

Práctica de laboratorio: ALU

Jorge A. Arbach Lizio; Bianca A. Fraga

Facultad de Ciencias Exactas, Físicas y Naturales

Arquitectura de Computadoras

Septiembre, 2025

jorge.arbach@mi.unc.edu.ar

bianca.fraga@mi.unc.edu.ar

INTRODUCCIÓN

Este trabajo práctico tiene como objetivo el diseño, implementación y validación de una Unidad Aritmético-Lógica (ALU) parametrizable, capaz de ejecutar operaciones básicas sobre datos binarios de 8 bits. La ALU constituye un componente esencial en arquitecturas digitales, ya que permite realizar cálculos y decisiones lógicas dentro de procesadores, microcontroladores y sistemas embebidos.

El desarrollo se estructura en módulos independientes que encapsulan la captura de operandos, el código de operación y la lógica de procesamiento. Esta modularidad facilita la trazabilidad, la reutilización y la validación por bloques. El sistema incluye control por reloj, reset síncrono y señales de habilitación, lo que permite simular un flujo de datos secuencial y controlado.

La verificación se realiza mediante un banco de pruebas automatizado que genera combinaciones aleatorias de entradas, compara los resultados con un modelo de referencia y reporta cualquier discrepancia. Se incluyen operaciones como suma, resta, AND, OR, XOR, NOR y desplazamientos.

Este enfoque permite no solo validar el comportamiento funcional de la ALU, sino también justificar cada decisión técnica, desde la parametrización hasta el manejo de flags. El trabajo se orienta a una defensa reproducible, ética y trazable, alineada con buenas prácticas de diseño digital.

DESARROLLO

El diseño se abordó de forma modular, dividiendo el sistema en bloques funcionales independientes que interactúan entre sí mediante señales de control y datos. Esta división permite una mayor claridad, reutilización y trazabilidad del comportamiento del sistema. A continuación se detallan los módulos principales y su función dentro del sistema.

1. ENFOQUE MODULAR

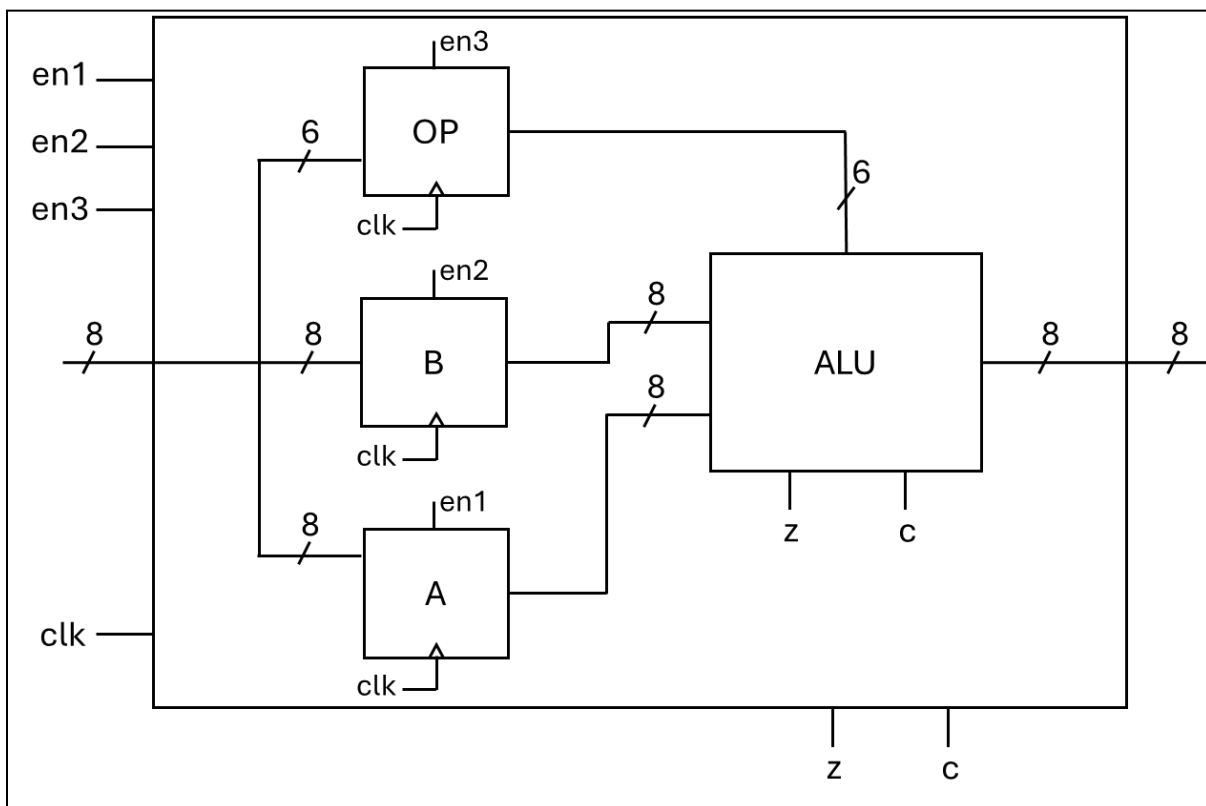


Figura 1

El diagrama anterior muestra claramente cómo se organiza el flujo de datos y control:

- Tres bloques de entrada (*A*, *B*, *OP*) reciben datos mediante una señal compartida (*data*) y se actualizan de forma independiente gracias a señales de habilitación (*en1*, *en2*, *en3*).
- Cada bloque está sincronizado por reloj (*clk*), lo que garantiza que las actualizaciones ocurran de forma ordenada y reproducible.

- Los datos capturados por **A** y **B** se envían a la ALU como operandos de 8 bits.
- El bloque **OP** envía un código de operación de 6 bits que indica qué cálculo debe ejecutar la ALU.
- La ALU procesa los datos y entrega tres salidas: el resultado (**8 bits**), el flag de cero (**z**) y el flag de acarreo (**c**).

1.1 Descripción de cada bloque

1.1.1 Bloques **A** y **B** - Captura de operandos

Estos módulos almacenan los operandos que serán utilizados por la ALU. Cada uno tiene:

- Entrada de reloj (**clk**) para sincronización.
- Señal de habilitación (**en1** para **A**, **en2** para **B**) que permite cargar el dato solo cuando se desea.
- Entrada de datos compartida (**data**), que se interpreta como el valor a almacenar.
- Salida de 8 bits que se conecta directamente a la ALU.

Permiten cargar los operandos de forma independiente, simulando el comportamiento de registros en un datapath real.

1.1.2 Bloque **OP** - Captura del código de operación

Este módulo almacena el código que indica qué operación debe ejecutar la ALU. Tiene:

- Entrada de reloj (**clk**) y habilitación (**en3**).
- Entrada de datos (**data**), de la cual se toman los 6 bits menos significativos.
- Salida de 6 bits (**opcode**) que se conecta a la ALU.

Su función es definir la operación a realizar (suma, resta, AND, OR, etc), permitiendo que el sistema sea flexible y controlado por software o estímulos externos,

1.1.3 Bloque **ALU** - Procesamiento de datos

La ALU recibe los operandos **A** y **B**, junto con el código de operación **OP**, y ejecuta la operación correspondiente. Sus salidas son:

- **result**: el valor calculado (8bits).
- **z**: flag que indica si el resultado es cero.
- **c**: flag de acarreo o préstamo, útil en operaciones aritméticas.

Su función es ejecutar la lógica central del sistema, procesando los datos según el código recibido y entregando el resultado junto con indicadores de estado.

1.2 Ventajas del enfoque modular

Podemos suponer que posee 4 ventajas principales:

- **Reutilización:** cada bloque puede ser usado en otros diseños sin modificaciones.
- **Trazabilidad:** es posible verificar cada parte por separado.
- **Escalabilidad:** se pueden agregar más operaciones o modificar el ancho de datos sin rediseñar todo el sistema.
- **Claridad estructural:** facilita la lectura, simulación y defensa técnica del diseño.
- **Parametrizable:** se puede variar el ancho de datos (N_{data}) y en el ancho del código de operación (N_{op}) para poder reutilizar la ALU para buses de 8, 16, 32 bits o más.

2. MÓDULOS

2.1 *data module*: Captura de datos sincronizada

Este módulo actúa como un registro controlado por reloj, reset y habilitación. Se utiliza para almacenar operandos (A, B) y el código de operación (***OpCode***) de forma independiente. Su estructura permite que cada dato se cargue en momentos distintos, simulando el comportamiento de un datapath real.

```
always @(posedge clk) begin
    if (rst)
        out <= {N{1'b0}};
    else if (enable)
        out <= in;
end
```

Figura 2

2.1.1 Aspectos vistos en clase:

- Uso de ***posedge clk*** para sincronización.
- Reset síncronico.
- Parametrización con N para reutilización del módulo.

2.2 alu: Unidad Aritmético-Lógica

Como dijimos anteriormente, la ALU recibe dos operandos y un código de operación, y ejecuta la operación correspondiente. Se implementaron operaciones aritméticas (*ADD*, *SUB*), lógicas (*AND*, *OR*, *XOR*, *NOR*) y desplazamientos (*SRL*, *SRA*). También se calculan dos flags: *carry*, *zero*.

```
case (OpCode)
  ADD: {carry, S} = A + B;           // suma
  SUB: {carry, S} = A - B;           // resta (no es un carry sino un borrow)
  AND_: S = A & B;                   // and
  OR_: S = A | B;                    // or
  XOR_: S = A ^ B;                   // xor
  SRA: S = $signed(A) >>> B[SHIFT_BITS-1:0]; // shift der aritmético
  SRL: S = A >> B[SHIFT_BITS-1:0];   // shift der logico
  NOR: S = ~(A | B);                 // nor
  default: S = 8'b00000000;          // por las dudas
endcase
```

Figura 3

2.2.1 Aspectos técnicos destacados:

- Uso de *{carry, S}* para capturar el bit extra en suma/resta.
- Desplazamiento aritmético con *\$signed* y *>>>*, que preserva el signo,
- Cálculo dinámico de bits de desplazamiento con *\$clog2(N_data)*.

2.2.2 Complemento a dos:

La ALU interpreta los datos como signed, lo que permite representar negativos en complemento a dos. Por ejemplo, $13 - 91 = -78$, que se representa como 234 en binario sin signo. Esta interpretación es coherente con el uso de registros *logic signed* y asegura compatibilidad con desplazamientos y comparaciones.

2.3 Módulo top module

Este módulo integra todo el sistema. Recibe una única señal de entrada (*data*) y señales de habilitación para cargar los operandos y el código de operación en momentos distintos. Esto simula el comportamiento de un datapath secuencial.

```
// Instanciamos los 2 modulos de los operandos
data_module #(.N(N_data)) data_a (
    .clk(clk),
    .rst(rst),
    .enable(enable_a),
    .in(data),
    .out(a)
);

data_module #(.N(N_data)) data_b (
    .clk(clk),
    .rst(rst),
    .enable(enable_b),
    .in(data),
    .out(b)
);

// Instanciamos el modulo de operacion
data_module #(.N(N_op)) data_op (
    .clk(clk),
    .rst(rst),
    .enable(enable_op),
    .in(data[N_op-1:0]), // sólo 6 bits
    .out(opcode)
);

// Instanciamos la ALU
alu alu (
    .A(a),
    .B(b),
    .OpCode(opcode),
    .S(result),
    .carry(carry),
    .zero(zero)
);

endmodule
```

Figura 4

2.3.1 Funciones:

- Coordina la carga secuencial de datos.
- Encapsula la lógica de control.
- Facilita la simulación y el testeo.

3. TESTBENCH

Una parte clave del desarrollo fue la implementación de un *testbench automatizado*, que permite verificar el comportamiento de la ALU de forma repetitiva, ordenada y trazable. Para lograr esto, se utilizó una estructura de control tipo *for*, que recorre múltiples combinaciones de entradas y operaciones.

```
for (i=0; i<100; i=i+1) begin
    // generar aleatorios proceduralmente
    A = $random;
    B = $random;
    Op = valid_ops[$urandom_range(0, 8)];

    // Cargar A
    @(posedge clk);
    data = A; enable_a = 1;
    @(posedge clk);
    enable_a = 0;

    // Cargar B
    @(posedge clk);
    data = B; enable_b = 1;
    @(posedge clk);
    enable_b = 0;

    // Cargar Op (se usan los 6 LSB de data en top_module)
    @(posedge clk);
    data = Op; enable_op = 1;
    @(posedge clk);
    enable_op = 0;

    // Darle un ciclo para que la ALU procese
    @(posedge clk);
    check_result();
end
```

Figura 5

Este bucle ejecuta 100 pruebas consecutivas, generando operandos aleatorios (*A*, *B*) y seleccionando una operación válida o inválida (*Op*). Cada conjunto de datos se carga en el sistema mediante señales de habilitación (*enable_a*, *enable_b*, *enable_op*), simulando el comportamiento secuencial de un datapath real.

Además, se utilizó una **tarea** (*task*) llamada *check_result*, que encapsula la lógica de verificación. Esta tarea compara el resultado entregado por la ALU (*result*, *carry*, *zero*) con el valor esperado calculado dentro del testbench. Si hay alguna discrepancia, se muestra un mensaje detallado y se detiene la simulación con *\$fatal*.

```
task check_result;
begin
    expected      = 0;
    expected_carry = 0;

    case (Op)
        6'b100000: {expected_carry, expected} = A + B;      // ADD
        6'b100010: {expected_carry, expected} = A - B;      // SUB
        6'b100100: expected = A & B;                        // AND
        6'b100101: expected = A | B;                        // OR
        6'b100110: expected = A ^ B;                        // XOR
        6'b000011: expected = $signed(A) >>> B[$clog2(N_data)-1:0]; // SRA
        6'b000010: expected = A >> B[$clog2(N_data)-1:0];   // SRL
        6'b100111: expected = ~(A | B);                    // NOR
        default:    expected = 0;
    endcase

    expected_zero = (expected == 0);

    // Verificación
    if (result != expected || carry != expected_carry || zero != expected_zero) begin
        $display("ERROR en i=%0d | Op=%b A=%h B=%h => result=%h(c=%b,z=%b) esperado=%h(c=%b,z=%b)",
            i, Op, A, B, result, carry, zero, expected, expected_carry, expected_zero);
        $fatal; // termina la simulación en el primer error
    end else begin
        $display("OK: Op=%b A=%h B=%h => result=%h (carry=%b zero=%b)",
            Op, A, B, result, carry, zero);
    end
end
endtask
```

Figura 6

3.1 Aspectos destacados:

- Automatiza la validación sin intervención manual.
- Simula el comportamiento secuencial por ciclos de reloj.
- Refuerza trazabilidad y reproducibilidad.
- Detecta errores de forma inmediata y precisa.

4. RESULTADOS

Durante la simulación se verificaron múltiples combinaciones de entradas. A continuación se presentan evidencias visuales que respaldan el funcionamiento correcto del sistema.

4.1 Resultados de simulación en forma de onda

La imagen muestra la evolución temporal de las señales *clk*, *reset*, *enable*, *dataA*, *dataB* y múltiples salidas (*result*, *result2*, ..., *result8*). Se puede observar cómo los datos se cargan en distintos ciclos y cómo las salidas cambian en función de las operaciones ejecutadas. Se observa cómo las señales *enable* y *reset* afectan directamente la carga de datos en los registros. Por ejemplo, cuando *enable* está en alto y *clk* hace flanco positivo, el valor de *dataA* se actualiza. Esto confirma que el control secuencial está funcionando como se diseñó.

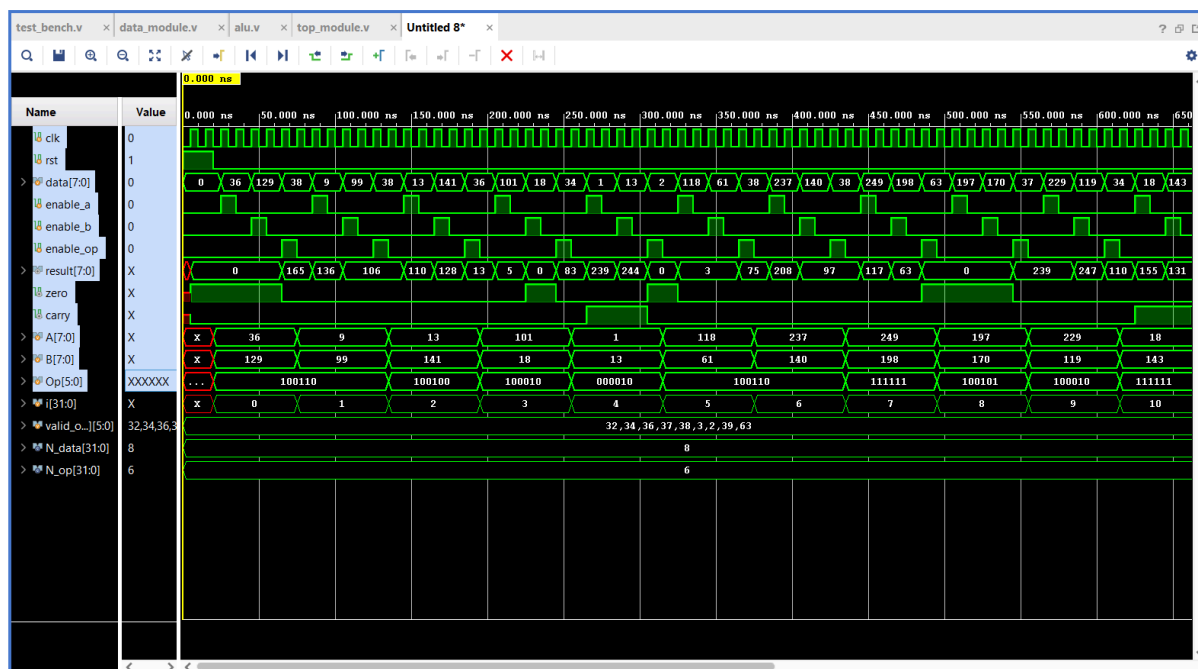


Figura 7

4.2 Verificación manual de resultados

La siguiente tabla muestra cómo se cargan los operandos y el código de operación en distintos ciclos, y cómo se obtiene el resultado correspondiente. Se incluyen operaciones como *XOR*, *AND* y *SUB*, con sus respectivos operandos y salidas.

Este chequeo manual permite validar paso a paso que el sistema interpreta correctamente las señales de habilitación y ejecuta la operación indicada.

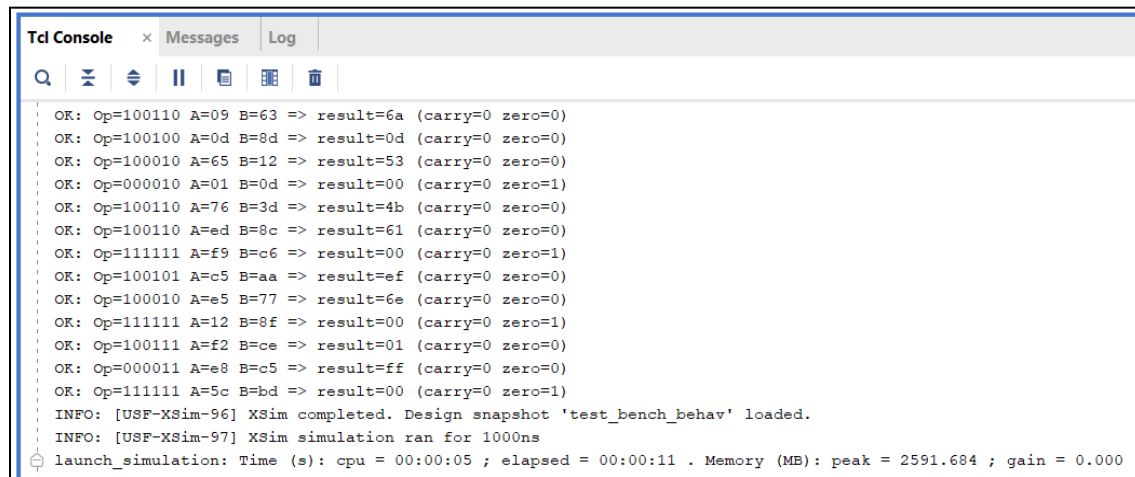
rst	data	en_A	en_B	en_OP	dato_A	dato_B	dato_OP	resul
1	0	0	0	0	0	0	0	0
0	36	1	0	0	36	0	0	0
0	129	0	1	0	36	129	0	0
0	XOR	0	0	1	36	129	XOR	165
0	9	1	0	0	9	129	XOR	136
0	99	0	1	0	9	99	XOR	106
0	XOR	0	0	1	9	99	XOR	106
0	13	1	0	0	13	99	XOR	110
0	141	0	1	0	13	141	XOR	128
0	AND	0	0	1	13	141	AND	13
0	101	1	0	0	101	141	AND	5
0	18	0	1	0	101	18	AND	0
0	SUB	0	0	1	101	18	SUB	83
0	1	1	0	0	1	18	SUB	239 (-17)
0	13	0	1	0	1	13	SUB	244 (-12)
0	SRL	0	0	1	1	13	SRL	0

Figura 8

4.3 Chequeo automático por consola

La imagen muestra la salida del testbench en la consola Tcl, donde se reportan operaciones ejecutadas, resultados y flags. Por ejemplo, se observa que $0001 + 0001 = 0002$ con $carry = 0$, lo que confirma que el modelo de referencia coincide con la salida de la ALU.

Este chequeo automático refuerza la trazabilidad del sistema y permite detectar errores sin intervención manual.



```

Tcl Console x Messages Log
[Icons: Search, Undo, Redo, Run, Stop, Refresh, Close]

OK: Op=100110 A=09 B=63 => result=6a (carry=0 zero=0)
OK: Op=100100 A=0d B=8d => result=0d (carry=0 zero=0)
OK: Op=100010 A=65 B=12 => result=53 (carry=0 zero=0)
OK: Op=000010 A=01 B=0d => result=00 (carry=0 zero=1)
OK: Op=100110 A=76 B=3d => result=4b (carry=0 zero=0)
OK: Op=100110 A=ed B=8c => result=61 (carry=0 zero=0)
OK: Op=111111 A=f9 B=c6 => result=00 (carry=0 zero=1)
OK: Op=100101 A=c5 B=aa => result=ef (carry=0 zero=0)
OK: Op=100010 A=e5 B=77 => result=6e (carry=0 zero=0)
OK: Op=111111 A=12 B=8f => result=00 (carry=0 zero=1)
OK: Op=100111 A=f2 B=ce => result=01 (carry=0 zero=0)
OK: Op=000011 A=e8 B=c5 => result=ff (carry=0 zero=0)
OK: Op=111111 A=5c B=bd => result=00 (carry=0 zero=1)

INFO: [USF-XSim-96] XSim completed. Design snapshot 'test_bench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:05 ; elapsed = 00:00:11 . Memory (MB): peak = 2591.684 ; gain = 0.000
  
```

Figura 9

CONCLUSIÓN

Este trabajo nos permitió entender mejor cómo funciona una ALU y cómo se puede diseñar de forma modular y controlada. A lo largo del desarrollo, se pudo implementar operaciones como suma, resta, AND, OR, XOR, NOR y desplazamientos, y comprobar que el sistema responde correctamente a cada una. También se aprendió a cómo representar números negativos usando complemento a dos, algo que al principio parecía confuso pero terminó siendo clave para que la ALU funcione bien con valores signed.

El uso de testbenches con datos aleatorios y comparación contra un modelo de referencia ayudó a validar el comportamiento del circuito y detectar errores rápidamente. Además, el diseño con módulos separados para los operandos y el código de operación hizo que todo sea más ordenado y fácil de probar.

En resumen, fue una experiencia útil para aplicar lo que venimos viendo en teoría, y nos dejó una base sólida para seguir trabajando con sistemas digitales más complejos en el futuro.