TRABAJO PRACTICO DE LABORATORIO #1

ALU

Integrantes:

- Medran, Constanza
- Berra, Facundo

Facultad de Ciencias Exactas, Físicas y Naturales
Arquitectura de Computadoras

INTRODUCCIÓN:

Este trabajo práctico tiene como objetivo el diseño, implementación y validación de una Unidad Aritmético-Lógica (ALU) parametrizable capaz de ejecutar operaciones sobre datos binarios de N bits en una FPGA Basys 3 (Artix-7, 100 MHz). La ALU es un bloque esencial en arquitecturas digitales, permitiendo cálculos y decisiones lógicas en procesadores y sistemas embebidos.

El desarrollo se estructura en módulos independientes que encapsulan captura de operandos, captura de opcode y lógica de procesamiento. Esta modularidad facilita trazabilidad, reutilización y validación por bloques. La verificación se realiza con un testbench automatizado que genera combinaciones aleatorias, compara contra un modelo de referencia y reporta discrepancias. Se incluyen operaciones ADD, SUB, AND, OR, XOR, NOR y desplazamientos SRL y SRA con manejo de flags de zero y carry/borrow.

DESARROLLO:

El sistema se organiza en tres bloques de entrada (A, B, OP) que reciben datos desde un bus compartido data y se actualizan mediante señales de habilitación independientes. Un reloj común coordina las cargas para asegurar comportamiento reproducible.

- A y B entregan operandos de N bits a la ALU.
- OP entrega un código de 6 bits que selecciona la operación.
- La ALU produce Result[N-1:0], Zero y Carry.

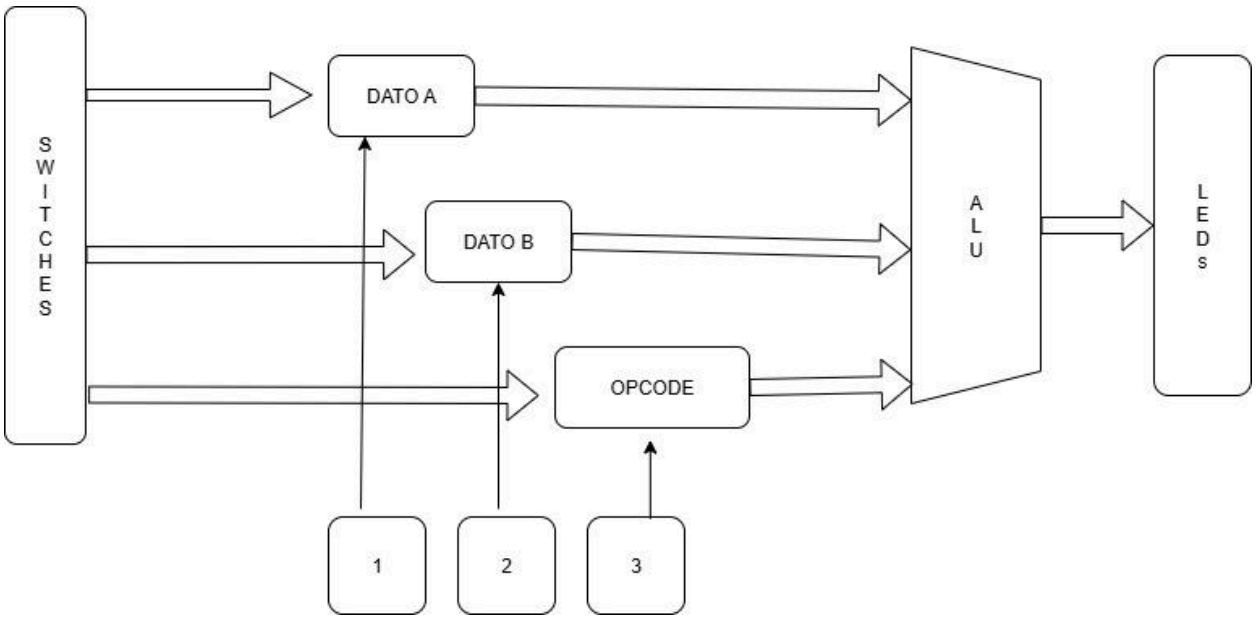
Ventajas principales del enfoque modular:

- Reutilización de bloques en otros diseños.
- Trazabilidad y pruebas por componente.
- Escalabilidad: cambio de N u operaciones sin rediseñar todo.
- Claridad estructural y defensa técnica.

Las operaciones de la ALU son las siguientes:

Operación	Opcode (6 bits)	Descripción
ADD	100000	Suma A + B. Carry indica acarreo
SUB	100010	Resta A − B. Carry indica borrow
AND	100100	AND bit a bit
OR	100101	OR bit a bit
XOR	100110	XOR bit a bit
NOR	100111	NOT (A OR B)
SRL	000010	Shift lógico der: A >> B[SHIFT_BITS-1:0]
SRA	000011	Shift aritmético der: $signed(A) >>> B[SHIFT_BITS-1:0]

A continuación se presenta el diagrama que guió nuestra implementación. Este esquema ilustra la arquitectura funcional de la ALU, mostrando cómo los switches y pulsadores de la placa proveen los operandos y códigos de operación que son procesados por la ALU, cuyo resultado se visualiza finalmente en los LEDs de la placa Basys 3.



Módulos:

Implementamos dos módulos principales:

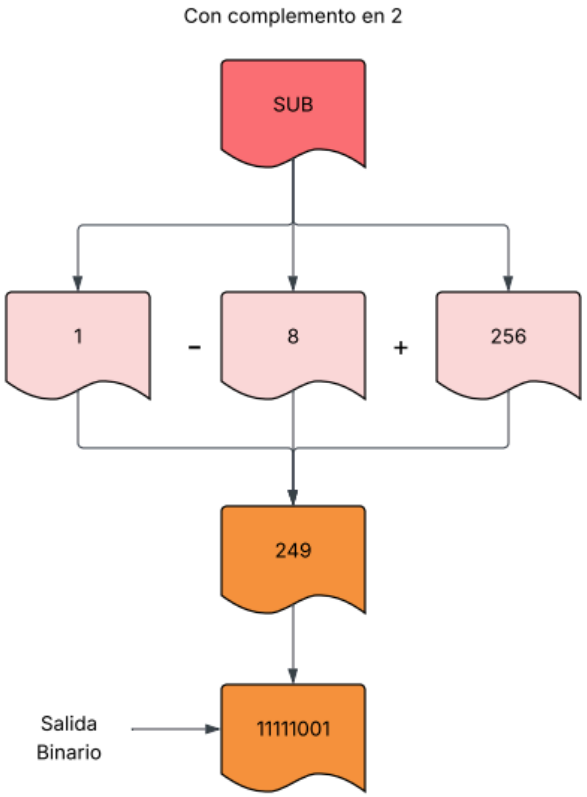
- ***alu***: contiene la implementación funcional de la ALU.
- ***top_module***: instancia la ALU y gestiona los componentes físicos de la placa (switches, pulsadores, LEDs).

Módulo ALU

Dentro del módulo ALU, se expresan los bloques A, B y Op, guardando los valores tanto en el bloque A como en el bloque B y ejecutando la operación seleccionada con respecto a la combinación de 6 bits seleccionada.

Dentro de este bloque se Implementan las operaciones dadas en la consigna (ADD, SUB, AND, OR, XOR, NOR, SRL y SRA). Como así también se establece el Cero y el Carry. Aclarando que el cero se va a mostrar cuando no se encuentre cargada alguna operación o cuando el resultado de la misma sea cero, y el carry en el caso de que haya algún préstamo o algún overflow.

Para la suma y resta se utiliza un acumulador temporal de N+1 bits para extraer el bit de acarreo o préstamo. Pero en el caso de diferencias negativas, se utiliza el complemento a 2, que lo explicamos en el siguiente gráfico (denotando como se ve la operación en decimal, y la salida en binario):



En el caso de los desplazamientos utilizamos:

- SHIFT_BITS = $clog2(N) para limitar la magnitud del shift.
- SRL: $unsigned(A) >> B[SHIFT_BITS-1:0].
- SRA: $signed(A) >>> B[SHIFT_BITS-1:0] para preservar el signo.

Para comprender bien la operatoria de los desplazamientos hay que entender que en el caso de este trabajo práctico, el número máximo está determinado por la variable “SHIFT_BITS” logrando un código más escalable, pero en nuestro caso como N es 8, entonces:

$$Log_2(N) = Log_2(8) = 3$$

Restandole 1 a “SHIFT_BITS” tanto en el SRL como en el SRA obtenemos como máximo 8 desplazamientos. Y debido a que estamos trabajando con números de 8 bits y al momento de excedernos de ese desplazamiento entonces, estaríamos desplazando más de la cantidad posible de la presentación del número mostrándose todos ceros o todos unos desde el L0 al L7.

Es necesario entender igualmente la diferencia entre el SRA y SRL puesto que ambos “shifteos” son iguales, pero el problema sucede cuando se encuentra un número negativo, el cual, en el caso del Shift Right Arithmetic debido a la preservación de signos, no sólo desplaza la cantidad de movimientos en el bloque B, sino que completa con cantidad de “1” hacia la izquierda del desplazamiento.

```
case (Op)
    6'b100000: begin  // ADD
        temp_result = A + B;
        Result = temp_result[N-1:0];
        Carry = temp_result[N];  // Bit de carry
    end

    6'b100010: begin  // SUB
        temp_result = A - B;
        Result = temp_result[N-1:0];
        Carry = temp_result[N];  // Borrow (préstamo)
    end

    6'b100100: Result = A & B;                    // AND
    6'b100101: Result = A | B;                    // OR
    6'b100110: Result = A ^ B;                    // XOR
    6'b100111: Result = ~(A | B);                 // NOR
    6'b000010: Result = A >> B[SHIFT_BITS-1:0];   //SRL
    6'b000011: Result = $signed(A) >>> B[SHIFT_BITS-1:0];    // SRA
    default:   Result = {N{1'b0}};               //por las dudas
endcase
```

top_module

A diferencia del módulo alu, que se enfoca puramente en la lógica computacional, el top_module maneja la interfaz entre el mundo digital lógico y los componentes físicos, como switches, botones y LEDs. Esto lo convierte en un puente esencial para la implementación práctica, permitiendo la carga de datos, ejecución de operaciones y visualización de resultados.

Funcionamiento detallado:

- **Sincronización y Detección de Flancos**: El módulo sincroniza los botones (pulsadores) con el reloj de 100 MHz de la placa para evitar rebotes o señales espurias. Utiliza detección de flancos de subida (rising edge) en señales como BTN_A, BTN_B,

BTN_OP y BTN_RESET. Esto significa que solo se activan las cargas cuando se detecta un pulso limpio, lo que mejora la estabilidad y reproducibilidad. Por ejemplo:

- BTN_RESET inicializa todos los registros en cero, reiniciando el estado del sistema.
- BTN_A carga el operando A desde los switches (SW[7:0]).
- BTN_B carga el operando B de manera similar.
- BTN_OP carga el opcode de 6 bits para seleccionar la operación.

- **Instanciación de la ALU**: Dentro del top_module, se instancia la ALU con parámetro N=8 (para operandos de 8 bits). Los operandos A y B, junto con el opcode OP, se pasan directamente a la ALU. La salida de la ALU (Result[7:0], Zero y Carry) se mapea a los LEDs:
  - LED[7:0] muestra el resultado binario.
  - LED_ZERO enciende si el resultado es cero o no hay operación cargada.
  - LED_CARRY indica acarreo (carry) en sumas, préstamo (borrow) en restas u overflow.

- **Gestión de Entradas y Salidas**: Los switches proporcionan un bus compartido para datos (data), y las señales de habilitación (enable) controlan qué bloque (A, B o OP) se actualiza. Esto permite una operación secuencial: primero cargas A, luego B, luego OP, y el resultado se actualiza en tiempo real. El diseño es síncrono, coordinado por el reloj común (CLK), lo que evita problemas de timing en la FPGA.

```
// Cargar valores cuando se presionan los botones
always @(posedge CLK) begin
    if (btn_reset_pulse) begin
        // RESET: limpiar todos los registros
        A  <= 8'd0;
        B  <= 8'd0;
        Op <= 6'd0;
    end else begin
        if (btn_a_pulse)  A  <= SW[7:0];    // SW0-SW7 → A
        if (btn_b_pulse)  B  <= SW[7:0];    // SW0-SW7 → B
        if (btn_op_pulse) Op <= SW[5:0];    // SW0-SW5 → Op
    end
end
```

```
// Instancia de la ALU
alu #(.N(8)) alu_inst (
    .A(A),
    .B(B),
    .Op(Op),
    .Result(Result),
    .Zero(Zero),
    .Carry(Carry)
);

// Mostrar resultado en LEDs
assign LED = Result;
assign LED_ZERO = Zero;      // LED se enciende cuando resultado = 0
assign LED_CARRY = Carry;    // LED se enciende cuando hay carry/overflow
```

Este módulo hace que la ALU sea operable en un entorno real, manejando aspectos prácticos como anti-rebote de botones y visualización, lo que es crucial para validar el diseño más allá de la simulación.

Constraints

Dentro de este archivo, es donde se definen las conexiones físicas de la placa y se le asignan las señales lógicas que definimos en el código de Verilog. Es decir, donde se mapean las señales del diseño a los pines específicos de la placa FPGA y se establecen las propiedades eléctricas de dichos pines, como por ejemplo, el estándar de voltaje.

```
# ---------- Clock 100MHz ----------
set_property PACKAGE_PIN W5 [get_ports CLK]
set_property IOSTANDARD LVCMOS33 [get_ports CLK]
create_clock -period 10.000 -name sys_clk_pin -waveform {0.000 5.000} -add [get_ports CLK]

# ---------- Switches (0-15) ----------
set_property PACKAGE_PIN V17 [get_ports {SW[0]}]
set_property PACKAGE_PIN V16 [get_ports {SW[1]}]
set_property PACKAGE_PIN W16 [get_ports {SW[2]}]
set_property PACKAGE_PIN W17 [get_ports {SW[3]}]
set_property PACKAGE_PIN W15 [get_ports {SW[4]}]
set_property PACKAGE_PIN V15 [get_ports {SW[5]}]
set_property PACKAGE_PIN W14 [get_ports {SW[6]}]
set_property PACKAGE_PIN W13 [get_ports {SW[7]}]
set_property PACKAGE_PIN V2  [get_ports {SW[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[*]}]
```

```
# ---------- LEDs (0-7) para resultado ----------
set_property PACKAGE_PIN U16 [get_ports {LED[0]}]
set_property PACKAGE_PIN E19 [get_ports {LED[1]}]
set_property PACKAGE_PIN U19 [get_ports {LED[2]}]
set_property PACKAGE_PIN V19 [get_ports {LED[3]}]
set_property PACKAGE_PIN W18 [get_ports {LED[4]}]
set_property PACKAGE_PIN U15 [get_ports {LED[5]}]
set_property PACKAGE_PIN U14 [get_ports {LED[6]}]
set_property PACKAGE_PIN V14 [get_ports {LED[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[*]}]

# ---------- LEDs para flags (L1 y P1 son LED14 y LED15) ----------
set_property PACKAGE_PIN L1  [get_ports LED_ZERO]     ;# LED14 - Flag Zero
set_property PACKAGE_PIN P1  [get_ports LED_CARRY]    ;# LED15 - Flag Carry
set_property IOSTANDARD LVCMOS33 [get_ports {LED_ZERO LED_CARRY}]

# ---------- Botones ----------
set_property PACKAGE_PIN W19 [get_ports BTN_A]        ;# Left button
set_property PACKAGE_PIN T17 [get_ports BTN_B]        ;# Right button
set_property PACKAGE_PIN T18 [get_ports BTN_OP]       ;# Up button
set_property PACKAGE_PIN U18 [get_ports BTN_RESET]    ;# Center button (RESET)
set_property IOSTANDARD LVCMOS33 [get_ports {BTN_A BTN_B BTN_OP BTN_RESET}]
```

Testbench

En esta parte es donde verificaremos nuestro diseño, en Verilog dentro de Vivado. Porque imitamos un laboratorio físico para probar el circuito y generamos las señales de entrada, para así poder evaluar las salidas del mismo.

El testbench instancia la ALU parametrizable con N=8 y genera estímulos dirigidos y aleatorios para validar cada operación. En la fase dirigida, fija combinaciones específicas de A, B y Op para ADD, SUB, AND, OR, XOR, NOR, SRL y SRA, y compara el resultado de la ALU con un valor esperado calculado en el mismo banco de pruebas. La comparación se encapsula en la tarea check_result, que espera un ciclo de simulación, contrasta Result con el esperado y registra “OK” o “ERROR” acumulando un contador de fallos. En la fase aleatoria, produce operandos y operaciones al azar, limita la magnitud de desplazamiento con B[2:0] y vuelve a calcular el resultado de referencia por software para cada caso. Al finalizar, informa la cantidad total de errores. De este modo, la verificación es automática y reproducible, cubre casos límite y reduce la intervención manual al mínimo.

Dentro del código, realizamos una función que nos indique por consola, si la operación fue exitosa, al comparar el resultado esperado con el resultado obtenido. Simplemente con leer las salidas de la consola, sabremos si nuestras operaciones se comportan de la forma deseada.

Salida de la consola Tcl:

```
=== Iniciando Test de ALU ===
OK: A= 10, B=  5, Op=100000 | Result= 15
OK: A= 10, B=  5, Op=100010 | Result=  5
OK: A=240, B=170, Op=100100 | Result=160
OK: A=240, B=170, Op=100101 | Result=250
OK: A=240, B=170, Op=100110 | Result= 90
OK: A=240, B=170, Op=100111 | Result=  5
OK: A=240, B=  2, Op=000010 | Result= 60
OK: A=240, B=  2, Op=000011 | Result=252

=== Pruebas Aleatorias ===
OK: A= 36, B=249, Op=100010 | Result= 43
OK: A= 99, B=  5, Op=100111 | Result=152
OK: A=101, B=250, Op=100010 | Result=107
OK: A= 13, B=  6, Op=100111 | Result=240
OK: A=237, B=  4, Op=100010 | Result=233
OK: A=198, B=253, Op=100100 | Result=196
OK: A=229, B=255, Op=100100 | Result=229
OK: A=143, B=  2, Op=000010 | Result= 35
OK: A=232, B=253, Op=100110 | Result= 21
OK: A=189, B=253, Op=100111 | Result=  2
OK: A= 99, B=  2, Op=100000 | Result=101
OK: A= 32, B=  2, Op=100111 | Result=221

=== Test Finalizado ===
Total de errores:           0
*** TODOS LOS TESTS PASARON ***
$finish called at time : 308 ns : File "C:/Users/conim/TP1_ALU/TP1_ALU.srcs/sim_1/new/tb.v" Line 99
INFO: [USF-XSim-96] XSim completed. Design snapshot 'testbench_alu_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:06 ; elapsed = 00:00:07 . Memory (MB): peak = 2306.582 ; gain = 0.000
```

Conclusión

Este trabajo práctico demuestra exitosamente el diseño de una ALU parametrizable en Verilog para la FPGA Basys 3, destacando la importancia de la modularidad en arquitecturas digitales. A través de la implementación de operaciones aritméticas (ADD, SUB) y lógicas (AND, OR, XOR, NOR, SRL, SRA), junto con el manejo de flags (Zero y Carry), se logra un sistema robusto y escalable que soporta operandos de N bits (en este caso, 8). El enfoque modular facilita la reutilización, trazabilidad y depuración, mientras que el testbench automatizado —con pruebas dirigidas y aleatorias— asegura una verificación exhaustiva, reportando cero errores en la consola Tcl, lo que confirma la corrección funcional.