Implementación de un ALU Básico de 16 bits en un CPLD

Implementation of a Basic ALU 16-bit in CPLD

Guillermo Tejada Muñoz[1]

Facultad de Ingeniería Electrónica y Eléctrica, Universidad Nacional Mayor de San Maros, Lima, Perú

***Resumen*— El presente trabajo describe la implementación de una Unidad Aritmética y Lógica - ALU de 16 bits sobre un Complex Programmable Logic Device – CPLD, para realizar las operaciones básicas aritméticas de suma, resta y las operaciones lógicas AND y XOR. Los bloques lógicos del ALU han sido codificados en VHDL (VHSIC-HDL: *Very High Speed Integrated Circuit - Hardware Description Language*). El software de distribución gratuita, Quartus II v. 9.1 fue utilizado para simular y programar el CPLD EPM7128S, el cual está inserto en el sistema de desarrollo UP2 Education Board.**

***Abstract*— This paper describes the implementation in a Complex Programmable Logic Device - CPLD of an ALU (Arithmetic and Logic Unit) of 16 bits, which can perform the basic arithmetic operations of addition, subtraction and logical operations AND and XOR. ALU logical blocks have been coded in VHDL (VHSIC-HDL: Very High Speed Integrated Circuit - Hardware Description Language). The free distribution software Quartus II v. 9.1 was used to simulate and program the CPLD EPM7128S, which is embedded in the development system UP2 Education Board.**

***Palabras Claves* - Unidad Aritmética y Lógica de 16 bits, ALU, CPLD, VHDL, QUARTUS, EPM7128S, UP2 Education Board.**

***Key Words* - Arithmetic and Logic Unit 16-bit, ALU, CPLD, VHDL, QUARTUS, EPM7128S, UP2 Education Board.**

I. INTRODUCCIÓN

La Unidad de Aritmética Lógica (ALU) es parte del computador que realiza las operaciones aritméticas y lógicas con los datos. El resto de los elementos del computador (Unidad de control, registros, memorias y E/S) están principalmente para suministrar datos a la ALU, a fin de que ésta los procese, y luego para recuperar los resultados [1].

Existen varios tipos de circuitos integrados (CIs) disponibles que se denominan unidades aritméticas y lógicas aunque no tienen la capacidad total de una unidad aritmética y lógica de una computadora. Estos CIs son capaces de realizar diversas operaciones aritméticas y lógicas con entrada de datos binarios. La operación específica que realiza un ALU se determina mediante un código binario aplicado a las entradas denominadas SELECT. Hay varios CIs disponibles para esta función, como por ejemplo el 74LS382 o el 74HC382, que son circuitos integrados de veinte pines que operan con dos números binarios de cuatro bits cada uno para producir un resultado de cuatro bits de salida. Pueden realizar hasta ocho diferentes operaciones dependiendo de los bits disponibles en SELECT [2]. Las funciones más importantes de un ALU son las operaciones de suma y resta aritmética, que se destacan por su velocidad de aquellas cuyas operaciones lógicas sean de arrastre anticipado. En los siguientes ítems se tratará sobre estos temas.

A. Sumador con arrastre anticipado

Considérese el circuito de un sumador completo de un bit como el mostrado la Fig.1, en donde se han definido las variables $P(i)$, denominado de arrastre propagado y $G(i)$, denominado de arrastre generado [3].

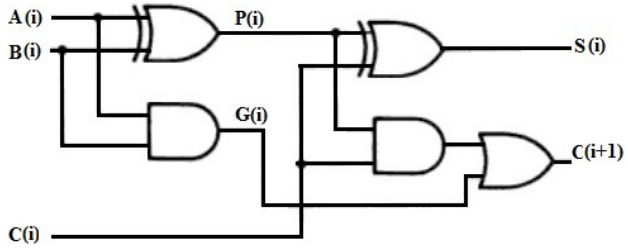

Fig.1. Sumador completo de un bit

[1] Guillermo Tejada Muñoz, e-mail: gtejadam@unmsm.edu.pe
Recibido: Abril 2014 / Aceptado: Mayo 2014

$P(i)$ y $G(i)$ están definidos como:

$$P(i) = A(i) \oplus B(i) \quad (1)$$
$$G(i) = A(i) \cdot B(i) \quad (2)$$

La suma de la salida y el arrastre se pueden expresar como:

$$S(i) = P(i) \oplus C(i) \quad (3)$$
$$C(i+1) = G(i) + P(i).C(i) \quad (4)$$

Para el caso de tener un sumador de 3 bits las ecuaciones serán:

$$P(0) = A(0) \oplus B(0) \quad (5)$$
$$P(1) = A(1) \oplus B(1) \quad (6)$$
$$P(2) = A(2) \oplus B(2) \quad (7)$$

$$G(0) = A(0).B(0) \quad (8)$$
$$G(1) = A(1).B(1) \quad (9)$$
$$G(2) = A(2).B(2) \quad (10)$$

$$C(1) = P(0).CI + G(0) \quad (11)$$
$$C(2) = P(1).P(0).CI + P(1).G(0) + G(1) \quad (12)$$
$$C(3) = P(2).P(1).P(0).CI + P(2).P(1).G(0) + \ldots + P(2).G(1) + G(2) \quad (13)$$

$$S(0) = P(0) \oplus CI \quad (14)$$
$$S(1) = P(1) \oplus C(1) \quad (15)$$
$$S(2) = P(2) \oplus C(2) \quad (16)$$

donde $C(0) = CI$. El circuito lógico resultante se muestra en la Fig. 2.

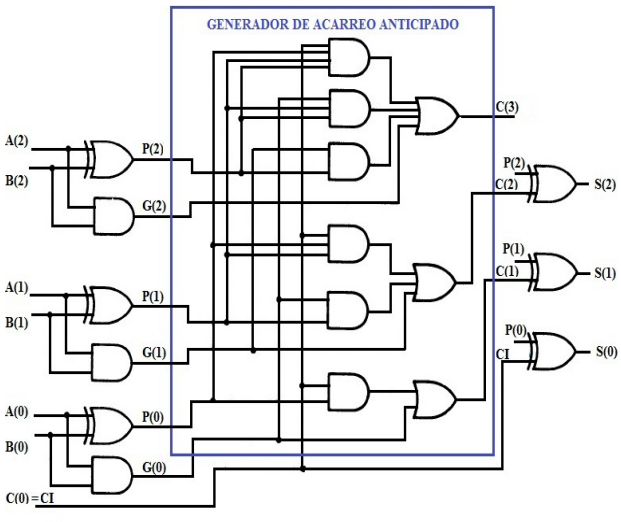

Fig.2. Sumador de 3 bits con acarreo anticipado.

B. Sumador restador

Si vamos a restar $A - B$, esta operación puede convertirse en una suma $A + (-B)$ donde el segundo sumando debe ser un número negativo. Sin embargo, se obtiene el mismo resultado si B lo complementamos a dos y efectuamos la operación de suma con A. El complemento a dos de un número binario se determina si a cada bit del número dado lo complementamos y al resultado de esa operación se le suma la unidad [4]. Es decir, si vamos a restar $A - B$, tendremos que realizar la siguiente operación de suma:

$$R = A - B = A + (/B + 1) = A + /B + 1 \quad (17)$$

Por lo tanto, el circuito de la Fig. 2 se modifica tal como se muestra en la Fig. 3.

Las puertas *XOR* agregadas a la entrada B actúan como inversores si CI = 1 en cuyo caso se invierte cada bit de B, el 1 de CI también se convierte en un sumando adicional, como se muestra en la Figura 3, comprobándose que la salida del circuito es coherente con la ecuación (17), es decir dando como resultado la resta de $A - B$. En el caso de que $CI = 0$, la entrada B no es afectada siendo el resultado la suma de $A+B$.

Las ecuaciones (1) y (2) para $P(i)$ y $G(i)$ respectivamente, son modificadas de la siguiente forma:

$$Pi = A(i) \oplus (B(i) \oplus CI) \quad (18)$$
$$Gi = A(i) \cdot (B(i) \oplus CI) \quad (19)$$

donde las expresiones para $C(i)$ y $S(i)$ permanecen iguales. En nuestro ejemplo sencillo para el Sumador/Restador de 3 bits, las ecuaciones resultantes para $P(i)$ y $G(i)$ son:

$$P(0) = A(0) \oplus (B(0) \oplus CI) \quad (20)$$
$$P(1) = A(1) \oplus (B(1) \oplus CI) \quad (21)$$
$$P(2) = A(2) \oplus (B(2) \oplus CI) \quad (22)$$

$$G(0) = A(0) \cdot (B(0) \oplus CI) \quad (23)$$
$$G(1) = A(1) \cdot (B(1) \oplus CI) \quad (24)$$
$$G(2) = A(2) \cdot (B(2) \oplus CI) \quad (25)$$

C. Dispositivos Lógicos Programables Complejos (CPLD).

Uno de los más trascendentales avances en la electrónica digital ha sido la introducción de los dispositivos lógicos programables (PLDs). Antes del desarrollo de los PLDs, los circuitos digitales se construyeron en integración a pequeña escala (SSI) e integración a media escala (MSI), donde estos dispositivos contenían puertas lógicas y otros circuitos

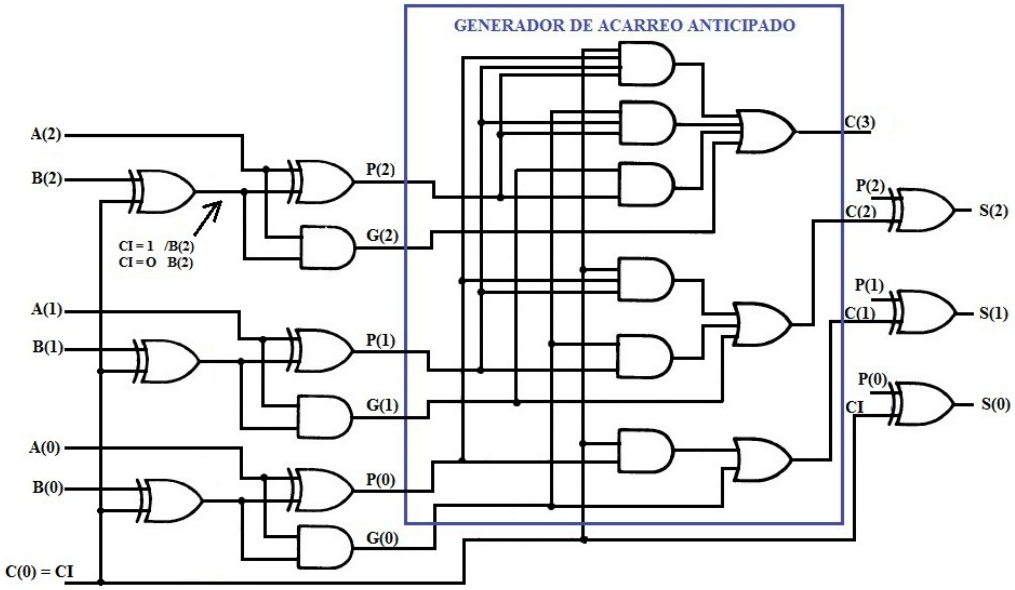

Fig.3. Sumador / Restador de 3 bits con acarreo anticipado.

digitales. Las funciones se determinaban en el momento de la fabricación y no podían ser cambiadas. Además, si un diseñador deseaba un dispositivo con una función en particular que no estaba en la lista de un fabricante, estaba obligado a utilizar varios dispositivos, algunos de los cuales podía contener funciones que no necesitaba, desperdiciando así el espacio en la tarjeta electrónica y tiempo de diseño. Los dispositivos lógicos programables nos proporcionan una solución a estos problemas.

Un PLD se suministra al usuario sin ninguna función lógica programada en absoluto, éstas son solo especificadas por el diseñador según sus necesidades. Dado que por lo general varias funciones se pueden combinar en el diseño interno del chip, el espacio utilizado en él puede reducirse. Además, si un diseño necesita ser cambiado, un PLD se puede reprogramar con la nueva información de diseño generalmente sin sacarlo del circuito. PLD es un término genérico, hay una amplia variedad de tipos de PLD, como por ejemplo: PAL (Arreglo Lógico Programable), GAL (Arreglo Lógico Genérico), EPLD (PLD Borrable), CPLD (PLD Complejo), FPGA (Arreglo de Puertas Programable en el Campo), así como otros. Aunque la terminología puede variar un poco, vamos a utilizar en el presente artículo el término CPLD para referirnos a un dispositivo con varias secciones de PLDs interconectados en un solo chip [5], [6]. En este trabajo se ha empleado concretamente el CPLD EPM7128SLC84-7 de la firma Altera, disponible en el Instituto de Investigación de la FIEE-UNMSM que por su dimensión satisface a los requerimientos del estudio.

II. METODOLOGÍA

El Diagrama de Bloques de la Unidad Aritmética y Lógica (ALU) mostrada en la Fig. 4, explica en forma general el funcionamiento del circuito implementado, donde cada una de las entradas de datos *A*[15..0] y *B*[15..0] tienen una longitud de dieciséis (16) bits, la salida esta constituida por *CO* y *OUT*[15..0] con una longitud total de diecisiete (17) bits, donde el bit más significativo (bit 17) es *CO*. Con los dos bits de entrada en los terminales de selección de operación *F1F0*, es posible seleccionar cuatro operaciones distintas, tal como se muestra en la Tabla I.

La asociación de los tres multiplexores 2 a 1 equivale a tener un multiplexor de 4 a 1 con datos de entrada de 16 bits. Dos de las entradas del multiplexor son alimentadas al mismo tiempo por la salida del bloque Sumador/Restador, es así que, cuando F0 = *F1*= *CI* = *SEL* = 0 el resultado de la suma se transfiere a *OUT*[15..0], mientras que cuando *F1*= 0 y *F0* = *CI* = *SEL* = 1 el resultado de la resta se transfiere a *OUT*[15..0]. De la misma forma el resultado de la operación lógica *AND* (bit a bit) y el resultado de la operación lógica *AND* (bit a bit) y el resultado de la operación lógica *XOR* (bit a bit) son conectadas a dos entradas distintas de otro multiplexor, de tal modo que

TABLA I.
FUNCIONAMIENTO DEL ALU

F1	F0	OUT (15..0)
0	0	Sumador (C0 = Carry)
0	1	Restador
1	0	AND
1	1	XOR

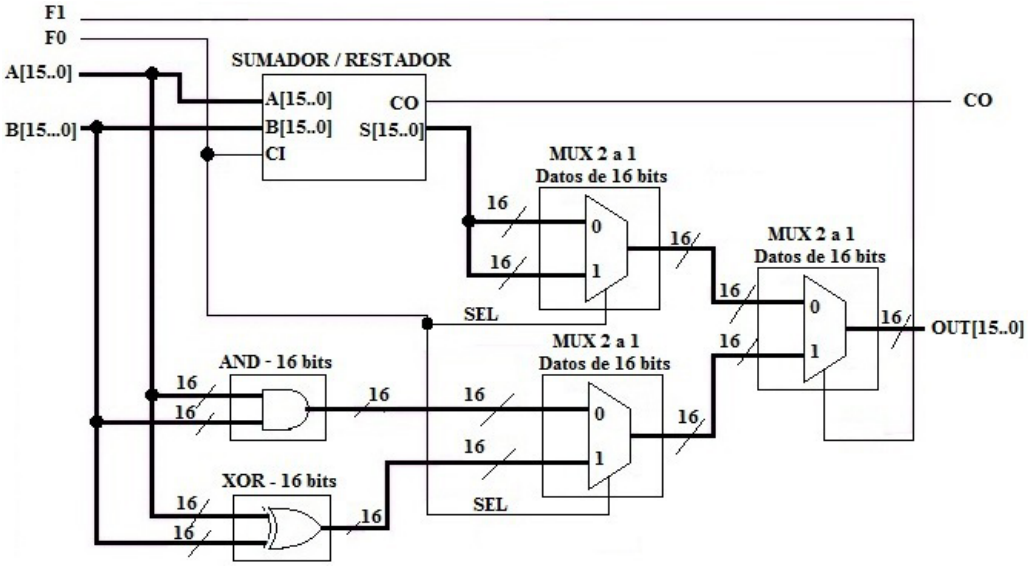

Fig.4. Diagrama de Bloques del ALU

cuando *SEL* = *F0* = 0 y *F1*=1 el resultado de la operación lógica *AND* se transfiere a *OUT*[15..0], mientras que si *SEL* = *F0* = *F1* = 1 el resultado de la operación lógica *XOR* se transfiere a *OUT*[15..0].

El Sumador/Restador ha sido diseñado en función a las ecuaciones (3), (4), (18) y (19). En el ítem C se ha desarrollado un ejemplo de aplicación de estas ecuaciones para un pequeño Sumador/Restador de 03 bits. Sin embargo, el Sumador/Restador de nuestra implementación es mucho más compleja porque se trata de 16 bits, aunque el procedimiento para su deducción es el mismo que el empleado en ese ejemplo.

Una vez deducida las ecuaciones del circuito Sumador/Restador, éstas se han codificado en VHDL con el editor del Quartus II V. 9.1 *Web Edition* de libre distribución. El código es bastante extenso, por este motivo para mostrarlo se ha separado en tres secciones. Así, en la Fig. 5 se muestra la sección del encabezado, el código para el arrastre propagado P(*i*) y el arrastre generado *G*(*i*) a la entidad se le ha denominado sumares16. En la Fig. 6, se muestra la sección para el código de arrastre *C*(*i*) y finalmente en la Fig.7, se muestra la sección del código para el resultado *S*(*i*). El lector no tendrá ningún problema en deducir la parte del código obviado, mostrado en puntos suspensivos, porque es reiterativo.

Luego, de haber realizado la codificación en VHDL del Sumador/Restador identificado con el nombre de sumares16 se le creó un símbolo lógico, es decir, se creó un bloque lógico cuyas entradas son *A*[15..0], *B*[15..0], *CI* y cuyas salidas son *S*[15..0] y *CO*, y se guardó para su posterior utilización. Para detalles de cómo se crea un símbolo lógico a partir de un código VHDL puede consultar la referencia **[7]**. De la misma forma se realizó la codificación en VHDL para un multiplexor de 2 a 1 (con datos de entrada de 16 bits) a la entidad se la denominó MUX16; la codificación para la operación lógica *AND* cuya entidad se denomino AND16

```
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.all;
entity sumares16 is
port( A:IN STD_LOGIC_VECTOR (15 downto 0);
    B:IN STD_LOGIC_VECTOR (15 downto 0);
    CI:IN STD_LOGIC;
    CO:OUT STD_LOGIC;
    S:OUT STD_LOGIC_VECTOR (15 downto 0));
end sumares16;
ARCHITECTURE ESTRUCTURAL OF sumares16 IS
signal P: STD_LOGIC_VECTOR (15 downto 0);
signal G: STD_LOGIC_VECTOR (15 downto 0);
signal C: STD_LOGIC_VECTOR (16 downto 1);
begin

P(0) <= A(0) xor (B(0) xor CI);
P(1) <= A(1) xor (B(1) xor CI);
P(2) <= A(2) xor (B(2) xor CI);
.
.
.
P(14) <= A(14) xor (B(14) xor CI);
P(15) <= A(15) xor (B(15) xor CI);

G(0) <= A(0) and (B(0) xor CI);
G(1) <= A(1) and (B(1) xor CI);
G(2) <= A(2) and (B(2) xor CI);
.
.
G(14) <= A(14) and (B(14) xor CI);
G(15) <= A(15) and (B(15) xor CI);
```

Fig. 5. Código VHDL del Sumador/Restador.

```
C(1) <= (P(0) and CI) or G(0);
C(2) <= (P(1) and P(0) and CI) or (P(1) and G(0)) or G(1);
C(3) <= (P(2) and P(1) and P(0) and CI) or (P(2) and P(1) and G(0)) or (P(2) and G(1)) or G(2);
.
.
.
C(16) <= (P(15) and P(14)and P(13) and P(12) and P(11) and P(10) and P(9) and P(8)and P(7)and P(6) and
P(5) and P(4) and P(3) and P(2) and P(1) and P(0) and CI) or (P(15) and P(14)and P(13) and P(12) and P(11)
and P(10) and P(9) and P(8) and P(7)and P(6) and P(5) and P(4) and P(3) and P(2)and P(1) and G(0)) or (P(15)
and P(14)and P(13) and P(12) and P(11) and P(10) and P(9) and P(8) and P(7)and P(6)and P(5) and P(4) and
P(3) and P(2) and G(1)) or (P(15) and P(14)and P(13) and P(12) and P(11) and P(10) and P(9) and P(8) and
P(7)and P(6) and P(5) and P(4) and P(3) and G(2)) or (P(15) and P(14)and P(13) and P(12) and P(11) and P(10)
and P(9) and P(8) and P(7)and P(6) and P(5) and P(4) and G(3)) or (P(15) and P(14)and P(13) and P(12) and
P(11) and P(10) and P(9) and P(8) and P(7)and P(6) and P(5)and G(4)) or (P(15) and P(14)and P(13) and P(12)
and P(11) and P(10) and P(9) and P(8) and P(7)and P(6)and G(5)) or (P(15) and P(14)and P(13) and P(12) and
P(11) and P(10) and P(9) and P(8) and P(7)and G(6)) or (P(15) and P(14)and P(13) and P(12) and P(11) and
P(10) and P(9) and P(8) and G(7)) or (P(15) and P(14)and P(13) and P(12) and P(11) and P(10) and P(9) and
G(8)) or (P(15) and P(14)and P(13) and P(12) and P(11) and P(10) and G(9)) or (P(15) and P(14)and P(13) and
P(12) and P(11) and G(10)) or (P(15) and P(14)and P(13) and P(12) and G(11)) or (P(15) and P(14)and P(13)
and G(12)) or (P(15) and P(14)and G(13)) or (P(15) and G(14)) or G(15);

CO <= C(16) xor CI; -- -- C(16) es el acarreo de la suma.
                    -- --  Para la resta hay que descartar su valor por eso como CI=1 y si C(16) es 1,
                    -- -- entonces C(16) xor 1 = 0. Para la suma CI=0, entonces C(16) xor 0 = C(16).
```

Fig. 6. Código VHDL del Sumador/Restador, sección Arrastre C(*i*)

```
S(0) <= P(0) xor CI;
S(1) <= P(1) xor C(1);
S(2) <= P(2) xor C(2);
.
.
.
S(12) <= P(12) xor C(12);
S(13) <= P(13) xor C(13);
S(14) <= P(14) xor C(14);
S(15) <= P(15) xor C(15);

end ESTRUCTURAL;
```

Fig. 7. Código VHDL del Sumador/Restador, sección resultado de la Suma o Resta *S*(*i*).

y la codificación para la operación lógica *XOR* cuya entidad se le denominó XOR16. Los códigos VHDL de cada uno de ellos se muestran respectivamente en la Fig. 8, Fig. 9 y Fig.10. En base a estos códigos en VHDL se crearon sus respectivos símbolos lógicos tal como se realizó para el caso del SUMADOR\RESTADOR.

Posteriormente, se creó con el editor del QUARTUS un esquemático en donde se unieron los símbolos lógicos (bloques lógicos) sumares16, MUX16, AND16 y XOR16. Luego, se simuló al circuito con la herramienta de simulación del QUARTUS. Finalmente, con el código generado se ha programado al CPLD EPM7128SLC84-7 de Altera, el cual está inserto en la tarjeta de desarrollo *University Program UP2.*

Como parte del proceso de la programación, fueron designados a los pines físicos del CPLD EPM7128SLC84-7CPLD las entradas y salidas del circuito. Se han utilizado en total 51 pines del CPLD, distribuidas de la siguiente forma: dieciséis (16) pines para la entrada *A*[15..0], dieciséis (16) pines para la entrada *B*[15..0], un (1) pin para la entrada *CI*, dos pines para las entradas de control *F1F0*, dieciséis (16) pines para la salida *OUT*[15..0] y un (1) pin para la salida *CO*. En la Fig. 11, se muestra el tipo de CPLD utilizado con la especificación de sus pines, y en la Fig. 12, se muestra el esquemático que se editó en QUARTUS, en donde además se pueden observar los pines que le corresponden físicamente a las entradas y salidas del circuito. Para mayores detalles de cómo programar a un CPLD se puede consultar con la referencia [8].

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

ENTITY MUX16 IS
  PORT(dataa  : IN std_logic_vector(15 DOWNTO 0);
       datab  : IN std_logic_vector(15 DOWNTO 0);
       sel    : IN std_logic;
       result : OUT std_logic_vector(15 DOWNTO 0));
END MUX16;

ARCHITECTURE COMPORTAMIENTO OF MUX16 IS
BEGIN
 PROCESS (sel, dataa, datab) IS
 BEGIN
  CASE sel IS
    WHEN '0' => result <= dataa;
    WHEN '1' => result <= datab;
  END CASE;
 END PROCESS;
END COMPORTAMIENTO;
```

Fig. 8. Código VHDL para el Multiplexor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

ENTITY AND16 IS
      PORT (A: IN std_logic_vector(15 DOWNTO 0);
            B: IN std_logic_vector(15 DOWNTO 0);
            OAND: OUT std_logic_vector(15 DOWNTO 0));
END AND16;

ARCHITECTURE ESTRUCTURAL OF AND16 IS
BEGIN
OAND(0) <= A(0) and B(0);
OAND(1) <= A(1) and B(1);
OAND(2) <= A(2) and B(2);
OAND(3) <= A(3) and B(3);
OAND(4) <= A(4) and B(4);
OAND(5) <= A(5) and B(5);
OAND(6) <= A(6) and B(6);
OAND(7) <= A(7) and B(7);
OAND(8) <= A(8) and B(8);
OAND(9) <= A(9) and B(9);
OAND(10) <= A(10) and B(10);
OAND(11) <= A(11) and B(11);
OAND(12) <= A(12) and B(12);
OAND(13) <= A(13) and B(13);
OAND(14) <= A(14) and B(14);
OAND(15) <= A(15) and B(15);
END ESTRUCTURAL;
```

Fig. 9. Código VHDL para función AND.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

ENTITY XOR16 IS
      PORT (A: IN std_logic_vector(15 DOWNTO 0);
            B: IN std_logic_vector(15 DOWNTO 0);
            OXOR: OUT std_logic_vector(15 DOWNTO 0));
END XOR16;

ARCHITECTURE ESTRUCTURAL OF XOR16 IS
BEGIN
OXOR(0) <= A(0) xor B(0);
OXOR(1) <= A(1) xor B(1);
OXOR(2) <= A(2) xor B(2);
OXOR(3) <= A(3) xor B(3);
OXOR(4) <= A(4) xor B(4);
OXOR(5) <= A(5) xor B(5);
OXOR(6) <= A(6) xor B(6);
OXOR(7) <= A(7) xor B(7);
OXOR(8) <= A(8) xor B(8);
OXOR(9) <= A(9) xor B(9);
OXOR(10) <= A(10) xor B(10);
OXOR(11) <= A(11) xor B(11);
OXOR(12) <= A(12) xor B(12);
OXOR(13) <= A(13) xor B(13);
OXOR(14) <= A(14) xor B(14);
OXOR(15) <= A(15) xor B(15);
END ESTRUCTURAL;
```

Fig. 10. Código VHDL para la función XOR.

Posteriormente, para probar físicamente el funcionamiento del circuito se han utilizado 51 cables, con los cuales se han expandido hacia tres placas de prueba o *protoboard* los pines de entradas y salidas del CPLD, tal como se muestra en la fotografía de la Fig.13. En los circuitos de experimentación se han colocado diodos LED para visualizar los estados lógicos de las entradas y salidas. Además, para las salidas se han utilizado diecisiete (17) transistores *drivers* con su correspondiente LED, la fuente de alimentación DC utilizada, fue la misma que alimentaba al CPLD.

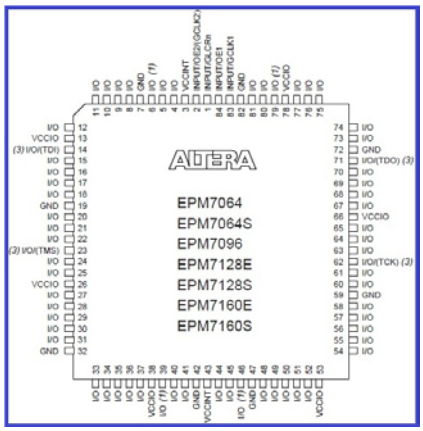

Fig.11. CPLD utilizado con la descripción de sus pines.

III. RESULTADOS

En esta sección se demuestra mediante imágenes de las Fig. 14, 15, 16 y 17 los resultados obtenidos, en ellas un LED encendido corresponde a uno (1) lógico, mientas que un LED apagado corresponde a un cero (0) lógico. Se colocaron dieciséis (16) LEDs para la entrada *A*, dieciséis (16) LEDs para la entrada *B* y diecisiete (17) LEDs para la salida *R*, tanto las entradas como las salidas son números en binario. En la Fig. 14, el ALU está configurado como un Sumador através de los pines de control (*F1F0*=00), el sumando *A* es: 1111111111111111 (65535 en decimal) y el sumando *B* es: 1111111111111111 (65535 en decimal) verificándose un resultado correcto de *R* igual a: 11111111111111110 (131070 en decimal). En la Fig.15, el ALU está configurado como un Restador mediante los pines de control (*F1F0*=01), el minuendo *A* es: 1000000000000000 (32768 en decimal) y el sustraendo *B* es: 0000000000000001 (1 en decimal), verificándose un resultado correcto de *R* igual a: 00111111111111111 (32767 en decimal).

En la Fig.16, el ALU está configurado como un AND mediante los pines de control (*F1F0*=10), los bits de *A* son 0111111111111111, mientras los bits de *B* son: 0010101111010101, de tal manera que luego de verificado la posición correcta de cada uno de los datos, la operación lógica AND bit a bit, en R resultó igual a: 0010101111010101. Finalmente, en la Fig.17, el ALU está configurado como un *XOR* mediante los pines de control (*F1F0*=11), los bits de *A* son 1111111111111111, mientras los bits de *B* son: 1010101111010101 verificándose correctamente la operación lógica XOR bit a bit, en R igual a: 0101010000101010.

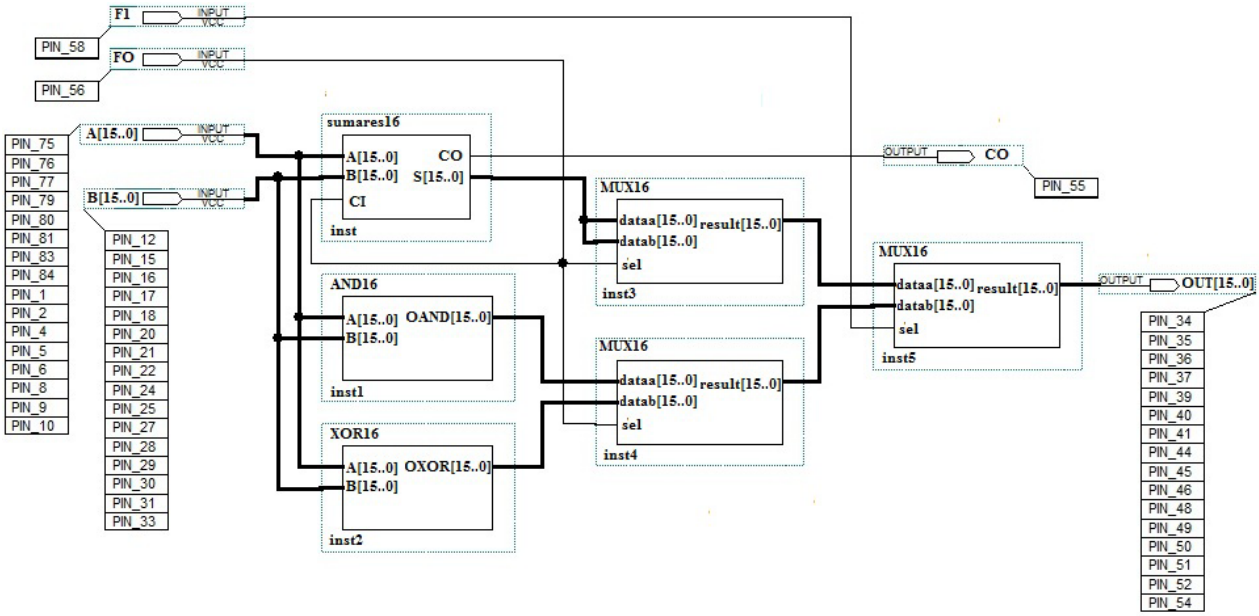

Fig.12. Esquemático del ALU extraído del editor del QUARTUS.

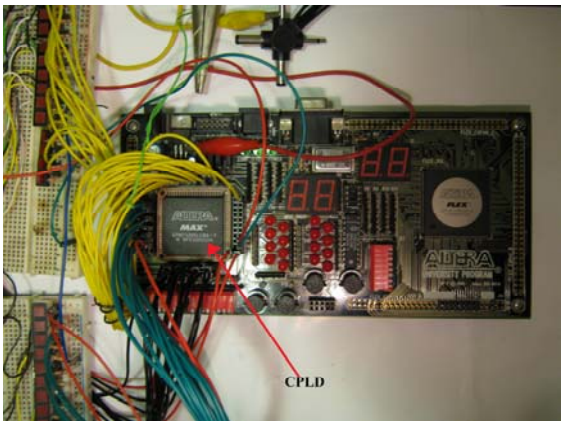

Fig.13. Pines del CPLD expandidos al *protoboard.*

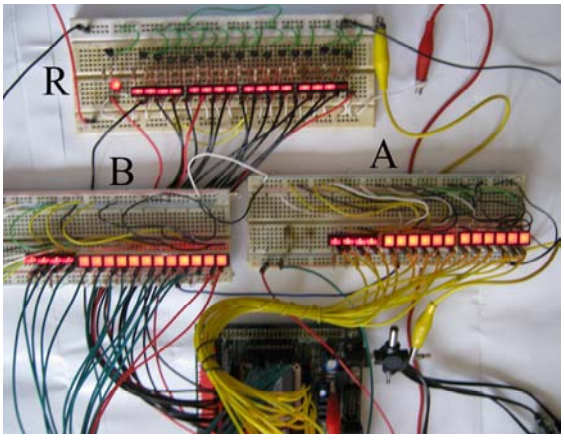

Fig.14. ALU como un sumador.

I. CONCLUSIONES

Se ha codificado e implementado con éxito un ALU de 16 bits, en la que se resalta el hecho de que el Sumador/Restador incorporado dentro del ALU corresponde a uno con arrastre anticipado que se destaca por su velocidad. Se han implementado cuatro operaciones lógicas, de allí que he utilizado internamente un arreglo de multiplexores equivalente a tener uno de 4 a 1. Si se desea ampliar el número de operaciones a un valor como *N* el multiplexor debe ser *N* a 1. De los datos proporcionados por el editor del QUARTUS se sabe que el Sumador/Restador ha utilizado sesentaiuno (61) macroceldas, los multiplexores han utilizado un total de veintidós (22) macroceldas y las puertas *AND* y *XOR* ambos han utilizado una (1) macrocelda, quedando aun por utilizar 44 macroceldas, lo que nos permitiría agregar muchas más funciones dentro del ALU. El porcentaje de pines utilizado ha sido del 81%, quedando por utilizar 13, lo que será más que suficiente si desearíamos aumentar los pines de selección del multiplexor y por lo tanto de las funciones del ALU.

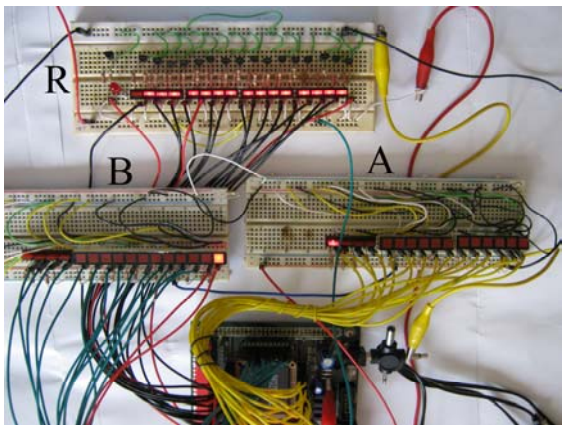

Fig. 15. ALU como un restador.

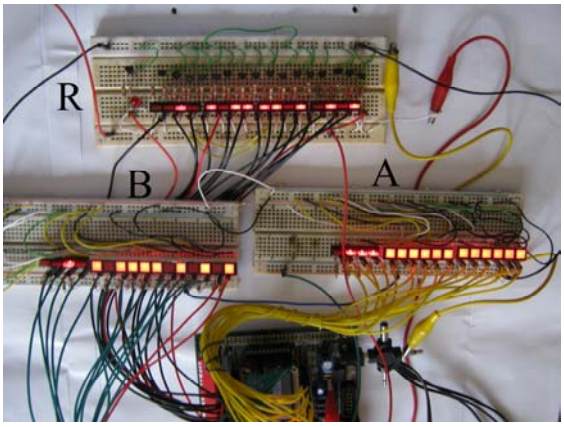

Fig.16. ALU como *AND*.

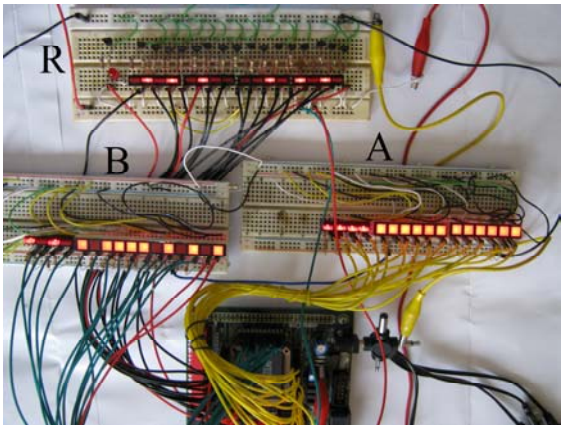

Fig.17. ALU como *XOR*.

REFERENCIAS

[1] Williams Stallings, "Organización y Arquitectura de Computadores", 7ª Edición, *Prentice Hall*, Madrid, 2005.

[2] Tocci, Ronald J. y Widmer, Neal S., "Sistemas Digitales", Octava edición, *Pearson Educación*, México, 2003.

[3] Morris Mano,"Diseño Digital", 3era Edición, *Pearson Prentice Hall*, 2003.

[4] José María Angulo Usategui, Ignacio Angulo Martínez, Mikel Etxceberria Isuskiza, Juan Carlos Hernández Martin, María Angeles Prieto, "Electrónica Digital y Microprogramable", *Ediciones Paraninfo*; Madrid, 2010.

[5] Robert Dueck, "Digital Design with CPLD and VHDL", *DELMAR Cengage Learning*, 2nd. edition, 2004.

[6] Guillermo Tejada M., "CPLD para el Control de un Motor Paso a Paso", *Revista Electrónica-UNMSM*, N° 31, pp.5-15, junio 2013.

[7] Diseño Lógico con Quartus II 10, http://www.youtube.com/watch?v=hajv4p6B8o4, fecha de visita: mayo 2012.

[8] Diseño Lógico con Quartus II 11, http://www.youtube.com/watch?v=99G_mK5_Tdo, fecha de visita: mayo del 2012.