Polibits

ISSN: 1870-9044

polibits@nlp.cic.ipn.mx

Instituto Politécnico Nacional

México

Álvarez, J. Antonio; Lindig B., Michael

Diseño de un Coprocesador Matemático de Precisión Simple usando el Spartan 3E

Polibits, vol. 38, 2008

Instituto Politécnico Nacional

Distrito Federal, México

Diseño de un Coprocesador Matemático de Precisión Simple usando el Spartan 3E

J. Antonio Álvarez y Michael Lindig B.

***Resumen*—Una Unidad de Punto Flotante (*Floating Point Unit* en inglés) o, más comúnmente conocido como coprocesador matemático, es un componente de la CPU especializado en las operaciones de punto flotante. Las operaciones básicas que toda FPU puede realizar son las aritméticas (suma y multiplicación), si bien algunos sistemas más complejos son capaces también de realizar cálculos trigonométricos o exponenciales. No todas las CPUs tienen una FPU dedicada. En ausencia de FPU, la CPU puede utilizar programas en microcódigo para emular una función en punto flotante a través de la unidad aritmético-lógica (ALU), la cual reduce el costo del hardware a cambio de una sensible pérdida de velocidad. El objetivo de este articulo, es mostrar como puede ser implementado un coprocesador matemático utilizando VHDL, para su implementación en cualquier FPGA.**

***Palabras Clave*—FPU, coprocesador matemático, VHDL, FPGA.**

DESIGN OF MATHEMATICAL COPROCESSOR OF SIMPLE PRECISION USING SPARTAN 3E

***Abstract*—Floating Point Unit (FPU) is also known as mathematical coprocessor and is a specialized component of the CPU dedicated to floating point operations. Basic operations of any FPU are arithmetic (sum and multiplication), though some more complex systems are also able to perform trigonometric or exponential calculations. Not all CPUs have an additional FPU. If there is no FPU present, then CPU can use some programs written in microcode for emulation of floating point operations using arithmetic-logical unit (ALU). This reduces the cost of the hardware but slow down the processing speed. The purpose of this paper is to propose an implementation of the mathematical coprocessor using VHDL, for its further implementation in FPGA.**

***Index Terms*—FPU, mathematical coprocessor, VHDL, FPGA.**

I. INTRODUCCIÓN

La primera computadora personal comercial, fue inventada por IBM en 1981. En su interior había un microprocesador de número 8088, de una empresa llamada Intel. Las características de velocidad de este dispositivo resultan risibles hoy en día: un chip de 8 bits trabajando a 4,77 MHz, aunque bastante razonables para una época en la que el chip de moda era el Z80 de Zilog, como microprocesador común en los sistemas Spectrum que fueron muy populares gracias sobre todo a los juegos increíbles que podían ejecutarse con más gracia y arte que muchos juegos actuales para Pentium.

El 8088 fue una versión de capacidades reducidas del 8086, el cual creo la serie 86 que se integro para los siguientes chips Intel: el 80186 (Control de periféricos), el 80286 (16 bits y hasta 20 MHz) y por fin, en 1987, el primer microprocesador de 32 bits, el 80386, llamado simplemente 386.

Al ser de 32 bits permitía idear software más moderno, con funcionalidades como multitarea real, es decir, disponer de más de un programa trabajando a la vez. A partir de entonces todos los chips compatibles Intel han sido de 32 bits, incluso el flamante Pentium II.

Un día llegó el microprocesador 80486 [1] , que era un microprocesador 80386 con un coprocesador matemático incorporado y una memoria caché integrada, lo que le hacía más rápido; desde entonces todos los chips tienen ambos en su interior, en la Fig. 1 se observa una computadora 486.

Fig. 1. Muestra una Microprocesador Amd modelo 486 (Un módelo 386 con Coprocesador Matemático (FPU)

II. EL COPROCESADOR MATEMÁTICO

El coprocesador matemático es un componente electrónico, que esta diseñado para que funcione en paralelo con el microprocesador. El conjunto de instrucciones incluye muchas operaciones extremadamente potentes para la operación de los datos en punto flotante.

El coprocesador trabaja internamente sólo en formato real, por lo que cualquier carga en los registros de coprocesador provocará que dicho valor sea convertido a punto flotante.

Sus registros están estructurados en forma de pila y se accede a ellos por el número de entrada que ocupan en la pila.

Los registros van desde R(0) hasta R(7), en total ocho registros de 80bits, como deben de manejarse en formato de pila, el coprocesador tiene un puntero de control de pila

Manuscrito recibido el 10 de mayo del 2008. Manuscrito aceptado para su publicación el 2 de septiembre del 2008.

J. A Alvarez Cedillo, Centro de Innovación y Desarrollo Tecnológico en Cómputo del Instituto Politécnico Nacional, México, D. F. (teléfono: 57296000 Ext. 52536; e-mail: jaalvarez@ipn.mx).

Michael Lindig B, Dirección de Cómputo y Telecomunicaciones del Instituto Politécnico Nacional, México, D. F. (e-mail: mlindig@ipn.mx).

llamado **St (state,estado)**, Toda interacción que tengamos que hacer con los registros del coprocesador se realiza a través del puntero de pila **St**, donde el último valor introducido es St o St(0) y si hubiéramos rellenado todos los registros el ultimo seria St(7)

Tipos de datos

El coprocesador puede obtener y escribir datos en memoria de los siguientes tipos.

- Entero
  - Words(16bits)
  - Dword(32 bits)
  - Qwords(64 bits)
- Real
  - Words(16 bits)
  - Dword(32 bits)
  - Qwords(64 bits)
  - Twords(80 bits)

Cada elemento de la pila puede almacenar un valor en formato real temporal de 10 bytes (80bits).

El puntero de la pila (ST) indica en todo momento el elemento en la cima. Puede valer entre 0 y 7.

Las instrucciones del 8087 pueden trabajar con datos en memoria, en el siguiente ejemplo se muestra código en ensamblador que explota esta capacidad:

```
finit
fild multiplicando
fild multiplicador
fmul st,st(1)
fist resultado
```

Las instrucciones del 8087 también pueden trabajar directamente con datos en la pila, en el siguiente ejemplo se muestra código en ensamblador que explota esta capacidad:

```
finit
fild multiplicando
fimul multiplicador
fist resultado
```

Todas las instrucciones del 8087 comienzan con la letra “F”. Si acaban con la letra “P” quiere decir que la pila se cargará con el resultado. Por ejemplo, FISTP VAR saca el resultado de la cima de la pila, lo guarda en la variable en memoria y lo quita de la pila (mueve el puntero).

Tipos de instrucciones:

Existen diferentes tipos de instrucciones, estas se encuentran clasificadas de acuerdo a una función primaria, estas funciones son las siguientes:

- • De transferencia de datos
- • Aritméticas
- • De comparación
- • De cálculo de funciones transcendentes
- • Relativas a constantes
- • De control

Para utilizar el 8087, el 8086 debe ejecutar la instrucción de inicialización FINIT.

El siguiente ejemplo en ensamblador, multiplica dos variables donde el cálculo de resultado es igual a la var1 * var2.

```
pila segment stack 'stack'
dw 100h dup (?)
pila ends
datos segment 'data'
var1 dw 7
var2 dw 3
resultado dw 0
datos ends
codigo segment 'code'
assume cs:codigo, ds:datos, ss:pila
main PROC
mov ax,datos
mov ds,ax
finit
fild var1
fimul var2
fist resultado
mov ax,resultado
call escribir_numero
mov ax,4C00h
int 21h
main ENDP
escribir_numero PROC NEAR
push ax
push dx
mov bx,10
mov dl,al
cmp ax,bx
jb escribir_resto
sub dx,dx
div bx
call escribir_numero
escribir_resto:
add dl,'0'
mov ah,2
int 21h
pop dx
pop ax
ret
escribir_numero ENDP
codigo ends
END main
```

III. Implementación en VHDL

VHDL es el acrónimo que representa la combinación de los conceptos VHSIC y HDL, donde VHSIC es el acrónimo de *Very High Speed Integrated Circuit* y HDL es a su vez el acrónimo de *Hardware Description Language* [2].

Es un lenguaje estándar definido por la IEEE (*Institute of Electrical and Electronics Engineers*), ANSI/IEEE 1076-1993 que se usa para diseñar circuitos digitales. Otros métodos para diseñar circuitos son la captura de esquemas con herramientas CAD y los diagramas de bloques, pero éstos no son prácticos

en diseños complejos. Otros lenguajes para el mismo propósito son Verilog y ABEL. Se usa principalmente para programar PLD (*Programable Logic Device* - Dispositivo Lógico Programable) y FPG] (*Field Programmable Gate Array*) [3]. En la Fig. 2 se muestra un *kit* de desarrollo Spartan 3).

El objetivo de este proyecto es diseñar una FPU de precisión simple usando VHDL, simularlo y sintetizarlo. Contara con las operaciones adición de punto sólo flotante, la substracción y la multiplicación. [4]

Fig. 2. *Kit* de desarrollo de Spartan 3.

Descripciones de Señal
Entradas:

clk reloj
opa y opb .- Entradas de los operandos A y B
rmode.- Redondeo (00 redondea al más cercano, 01 redondea a cero, 10 redondea a inf, 11 redondea a-inf)
fpu_op .- Operaciones de punto flotante
0 – Suma
1 – Resta
2 -Multiplicación
3 -División
4 -Int.- Convierte en entero,
5 - Flot.- Conversión a int.

Salidas:

fout - salida del resultado
inf.- Es el valor especial INF
ine -Calculo inexacto
overflow .- Salida de *overflow*, por ejemplo el número es mas largo de lo que puede ser representado.
div_by_zero .- División sobre cero.
snan .- SNAN
qnan .- QNAN

La Fig. 3 muestra la entidad del proyecto a construir.

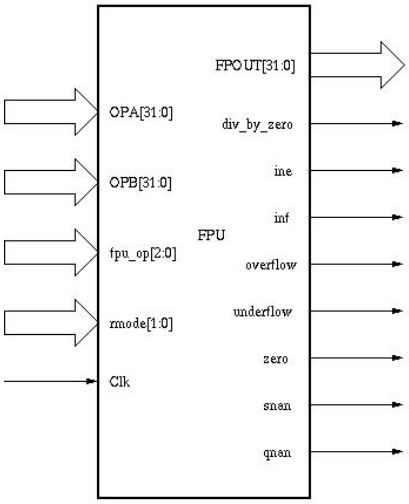

Fig. 3. Muestra una Microprocesador Amd modelo 486 (Un módelo 386 con Coprocesador Matemático (FPU)

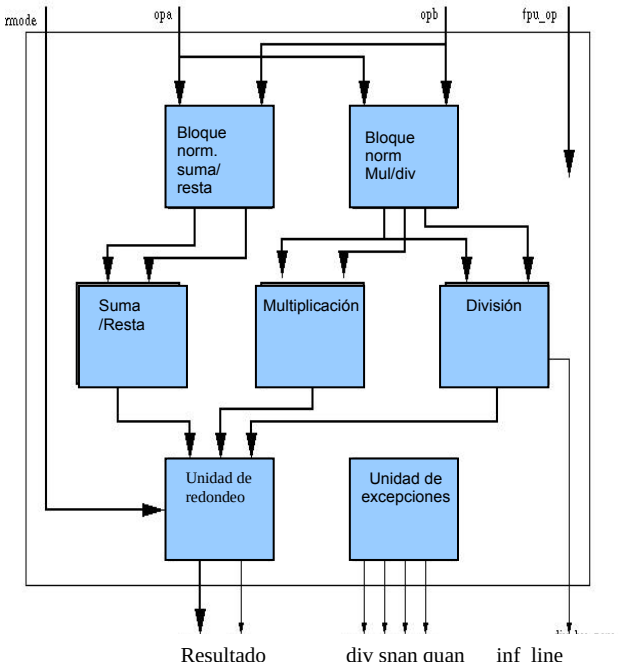

Fig. 4. Muestra el núcleo del proyecto [5].

El núcleo de la FPU (Fig. 4) está formado por las siguientes unidades.

Un bloque normalizado de suma y resta.- Calcula la diferencia entre el exponente mas grande y el pequeño. Ajusta la fracción pequeña por la derecha y determina si la operación es una suma o una resta, después resuelve los bits del signo

Un bloque normalizado de multiplicación y división.- Conmuta la suma o resta de exponentes, checa si existe un exponente en *overflow*, o la condición de *underflow* y el valor INF sobre una entrada.

Unidad de redondeo – Normaliza la fracción y el exponente. Todos los redondeos los hace en paralelo y selecciona la salida correspondiente.

Unidad de Excepciones – Genera y maneja las excepciones.

El diagrama de la Fig. 3 muestra el bloque general de circuitos dentro de la integración del proyecto.

En la Fig. 5 se muestra el *datapath* y el *pipeline*:

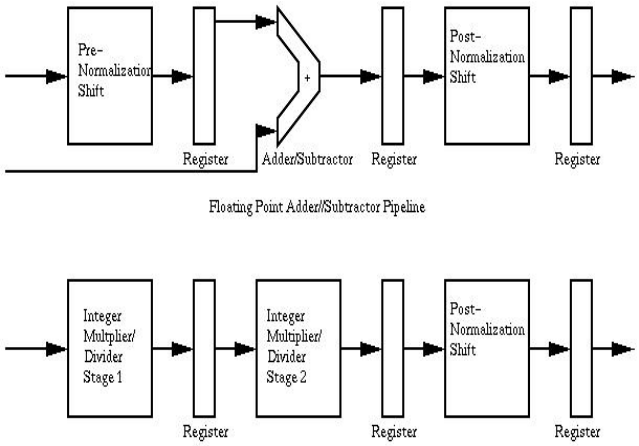

Figura 5.- *Datapath* y *pipeline*

El código en VHDL general es el siguiente:

```
LIBRARY ieee ;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_misc.ALL;
USE ieee.std_logic_unsigned.ALL;

LIBRARY work;

------------------------------------------------------------------------------
-- FPU Operations (fpu_op):
-- 0 = add
-- 1 = sub
-- 2 = mul
-- 3 = div
-- 4 =
-- 5 =
-- 6 =
-- 7 =
------------------------------------------------------------------------------

------------------------------------------------------------------------------
-- Rounding Modes (rmode):

-- 0 = round_nearest_even
-- 1 = round_to_zero
-- 2 = round_up
-- 3 = round_down
------------------------------------------------------------------------------

ENTITY fpu IS
  PORT(
    clk        : IN     std_logic  ;
    fpu_op     : IN     std_logic_vector (2 downto 0) ;
    opa        : IN     std_logic_vector (31 downto 0) ;
    opb        : IN     std_logic_vector (31 downto 0) ;
    rmode      : IN     std_logic_vector (1 downto 0) ;
    div_by_zero : OUT    std_logic  ;
    fpout      : OUT    std_logic_vector (31 downto 0) ;
    ine        : OUT    std_logic  ;
    inf        : OUT    std_logic  ;
    overflow   : OUT    std_logic  ;
    qnan       : OUT    std_logic  ;
    snan       : OUT    std_logic  ;
    underflow  : OUT    std_logic  ;
    zero       : OUT    std_logic
  );
END fpu ;

ARCHITECTURE arch OF fpu IS
  signal opa_r, opb_r : std_logic_vector (31 downto 0);
  signal signa, signb : std_logic ;
  signal sign_fasu : std_logic ;
  signal fracta, fractb : std_logic_vector (26 downto 0);
  signal exp_fasu : std_logic_vector (7 downto 0);
  signal exp_r : std_logic_vector (7 downto 0);
  signal fract_out_d : std_logic_vector (26 downto 0);
  signal co : std_logic ;
  signal fract_out_q : std_logic_vector (27 downto 0);
  signal out_d : std_logic_vector (30 downto 0);
  signal overflow_d, underflow_d : std_logic ;
  signal mul_inf, div_inf : std_logic ;
  signal mul_00, div_00 : std_logic ;
  signal inf_d, ind_d, qnan_d, snan_d, opa_nan, opb_nan : std_logic ;
  signal opa_00, opb_00 : std_logic ;
  signal opa_inf, opb_inf : std_logic ;
  signal opa_dn, opb_dn : std_logic ;
  signal nan_sign_d, result_zero_sign_d : std_logic ;
  signal sign_fasu_r : std_logic ;
  signal exp_mul : std_logic_vector (7 downto 0);
  signal sign_mul : std_logic ;
  signal sign_mul_r : std_logic ;
  signal fracta_mul, fractb_mul : std_logic_vector (23 downto 0);
  signal inf_mul : std_logic ;
  signal inf_mul_r : std_logic ;
  signal exp_ovf : std_logic_vector (1 downto 0);
  signal exp_ovf_r : std_logic_vector (1 downto 0);
  signal sign_exe : std_logic ;
  signal sign_exe_r : std_logic ;
  signal underflow_fmul1_p1, underflow_fmul1_p2, underflow_fmul1_p3 : std_logic ;
  signal underflow_fmul_d : std_logic_vector (2 downto 0);
  signal prod : std_logic_vector (47 downto 0);
  signal quo : std_logic_vector (49 downto 0);
  signal fdiv_opa : std_logic_vector (49 downto 0);
  signal remainder : std_logic_vector (49 downto 0);
  signal remainder_00 : std_logic ;
  signal div_opa_ldz_d, div_opa_ldz_r1, div_opa_ldz_r2 : std_logic_vector (4 downto
0);
  signal ine_d : std_logic ;
  signal fract_denorm : std_logic_vector (47 downto 0);
  signal fract_div : std_logic_vector (47 downto 0);
  signal sign_d : std_logic ;
  signal sign : std_logic ;
  signal opa_r1 : std_logic_vector (30 downto 0);
  signal fract_i2f : std_logic_vector (47 downto 0);
  signal opas_r1, opas_r2 : std_logic ;
  signal f2i_out_sign : std_logic ;
  signal fasu_op_r1, fasu_op_r2 : std_logic ;
  signal out_fixed : std_logic_vector (30 downto 0);
  signal output_zero_fasu : std_logic ;
  signal output_zero_fdiv : std_logic ;
  signal output_zero_fmul : std_logic ;
  signal inf_mul2 : std_logic ;
  signal overflow_fasu : std_logic ;
  signal overflow_fmul : std_logic ;
  signal overflow_fdiv : std_logic ;
  signal inf_fmul : std_logic ;
  signal sign_mul_final : std_logic ;
  signal out_d_00 : std_logic ;
  signal sign_div_final : std_logic ;
  signal ine_mul, ine_mula, ine_div, ine_fasu : std_logic ;
  signal underflow_fasu, underflow_fmul, underflow_fdiv : std_logic ;
  signal underflow_fmul1 : std_logic ;
  signal underflow_fmul_r : std_logic_vector (2 downto 0);
  signal opa_nan_r : std_logic ;
  signal mul_uf_del : std_logic ;
  signal uf2_del, ufb2_del, ufc2_del, underflow_d_del : std_logic ;
  signal co_del : std_logic ;
  signal out_d_del : std_logic_vector (30 downto 0);
  signal ov_fasu_del, ov_fmul_del : std_logic ;
  signal fop : std_logic_vector (2 downto 0);
  signal ldza_del : std_logic_vector (4 downto 0);
  signal quo_del : std_logic_vector (49 downto 0);
  signal rmode_r1, rmode_r2, rmode_r3 : std_logic_vector (1 downto 0);
  signal fpu_op_r1, fpu_op_r2, fpu_op_r3 : std_logic_vector (2 downto 0);
  signal fpu_op_r1_0_not : std_logic ;
  signal fasu_op, co_d : std_logic ;
  signal post_norm_output_zero : std_logic ;

  CONSTANT INF_VAL : std_logic_vector(31 DOWNTO 0) := X"7f800000";
  CONSTANT QNAN_VAL : std_logic_vector(31 DOWNTO 0) := X"7fc00001";
  CONSTANT SNAN_VAL : std_logic_vector(31 DOWNTO 0) := X"7f800001";

  COMPONENT add_sub27
    PORT(
      add : IN     std_logic  ;
      opa : IN     std_logic_vector (26 downto 0) ;
      opb : IN     std_logic_vector (26 downto 0) ;
      co  : OUT    std_logic  ;
      sum : OUT    std_logic_vector (26 downto 0)
    );
  END COMPONENT;
```

```
COMPONENT div_r2
  PORT(
    clk      : IN    std_logic  ;
    opa      : IN    std_logic_vector (49 downto 0) ;
    opb      : IN    std_logic_vector (23 downto 0) ;
    quo      : OUT   std_logic_vector (49 downto 0) ;
    remainder : OUT   std_logic_vector (49 downto 0)
  );
END COMPONENT;

COMPONENT except IS
  PORT(
    clk    : IN    std_logic  ;
    opa    : IN    std_logic_vector (31 downto 0) ;
    opb    : IN    std_logic_vector (31 downto 0) ;
    ind    : OUT   std_logic  ;
    inf    : OUT   std_logic  ;
    opa_00  : OUT   std_logic  ;
    opa_dn  : OUT   std_logic  ;
    opa_inf : OUT   std_logic  ;
    opa_nan : OUT   std_logic  ;
    opb_00  : OUT   std_logic  ;
    opb_dn  : OUT   std_logic  ;
    opb_inf : OUT   std_logic  ;
    opb_nan : OUT   std_logic  ;
    qnan   : OUT   std_logic  ;
    snan   : OUT   std_logic
  );
END COMPONENT ;

COMPONENT mul_r2 IS
  PORT(
    clk  : IN    std_logic  ;
    opa  : IN    std_logic_vector (23 downto 0) ;
    opb  : IN    std_logic_vector (23 downto 0) ;
    prod : OUT   std_logic_vector (47 downto 0)
  );
END COMPONENT;

COMPONENT post_norm IS
  PORT(
    clk        : IN    std_logic  ;
    div_opa_ldz  : IN    std_logic_vector (4 downto 0) ;
    exp_in     : IN    std_logic_vector (7 downto 0) ;
    exp_ovf    : IN    std_logic_vector (1 downto 0) ;
    fpu_op     : IN    std_logic_vector (2 downto 0) ;
    fract_in   : IN    std_logic_vector (47 downto 0) ;
    opa_dn     : IN    std_logic  ;
    opas       : IN    std_logic  ;
    opb_dn     : IN    std_logic  ;
    output_zero  : IN    std_logic  ;
    rem_00     : IN    std_logic  ;
    rmode      : IN    std_logic_vector (1 downto 0) ;
    sign       : IN    std_logic  ;
    f2i_out_sign : OUT   std_logic  ;
    fpout      : OUT   std_logic_vector (30 downto 0) ;
    ine        : OUT   std_logic  ;
    overflow   : OUT   std_logic  ;
    underflow  : OUT   std_logic
  );
END COMPONENT;

COMPONENT pre_norm IS
  PORT(
    add           : IN    std_logic  ;
    clk           : IN    std_logic  ;
    opa           : IN    std_logic_vector (31 downto 0) ;
    opa_nan       : IN    std_logic  ;
    opb           : IN    std_logic_vector (31 downto 0) ;
    opb_nan       : IN    std_logic  ;
    rmode         : IN    std_logic_vector (1 downto 0) ;
    exp_dn_out    : OUT   std_logic_vector (7 downto 0) ;
    fasu_op       : OUT   std_logic  ;
    fracta_out    : OUT   std_logic_vector (26 downto 0) ;
    fractb_out    : OUT   std_logic_vector (26 downto 0) ;
    nan_sign      : OUT   std_logic  ;
    result_zero_sign : OUT   std_logic  ;
    sign          : OUT   std_logic
  );
END COMPONENT;

COMPONENT pre_norm_fmul IS
  PORT(
    clk      : IN    std_logic  ;
    fpu_op   : IN    std_logic_vector (2 downto 0) ;
    opa      : IN    std_logic_vector (31 downto 0) ;
    opb      : IN    std_logic_vector (31 downto 0) ;
    exp_out  : OUT   std_logic_vector (7 downto 0) ;
    exp_ovf  : OUT   std_logic_vector (1 downto 0) ;
    fracta   : OUT   std_logic_vector (23 downto 0) ;
    fractb   : OUT   std_logic_vector (23 downto 0) ;
    inf      : OUT   std_logic  ;
    sign     : OUT   std_logic  ;
    sign_exe : OUT   std_logic  ;
    underflow : OUT   std_logic_vector (2 downto 0)
  );
END COMPONENT;

BEGIN

  PROCESS (clk)
  BEGIN
    IF clk'event AND clk = '1' THEN
      opa_r <=  opa;
      opb_r <=  opb;
      rmode_r1 <=  rmode;
      rmode_r2 <=  rmode_r1;
      rmode_r3 <=  rmode_r2;
      fpu_op_r1 <=  fpu_op;
      fpu_op_r2 <=  fpu_op_r1;
      fpu_op_r3 <=  fpu_op_r2;
    END IF;
  END PROCESS;

  ----------------------------------------------------------------------
  -- Exceptions block
  ----------------------------------------------------------------------
  u0 : except
    PORT MAP (
      clk => clk,
      opa => opa_r,
      opb => opb_r,
      inf => inf_d,
      ind => ind_d,
      qnan => qnan_d,
      snan => snan_d,
      opa_nan => opa_nan,
      opb_nan => opb_nan,
      opa_00 => opa_00,
      opb_00 => opb_00,
      opa_inf => opa_inf,
      opb_inf => opb_inf,
      opa_dn => opa_dn,
      opb_dn => opb_dn
      );

  ----------------------------------------------------------------------
  -- Pre-Normalize block
  -- Adjusts the numbers to equal exponents and sorts them
  -- determine result sign
  -- determine actual operation to perform (add or sub)
  ----------------------------------------------------------------------
  fpu_op_r1_0_not <= NOT fpu_op_r1(0);
  u1 : pre_norm
    PORT MAP (
      clk => clk,                     -- System Clock
      rmode => rmode_r2,                  -- Roundin Mode
      add => fpu_op_r1_0_not,               -- Add/Sub Input
      opa => opa_r,
      opb => opb_r,                   -- Registered OP Inputs
      opa_nan => opa_nan,                 -- OpA is a NAN indicator
      opb_nan => opb_nan,                 -- OpB is a NAN indicator
      fracta_out => fracta,               -- Equalized and sorted fraction
      fractb_out => fractb,               -- outputs (Registered
      exp_dn_out => exp_fasu,             -- Selected exponent output (registered;
      sign => sign_fasu,                  -- Encoded output Sign (registered)
      nan_sign => nan_sign_d,             -- Output Sign for NANs (registered)
      result_zero_sign => result_zero_sign_d,  -- Output Sign for zero result
(registered)
      fasu_op => fasu_op                  -- Actual fasu operation output (registered)
      );

  u2 : pre_norm_fmul
    PORT MAP (
      clk => clk,
      fpu_op => fpu_op_r1,
      opa => opa_r,
      opb => opb_r,
```

```
        fracta => fracta_mul,
        fractb => fractb_mul,
        exp_out => exp_mul,      -- FMUL exponent output  => registered)
        sign => sign_mul,        -- FMUL sign output (registered)
        sign_exe => sign_exe,    -- FMUL exception sign output (registered)
        inf => inf_mul,          -- FMUL inf output (registered)
        exp_ovf => exp_ovf,      -- FMUL exponnent overflow output (registered)
        underflow => underflow_fmul_d
        );


   PROCESS (clk)
   BEGIN
      IF clk'event AND clk = '1' THEN
         sign_mul_r <= sign_mul;
         sign_exe_r <= sign_exe;
         inf_mul_r <= inf_mul;
         exp_ovf_r <= exp_ovf;
         sign_fasu_r <= sign_fasu;
      END IF;
   END PROCESS;


------------------------------------------------------------------------
--
-- Add/Sub
--

 u3 : add_sub27
 PORT MAP (
       add => fasu_op,              -- Add/Sub
       opa => fracta,               -- Fraction A input
       opb => fractb,               -- Fraction B Input
       sum => fract_out_d,          -- SUM output
       co => co_d );                -- Carry Output

   PROCESS (clk)
   BEGIN
      IF clk'event AND clk = '1' THEN
         fract_out_q <= co_d & fract_out_d;
      END IF;
   END PROCESS;


------------------------------------------------------------------------
--
-- Mul
--

   u5 : mul_r2 PORT MAP (clk => clk, opa => fracta_mul, opb => fractb_mul, prod
=> prod);

------------------------------------------------------------------------
--
-- Divide
--
PROCESS (fracta_mul)
BEGIN
       IF fracta_mul(22) = '1' THEN  div_opa_ldz_d <= conv_std_logic_vector(1,5);
       ELSIF fracta_mul(22 DOWNTO 21) = "01" THEN  div_opa_ldz_d <=
conv_std_logic_vector(2,5);
       ELSIF fracta_mul(22 DOWNTO 20) = "001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(3,5);
       ELSIF fracta_mul(22 DOWNTO 19) = "0001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(4,5);
       ELSIF fracta_mul(22 DOWNTO 18) = "00001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(5,5);
       ELSIF fracta_mul(22 DOWNTO 17) = "000001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(6,5);
       ELSIF fracta_mul(22 DOWNTO 16) = "0000001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(7,5);
       ELSIF fracta_mul(22 DOWNTO 15) = "00000001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(8,5);
       ELSIF fracta_mul(22 DOWNTO 14) = "000000001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(9,5);
       ELSIF fracta_mul(22 DOWNTO 13) = "0000000001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(10,5);
       ELSIF fracta_mul(22 DOWNTO 12) = "00000000001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(11,5);
       ELSIF fracta_mul(22 DOWNTO 11) = "000000000001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(12,5);
       ELSIF fracta_mul(22 DOWNTO 10) = "0000000000001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(13,5);
       ELSIF fracta_mul(22 DOWNTO 9) = "00000000000001" THEN  div_opa_ldz_d <=
conv_std_logic_vector(14,5);
       ELSIF fracta_mul(22 DOWNTO 8) = "000000000000001" THEN  div_opa_ldz_d
<= conv_std_logic_vector(15,5);
       ELSIF fracta_mul(22 DOWNTO 7) = "0000000000000001" THEN  div_opa_ldz_d
<= conv_std_logic_vector(16,5);
       ELSIF fracta_mul(22 DOWNTO 6) = "00000000000000001" THEN  div_opa_ldz_d
<= conv_std_logic_vector(17,5);
       ELSIF fracta_mul(22 DOWNTO 5) = "000000000000000001" THEN
div_opa_ldz_d <= conv_std_logic_vector(18,5);
       ELSIF fracta_mul(22 DOWNTO 4) = "0000000000000000001" THEN
div_opa_ldz_d <= conv_std_logic_vector(19,5);
       ELSIF fracta_mul(22 DOWNTO 3) = "00000000000000000001" THEN
div_opa_ldz_d <= conv_std_logic_vector(20,5);
       ELSIF fracta_mul(22 DOWNTO 2) = "000000000000000000001" THEN
div_opa_ldz_d <= conv_std_logic_vector(21,5);
       ELSIF fracta_mul(22 DOWNTO 1) = "0000000000000000000001" THEN
div_opa_ldz_d <= conv_std_logic_vector(22,5);
       ELSIF fracta_mul(22 DOWNTO 1) = "0000000000000000000000" THEN
div_opa_ldz_d <= conv_std_logic_vector(23,5);
       ELSE div_opa_ldz_d <= (OTHERS => 'X');
       END IF;
END PROCESS;

       fdiv_opa <= ((SHL(fracta_mul,div_opa_ldz_d)) &
                "00" & X"000000") WHEN
                ((or_reduce(opa_r(30 DOWNTO 23)))='0') ELSE
                (fracta_mul & "00" & X"000000");

       u6 : div_r2 PORT MAP (clk => clk, opa => fdiv_opa, opb => fractb_mul,
                       quo => quo,
                       remainder => remainder);

       remainder_00 <= NOT or_reduce(remainder);

       PROCESS (clk)
       BEGIN
          IF clk'event AND clk = '1' THEN
             div_opa_ldz_r1 <= div_opa_ldz_d;
             div_opa_ldz_r2 <= div_opa_ldz_r1;
          END IF;
       END PROCESS;

------------------------------------------------------------------------
--
-- Normalize Result
--

   PROCESS (clk)
   BEGIN
      IF clk'event AND clk = '1' THEN
         CASE fpu_op_r2 IS
            WHEN "000" => exp_r <= exp_fasu;
            WHEN "001" => exp_r <= exp_fasu;
            WHEN "010" => exp_r <= exp_mul;
            WHEN "011" => exp_r <= exp_mul;
            WHEN "100" => exp_r <= (others => '0');
            WHEN "101" => exp_r <= opa_r1(30 downto 23);
            WHEN OTHERS  => exp_r <= (others => '0');
         END case;
      END IF;
   END PROCESS;


   fract_div <= quo(49 DOWNTO 2) WHEN (opb_dn = '1') ELSE
             (quo(26 DOWNTO 0) & '0' & X"00000");

   PROCESS (clk)
   BEGIN
      IF clk'event AND clk = '1' THEN
         opa_r1 <= opa_r(30 DOWNTO 0);
         IF fpu_op_r2="101" THEN
            IF sign_d = '1' THEN
               fract_i2f <= conv_std_logic_vector(1,48)-(X"000000" &
                        (or_reduce(opa_r1(30 downto 23))) &
                         opa_r1(22 DOWNTO 0))-conv_std_logic_vector(1,48);
            ELSE
               fract_i2f <= (X"000000" &
                        (or_reduce(opa_r1(30 downto 23))) &
                        opa_r1(22 DOWNTO 0));
            END IF;
         ELSE
            IF sign_d = '1' THEN
               fract_i2f <= conv_std_logic_vector(1,48) - (opa_r1 & X"0000" & '1');
            ELSE
               fract_i2f <= (opa_r1 & '0' & X"0000");
            END IF;
```

```
      END IF;
    END IF;
  END PROCESS;

  PROCESS (fpu_op_r3,fract_out_q,prod,fract_div,fract_i2f)
  BEGIN
    CASE fpu_op_r3 IS
      WHEN "000" => fract_denorm <= (fract_out_q & X"00000");
      WHEN "001" => fract_denorm <= (fract_out_q & X"00000");
      WHEN "010" => fract_denorm <= prod;
      WHEN "011" => fract_denorm <= fract_div;
      WHEN "100" => fract_denorm <= fract_i2f;
      WHEN "101" => fract_denorm <= fract_i2f;
      WHEN OTHERS  => fract_denorm <= (others => '0');
    END case;
  END PROCESS;



  PROCESS (clk, opa_r(31),opas_r1,rmode_r2,sign_d)
  BEGIN
    IF clk'event AND clk = '1' THEN
      opas_r1 <= opa_r(31);
      opas_r2 <= opas_r1;
      IF rmode_r2="11" THEN
        sign <= NOT sign_d;
      ELSE
        sign <= sign_d;
      END IF;
    END if;
  END PROCESS;

  sign_d <= sign_mul WHEN (fpu_op_r2(1) = '1') ELSE sign_fasu;

  post_norm_output_zero <= mul_00 or div_00;
  u4 : post_norm
  PORT MAP (
    clk => clk,               -- System Clock
    fpu_op => fpu_op_r3,          -- Floating Point Operation
    opas => opas_r2,              -- OPA Sign
    sign => sign,                 -- Sign of the result
    rmode => rmode_r3,            -- Rounding mode
    fract_in => fract_denorm,      -- Fraction Input
    exp_ovf => exp_ovf_r,         -- Exponent Overflow
    exp_in => exp_r,              -- Exponent Input
    opa_dn => opa_dn,             -- Operand A Denormalized
    opb_dn => opb_dn,             -- Operand A Denormalized
    rem_00 => remainder_00,       -- Diveide Remainder is zero
    div_opa_ldz => div_opa_ldz_r2,  -- Divide opa leading zeros count
    output_zero => post_norm_output_zero,  -- Force output to Zero
    fpout => out_d,               -- Normalized output (un-registered)
    ine => ine_d,                 -- Result Inexact output (un-registered)
    overflow => overflow_d,       -- Overflow output (un-registered)
    underflow => underflow_d,     -- Underflow output (un-registered)
    f2i_out_sign => f2i_out_sign    -- F2I Output Sign
    );

-------------------------------------------------------------------------
--
-- FPU Outputs
--

  PROCESS (clk)
  BEGIN
    IF clk'event AND clk = '1' THEN
      fasu_op_r1 <= fasu_op;
      fasu_op_r2 <= fasu_op_r1;
      IF exp_mul = X"ff" THEN
        inf_mul2 <= '1';
      ELSE
        inf_mul2 <= '0';
      END IF;
    END IF;
  END PROCESS;


  -- Force pre-set values for non numerical output
  mul_inf <= '1' WHEN ((fpu_op_r3="010") and ((inf_mul_r or inf_mul2)='1') and
              (rmode_r3="00")) else '0';

  div_inf <= '1' WHEN ((fpu_op_r3="011") and
              ((opb_00 or opa_inf)='1')) ELSE '0';

  mul_00 <= '1' WHEN ((fpu_op_r3="010") and ((opa_00 or opb_00)='1')) ELSE '0';
  div_00 <= '1' WHEN ((fpu_op_r3="011") and ((opa_00 or opb_inf)='1')) else '0';

  out_fixed <= QNAN_VAL(30 DOWNTO 0) WHEN
          (((qnan_d OR snan_d) OR (ind_d AND NOT fasu_op_r2) OR
           ((NOT fpu_op_r3(2) AND fpu_op_r3(1) AND fpu_op_r3(0)) AND opb_00
AND opa_00) OR
           (((opa_inf AND opb_00) OR (opb_inf AND opa_00 )) AND
           (NOT fpu_op_r3(2) AND fpu_op_r3(1) AND NOT fpu_op_r3(0)))
           )='1')
          ELSE INF_VAL(30 DOWNTO 0);


  PROCESS (clk)
  BEGIN
    IF clk'event AND clk = '1' THEN
      IF ( ((mul_inf='1') or (div_inf='1') or
          ((inf_d='1') and (fpu_op_r3/="011") and (fpu_op_r3/="101")) or
          (snan_d='1') or (qnan_d='1')) and (fpu_op_r3/="100"))  THEN
        fpout(30 DOWNTO 0) <= out_fixed;
      ELSE
        fpout(30 DOWNTO 0) <= out_d;
      END IF;
    END IF;
  END PROCESS;

  out_d_00 <= NOT or_reduce(out_d);

  sign_mul_final <= NOT sign_mul_r WHEN
            ((sign_exe_r AND  ((opa_00 AND opb_inf) OR
                    (opb_00 AND opa_inf)))='1')
            ELSE sign_mul_r;
  sign_div_final <= NOT sign_mul_r WHEN
            ((sign_exe_r and (opa_inf and opb_inf))='1')
            ELSE (sign_mul_r or (opa_00 and opb_00));

  PROCESS (clk)
  BEGIN
    IF clk'event AND clk = '1' THEN
      If ((fpu_op_r3="101") and (out_d_00='1')) THEN
        fpout(31) <= (f2i_out_sign and not(qnan_d OR snan_d) );
      ELSIF ((fpu_op_r3="010") and ((snan_d or qnan_d)='0')) THEN
        fpout(31) <= sign_mul_final;
      ELSIF ((fpu_op_r3="011") and ((snan_d or qnan_d)='0')) THEN
        fpout(31) <= sign_div_final;
      ELSIF ((snan_d or qnan_d or ind_d) = '1') THEN
        fpout(31) <= nan_sign_d;
      ELSIF (output_zero_fasu = '1') THEN
        fpout(31) <= result_zero_sign_d;
      ELSE
        fpout(31) <= sign_fasu_r;
      END IF;
    END IF;
  END PROCESS;


-- Exception Outputs
  ine_mula <= ((inf_mul_r OR inf_mul2 OR opa_inf OR opb_inf) AND
          (NOT rmode_r3(1) AND rmode_r3(0)) and
           NOT ((opa_inf AND opb_00) OR (opb_inf AND opa_00 )) AND
fpu_op_r3(1));

  ine_mul <= (ine_mula OR ine_d OR inf_fmul OR out_d_00 OR overflow_d OR
underflow_d) AND
          NOT opa_00 and NOT opb_00 and NOT (snan_d OR qnan_d OR inf_d);
  ine_div <= (ine_d OR  overflow_d OR  underflow_d) AND NOT (opb_00 OR snan_d
OR qnan_d OR inf_d);
  ine_fasu <= (ine_d OR overflow_d OR underflow_d) AND NOT (snan_d OR qnan_d
OR inf_d);

  PROCESS (clk)
  BEGIN
    IF clk'event AND clk = '1' THEN
      IF fpu_op_r3(2) = '1' THEN
        ine <= ine_d;
      ELSIF fpu_op_r3(1) = '0' THEN
        ine <= ine_fasu;
      ELSIF fpu_op_r3(0)='1' THEN
        ine <= ine_div;
      ELSE
       ine <= ine_mul;
      END IF;
    END IF;
  END PROCESS;

  overflow_fasu <= overflow_d AND NOT (snan_d OR qnan_d OR inf_d);
  overflow_fmul <= NOT inf_d AND
```

```
                (inf_mul_r OR inf_mul2 OR overflow_d) AND
                NOT (snan_d OR qnan_d);

   overflow_fdiv <= (overflow_d AND NOT (opb_00 OR inf_d OR snan_d OR qnan_d));

   PROCESS (clk)
   BEGIN
      IF clk'event AND clk = '1' THEN
         underflow_fmul_r <= underflow_fmul_d;
         IF fpu_op_r3(2) ='1' THEN
            overflow <= '0';
         ELSIF fpu_op_r3(1) = '0' THEN
            overflow <= overflow_fasu;
         ELSIF fpu_op_r3(0) = '1' THEN
            overflow <= overflow_fdiv;
         ELSE
            overflow <= overflow_fmul;
         END IF;
      END IF;
   END PROCESS;

   underflow_fmul1_p1 <= '1' WHEN (out_d(30 DOWNTO 23) = X"00") else '0';
   underflow_fmul1_p2 <= '1' WHEN (out_d(22 DOWNTO 0) = ("000" & X"00000"))
else '0';
   underflow_fmul1_p3 <= '1' WHEN (prod/=conv_std_logic_vector(0,48)) else '0';

   underflow_fmul1 <= underflow_fmul_r(0) or
                (underflow_fmul_r(1) and underflow_d ) or
                ((opa_dn or opb_dn) and out_d_00 and (underflow_fmul1_p3) and sign)
or
                (underflow_fmul_r(2) AND
                ((underflow_fmul1_p1) or (underflow_fmul1_p2)));

   underflow_fasu <= underflow_d AND  NOT (inf_d or snan_d or qnan_d);
   underflow_fmul <= underflow_fmul1 AND NOT (snan_d or qnan_d or inf_mul_r);
   underflow_fdiv <= underflow_fasu AND NOT opb_00;

   PROCESS (clk)
   BEGIN
      IF clk'event AND clk = '1' THEN
         IF fpu_op_r3(2) = '1' THEN
            underflow <= '0';
         ELSIF fpu_op_r3(1) = '0' THEN
            underflow <= underflow_fasu;
         ELSIF fpu_op_r3(0) = '1' THEN
            underflow <= underflow_fdiv;
         ELSE
            underflow <= underflow_fmul;
         END IF;
         snan <= snan_d;
      END IF;
   END PROCESS;


-- Status Outputs
   PROCESS (clk)
   BEGIN
      IF clk'event AND clk = '1' THEN
         IF fpu_op_r3(2)='1' THEN
            qnan <= '0';
         ELSE
            qnan <= snan_d OR qnan_d OR (ind_d AND NOT fasu_op_r2) OR
                  (opa_00 AND  opb_00 AND
                  (NOT fpu_op_r3(2) AND fpu_op_r3(1) AND fpu_op_r3(0))) OR
                  (((opa_inf AND opb_00) OR (opb_inf AND opa_00 )) AND
                   (NOT fpu_op_r3(2) AND fpu_op_r3(1) AND NOT fpu_op_r3(0)));

         END IF;
      END IF;
   END PROCESS;

   inf_fmul <= (((inf_mul_r OR inf_mul2) AND (NOT rmode_r3(1) AND NOT
rmode_r3(0)))
             OR opa_inf OR opb_inf) AND
            NOT ((opa_inf AND opb_00) OR (opb_inf AND opa_00)) AND
            (NOT fpu_op_r3(2) AND fpu_op_r3(1) AND NOT fpu_op_r3(0));

   PROCESS (clk)
   BEGIN
      IF clk'event AND clk = '1' THEN
         IF fpu_op_r3(2) = '1' THEN
            inf <= '0';
         ELSE
            inf <= (NOT (qnan_d OR snan_d) AND
                  (((and_reduce(out_d(30 DOWNTO 23))) AND
                   NOT (or_reduce(out_d(22 downto 0))) AND
                   NOT(opb_00 AND NOT fpu_op_r3(2) AND fpu_op_r3(1) AND
fpu_op_r3(0))) OR
                  (inf_d AND NOT (ind_d AND NOT fasu_op_r2) AND NOT fpu_op_r3(1))
OR
                  inf_fmul OR
                  (NOT opa_00 AND  opb_00 AND
                   NOT fpu_op_r3(2) AND fpu_op_r3(1) AND fpu_op_r3(0)) or
                  (NOT fpu_op_r3(2) AND fpu_op_r3(1) AND fpu_op_r3(0) AND
                   opa_inf AND  NOT opb_inf)
                  )
                 );
         END IF;
      END IF;
   END PROCESS;


   output_zero_fasu <= out_d_00 AND NOT (inf_d OR snan_d OR qnan_d);
   output_zero_fdiv <= (div_00 OR (out_d_00 AND NOT opb_00)) AND NOT (opa_inf
AND opb_inf) AND
                NOT (opa_00 AND opb_00) AND NOT (qnan_d OR snan_d);
   output_zero_fmul <= (out_d_00 OR opa_00 OR opb_00) AND
                NOT (inf_mul_r OR inf_mul2 OR opa_inf OR opb_inf OR
                   snan_d OR qnan_d) AND
                NOT (opa_inf AND opb_00) AND NOT (opb_inf AND opa_00);

   PROCESS (clk)
   BEGIN
      IF clk'event AND clk = '1' THEN
         IF fpu_op_r3="101" THEN
            zero <= out_d_00 and NOT (snan_d or qnan_d);
         ELSIF fpu_op_r3="011" THEN
            zero <= output_zero_fdiv;
         ELSIF fpu_op_r3="010" THEN
            zero <= output_zero_fmul;
         ELSE
            zero <= output_zero_fasu;
         END IF;
         IF (opa_nan = '0') AND (fpu_op_r2="011") THEN
            opa_nan_r <= '1';
         ELSE
            opa_nan_r <= '0';
         END IF;
         div_by_zero <= opa_nan_r AND NOT opa_00 AND NOT opa_inf AND opb_00;
      END IF;
   END PROCESS;

END arch;
```

En la Fig. 6 se muestra la entidad elaborada.

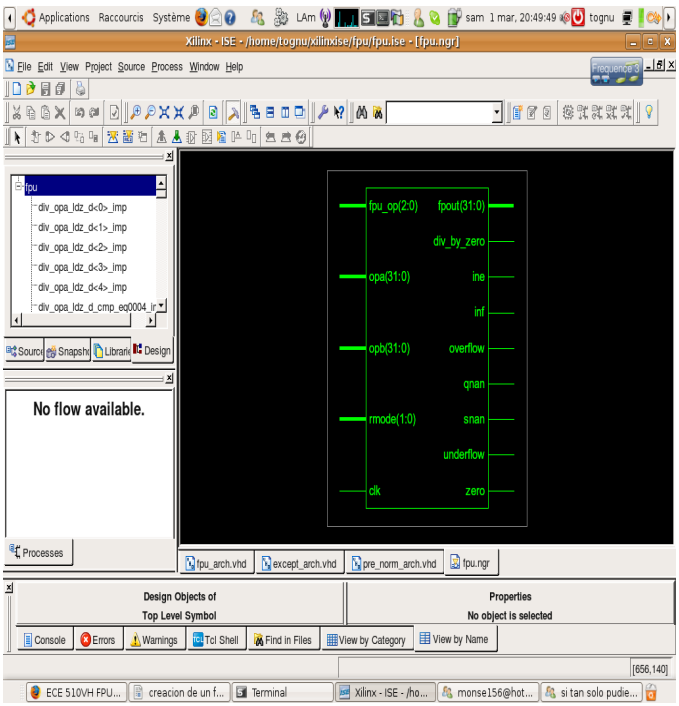

Fig. 6. Pantalla del ise de Xilinx con la entidad general

IV. Conclusiones

En este proyecto se mostró las características y el poder existente en el uso de VHDL y su simplicidad para usarlos en proyectos mas robustos para aplicarlos en FPGA, En el caso de la unidad de punto flotante construida, se pudo observar el mismo comportamiento y se implementaron en ellas las operaciones comunes como es la suma , la multiplicación y la división, todas las operaciones de simple precisión y de punto flotante, también se muestra el *pipeline*, estrategias de verificación y síntesis.

Referencias

[1] Rudolf Usselman. Documentation for Floating Point Unit, http://www.opencores.org.

[1] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. 2nd Edition, Morgan Kaufmann, 2007, 708 p.

[2] Peter J. Ashenden. The Designer's Guide to VHDL, Morgan Kaufmann, 1995, 688 p.

[3] Donald E. Thomas and Philip R. Moorby. The Verilog Hardware Description Language, Kluwer Academic Publishers, 2002, 408 p.

[4] Stuart Oberman. Design Issues in High Performance Floating-Point Arithmetic Units. Stanford University, Technical report, 1996.

[5] IEEE, IEEE-754-1985 Standard for binary floating-point arithmetic.