

Unidad Aritmética Lógica (ALU)

Tyrone Novillo

Facultad de ingeniería

Universidad de Cuenca

Cuenca, Ecuador

tyrone.novillo@ucuenca.edu.ec

Resumen—This report explains the design and implementation of a Arithmetic Logic Unit (ALU) using VHDL, a standard for digital system design. Key VHDL components include binary word addition, arithmetic operations (such as addition and subtraction), logical operations (like AND and OR), and a multiplexer for output selection. Components lead a modular and efficient approach to digital hardware implementation. Emphasizing scalability and flexibility, this study underscores the importance of modular design in facilitating system construction and maintenance.

Keywords— Arithmetic Logic Unit (ALU), VHDL, binary addition, arithmetic operations, logical operations, multiplexer, modular design

I. INTRODUCCIÓN

La Unidad Aritmético-Lógica (ALU) es un componente fundamental en la arquitectura de computadores que realiza operaciones tanto aritméticas como lógicas sobre datos binarios. En este trabajo, se aborda la implementación de una ALU utilizando el lenguaje VHDL. Se han desarrollado varios componentes VHDL esenciales para la funcionalidad de la ALU, incluyendo la suma de palabras binarias, la unidad aritmética para operaciones como sumas y restas, la unidad lógica para operaciones booleanas como AND y OR, y un multiplexor para seleccionar entre múltiples salidas. Cada componente se instancia según las especificaciones de diseño para cumplir con las funcionalidades requeridas, promoviendo así una implementación modular y eficiente del hardware digital. Este enfoque modular no solo facilita la construcción y mantenimiento del sistema, sino que también asegura un diseño escalable.

II. DESARROLLO

Se requiere implementar un hardware cuya estructura está descrita en la Figura 1 con las funcionalidades de la Figura 2.

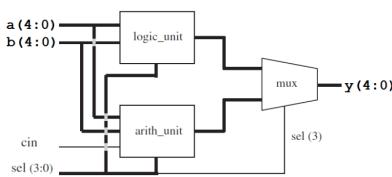


Figura 1: Diagrama de la ALU a implementar

sel	Operation	Function	Unit
0000	$y \leq a$	Transfer a	Arithmetic
0001	$y \leq a+1$	Increment a	
0010	$y \leq a-1$	Decrement a	
0011	$y \leq b$	Transfer b	
0100	$y \leq b+1$	Increment b	
0101	$y \leq b-1$	Decrement b	
0110	$y \leq a+b$	Add a and b	
0111	$y \leq a+b+cin$	Add a and b with carry	
1000	$y \leq \text{NOT } a$	Complement a	
1001	$y \leq \text{NOT } b$	Complement b	
1010	$y \leq a \text{ AND } b$	AND	Logic
1011	$y \leq a \text{ OR } b$	OR	
1100	$y \leq a \text{ NAND } b$	NAND	
1101	$y \leq a \text{ NOR } b$	NOR	
1110	$y \leq a \text{ XOR } b$	XOR	
1111	$y \leq a \text{ XNOR } b$	XNOR	

Figura 2: Funcionalidades de la ALU a implementar

Para ello, se utilizarán componentes para cada unidad, tanto la aritmética como la lógica. Para esta práctica se evitará el uso de la librería `ieee.std_logic_1164.all`. Los componentes que se implementarán serán los siguientes:

- Componente para dividir frecuencia definido en prácticas anteriores.
- Componente que describe la operación de la unidad lógica.
- Componente que describe la operación de la unidad aritmética.
- Componente que define la suma de dos valores en binario.
- Componente de multiplexación de unidades.

II-A. Componente de suma de palabras binarias

La entidad `suma_palabras` define un componente que realiza la suma de dos vectores de 4 bits (a y b), generando un vector de 4 bits como resultado. En la arquitectura, se utiliza una señal interna llamada `carry` para gestionar los acarreos durante la suma bit a bit. Un proceso sensible a las entradas a y b se encarga de calcular la suma utilizando operadores XOR y AND, y gestionando los acarreos mediante operaciones lógicas. El proceso comienza sumando los bits menos significativos, almacenando el acarreo correspondiente, y continúa sumando los siguientes bits junto con los acarreos generados previamente. Finalmente, se asigna el bit de acarreo más significativo al vector `sum` y se asigna el resultado de la suma a la salida `resultado`.

II-B. Componente de unidad aritmética

La entidad `aritmetica` define un componente que realiza varias operaciones aritméticas básicas basadas en la entrada de un vector de selección (`sel`). Este componente utiliza el componente `suma_palabras` para realizar sumas y restas de vectores de 4 bits. Las entradas al componente `aritmetica` son a y b , que son vectores de 3 bits, y un bit de acarreo de entrada (`cin`). Las salidas son el vector `resultado` de 4 bits. Dentro de la arquitectura, se definen varias señales internas para gestionar las operaciones y convertir las entradas de 3 bits a 4 bits mediante la concatenación de un bit de '0' al principio. Estas señales se utilizan para realizar las operaciones requeridas basadas en el valor del vector de selección `sel`.

El componente instancia seis veces el componente `suma_palabras` para realizar sumas y restas con a y b .

Estas instancias calculan la suma de a más 1, la resta de a menos 1, la suma de b más 1, la resta de b menos 1, la suma de a y b, y la suma de a y b con el acarreo de entrada. Un proceso sensible a las señales de selección `sel`, a, y b determina qué operación se realiza y asigna el resultado correspondiente a la salida `resultado`. Dependiendo del valor de `sel`, el resultado puede ser directamente el vector a extendido a 4 bits, a más 1, a menos 1, b extendido a 4 bits, b más 1, b menos 1, la suma de a y b, o la suma de a y b con el acarreo de entrada. Si `sel` tiene un valor diferente a los especificados, el resultado será "0000".

II-C. Componente de unidad lógica

La entidad `logica` está diseñada para realizar varias operaciones lógicas básicas sobre dos vectores de entrada de 3 bits, a y b, basándose en un vector de selección de 4 bits (`sel`). La salida del componente es un vector de 4 bits llamado `led_resultado`. Dentro de la arquitectura Behavioral, se define una señal interna `resultado` para almacenar el resultado de las operaciones lógicas, y otra señal `resultado_final` para almacenar el resultado final extendido a 4 bits mediante la concatenación de un bit de '0' al principio.

El componente contiene un proceso sensible a las señales `sel`, a y b, que determina qué operación lógica se debe realizar en función del valor de `sel`. Las operaciones lógicas disponibles son: negación de a (not a), negación de b (not b), AND (a and b), OR (a or b), NAND (a nand b), NOR (a nor b), XOR (a xor b), y XNOR (a xnor b). Si el valor de `sel` no coincide con ninguna de las especificadas, el resultado es "000". Después de calcular la operación lógica correspondiente, el resultado se extiende a 4 bits y se asigna a `led_resultado`.

II-D. Componente de multiplexación

La entidad `mux` representa un multiplexor de 4 bits, el cual selecciona entre dos vectores de entrada a y b en función del valor de la señal de selección `sel`. La salida del multiplexor se denota como x, y es un vector de 4 bits.

Dentro de la arquitectura denominada `mux`, se utiliza una declaración `with ... select` para definir cómo la señal de salida x se asigna en función del valor de `sel`. Si `sel` es '0', x toma el valor de a. En cualquier otro caso (representado por `others`), x toma el valor de b. Esta estructura proporciona una manera clara y concisa de implementar el comportamiento de un multiplexor, seleccionando una de las dos entradas en función de una señal de control.

II-E. Programa principal

La entidad `leccion2` se encarga de realizar operaciones aritméticas y lógicas utilizando señales de entrada, actualizar un display de 7 segmentos y controlar un conjunto de LEDs. Las señales de entrada son el reloj (`clk`), los vectores de entrada (a y b) que son de 3 bits cada uno, y las señales de control (`sel` de 4 bits y `cin` de 1 bit). Las salidas incluyen `act_display` para activar el display, `display_7_segmentos` para mostrar el resultado en un display de 7 segmentos, y `led_resultado` para mostrar el resultado en LEDs. Internamente, se definen varias señales intermedias como `resultado_aritmetico`, `resultado`, `resultado_led`, y `resultado_bcd` para manejar los resultados de las operaciones, un contador de LEDs (`contador_led`) y una señal BCD (BCD) para la representación en el display.

El diseño utiliza varios componentes definidos previamente: `divisor_frecuencia`, `aritmetica`, `logica`, y `mux`. El componente `divisor_frecuencia` se instancia con un divisor para generar una señal de reloj de 1 ms, que se usa para sincronizar la actualización de los LEDs y el display de 7 segmentos. Los componentes `aritmetica` y `logica` se instancian para realizar operaciones aritméticas y lógicas respectivamente, basándose en la señal de control `sel`. El componente `mux` se utiliza para seleccionar

entre los resultados aritméticos y lógicos, controlado por el bit más significativo de `sel` (`sel(3)`). Estas instancias aseguran una separación modular y clara de las funciones dentro del diseño, facilitando su mantenimiento y escalabilidad.

El proceso de multiplexación para los displays de 7 segmentos se maneja mediante varios procesos secuenciales. Un proceso se encarga de incrementar el contador de LEDs (`contador_led`) cada 1 ms usando la señal de reloj generada por `divisor_frecuencia`. Dependiendo del valor de `contador_led`, se selecciona y se muestra un valor diferente en el display de 7 segmentos. Por ejemplo, cuando el contador es 0, se muestra el valor de a, cuando es 1, se muestra b, y cuando es 2 o 3, se muestran los resultados de las operaciones basadas en las condiciones de selección. El valor BCD correspondiente se asigna a `display_7_segmentos` mediante un proceso que mapea los valores binarios a los segmentos del display. Este proceso garantiza que los resultados se visualicen correctamente y de manera sincronizada, asegurando una actualización fluida y precisa del display.

II-F. Constraints

El archivo `.xdc` define restricciones de asignación de pines en la placa Basys3 rev B. Estas restricciones relaciona señales lógicas como la del reloj principal (`clk`) proporcionado por la Basys 3 de 100 MHz, las entradas de los vectores a y b, así como las señales de selección (`sel`) y de acarreo (`cin`). También se asignan pines para la salida hacia los LEDs (`led_resultado`) y el display de 7 segmentos (`display_7_segmentos`).

III. CONCLUSIONES

La implementación de componentes en el diseño de una Unidad Aritmético-Lógica (ALU) ofrece ventajas permite entender con más claridad el programa, permite un mejor mantenimiento y la reutilización del código. Se ha dividido las operaciones en entidades separadas como la suma de palabras binarias, la unidad aritmética, la unidad lógica y el multiplexor. La ALU es responsable de ejecutar operaciones aritméticas (como suma y resta) y lógicas (como AND, OR) sobre datos binarios.

IV. ANEXOS

IV-A. Archivo leccion2.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity leccion2 is
Port (
clk : in std_logic ;
a: in std_logic_vector(2 downto 0);
b: in std_logic_vector(2 downto 0);
act_display : out std_logic_vector(3 downto 0);
display_7_segmentos: out std_logic_vector(6 downto 0);
led_resultado : out std_logic_vector(3 downto 0);
sel: in std_logic_vector(3 downto 0);
cin: in std_logic
);
end leccion2;

architecture Behavioral of leccion2 is

signal contador_led: integer := 0;
signal BCD: std_logic_vector(3 downto 0);
constant CONTADOR1MS: integer := 50_000;
signal clk_1ms : std_logic ;

type state_type is (arit, log);
signal PS, NS : state_type;
```

```

signal resultado_aritmetico: std_logic_vector(3 downto 0);
signal resultado: std_logic_vector(3 downto 0);
signal resultado_led: std_logic_vector(3 downto 0) process (clk)
signal resultado_bcd: std_logic_vector(7 downto 0) begin
    if rising_edge(clk) then
        if ( sel = "0---" ) then
            led_resultado <= "0000";
            mascara_anodo <= "0000";
        else
            led_resultado <= resultado_led;
            mascara_anodo <= "1111";
        end if;
    end if;
end process;

--Declaracion de componentes
component divisor_frecuencia
generic (
    divisor : integer
);
Port (
    clk_in : in STD_LOGIC;
    clk_out : out STD_LOGIC
);
end component;

component aritmetica
Port (
    sel: in std_logic_vector(3 downto 0);
    a: in std_logic_vector(2 downto 0);
    b: in std_logic_vector(2 downto 0);
    cin: in std_logic;
    resultado: out std_logic_vector(3 downto 0)
);
end component;

component logica is
Port (
    sel: in std_logic_vector(3 downto 0);
    a: in std_logic_vector(2 downto 0);
    b: in std_logic_vector(2 downto 0);
    led_resultado: out std_logic_vector(3 downto 0)
);
end component;

component mux is
Port (
    a, b: in std_logic_vector (3 downto 0);
    sel: in std_logic;
    x : out std_logic_vector (3 downto 0)
);
end component;

begin
    ins_aritmetica : aritmetica port map (sel => sel, resultado => resultado_aritmetico, ci
    ins_logica : logica port map (sel => sel, a => a, b, led_resultado => resultado_led);
    multiplexor: mux port map (resultado_aritmetico, resultado_bcd(7 downto 4), resultado_bcd(3 downto 0));
    --Generar la señal de lms (tasa de refresco de los led)
    div1ms : divisor_frecuencia
    generic map (divisor => CONTADOR1MS)
    port map (clk_in => clk, clk_out => clk_1ms);

    process (clk_1ms)
    begin
        if (clk_1ms'event and clk_1ms = '1') then
            if contador_led < 3 then
                contador_led <= contador_led + 1;
            else
                contador_led <= 0;
            end if;
        end if;
    end process;

```

```

end case;
end process;

with BCD select
display_7_segmentos <=
"0000001" when "0000",
"1001111" when "0001",
"0010010" when "0010",
"0000110" when "0011",
"1001100" when "0100",
"0100100" when "0101",
"0100000" when "0110",
"0001111" when "0111",
"0000000" when "1000",
"0000100" when "1001",
"1111110" when "1111",
"1111111" when others;

end Behavioral;

IV-B. Archivo aritmetica.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity aritmetica is
Port (
sel: in std_logic_vector(3 downto 0);
a: in std_logic_vector(2 downto 0);
b: in std_logic_vector(2 downto 0);
resultado: out std_logic_vector(3 downto 0);
cin: in std_logic
);
end aritmetica;

architecture Behavioral of aritmetica is

component suma_palabras
Port(
a: in std_logic_vector(3 downto 0);
b: in std_logic_vector(3 downto 0);
resultado: out std_logic_vector(3 downto 0)
);
end component;

signal resultado_suma_1_a : std_logic_vector(3 downto 0);
signal resultado_resta_1_a : std_logic_vector(3 downto 0);
signal resultado_suma_1_b : std_logic_vector(3 downto 0);
signal resultado_resta_1_b : std_logic_vector(3 downto 0);
signal resultado_suma : std_logic_vector(3 downto 0);
signal resultado_suma_acarreo : std_logic_vector(3 downto 0);

begin
acarreo_operacion : std_logic_vector(3 downto 0);
a_4bits : std_logic_vector(3 downto 0);
b_4bits : std_logic_vector(3 downto 0);

begin
acarreo_operacion <= "000" & cin ;
a_4bits <= "0" & a ;
b_4bits <= "0" & b ;

suma_1_a : suma_palabras port map (a => a_4bits, resultado => resultado_suma_1_a );
resta_1_a : suma_palabras port map (a => a_4bits, resultado => resultado_resta_1_a);
suma_1_b : suma_palabras port map (a => b_4bits, resultado => resultado_suma_1_b);
resta_1_b : suma_palabras port map (a => b_4bits, resultado => resultado_resta_1_b);

end Behavioral;

process(sel, a, b)
begin
case sel is
when "0000" =>
resultado <= a_4bits ;
when "0001" =>
resultado <= resultado_suma_1_a ;
when "0010" =>
resultado <= resultado_resta_1_a ;
when "0011" =>
resultado <= b_4bits ;
when "0100" =>
resultado <= resultado_suma_1_b ;
when "0101" =>
resultado <= resultado_resta_1_b ;
when "0110" =>
resultado <= resultado_suma ;
when "0111" =>
resultado <= resultado_suma_acarreo;
when others =>
resultado <= "0000";
end case;
end process;

end Behavioral;

IV-C. Archivo logica.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity logica is
Port (
sel: in std_logic_vector(3 downto 0);
a: in std_logic_vector(2 downto 0);
b: in std_logic_vector(2 downto 0);
led_resultado: out std_logic_vector(3 downto 0)
);
end logica;

architecture Behavioral of logica is

begin
acarreo_operacion : std_logic_vector(3 downto 0);
a_4bits : std_logic_vector(3 downto 0);
b_4bits : std_logic_vector(3 downto 0);

begin
acarreo_operacion <= "000" & cin ;
a_4bits <= "0" & a ;
b_4bits <= "0" & b ;

suma_1_a : suma_palabras port map (a => a_4bits, resultado => resultado_suma_1_a );
resta_1_a : suma_palabras port map (a => a_4bits, resultado => resultado_resta_1_a);
suma_1_b : suma_palabras port map (a => b_4bits, resultado => resultado_suma_1_b);
resta_1_b : suma_palabras port map (a => b_4bits, resultado => resultado_resta_1_b);

end Behavioral;

process(sel, a, b)
begin
case sel is
when "0000" =>
resultado <= a_4bits ;
when "0001" =>
resultado <= resultado_suma_1_a ;
when "0010" =>
resultado <= resultado_resta_1_a ;
when "0011" =>
resultado <= b_4bits ;
when "0100" =>
resultado <= resultado_suma_1_b ;
when "0101" =>
resultado <= resultado_resta_1_b ;
when "0110" =>
resultado <= resultado_suma ;
when "0111" =>
resultado <= resultado_suma_acarreo;
when others =>
resultado <= "0000";
end case;
end process;

```

```

when "1111" =>
resultado <= a xnor b;
when others =>
resultado <= "000";
end case;
resultado_final <= "0" & resultado ;
end process;
led_resultado <= resultado_final;
end Behavioral;

```

IV-D. Archivo suma_palabras.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity suma_palabras is
Port(
a: in std_logic_vector(3 downto 0);
b: in std_logic_vector(3 downto 0);
resultado: out std_logic_vector(3 downto 0)
);
end suma_palabras;

architecture Behavioral of suma_palabras is
signal carry : STD_LOGIC_VECTOR(3 downto 0);
begin
process(a, b)
variable sum: std_logic_vector(4 downto 0);
begin
-- Suma de los bits menos significativos
sum(0) := a(0) XOR b(0);
carry(0) <= a(0) AND b(0);

-- Suma de los bits del segundo bit
sum(1) := a(1) XOR b(1) XOR carry(0);
carry(1) <= (a(1) AND b(1)) OR (a(1) AND carry(0)) OR (b(1) AND carry(0));

-- Suma de los bits del tercer bit
sum(2) := a(2) XOR b(2) XOR carry(1);
carry(2) <= (a(2) AND b(2)) OR (a(2) AND carry(1)) OR (b(2) AND carry(1));

-- Suma de los bits más significativos
sum(3) := a(3) XOR b(3) XOR carry(2);
carry(3) <= (a(3) AND b(3)) OR (a(3) AND carry(2)) OR (b(3) AND carry(2));

-- Asignar el bit de acarreo más significativo a la salida
sum(4) := carry(3);

-- Asignar la salida
resultado <= sum(3 downto 0);
end process;
end Behavioral;

```

IV-E. Archivo mux.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```

```

entity mux is
Port (
a, b: in std_logic_vector (3 downto 0);
sel: in std_logic;

```

```

x : out std_logic_vector (3 downto 0)
);
end mux;

architecture mux of mux is

begin
with sel select
x <= a when '0',
b when others;
end mux;

```

IV-F. Archivo de Constraints Basys-3-Master.xdc

```

# Clock signal
set_property -dict { PACKAGE_PIN W5
→ IOSTANDARD LVCMOS33 } [get_ports clk]
#create_clock -add -name sys_clk_pin -period
→ 10.00 -waveform {0 5} [get_ports clk]

# Switches
set_property -dict { PACKAGE_PIN V17
→ IOSTANDARD LVCMOS33 } [get_ports {b[0]}]
set_property -dict { PACKAGE_PIN V16
→ IOSTANDARD LVCMOS33 } [get_ports {b[1]}]
set_property -dict { PACKAGE_PIN W16
→ IOSTANDARD LVCMOS33 } [get_ports {b[2]}]
set_property -dict { PACKAGE_PIN W17
→ IOSTANDARD LVCMOS33 } [get_ports {sel[0]}]
set_property -dict { PACKAGE_PIN W15
→ IOSTANDARD LVCMOS33 } [get_ports {sel[1]}]
set_property -dict { PACKAGE_PIN V15
→ IOSTANDARD LVCMOS33 } [get_ports {sel[2]}]
set_property -dict { PACKAGE_PIN W14
→ IOSTANDARD LVCMOS33 } [get_ports {sel[3]}]
set_property -dict { PACKAGE_PIN W2
→ IOSTANDARD LVCMOS33 } [get_ports {cin}]
set_property -dict { PACKAGE_PIN U1
→ IOSTANDARD LVCMOS33 } [get_ports {a[0]}]
set_property -dict { PACKAGE_PIN T1
→ IOSTANDARD LVCMOS33 } [get_ports {a[1]}]
set_property -dict { PACKAGE_PIN R2
→ IOSTANDARD LVCMOS33 } [get_ports {a[2]}]

# LEDs
set_property -dict { PACKAGE_PIN W18
→ IOSTANDARD LVCMOS33 } [get_ports {led_resultado[0]}]
set_property -dict { PACKAGE_PIN U15
→ IOSTANDARD LVCMOS33 } [get_ports {led_resultado[1]}]
set_property -dict { PACKAGE_PIN U14
→ IOSTANDARD LVCMOS33 } [get_ports {led_resultado[2]}]
set_property -dict { PACKAGE_PIN V14
→ IOSTANDARD LVCMOS33 } [get_ports {led_resultado[3]}]

#7 Segment Display
set_property -dict { PACKAGE_PIN W7
→ IOSTANDARD LVCMOS33 } [get_ports {display_7_segmentos[6]}]

```

```

set_property -dict { PACKAGE_PIN W6
    ↳ IOSTANDARD LVCMOS33 } [get_ports
    ↳ {display_7_segmentos[5]}]
set_property -dict { PACKAGE_PIN U8
    ↳ IOSTANDARD LVCMOS33 } [get_ports
    ↳ {display_7_segmentos[4]}]
set_property -dict { PACKAGE_PIN V8
    ↳ IOSTANDARD LVCMOS33 } [get_ports
    ↳ {display_7_segmentos[3]}]
set_property -dict { PACKAGE_PIN U5
    ↳ IOSTANDARD LVCMOS33 } [get_ports
    ↳ {display_7_segmentos[2]}]
set_property -dict { PACKAGE_PIN V5
    ↳ IOSTANDARD LVCMOS33 } [get_ports
    ↳ {display_7_segmentos[1]}]
set_property -dict { PACKAGE_PIN U7
    ↳ IOSTANDARD LVCMOS33 } [get_ports
    ↳ {display_7_segmentos[0]}]

#set_property -dict { PACKAGE_PIN V7
#    ↳ IOSTANDARD LVCMOS33 } [get_ports dp]

set_property -dict { PACKAGE_PIN U2
    ↳ IOSTANDARD LVCMOS33 } [get_ports
    ↳ {act_display[0]}]
set_property -dict { PACKAGE_PIN U4
    ↳ IOSTANDARD LVCMOS33 } [get_ports
    ↳ {act_display[1]}]
set_property -dict { PACKAGE_PIN V4
    ↳ IOSTANDARD LVCMOS33 } [get_ports
    ↳ {act_display[2]}]
set_property -dict { PACKAGE_PIN W4
    ↳ IOSTANDARD LVCMOS33 } [get_ports
    ↳ {act_display[3]}]

## Configuration options, can be used for all
## designs
set_property CONFIG_VOLTAGE 3.3
    ↳ [current_design]
set_property CFGBVS VCCO [current_design]

## SPI configuration mode options for QSPI
## boot, can be used for all designs
set_property BITSTREAM.GENERAL.COMPRESS TRUE
    ↳ [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 33
    ↳ [current_design]
set_property CONFIG_MODE SPIx4
    ↳ [current_design]

```

IV-G. Funcionamiento de la ALU en la Basys 3

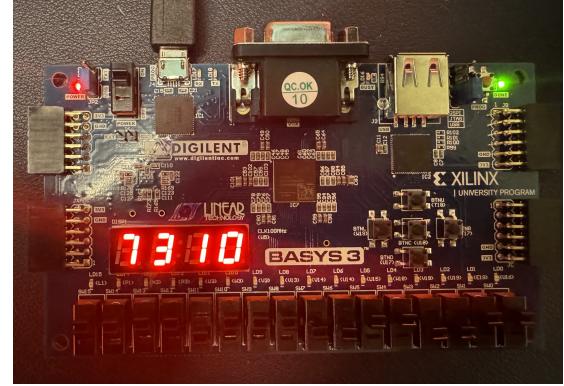


Figura 3: Operación de Suma entre a y b

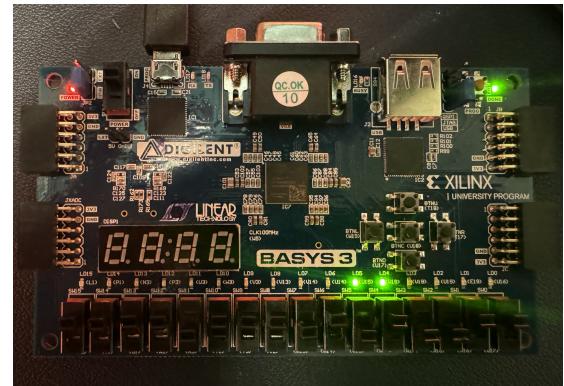


Figura 4: Operación and entre a y b

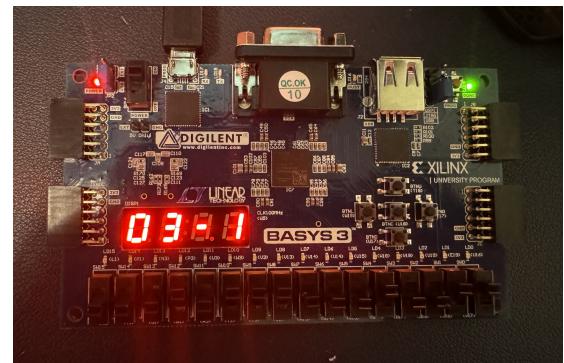


Figura 5: Decremento de a