herramienta de software la cual se encarga de construir una especificación en VHDL para un filtro a partir de las características introducidas. En la figura 4 (a) se puede apreciar la interfaz gráfica de dicho sistema.

Para el caso de filtros digitales tipo IIR es posible seleccionar entre los métodos de aproximación, Chebyshev y Butterworth, ya sea para filtros pasa-bajo, pasa-alto y pasa-banda. Los algoritmos para encontrar los polos y ceros correspondientes al sistema pueden verse en [8]. Una vez encontrados los polos y los ceros pueden calcularse los coeficientes del sistema, que luego nos permiten construir la LUT. La herramienta además nos permite seleccionar la cantidad de bits tanto para las señales de entradas y salidas como para la LUT. El orden del filtro también puede ser seleccionado, como así también la frecuencia de corte, frecuencia de muestreo con la que trabajara el filtro y la frecuencia de la FPGA. En base a estos dos últimos valores y a la cantidad de bits de la señal de entrada se construye un divisor de frecuencia, con lo cual el filtro diseñado trabajara a la misma velocidad que el conversor analógico-digital.



(a) Diseño. (b) Simulación.

Figura 4: Interfaz gráfica.

Para el caso del FIR se introducen los mismos parámetros, con la diferencia que la familia a seleccionar puede ser Hamming, Von Hann y Rectangular. Los algoritmos para calcular los coeficientes también pueden verse en [8].

Cuando se intenta construir filtros de orden superior la herramienta hace uso de los esquemas propuestos para ambos casos.

Además, esta herramienta nos provee la posibilidad de realizar una simulación del filtro diseñado con el fin de verificar que el comportamiento sea el deseado antes de realizar la síntesis del mismo. Para esto se pueden ingresar dos señales de entrada, indicando su frecuencia respectivas, y se obtiene de manera grafica los resultados de la simulación. La simulación es realizada por medio del simulador GHDL[1], el cual es invocado por la herramienta. En la figura 4 (b) puede verse la la interfaz grafica del sistema para la simulación.

Una vez que se encontró el filtro deseado y se obtuvo su especificación en VHDL, la misma puede ser sintetizada directamente sobre una FPGA mediante las herramientas de síntesis existentes en la actualidad.

4. RESULTADOS EXPERIMENTALES

Diversos filtros digitales con distintas características fueron diseñados a fin de analizar su comportamiento, desempeño y utilización de celdas lógicas en una FPGA. La FPGA seleccionada para trabajar fue una Actel ProAsic250, la herramienta usada para realizar la síntesis fue ACTEL LIBERO IDE v7.3.

En el cuadro 1 se muestran los resultados obtenidos al realizar la síntesis de filtros FIR de distintos ordenes, con 10 bits tanto para la LUT como para las señales de entrada y salida, los cuales fueron obtenidos con la herramienta. Además se muestran los resultados obtenidos al implementar filtros sin la mejora propuesta en este artículo, tal como se plantea en [1], a fin de observar la mejora obtenida.

Los resultados muestran claramente que si la simetría de los coeficientes es aprovechada, obtenemos una mayor ganancia en cuanto a espacio utilizado en la FPGA, a la vez que aumentamos la frecuencia de reloj obtenida, con lo cual es posible sintetizar una mayor cantidad de coeficientes en una única FPGA, como mencionamos previamente, con la ventaja adicional de realizar una menor degradación del desempeño.

[1] http://ghdl.free.fr/

Cuadro 1: Resultados de síntesis de filtros FIR.

Orden del Filtro	Usando Simetría	Celdas Lógicas Usadas	Frecuencia de Reloj (MHz)
14	Si	6.18 %	101.153
14	No	6.22 %	87.658
30	Si	12.16 %	87.897
30	No	13.49 %	85.092
70	Si	28.40 %	90.465
70	No	31.14 %	78.425

Cuadro 2: Resultados de síntesis de filtros IIR.

Orden del Filtro	Celdas Lógicas Usadas	Frecuencia de Reloj (MHz)
2	3.06 %	112.02
4	6.28 %	102.28
5	7.64 %	101.54

En el cuadro 2, se muestran los resultados que se obtuvieron al realizar la síntesis de filtros IIR, donde se puede observar que un filtro de orden 5, el cual se considera como un filtro con un muy buen desempeño, ocupa un 7.64 % de la FPGA, con lo cual es posible tener otro sistema en ejecucion sobre ésta.

Durante los experimentos realizados, se pudo observar que los filtros con 10 bits de precisión para la LUT ofrecen un desempeño adecuado. Para mostrar esto se presentan los resultados obtenidos en la simulación para un filtro FIR pasa-bajo de orden 70, usando el tipo de ventana Hamming, 10 bits para la tabla LUT así como para las señales de entrada y salida, una frecuencia de corte de 60 Hz y frecuencia de muestreo de 1200 Hz. El mismo fue usado para filtrar una señal de entrada compuesta por una señal de 10 Hz contaminada por una señal de 200 Hz. La figura 5 muestra el efecto del filtro sobre la señal recién mencionada. Se puede observar que la señal de 200 Hz ha sido filtrada dejando pasar la señal de 10 Hz, como se esperaba.

A modo de ejemplo, se muestra en el anexo 1 un código VHDL correspondiente a un filtro IIR de orden 2, pasa-bajo, con frecuencia de corte a 60 Hz y que trabaja a una frecuencia de muestreo de 1200 Hz. La LUT y las señales de entrada y salida son de 10 bits. Los coeficientes del mismo y el valor de ganancia fueron obtenidos con la herramienta. La ecuación característica para dicho filtro es la siguiente:

$$y[n] = 0{,}020083 \cdot (x[n] + 2 \cdot x[n-1] + x[n-2]) + 1{,}561018 \cdot y[n-1] - 0{,}641352 \cdot y[n-1] \quad (7)$$

5. CONCLUSIONES

Como se pudo apreciar a lo largo del artículo, las técnicas propuestas para la implementación de filtros digitales de orden superior permiten una importante reducción en la utilización de celdas lógicas de una FPGA, produciendo filtros cuyos tamaños crecen de forma lineal con la cantidad de coeficientes y no de forma exponencial, ya que se utilizan un conjunto de pequeñas LUTs en lugar de una única

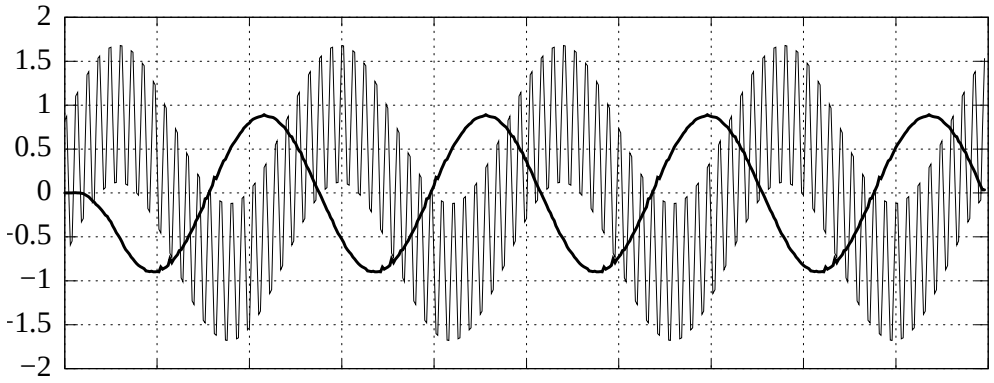


Figura 5: Resultados de simulación.

LUT de gran tamaño. Además para el caso de filtros FIR se pudo observar que es posible reducir aún más la cantidad de celdas lógicas utilizadas, debido a que la simetría de los coeficientes nos permiten agruparlos de manera tal que varias entradas de una LUT sean iguales, permitiendo a las herramientas de síntesis realizar optimizaciones sobre la especificación.

Por otro lado, la herramienta construida permite que usuarios, tanto experimentados como no experimentados, puedan obtener especificaciones en VHDL rápidamente, con sólo introducir algunas características del filtro deseado, pudiendo sintetizarlas directamente sin necesidad de conocer como es implementado internamente el filtro, ni como se obtienen los coeficientes del mismo.

REFERENCIAS

[1] Arroyuelo Mónica, Arroyuelo Jorge y Grosso Alejandro, "FPGA-Based Digital Filters Using Bit-Serial Arithmetic". *XIII Congreso Argentino de Ciencias de la Computación.* Octubre de 2007. Corrientes y Resistencia, Argentina.

[2] Rawski, Tomaszewicz, Selvaraj and Luba. "Efficient Implementation of Digital Filters with Use of Advanced Synthesis Methods Targeted FPGA Architectures". *Digital System Design, 2005. Proceedings. 8Th Euromicro Conference on.* 30 Aug. - 3 Sept. 2005. Pages 460-466.

[3] Knut Arne Vinger and Jim Torrensen. "Implementing Evolution of FIR-Filters Efficiently in an FPGA". *Evolvable Hardware, 2003. Proceedings. NASA/DoD Conference on.* July 9-11, 2003. Pages 26-29.

[4] Kalivas, Tsirikos, Bougas and Pekmestzi. "100 % Operational Efficient Bit-Serial Programmable FIR Digital Filters". *EUSIPCO 2005 - 13Th European Signal Processing Conference.* September 4-8, 2005. Antalya, Turkey.

[5] Chi-Jui Chou, Satish Mohanakrishnan and Joseph Evans. "FPGA Implementation of Digital Filters". *International Conference on Signal Processing Applications and Technology.* Berlin, 1993. Pages 251-255.

[6] Sang-Hun Yoon, Jong-wha Chong and Chi-Ho Lin. "An Area Optimization Method for Digital Filter Design". *ETRI Journal, volume 26, Number 6.* December 2004. Pages 545-553.

[7] behrooz Parhami. "Computer Arithmetic: Algorithms and Hardware Designs". New York: Oxford University Press, 2000.

[8] Paul a. Lynn and Wolfgang Fuerst. "Introductory Digital Signal Processing with Computer Applications". Revised Edition. *John Wiley & Sons.* 1994.

ANEXO I (CÓDIGO VHDL)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity dig_filter is port (
       x    : in  signed(0 to 9);
       clk  : in  std_logic;
       rst  : in  std_logic;
       y    : out signed(0 to 9));
end;

architecture df of dig_filter is
   constant cBitsIO        : integer := 10;
   constant cCoefO1        : integer := 3;
   constant cCoefO2        : integer := 5;
   constant cBitsCoef      : integer := 13;

   type tTableCoefO1 is array(0 to 2**cCoefO1-1) of signed(0 to cBitsCoef-1);
   type tTableCoefO2 is array(0 to 2**cCoefO2-1) of signed(0 to cBitsCoef-1);
   constant SF1_cTableCoef : tTableCoefO2 :=(
               "0000000000000",
               "1111010110111",
               "0001100011111",
               "0000111010110",
               "0000000001010",
               "1111011000001",
               "0001100101001",
               "0000111100001",
               "0000000010100",
               "1111011001100",
               "0001100110011",
               "0000111101011",
               "0000000011110",
               "1111011010110",
               "0001100111110",
               "0000111110101",
               "0000000001010",
               "1111011000001",
               "0001100101001",
               "0000111100001",
               "0000000010100",
               "1111011001100",
               "0001100110011",
               "0000111101011",
               "0000000011110",
               "1111011010110",
               "0001100111110",
               "0000111110101",
               "0000000101001",
               "1111011100000",
               "0001101001000",
               "0001000000000"
   );
   signal Counter_reg      : signed(0 to cBitsIO-1);
   signal Counter_input    : signed(0 to cBitsIO-1);
```

```
   signal SF1_x_n_reg      : signed(0 to cBitsIO-1);
   signal SF1_x_n_input    : signed(0 to cBitsIO-1);
   signal SF1_x_n_1_reg    : signed(0 to cBitsIO-1);
   signal SF1_x_n_1_input  : signed(0 to cBitsIO-1);
   signal SF1_x_n_2_reg    : signed(0 to cBitsIO-1);
   signal SF1_x_n_2_input  : signed(0 to cBitsIO-1);
   signal SF1_y_n_reg      : signed(0 to cBitsIO-1);
   signal SF1_y_n_input    : signed(0 to cBitsIO-1);
   signal SF1_y_n_1_reg    : signed(0 to cBitsIO-1);
   signal SF1_y_n_1_input  : signed(0 to cBitsIO-1);
   signal SF1_y_n_2_reg    : signed(0 to cBitsIO-1);
   signal SF1_y_n_2_input  : signed(0 to cBitsIO-1);
   signal SF1_s_reg        : signed(0 to cBitsCoef-1);
   signal SF1_s_input      : signed(0 to cBitsCoef-1);
   signal SF1_f            : signed(0 to cBitsCoef-1);
   signal SF1_opndo_1      : signed(0 to cBitsCoef-1+2);
   signal SF1_opndo_2      : signed(0 to cBitsCoef-1+2);
   signal SF1_add          : signed(0 to cBitsCoef-1+2);
   signal SF1_address      : signed(0 to cCoefO2-1);

-------------------------------------------------------------
begin
-------------------------------------------------------------
  counter_input <= counter_reg(counter_reg'high) &
                   counter_reg(0 to counter_reg'high-1);
  SF1_x_n_input <= x when counter_reg(counter_reg'high)='1' else
                    '0' & SF1_x_n_reg(0 to SF1_x_n_reg'high-1);
  SF1_x_n_1_input <= SF1_x_n_reg(SF1_x_n_reg'high) &
                     SF1_x_n_1_reg(0 to SF1_x_n_1_reg'high-1);
  SF1_x_n_2_input <= SF1_x_n_1_reg(SF1_x_n_1_reg'high) &
                     SF1_x_n_2_reg(0 to SF1_x_n_2_reg'high-1);
  SF1_y_n_2_input <= SF1_y_n_1_reg(SF1_y_n_1_reg'high) &
                     SF1_y_n_2_reg(0 to SF1_y_n_2_reg'high-1);
  SF1_y_n_1_input <= SF1_add(4 to 4+SF1_y_n_1_input'high) when
                     counter_reg(counter_reg'high)='1' else
                     '0' & SF1_y_n_1_reg(0 to SF1_y_n_1_reg'high-1);
  SF1_y_n_input <= SF1_add(4 to 4+SF1_y_n_1_input'high) when
                    counter_reg(counter_reg'high)='1' else
                    SF1_y_n_reg;
  SF1_opndo_1 <= '0' & SF1_s_reg(0) & SF1_s_reg(0 to cBitsCoef-2) & '1';
  SF1_opndo_2 <= '0'&(SF1_f xor (0 to (cBitsCoef-1) => counter_reg(counter_reg'high)))&
                   counter_reg(counter_reg'high);
  SF1_add <= SF1_opndo_1 + SF1_opndo_2;
  SF1_s_input <= (others => '0') when counter_reg(counter_reg'high) = '1' else SF1_add(1 t
  SF1_address <= (SF1_x_n_reg(SF1_x_n_reg'high),
                  SF1_x_n_1_reg(SF1_x_n_1_reg'high),
                  SF1_x_n_2_reg(SF1_x_n_2_reg'high),
                  SF1_y_n_1_reg(SF1_y_n_1_reg'high),
                  SF1_y_n_2_reg(SF1_y_n_2_reg'high));
  with SF1_address select SF1_f <=
       SF1_cTableCoef(0)  when "00000",
       SF1_cTableCoef(1)  when "00001",
       SF1_cTableCoef(2)  when "00010",
       SF1_cTableCoef(3)  when "00011",
       SF1_cTableCoef(4)  when "00100",
       SF1_cTableCoef(5)  when "00101",
       SF1_cTableCoef(6)  when "00110",
```

```
      SF1_cTableCoef(7)  when "00111",
      SF1_cTableCoef(8)  when "01000",
      SF1_cTableCoef(9)  when "01001",
      SF1_cTableCoef(10) when "01010",
      SF1_cTableCoef(11) when "01011",
      SF1_cTableCoef(12) when "01100",
      SF1_cTableCoef(13) when "01101",
      SF1_cTableCoef(14) when "01110",
      SF1_cTableCoef(15) when "01111",
      SF1_cTableCoef(16) when "10000",
      SF1_cTableCoef(17) when "10001",
      SF1_cTableCoef(18) when "10010",
      SF1_cTableCoef(19) when "10011",
      SF1_cTableCoef(20) when "10100",
      SF1_cTableCoef(21) when "10101",
      SF1_cTableCoef(22) when "10110",
      SF1_cTableCoef(23) when "10111",
      SF1_cTableCoef(24) when "11000",
      SF1_cTableCoef(25) when "11001",
      SF1_cTableCoef(26) when "11010",
      SF1_cTableCoef(27) when "11011",
      SF1_cTableCoef(28) when "11100",
      SF1_cTableCoef(29) when "11101",
      SF1_cTableCoef(30) when "11110",
      SF1_cTableCoef(31) when others;
  y <= SF1_y_n_reg;

   write: process(clk, rst)
   begin
     if rst'event and rst = '1' then
        SF1_s_reg     <= (others => '0');
        SF1_x_n_reg   <= (others => '0');
        SF1_x_n_1_reg <= (others => '0');
        SF1_x_n_2_reg <= (others => '0');
        SF1_y_n_reg   <= (others => '0');
        SF1_y_n_1_reg <= (others => '0');
        SF1_y_n_2_reg <= (others => '0');
        counter_reg   <= (0 to counter_reg'high-1 => '0',
                          counter_reg'high => '1');
     elsif clk'event and clk = '1' then
        SF1_s_reg     <= SF1_s_input;
        SF1_x_n_reg   <= SF1_x_n_input;
        SF1_x_n_1_reg <= SF1_x_n_1_input;
        SF1_x_n_2_reg <= SF1_x_n_2_input;
        SF1_y_n_reg   <= SF1_y_n_input;
        SF1_y_n_1_reg <= SF1_y_n_1_input;
        SF1_y_n_2_reg <= SF1_y_n_2_input;
        counter_reg   <= counter_input;
     end if;
   end process;
end df;
```