IMPLEMENTACIÓN EN HARDWARE DE UN SUMADOR DE PUNTO FLOTANTE BASADO EN EL ESTÁNDAR IEEE 754-2008

IMPLEMENTATION IN HARDWARE OF A FLOATING POINT ADDER BASED IN IEEE 754-2008 STANDARD

Juan José Raygoza P.[1], Susana Ortega C. [1], Miguel A. Carrazco [1], Adrian Pedroza C.[1]
juan.raygoza@cucei.udg.mx / Susana.ortega@cucei.udg.mx / miguel.carrazco@red.cucei.udg.mx
adrian.pedroza@red.cucei.udg.mx

Recibido: 03 marzo, 2009 / Aceptado: 28 octubre 28, 2009 / Publicado: 31 diciembre, 2009

RESUMEN. Este artículo presenta el diseño de un sumador de punto flotante descrito en lenguaje VHDL, basado en el estándar para Aritmética de Punto Flotante de IEEE (754™-2008) para microprocesadores, del cual se utiliza el formato binario para precisión simple de 32 bits. El estándar define formatos para representar diferentes tipos de datos los cuales son: normal, subnormal, cero positivo, cero negativo, infinito positivo, infinito negativo y un no número (NaN). Muchas aplicaciones basadas en procesadores embebidos requieren la capacidad para realizar operaciones aritméticas de punto flotante, lo cual es fundamental para una mejor precisión y desempeño del sistema en el procesamiento de los datos. El sumador ha sido diseñado considerando los parámetros de velocidad, área utilizada dentro de la FPGA y consumo de potencia estimada; además el circuito ha sido sintetizado y simulado sobre las FPGAs Spartan®3 (3s200ft256-4), Virtex® II (2v1000fg256-4) y Virtex® 4 (4vfx12sf363-12) de la familia Xilinx®. El sumador ha sido diseñado por bloques de modo que podamos optimizar el proceso de cálculo por medio de las líneas de control, para que sólo la unidad indicada procese los datos. El circuito ha sido interconectado en un diagrama esquemático principal para la fácil incorporación de los bloques de control, entradas, salidas, cálculo simbólico y aritmético.

PALABRAS CLAVE: Aritmética Binaria, FPGA, VHDL, Sistemas Embebidos.

ABSTRACT. This paper presents a floating point adder described in VHDL language, based on IEEE Standard for Floating-Point Arithmetic (754™-2008) for microprocessors, which it uses the binary format for simple precision of 32 bits. The standard defines formats to represent different data types which are: normal, subnormal, positive zero, negative zero, positive infinity, negative infinity, and not-a-number (NaN). Many applications based on embedded processors require the capability to realize arithmetical operations of floating point, which is fundamental to improve the precision and performance of the system into the data process. The adder has been designed considering the parameter of speed, used area into FPGA and estimated power consumption, in addition the circuit has been synthesized and simulated on the FPGA Spartan® 3 (3s200ft256-4), Virtex® II (2v1000fg256-4) and Virtex® 4 (4vfx12sf363-12) of Xilinx® family. The adder has been designed by blocks, so that we may optimize the process of calculation by means of the control lines so that only the indicated unit processes the data. The circuit has been interconnected in a main schematic diagram to easy incorporation of control, input, output, symbolic and arithmetic blocks.

KEYWORDS: Binary Arithmetic, FPGA, VHDL, Embedded Systems.

1. Introducción

El estándar de la IEEE para aritmética en punto flotante (IEEE Standard for Floating-Point Arithmetic [1]), especifica formatos para su uso en los sistemas informáticos, también define la precisión simple, doble y extendida así como los métodos para el intercambio de datos. Este estándar es el más utilizado para resolver operaciones de alta precisión. Sin embargo, el problema nace cuando los números enteros ya no son suficientes para aplicaciones donde se utilizan números muy pequeños o demasiado grandes y es indispensable incluir una parte fraccionaria o una notación científica.

Se define que *'x'* representa bits, *"x"* nombres de bloques, los números sin comillas son decimales y las simulaciones son representadas en números hexadecimales para una mejor la visualización de los datos.

[1] Centro Universitario de Ciencias Exactas e Ingenierías (CUCEI), Blvd. Marcelino García Barragán #1421, Guadalajara, Jalisco, 44430, México. www.cucei.udg.mx

Se utiliza el formato de precisión simple de 32 bits donde un dato está compuesto de signo, exponente y mantisa debidamente ordenados; el código esta descrito, sintetizado y simulado en el software ISE™ 7.1i [2] de Xilinx® [3].

2. Formato de punto Flotante (Estándar IEEE 754-2008)

El formato con punto flotante de precisión sencilla [4] se representa en la aritmética binaria [5] con un bit de signo (31), 8 bits de exponente (30 al 23) -éste está sesgado con 127 (bias) y es un entero- y 23 bits de mantisa (22 al 0) como se ilustra en la figura 1. Para obtener una mayor precisión, se omite el bit correspondiente a la parte entera de la mantisa y se supone ‘1’ cuando es un número normal y ‘0’ cuando es un subnormal, obteniendo una precisión real de 24 bits de mantisa.

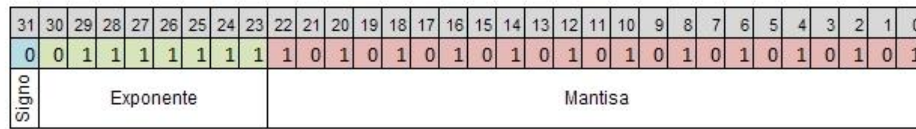

Figura 1. Formato de precisión sencilla.

Tabla 1. Representaciones de tipos de datos y sus valores.

TIPO DE DATO	*SIGNO*	*EXPONENTE*	*MANTISA*	*VALOR DECIMAL*
Cero Positivo	0	E = 0	M = 0	0
Cero Negativo	1	E = 0	M = 0	-0
Subnormal	S	E = 0	M > 0	$(-1)^S*(0.M)*2^{-126}$
Normal	S	0 < E < 255	M = X	$(-1)^S *(1.M)*2^{E-127}$
Infinito	S	E = 255	M = 0	$(-1)^S *\infty$
No un Número (NaN)	S	E = 255	M > 0	$(-1)^S$ *NaN

Los tipos de datos que se pueden procesar según el estándar IEEE 754-2008 se ilustran en la tabla 1 y para encontrar el valor de *M* de la se utiliza la siguiente ecuación:

$$M = \sum_{k=0}^{22} m_{(22-k)} 2^{-(1+k)} \tag{1}$$

Donde *m* son los bits 22 a 0 de la mantisa ($m_{22}, m_{21}, \ldots, m_1, m_0$).

Una vez obtenida *M*, se convierte el valor binario a decimal de los datos normales y subnormales.

3. Implementación en FPGAs

La figura 2 ilustra el diagrama esquemático principal [6] del circuito sumador de punto flotante de 32 bits, el cual está formado a base de bloques diseñados para el cálculo de cada tipo de datos y sus combinaciones. Éste es implementado en FPGAs [7] de Xilinx ®.

El diagrama esquemático se compone de dos entradas de datos a sumar y de 11 bloques: "datatype", "e_case", "n_case", "dn_adder", "dn_sel", "dn_sum", "dexp", "shift", "sum", "decod" y "mux", de los cuales sólo el primero y los dos últimos están funcionando continuamente, mientras los demás bloques únicamente procesan los datos dependiendo de la combinación del tipo de dato de A y B, dado que cada uno desempeña una función propia y procesan tipos de datos diferentes a los demás bloques.

Figura 2. Diagrama esquemático principal.

A continuación se describe la función específica que desempeña cada bloque:

• Bloque "datatype":
Este bloque es el primero de los módulos y se encarga de identificar los datos de entrada A y B especificados en el resumen, como se muestra en la tabla 1. Por cada dato de entrada se genera una señal de 3 bits de salida con la que se indica el tipo de dato de cada una de las entradas, como se muestra en la tabla 2.

Tabla 2. Señales de control de cada tipo de dato.

TIPO DE DATO	*OC(2)*	*OC(1)*	*OC(0)*	*DECIMAL*
Cero	'0'	'0'	'0'	0
Subnormal	'0'	'0'	'1'	1
Normal	'1'	'0'	'X'	4,5
Infinito	'1'	'1'	'0'	6
No un Número	'1'	'1'	'1'	7

• Bloque "e_case":

Cuando se realiza la suma de los datos A y B se pueden tener varias combinaciones entre los datos que impliquen cálculos simbólicos como la adición de un número infinito con un número normal, un NaN con un cero, etc. El bloque "e_case" se activa sólo en caso de que estas combinaciones de datos se den y las resuelve conforme a la tabla 3, en donde se muestran las 12 posibles combinaciones en las que no es necesario realizar la suma para obtener el resultado.

Tabla 3 Resultados del procesamiento de datos del bloque "e_case".

SIGNO A	*OCA(2:0)*	*DATO A*	*SIGNO B*	*OCB(2:0)*	*DATO B*	*SIGNO S*	*SALIDA*
'SA'	'111'	NaN	'SB'	'XXX'	Dato B	'1'	NaN
'SA'	'XXX'	Dato A	'SB'	'111'	NaN	'1'	NaN
'1'	'110'	Infinito	'1'	'110'	Infinito	'1'	Infinito
'1'	'110'	Infinito	'0'	'110'	Infinito	'1'	NaN
'0'	'110'	Infinito	'1'	'110'	Infinito	'1'	NaN
'0'	'110'	Infinito	'0'	'110'	Infinito	'0'	Infinito
'SA'	'110'	Infinito	'SB'	'11X'	Normal	'SA'	Infinito
'SA'	'110'	Infinito	'SB'	'001'	Subnormal	'SA'	Infinito
'SA'	'11X'	Normal	'SB'	'110'	Infinito	'SB'	Infinito
'SA'	'001'	Subnormal	'SB'	'110'	Infinito	'SB'	Infinito
'SA'	'000'	Cero	'SB'	'110'	Infinito	'SB'	Infinito
'SA'	'000'	Cero	'SB'	'11X'	Normal	'SB'	Normal
'SA'	'000'	Cero	'SB'	'001'	Subnormal	'SB'	Subnormal
'SA'	'XXX'	Dato A	'SB'	'000'	Cero	'SA'	Dato A

SA, SB, X Representan 1/0

• Bloque "n_case":

Éste trabaja como un decodificador que, a partir de las dos señales del bloque "datatype", detecta si los datos A y B son del tipo normal, subnormal o su combinación, y nos generan cuatro salidas de habilitación para los bloques "dd_adder", "dn_sel" y "dexp", de los cuales solo uno puede estar activo a la vez.

•Bloque "dn_sel", "dn_sum":

El bloque "dn_sel" recibe los dos datos y los ordena como subnormal o normal en un registro especifico para cada uno de ellos, los cuales se envían a "dn_sum" el cual concatena un '1' a la izquierda de la mantisa del dato normal y '0' al subnormal para procesar la suma. La figura 3 muestra la alineación del punto de las mantisas, y el proceso de normalización en el caso que la mantisa resultante sea subnormal.

```
Alineación de Mantisas
if iec <= "00010111"  then                              -- si el exponente del dato normal es <= 23
  r := (conv_integer(iec));                             -- conversión a entero de iec
  sden := vz(r-1 downto 0) & mden(22 downto r-1);       -- desplazamiento del número subormalizado
  snor := '1' & mnor;                                   -- concatena '1' al normal
  ec := iec;
Proceso de Normalización                                -- sum: suma de las mantisas de A y B
if  sum(23) = '0'  and iec > "00000001" then            -- si sum(23)=0, y iec>1: el resultado es subnormal
  sum := sum(22 downto 0) & '0';                        -- normalización de la mantisa
  ec := iec - '1';                                      -- decremento de exponente por la normalización
elsif sum(23) = '0' and iec = "00000001" then           -- si sum(23)=0 y iec=1: el resultado queda subnormal
  ec := "00000000";                                     -- exponente de salida = 0
end if;
```

Figura 3. Fragmento del código para normalizar en el "dn_sum".

• Bloques "dexp", "shift", "sum":

Estos bloques se encargan de la suma de números normales. El bloque "dexp" entrega en registros el dato mayor, el dato menor y la diferencia entre exponentes de A y B. El bloque "shift" concatena un '1' en ambas mantisas y desplaza a la derecha la mantisa correspondiente al dato menor para igualar exponentes. El bloque "sum" se encarga de procesar la suma y cuenta con un bit de acarreo, el cual incrementara el exponente en 1; en el caso que el exponente del dato mayor sea 254 y se obtenga un acarreo en la suma de las mantisas se genera a la salida un infinito.

En algunos casos, en la suma de dos datos normales con signo diferente puede resultar una mantisa subnormal con un exponente diferente de cero y se tendrá que normalizar, como se muestra en la figura 4, este código es basado en la ecuación (2). El método de redondeo utilizado en los bloques "sum" y "dn_sum" es el redondeo hacia cero ya que agiliza el procesamiento despreciando los bits que se pierden al desplazar las mantisas.

```
ze := "00000000";                                   -- ze,mc: son registros
mc := "000000000000000000000000";
cicloa:for i in 23 downto 0 loop                    -- ciclo for: para contar los 0's
   if ms(i) = '0' then                              -- ms: resultado de la suma
      ze := ze + '1';                               -- ze: contiene el conteo de 0's
   else
      exit cicloa;                                  -- si encuentra un '1' sale del ciclo for
   end if;
end loop;
if ze = "00011000" then                             -- EL dato es Zero
   oes <= "00000000";                               -- oes: exponente de salida
   es := "00000000";
elsif ieu > ze then                                 -- El dato es Normal
   oes <= ieu - ze;                                 -- ieu: exponente mayor de los datos A y B
   es := ze;
else -- ieu <= ze                                   -- El dato es Subnormal
   oes <= "00000000";
   es := ieu - '1';                                 -- es: bits a desplazar
end if;
mc(23 downto conv_integer(es)) := ms((23 - conv_integer(es)) downto 0);
oms <= mc(22 downto 0);                             -- oms: mantisa de salida
```

Figura 4. Código para la normalización de mantisa.

Desplazamiento de mantisa.

En los casos que se obtiene como resultante una mantisa subnormal *m* con un exponente mayor a cero, el método utilizado es contar los *n* ceros a la izquierda de la mantisa, después desplazar *n* bits a la izquierda y transferirlos a la salida. La ecuación para obtener la mantisa normalizada (*mn)* se muestra a continuación:

$$mn_{(k)} = m_{(k-n)}\, u(k-n) \quad (2)$$

Donde $k = [0,1,\ldots, 22,23]$ y $u(x)$ es la función escalón unitario

$$u(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

• Bloque “dd_adder”:

Este bloque se encarga de la adición de números subnormales. Se concatena un ‘0’ a la mantisa por la izquierda, que se utiliza en caso de acarreo. Parte del código se muestra en la figura 5.

```
om <= ms(22 downto 0);      -- ms: suma  de mantisas A y B, om: mantisa de salida
oe   <= ee + ms(23);        -- ms(23): bit de acarreo, oe: exponente de salida, ee = "00000000"
oen <= ien;                 -- oen: señal de habilitación
```

Figura 5. Fragmento del código para el acarreo.

• Bloques “decod” y “mux”:

El bloque “decod” decodifica las señales de habilitación que recibe de “e_case”, “dd_adder”, “dn_sum” y “sum” que indican cuál de los bloques contiene la suma de los datos de entrada y genera una señal binaria de dos bits que controlan el bloque “mux”. Este último, dependiendo de las señales de control, selecciona la salida adecuada para la suma de los datos A y B.

4. Resultados de la implementación

La implementación del sumador de punto flotante se describió en VHDL y se realizó en 3 familias de FPGAs de Xilinx, Virtex® 4, Virtex® II y Spartan® 3.

Los resultados de ocupación de los recursos en FPGAs se muestran en la tabla 4 y la figura 6, en donde se puede observar que el circuito sumador puede ser implementado en las 3 FPGAs.

Tabla 4 Tabla de ocupación para Spartan® 3, Virtex® II y Virtex® 4.

Dispositivo Utilizado	*Spartan 3 (3s200ft256-4)*			*Virtex 2 (2v1000fg256-4)*			*Virtex 4 (4vfx12sf363-12)*		
Ocupación	*Usadas*	*Total*	*%*	*Usadas*	*Total*	*%*	*Usadas*	*Total*	*%*
Número de Slices	788	1920	41.04%	771	5120	15.06%	771	5472	14.09%
Número de LUTs	1385	3840	36.07%	1385	10240	13.53%	1385	10944	12.66%
Número de IOBs utilizadas	96	173	55.49%	96	172	55.81%	96	240	40.00%

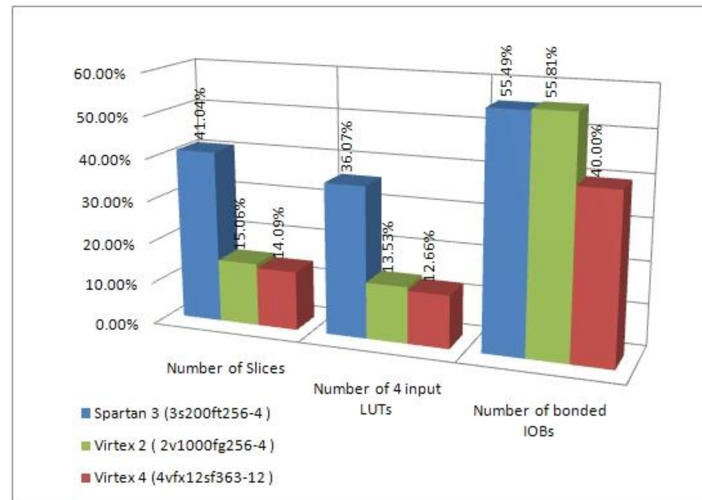

Figura 7. Gráfica de ocupación para las 3 tecnologías.

Con respecto a los tiempos de retardo dentro de las FPGAs encontramos que la Virtex 4 es en promedio dos veces más rápida que la Virtex II y la Spartan 3. En la tabla 5 y la figura 7 se muestran los retardos para las 3 familias de FPGAs.

Tabla 5 Tabla de tiempos de retardo (ns).

Pad Origen - destino	*Spartan 3 (3s200ft256-4)*	*Virtex 2 (2v1000fg256-4)*	*Virtex 4 (4vfx12sf363-12)*
EXP_A(0) - EXP_S(0)	27.827	26.039	13.254
EXP_A(0) - MAN_S(0)	28.478	26.578	13.5
EXP_A(0) - SIG_S	11.677	9.931	5.983
MAN_A(0) - EXP_S(0)	27.061	24.905	13.536
MAN_A(0) - MAN_S(0)	27.712	27.061	13.782
MAN_A(0) - SIG_S	12.54	11.009	7.228
SIG_A - EXP_S(0)	13.143	11.814	7.418
SIG_A - MAN_S(0)	10.015	8.486	5.231
SIG_A - SIG_S	10.015	8.417	5.202

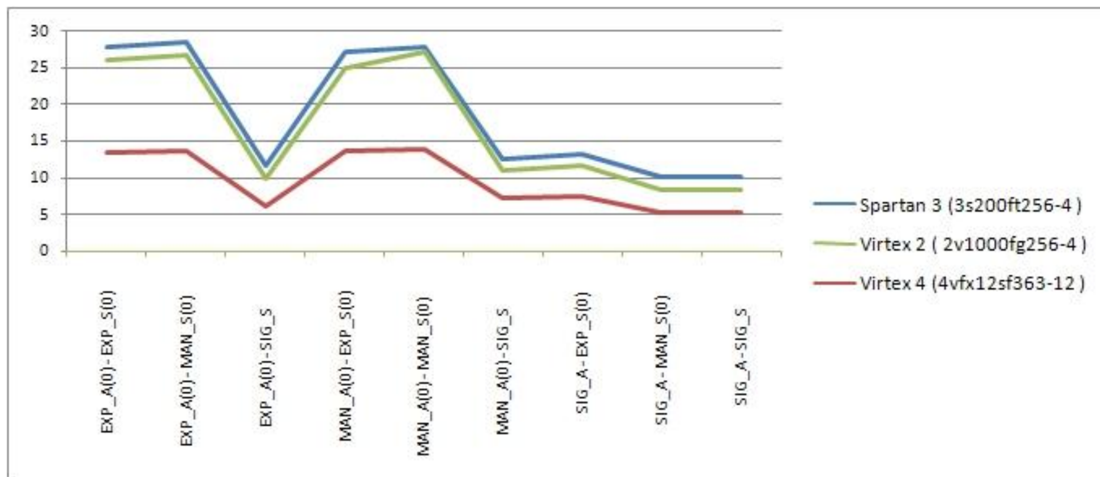

Figura 8. Gráfica de retardos de tiempo para las tres tecnologías.

Los resultados de la simulación del circuito sumador de punto flotante se muestran en la figura 9 en donde el cursor 20.8 indica el dato *A* que contiene ‘11111111’ de exponente, y la mantisa es diferente a cero. Revisando la tabla 2, observamos que éste es un dato NaN y, de igual manera, que el dato B que contiene ‘10101010’ de exponente es un número normal. La tabla 3 indica que el resultado de la suma es un dato NaN, lo que concuerda con la salida *S*.

Un segundo caso lo podemos ver en la misma figura 9 con el segundo dato A que es un número normal pues tiene un ‘0’ en el signo, ‘11111110’ en el exponente y ‘10101010101010101010101’ de mantisa. Utilizando la ecuación (1) se obtiene el valor decimal de *M* (0.66666662693023) y la tabla 1 indica el valor decimal $2.83568632339979*10^{+38}$. El dato *B* también es un número normal, su valor decimal es $-2.8356855121034*10^{+38}$ y la salida *S* es $8.11296384146067*10^{+31}$, lo cual concuerda con el valor real de la suma.

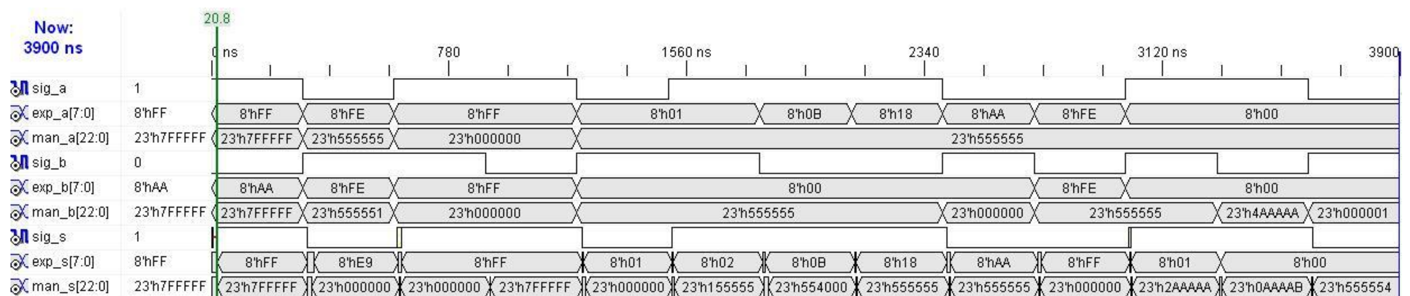

Figura 9. Simulación del sumador de punto flotante de 32 bits incluyendo todos los tipos de datos.

5. Conclusiones y trabajos futuros.

Se comprobó que el circuito propuesto puede ser implementado en las 3 familias de FPGAs de Xilinx debido a su ocupación. Una de las ventajas de implementar este sumador en FPGAs es que tenemos procesos independientes al mismo tiempo y se agilizan los cálculos.

El tiempo de latencia para realizar la suma es de 17 ns en virtex, 4, 25 ns en Virtex II y 27 ns en Spartan 3, lo que indica que la velocidad de cálculo en la Virtex 4 es casi el doble que las otras dos familias de FPGAs. Además se obtuvo un estimado de potencia en la familia Spartan® 3 de 62 mW y en la Virtex® II de 41 mW.

La utilización de las FPGAs nos brinda la flexibilidad para reconfigurar e implementar o modificar el diseño fácilmente y con bajo costo. Además, el circuito está hecho en VHDL estándar, por lo que es fácil de exportar a otras tecnologías de FPGAs y ASIC.

Como trabajo futuro se podría implementar en hardware una FPU (*Floating Point Unit*) que permita utilizar sólo el bloque adecuado para resolver la operación aritmética y, al mismo tiempo, mantener los demás en modo de bajo consumo, ahorrando potencia y mejorando la velocidad de procesamiento. Debido a la facilidad de descripción de hardware en el código VHDL, se podría modificar el tamaño del dato para trabajar con 32, 64 y 128 bits, que específica la norma.

Referencias

1. Institute of Electrical and Electronics Engineers, Inc. (29 agosto 2001). IEEE Standard for Floating-Point Arithmetic. *IEEE Xplore* ® [online] <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=4610935>, Consultado: 06 Enero 2009.
2. Xilinx Inc. (2005), Xilinx ISE 7 Software Manuals and Help, <http://www.xilinx.com/itp/xilinx7/books/manuals.pdf>
3. Xilinx Inc. <http://www.xilinx.com>
4. Stephen Brown, Zvonko Vranesic. (2006). *Fundamentos de lógica digital con diseño VHDL.* Segunda edición 295-297. Mc Graw Hill. México, D.F.
5. Yaohan Chu. (1975). *Organización y microprogramación del ordenador.*191-230. Editorial Reverté, s.a. Barcelona, España.
6. David G. Maxinez, Jessica Alcalá. (2008). *VHDL El arte de programar sistemas digitales*. Sexta reimpresión, 207-208. Grupo Editorial Patria México, D.F.
7. Stephen Brown, Jonathan Rose. (1996). Architecture of FPGAs and CPLDs: A Tutorial. *IEEE Design and Test of Computers,* Vol. 13, No. 2, pp. 42-57. [Online]. <http://www.eecg.toronto.edu/~jayar/pubs/brown/survey.pdf>, Consultado: 06 enero 2009.
8. Scientific Literature Digital Library and Search Engine. (2001). Floating point to fixed-point compilation and embedded architectural support. *Citeseerx* [online] < http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.5869>, Consultado: 06 enero 2009.