

# Universidad Nacional de Córdoba



Facultad de Ciencias Exactas, Físicas y Naturales

---

## Cátedra de Arquitectura de Computadoras Trabajo Práctico 1: ALU

---

**Profesores:** - Martin Pereyra, Santiago Rodriguez

**Integrantes:**

Pallardó Agustín - apallardo@mi.unc.edu.ar

Trachta Agustín - agutrachta@mi.unc.edu.ar

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Objetivos</b>	<b>3</b>
2.1. Objetivo general . . . . .	3
2.2. Objetivos específicos . . . . .	3
<b>3. Marco teórico</b>	<b>4</b>
3.1. La ALU en el contexto de un procesador . . . . .	4
3.2. Representación de datos: complemento a dos y <code>signed/unsigned</code> . . . . .	4
3.3. Operaciones implementadas . . . . .	4
3.4. Corrimientos: aritmético vs. lógico . . . . .	4
3.5. Parametrización por ancho de datos . . . . .	5
3.6. Señales de estado (flags) y saturación (consideraciones) . . . . .	5
3.7. Multiplexado de display de 7 segmentos . . . . .	5
<b>4. Desarrollo</b>	<b>6</b>
4.1. Descripción general de la arquitectura . . . . .	6
4.2. ALU: consideraciones de diseño . . . . .	6
4.3. TOP: captura de entradas y ruteo de salidas . . . . .	7
4.4. Módulos de visualización . . . . .	7
<b>5. Simulación del <i>testbench</i></b>	<b>8</b>
5.1. Descripción del procedimiento . . . . .	8
5.2. Resultados . . . . .	8
<b>6. Síntesis y diagrama RTL detallado</b>	<b>10</b>
6.1. ALU . . . . .	10
6.2. Módulo de visualización ( <code>sevenseg_hex</code> ) . . . . .	10
6.3. Módulo <code>top</code> . . . . .	11
<b>7. Prueba en hardware real (FPGA Basys 3)</b>	<b>12</b>
7.1. Secuencia de pruebas . . . . .	12
7.2. Resultados . . . . .	12

# 1. Introducción

En este informe se documenta el desarrollo de una Unidad Aritmético-Lógica (ALU) implementada en una FPGA Basys 3. La ALU constituye un bloque fundamental en el diseño de procesadores y sistemas digitales, ya que permite realizar operaciones aritméticas, lógicas y de desplazamiento sobre datos binarios.

El objetivo principal de este trabajo práctico es adquirir experiencia en la descripción de hardware mediante Verilog, la validación con bancos de prueba, la simulación usando la herramienta Vivado, y la implementación física real.

Las herramientas utilizadas para el desarrollo fueron:

- **Lenguaje Verilog** para la descripción del hardware.
- **Vivado** como entorno de síntesis, simulación e implementación.
- **FPGA Basys 3** como plataforma de ejecución del diseño.

## **2. Objetivos**

### **2.1. Objetivo general**

Implementar y validar una Unidad Aritmético-Lógica (ALU) en una FPGA Basys 3.

### **2.2. Objetivos específicos**

- Implementar una ALU parametrizable en Verilog, de modo que el ancho del bus de datos pueda ajustarse según futuras necesidades.
- Validar la implementación mediante un banco de pruebas (*testbench*) que genere entradas asignadas y contemple un mecanismo de validación.
- Integrar la ALU con los periféricos de la Basys 3 (switches, botones, LEDs y display de 7 segmentos) para verificar el correcto funcionamiento en hardware real.

### 3. Marco teórico

#### 3.1. La ALU en el contexto de un procesador

La Unidad Aritmético-Lógica (ALU) es el bloque funcional encargado de ejecutar operaciones aritméticas, lógicas y de desplazamiento sobre operandos binarios. En un procesador, la ALU recibe los datos desde el banco de registros y ejecuta la operación seleccionada por la unidad de control. Su salida alimenta de vuelta el banco de registros y/o el camino de datos (bus) para su posterior uso. En arquitecturas *RISC*, la ALU suele implementar un conjunto mínimo pero completo de operaciones combinacionales de un ciclo, mientras que operaciones más complejas (multiplicación, división) pueden resolverse por hardware dedicado o microcódigo.

#### 3.2. Representación de datos: complemento a dos y signed/unsigned

En hardware digital, los enteros con signo suelen representarse en *complemento a dos*. Para un ancho de palabra  $N$ , el rango representable es:

$$[-2^{N-1}, 2^{N-1} - 1]$$

En Verilog, declarar un bus como `signed` hace que las operaciones aritméticas y de desplazamiento aritmético consideren el bit más significativo (MSB) como bit de signo. Para operaciones lógicas (`&`, `|`, `^`) el signo no afecta el resultado, pero sí es relevante en sumas/restas y corrimientos aritméticos.

#### 3.3. Operaciones implementadas

La ALU de este trabajo implementa las siguientes operaciones (las etiquetas son códigos de 6 bits que representan la operación seleccionada por `i_op`):

Código	Operandos	Descripción
100000	ADD	Suma de <code>i_data_a</code> e <code>i_data_b</code> (enteros con signo).
100010	SUB	Resta <code>i_data_a</code> - <code>i_data_b</code> (con signo).
100100	AND	AND bit a bit.
100101	OR	OR bit a bit.
100110	XOR	XOR bit a bit.
000011	SRA	Corrimiento a derecha <b>aritmético</b> : preserva el bit de signo.
000010	SRL	Corrimiento a derecha <b>lógico</b> : ingresa ceros por la izquierda.
100111	NOR	NOR bit a bit: $\sim(A \mid B)$ .

#### 3.4. Corrimientos: aritmético vs. lógico

- **SRA** (Arithmetic Right Shift) preserva el bit de signo: si el operando es negativo, se rellenan unos por la izquierda. Esto aproxima una división entre 2 para enteros

con signo (redondeo hacia  $-\infty$  en representación a dos).

- **SRL** (Logical Right Shift) inserta ceros por la izquierda, apropiado para datos sin signo o para manipulación de campos de bits.

Al parametrizar el ancho de dato a  $N$  bits, el corrimiento máximo significativo es  $N - 1$ . Es buena práctica enmascarar la magnitud de corrimiento con  $\log_2(N)$  bits para evitar comportamientos dependientes de la herramienta:

$$\text{shift\_amt} = \text{i\_data\_b}[\lceil \log_2(N) \rceil - 1 : 0]$$

### 3.5. Parametrización por ancho de datos

La ALU se parametriza mediante **NB\_DATA** (ancho de datos) y **NB\_OP** (ancho del código de operación). Esto permite reutilizar el diseño para 8, 16 o más bits cambiando sólo parámetros de síntesis, sin modificar el cuerpo del módulo.

### 3.6. Señales de estado (flags) y saturación (consideraciones)

En esta práctica no se implementan *flags* (Carry, Zero, Overflow, Negative) ni saturación; el resultado aritmético se trunca a **NB\_DATA** bits (comportamiento de *wrap-around*). En extensiones futuras se pueden agregar:

- **Zero**: `o_result == 0`
- **Negative**: `o_result[NB_DATA-1]`
- **Carry/Overflow**: a partir del bit extra en la suma/resta.
- **Saturación**: limitar el resultado al máximo/mínimo representable.

### 3.7. Multiplexado de display de 7 segmentos

La Basys 3 posee 4 dígitos de 7 segmentos con *ánodo común*, controlados por líneas **o\_an** (bajo activo) y segmentos **o\_seg** (bajo activo). Se multiplexan los dígitos activando uno por vez a alta velocidad. En este diseño:

- Se usa un contador de 16 bits (**div**) a 100 MHz.
- La selección `sel = div[15:14]` genera 4 estados, uno por dígito.
- La tasa de avance de estados es  $\frac{100 \text{ MHz}}{2^{14}} \approx 6103 \text{ Hz}$ ; el refresco por dígito es  $\frac{6103}{4} \approx 1526 \text{ Hz}$ , suficiente para evitar parpadeo.
- Se muestra el **resultado** de la ALU en hexadecimal (2 dígitos activos; los superiores en cero).

## 4. Desarrollo

### 4.1. Descripción general de la arquitectura

La arquitectura implementada se compone de cinco bloques principales:

1. **ALU**: bloque combinacional parametrizable que ejecuta las operaciones definidas por `i_op`.
2. **TOP**: bloque secuencial que registra operandos A/B y el código de operación a partir de los switches y botones de la Basys 3; además enruta resultados a LEDs y display.
3. **Hex\_to\_sseg**: decodifica un nibble hexadecimal (0–F) a segmentos (bajo activo).
4. **SevenSeg\_hex**: multiplexa cuatro dígitos, selecciona el nibble a mostrar y controla `o_an/o_seg`.
5. **Testbench**: banco de prueba autocontenido que aplica estímulos representativos y observa salidas.

### 4.2. ALU: consideraciones de diseño

**Interfaz y parámetros.** La ALU recibe dos operandos `signed` de `NB_DATA` bits y un código de operación de `NB_OP` bits. La salida `o_result` es `signed` y del mismo ancho que los operandos.

**Ruta de datos interna.** Se utiliza un registro interno `r_result` de `NB_DATA+1` bits para computar resultados de suma/resta con un bit extra (posible acarreo/overflow). Finalmente se asigna a `o_result` truncando a `NB_DATA` bits (sin saturación).

#### Operaciones de corrimiento.

- **SRA**: `i_data_a >>> i_data_b`. Al ser `i_data_a signed`, el operador `>>>` replica el bit de signo.
- **SRL**: se fuerza lógico enmascarando el corrimiento: `$unsigned(i_data_a)>> i_data_b[$clog2(NB_DATA)-1:0]`.

*Recomendación de robustez:* también enmascarar el corrimiento en SRA con `i_data_b[$clog2(NB_DATA)-1:0]` para hacer explícito el límite del ancho, evitando dependencias de implementación.

**Estilo de codificación.** El bloque es combinacional (`always @(*)`) con asignaciones bloqueantes para la lógica interna; las señales de salida se asignan por `assign`. Este estilo evita *latches* y facilita el mapeo a LUTs/sumadores del FPGA.

### 4.3. TOP: captura de entradas y ruteo de salidas

#### Captura de operandos y operación.

- **Reset:** síncrono con `i_clk`; limpia registros `data_a`, `data_b`, `op`.
- **Botones:** al detectar `i_btn_a`, `i_btn_b`, `i_btn_op` se capturan respectivamente `i_sw_data` (operandos) y `i_sw_data[NB_OP-1:0]` (operación).
- **Nota:** en hardware real, los botones *rebotan*. Este diseño funcional no incluye *debounce*; para robustez en placa se recomienda añadir sincronización doble y filtro (p.ej., contador temporal).

#### Visualización en LEDs y display.

- `o_led_now` refleja en tiempo real `i_sw_data` (útil para cargar operandos/códigos).
- `o_led_res` muestra el resultado de la ALU.
- **Display 7-seg:** `disp_val = {8'h00, alu_result[7:4], alu_result[3:0]}` muestra el resultado en dos dígitos hex; los dos dígitos altos quedan en cero. El punto decimal (`o_dp`) permanece apagado.

### 4.4. Módulos de visualización

`hex_to_sseg`. Tabla de verdad que mapea un nibble (0–F) a segmentos **bajo activo**. La Basys 3 utiliza ánodos comunes por dígito y cátodos compartidos para segmentos, por eso la codificación de 0 es `7'b1000000` (todos los segmentos activos menos el punto).

#### sevenseg\_hex.

- **Divisor:** contador de 16 bits a 100 MHz. Con `sel = div[15:14]` se recorre cada dígito a  $\sim 6.1$  kHz y cada dígito se refresca a  $\sim 1.5$  kHz (sin parpadeo).
- **Selección de nibble:** según `sel` se elige el nibble correspondiente de `i_value`.
- **Anodos:** `o_an` es bajo activo; se activa sólo el dígito seleccionado.

## 5. Simulación del *testbench*

Con el fin de validar el correcto funcionamiento de la ALU y su integración con el bloque `top`, se desarrolló un *testbench* (`tb_top_alu`) autocontenido. Dicho banco de pruebas aplica estímulos representativos sobre las entradas de la unidad y observa las señales de salida relevantes.

### 5.1. Descripción del procedimiento

El *testbench* genera un reloj de 100 MHz mediante el proceso `always #5 clk = ~clk` y define tareas (`tasks`) para automatizar la carga de operandos y operación:

- `load_A(v)`: coloca el valor `v` en los switches y pulsa `btn_a`.
- `load_B(v)`: coloca `v` en los switches y pulsa `btn_b`.
- `load_OP(v)`: coloca el código de operación en los switches (6 bits) y pulsa `btn_op`.

Se probaron las siguientes secuencias:

1. Reset inicial.
2.  $A = -5$  (0xFB),  $B = 3$ .
3. **ADD**  $\Rightarrow$  0xFE (-2).
4. **SUB**  $\Rightarrow$  0xF8 (-8).
5. Cargar  $B = 1$  para corrimientos:
  - **SRA**( $A, 1$ )  $\Rightarrow$  0xFD (-3).
  - **SRL**( $A, 1$ )  $\Rightarrow$  0x7D (125).
6. **NOR**( $A, 1$ )  $\Rightarrow$  0x04.

### 5.2. Resultados

La Figura 1 muestra la forma de onda obtenida. Puede observarse que las salidas `led_res` y `led_now` coinciden con los resultados esperados para cada operación.

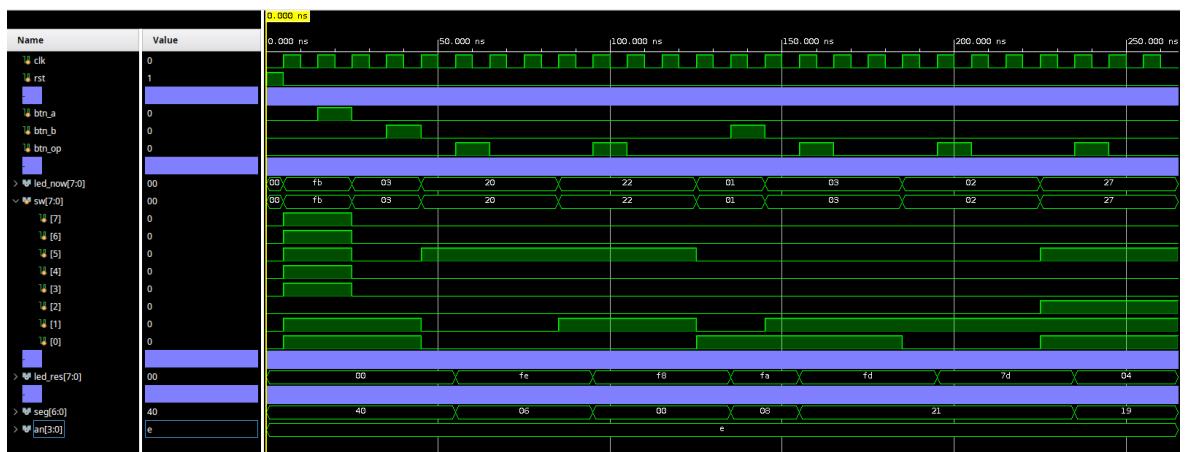


Figura 1: Forma de onda de la simulación del *testbench*.

## 6. Síntesis y diagrama RTL detallado

A continuación se presentan los diagramas RTL generados por Vivado a partir del código Verilog, que muestran la implementación estructural interna de los módulos principales del diseño.

### 6.1. ALU

En la Figura 2 se observa la implementación detallada de la Unidad Aritmético-Lógica. Cada operación aritmética/lógica se sintetiza en un bloque dedicado (ADD, SUB, AND, OR, XOR, corrimientos aritmético y lógico, INV). Sus salidas se combinan mediante un multiplexor (RTL\_MUX) gobernado por el código de operación *i\_op*.

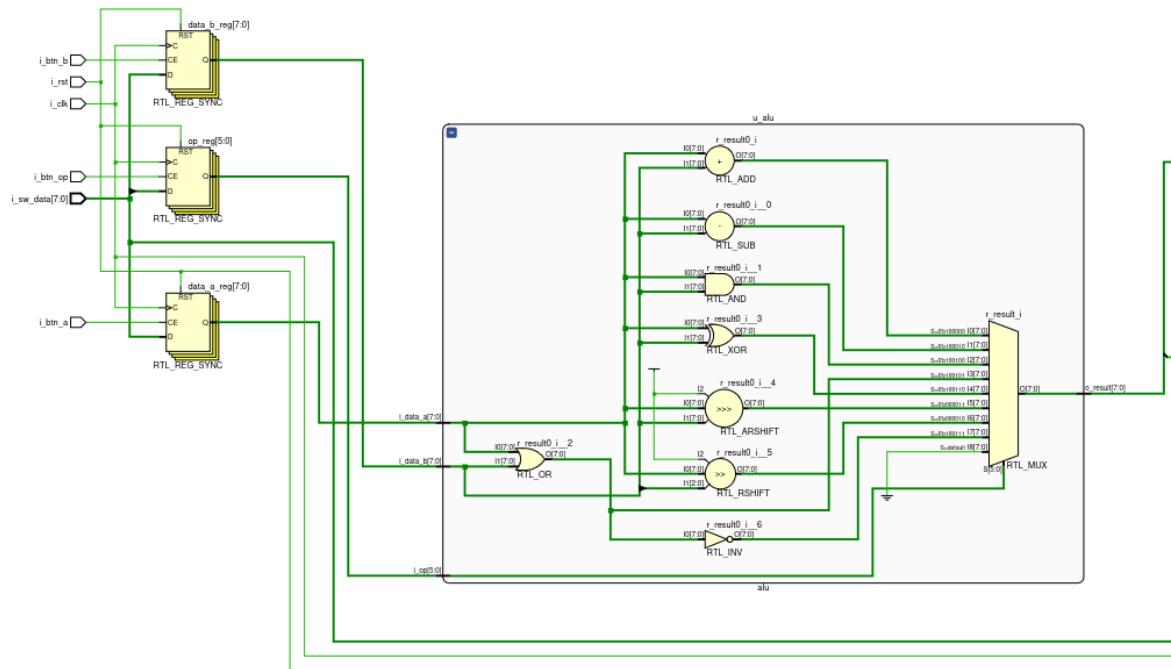


Figura 2: Diagrama RTL del módulo *alu*.

### 6.2. Módulo de visualización (sevenseg\_hex)

La Figura 3 muestra el módulo de visualización. El bloque incluye:

- Un divisor de frecuencia (*div\_reg*) que genera la señal de selección de dígito.
- Un multiplexor de nibbles (RTL\_MUX) para elegir cuál de los cuatro dígitos se presenta.
- Un decodificador hexadecimal a 7 segmentos (RTL\_ROM, *hex\_to\_sseg*).
- Un combinador (RTL\_BMERGE) para activar los ánodos (*o\_an*) de forma secuencial.

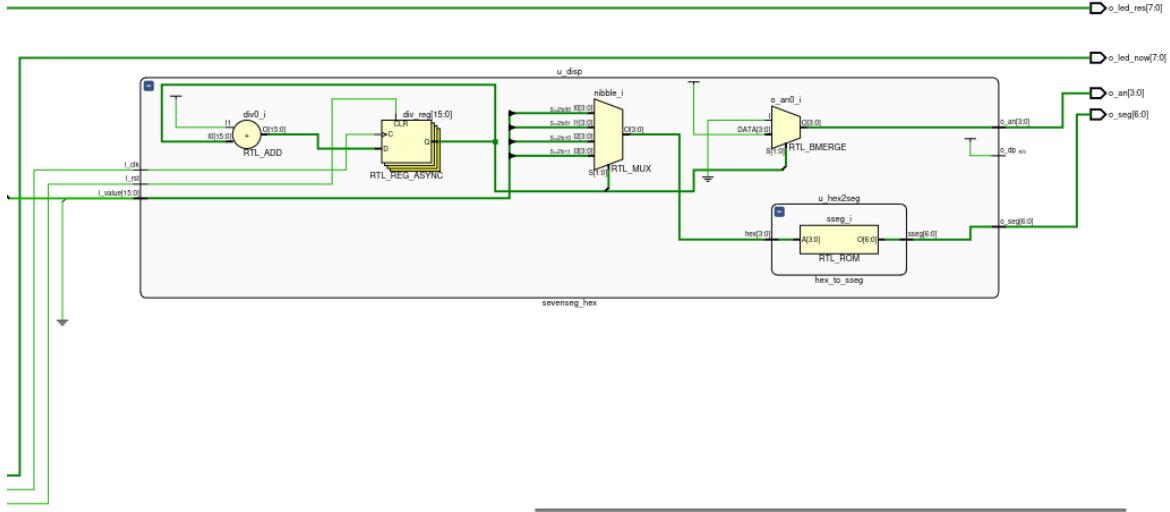


Figura 3: Diagrama RTL del módulo de visualización `sevenseg_hex`.

### 6.3. Módulo top

Finalmente, en la Figura 4 se muestra el diagrama RTL completo del bloque `top`, que integra los registros de entrada, la ALU y el módulo de display. Se observa cómo las señales de los botones y switches son capturadas en registros sincronizados, las operaciones se ejecutan en la ALU y el resultado se envía tanto a los LEDs (`o_led_res`) como al display de 7 segmentos.

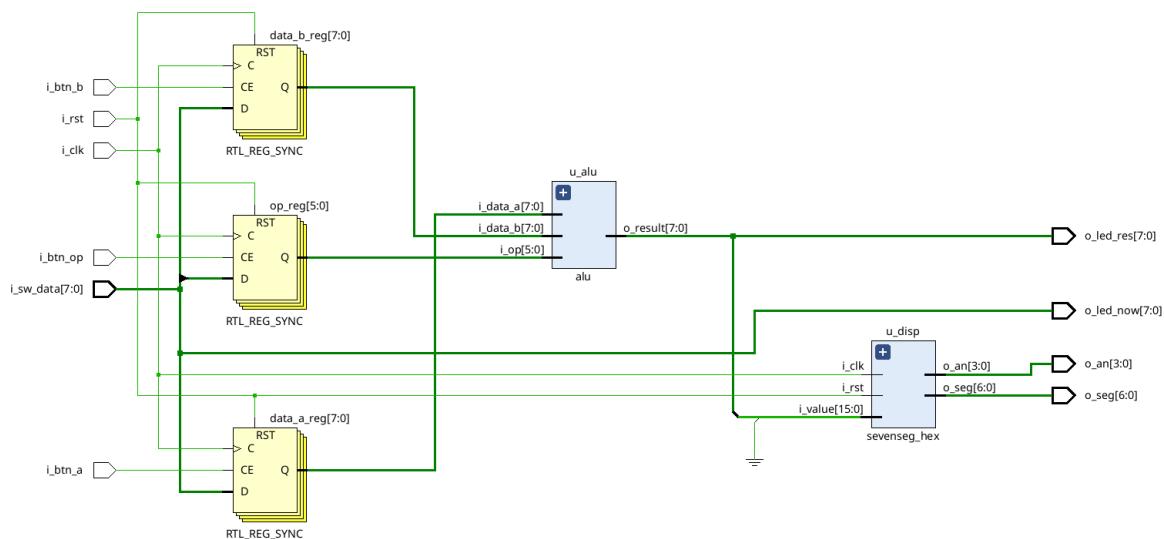


Figura 4: Diagrama RTL del bloque superior `top`.

## 7. Prueba en hardware real (FPGA Basys 3)

Finalmente, se validó la implementación sintetizada en la FPGA Basys 3. Se cargó el bitstream generado por Vivado y se ejecutaron pruebas controladas de operaciones aritméticas y lógicas, utilizando los switches como entradas y los botones para cargar operandos/operaciones. Los resultados se observaron en los LEDs y en el display de 7 segmentos integrado en la placa.

### 7.1. Secuencia de pruebas

1. **Reset inicial:** tras aplicar `rst`, la placa muestra el estado limpio en el display (Figura 5).
2. **Carga de  $A = 10000$ :** se coloca el valor en los switches y se pulsa `btn_a` (Figura 6).
3. **Suma con operandos cargados:** al cargar la operación de suma, se observa el resultado correcto en el display (Figuras 7 y 8).
4. **Operación SRA con  $A = 11$ :** se carga el operando y la operación de corrimiento aritmético a derecha. El resultado en el display coincide con lo esperado (Figura 9).
5. **Variación de B:** se prueba con diferentes valores de B (1, 0 y 100), confirmando el funcionamiento de la ALU en cada caso (Figuras 10, 11 y 12).

### 7.2. Resultados

En todos los casos, los resultados observados en LEDs y display coinciden con los valores esperados de acuerdo con la simulación previa, validando la correcta implementación del diseño.

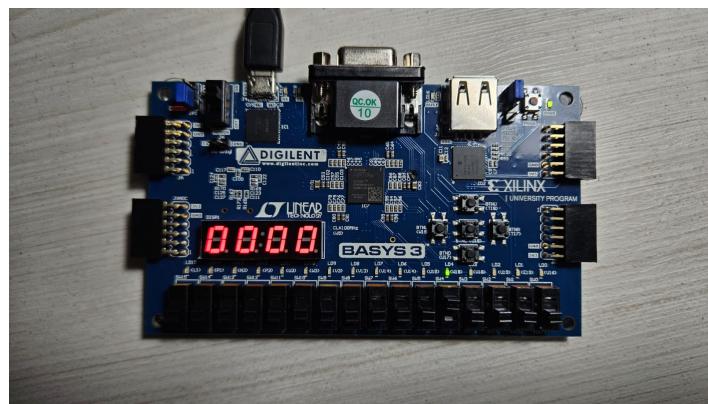


Figura 5: Carga de  $A = 10000$  después del `reset`.

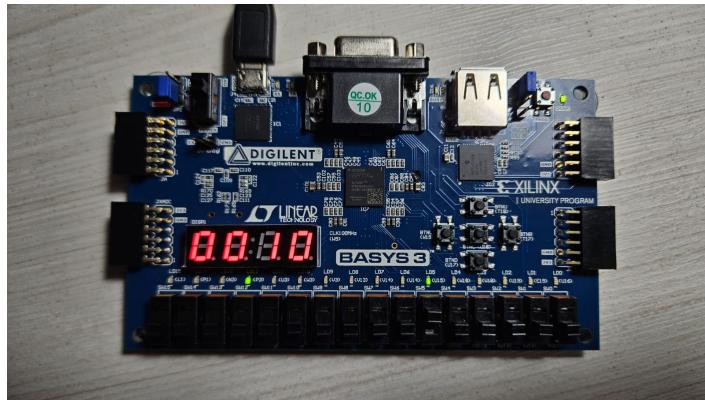


Figura 6: Carga de la suma después de cargar A.

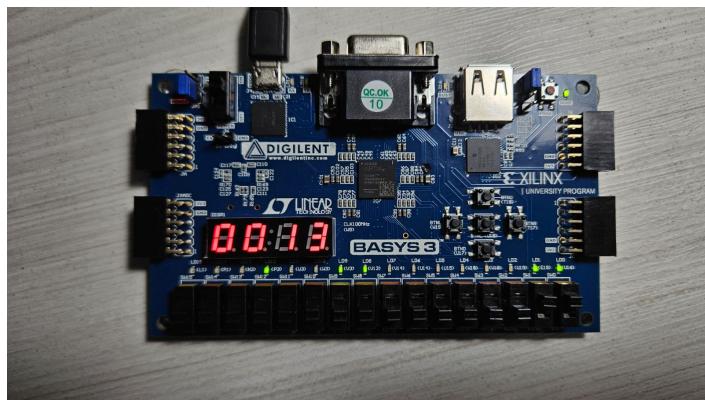


Figura 7: Carga de  $B = 11$ , se observa el resultado de la suma (10011).

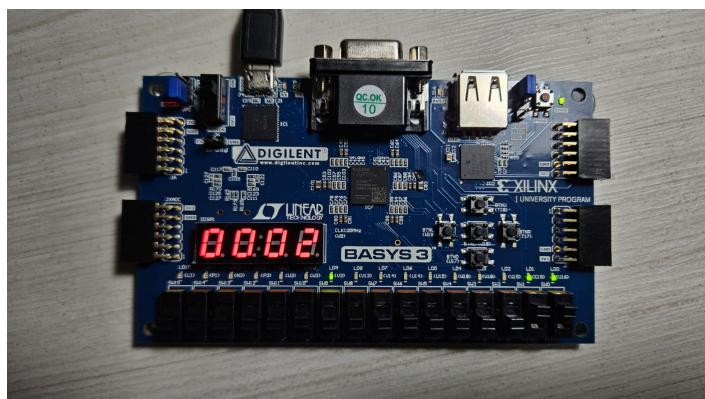


Figura 8: Carga de la operación SRA con  $A = 11$ .

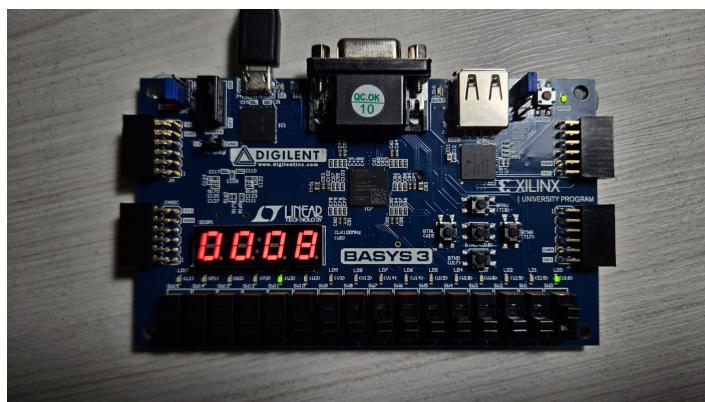


Figura 9: Prueba con  $B = 1$ .

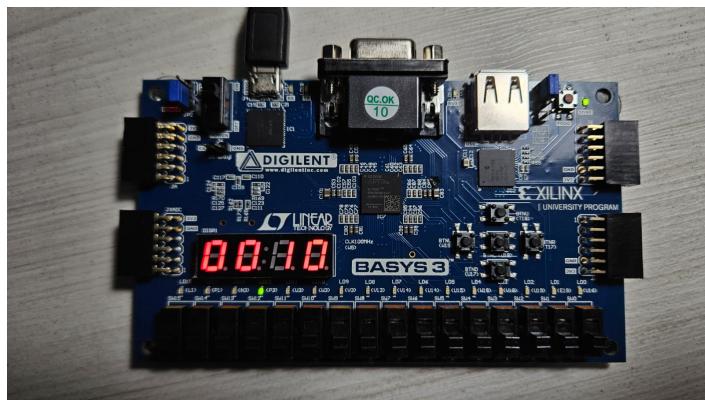


Figura 10: Prueba con  $B = 0$ .

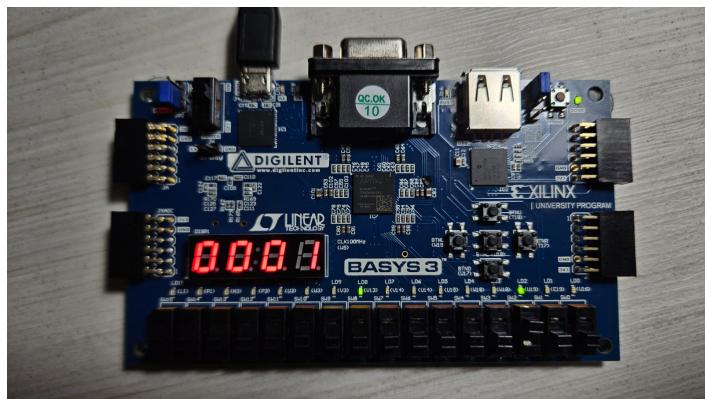


Figura 11: Prueba con  $B = 100$ .

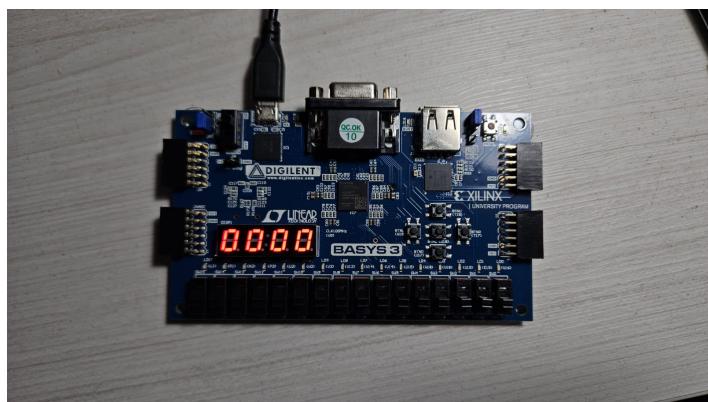


Figura 12: FPGA despues del reset