

**Centro de Investigación y de Estudios
Avanzados del IPN, Unidad Guadalajara**



DISEÑO ELECTRÓNICO

ALU VECTORIAL

Presentan:

DANIEL ALDAHIR GARCÍA GARCÍA

IVÁN ALONZO BAYLÓN NAVA

JESÚS EDUARDO ESPARZA SOTO

Supervisión:

MIGUEL ANGEL RIVERA COSTA

MARTÍN GONZÁLEZ PÉREZ

Contenido

RTL.....	3
Módulo alu	3
Módulo vector_alu	3
Testbench	5
interfaz	6
Testscases.....	7
Covergroups.....	9
Asserts	12
Conclusiones	14

RTL

En la primera fase se realizó el diseño del circuito que generará la ALU vectorial. Esta ALU cuenta con 4 operaciones aritméticas y 2 lógicas: Suma, resta, multiplicación, división, bitwise AND y bitwise OR.

El diseño consiste de dos módulos, el Módulo alu que se encarga de definir las operaciones y el selector de éstas.

Módulo alu

El módulo se define mediante un parámetro que determina la longitud (numero de bits) de las entradas A y B que representan los números de entrada a operar y de Z que representa la salida o resultado de la operación. Se puede notar que Z tiene el doble de bits de las entradas, esto asegura que no haya errores porque falten bits para las operaciones, la multiplicación, por ejemplo, en la que si la salida fuera de la misma longitud causaría errores por ser un número de mayor extensión.

El comportamiento del circuito esta dado por un proceso always, que como es de esperarse siempre se está ejecutando y que por medio de un bloque case se encarga de preguntar el estado del selector de 3 bits, que representan las 6 operaciones, de acuerdo con el valor de Sel se define la operación a realizar.

```
module alu #(
    parameter WIDTH = 8
) (
    input logic [WIDTH-1:0] a, // Entradas
    input logic [WIDTH-1:0] b, // Entradas
    input logic [2:0] sel,      // Selección de operación
    output logic [WIDTH*2-1:0] z // Salida de la operación
);
    always_comb begin
        case (sel)
            3'b000: z = a + b;          // Suma
            3'b001: z = a - b;          // Resta
            3'b010: z = a & b;          // AND
            3'b011: z = a | b;          // OR
            3'b100: z = a * b;          // Multiplicación
            3'b101: z = (b != 0) ? a / b : '0; // División protegida contra división por cero
            default: z = '0;            // Salida por defecto
        endcase
    end
endmodule
```

Ilustración 1 Módulo alu

Módulo vector_alu

Por otro lado el módulo vector_alu es una unidad que contiene **N ALUs** en paralelo, cada una de las cuales realiza operaciones entre dos vectores de datos de entrada A y B, y tiene una selección de operación independiente para cada ALU.

Este módulo está definido por dos parámetros, al igual que el anterior tiene un parámetro **WIDTH** que define el tamaño de los valores de entrada y de salida, además cuenta con un parámetro **N** que determina el número de ALU que ejecutarán las operaciones.

Cuenta con las entradas:

- **clk**: La señal de reloj que sincroniza la operación de las ALUs.
- **arst**: Señal de reinicio asíncrono (si se activa, reinicia las salidas de todas las ALUs).
- **A**: Un arreglo de N entradas, cada una con **WIDTH** bits. Son los operandos de entrada para las ALUs.
- **B**: Un arreglo de N entradas, cada una con **WIDTH** bits. Son los operandos de entrada para las ALUs.
- **sel**: Un arreglo de N entradas de 3 bits, cada una seleccionando qué operación realizar en cada
- **ALU.enable**: Un arreglo de N bits, donde cada bit habilita o deshabilita la salida de cada ALU.

La única salida es:

Z: Un arreglo de N salidas, cada una de **WIDTH*2** bits. Representa los resultados de las ALUs, registradas y condicionadas a la habilitación.

Su funcionamiento está dado por un bloque generate que crea N instancias del módulo ALU. Cada ALU se conecta con los correspondientes elementos de los arreglos **A[i]**, **B[i]** y **sel[i]**. La salida de cada ALU (**alu_out[i]**) es registrada de manera sincrónica con el reloj **clk** o se pone a cero si se activa la señal de reset **arst**. Si el bit de habilitación **enable[i]** es 1, la salida **Z[i]** toma el valor de la salida de la ALU; de lo contrario, se mantiene en cero.

```
module vector_alu #(
    parameter N = 4, // Cantidad de ALUs internas
    parameter WIDTH = 8 // Ancho de las entradas y salidas
) (
    input logic clk,
    input logic arst, // Reset asíncrono
    input logic [WIDTH-1:0] A [N-1:0], // Entradas A
    input logic [WIDTH-1:0] B [N-1:0], // Entradas B
    input logic [2:0] sel [N-1:0], // Selección de operación (por ALU)
    input logic [N-1:0] enable, // Bus de habilitación
    output logic [WIDTH*2-1:0] Z [N-1:0] // Salida registrada de las ALUs
);

// Señales intermedias para las salidas combinacionales de las ALUs
logic [WIDTH*2-1:0] alu_out [N-1:0];

// Generar N instancias de la ALU
generate
    for (i = 0; i < N; i++) begin : ALU_INSTANCES
        // Instancia de una ALU interna
        alu #(WIDTH(WIDTH)) alu_inst (
            .a(A[i]),
            .b(B[i]),
            .sel(sel[i]),
            .c(alu_out[i])
        );

        // Salida registrada con habilitación
        always_ff @(posedge clk or posedge arst) begin
            if (arst)

```

Ilustración 2 Módulo vector_alu pt. 1

```

        Z[i] <= '0'; // Reset de la salida
    else if (enable[i])
        Z[i] = alu_out[i]; // Lógica combinacional
        //Z[i] <= alu_out[i]; // Actualizar salida
    else
        Z[i] <= '0'; // Salida en zeros si está deshabilitada
    end
end
endgenerate
endmodule

```

Ilustración 3 Módulo vector_alu pt. 2

A continuación, se muestra una ilustración del RTL generado:

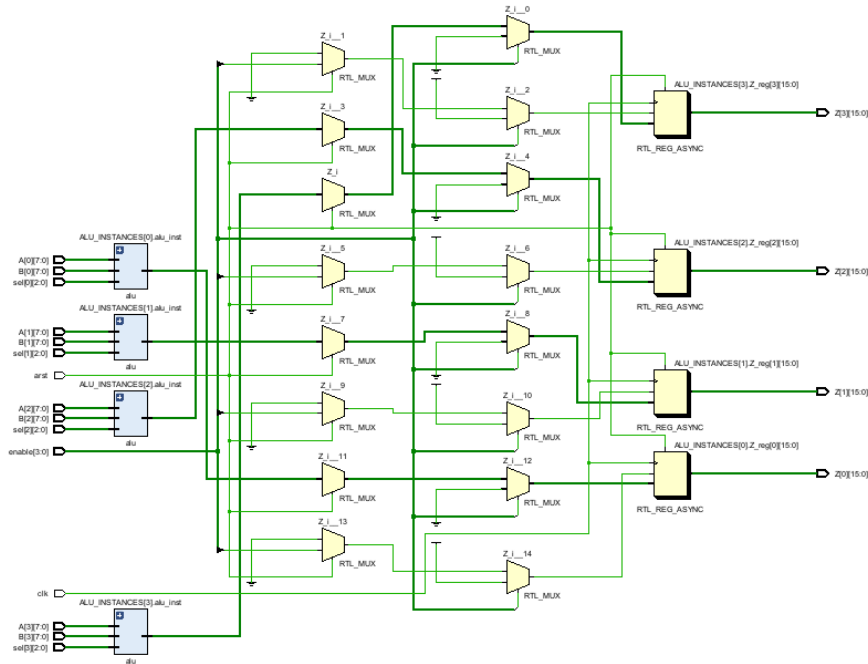


Ilustración 4 RTL de la ALU Vectorial

Testbench

Para poder probar el funcionamiento de la ALU vectorial se realizó un testbench que además de corroborar el funcionamiento del selector y de la ejecución de cada una de las operaciones, también tenía como objetivo corroborar que no hubiera errores por el número de bits o por los signos negativos.

La manera de hacer las pruebas de este DUT fue a través de una Interface, que es la que define los estímulos que tendrá cada entrada, A, B, Sel y enable, generando así valores aleatorios o fijos para cada una de estas.

interfaz

Como se puede observar la interfaz tiene un logic asociado a las entradas y salidas del DUT, A, B, etc. Dentro de la interfaz se desarrollaron 12 tasks que dan valores a A, B y Sel y enable.

La función que tiene cada uno de esos task se define a continuación:

1. generate_random_stimuli:

Genera estímulos aleatorios para las señales A, B, sel y enable, asegurando que los valores estén dentro de sus respectivos rangos.

2. add_a_b_random:

Genera valores aleatorios para A y B y configura sel para realizar una operación de suma (sel[i] = 3'b000).

3. sub_a_b_random:

Genera valores aleatorios para A y B y configura sel para realizar una operación de resta (sel[i] = 3'b001).

4. and_a_b_random:

Genera valores aleatorios para A y B y configura sel para realizar una operación AND (sel[i] = 3'b010).

5. or_a_b_random:

Genera valores aleatorios para A y B y configura sel para realizar una operación OR (sel[i] = 3'b011).

6. mul_a_b_random:

Genera valores aleatorios para A y B y configura sel para realizar una operación de multiplicación (sel[i] = 3'b100).

7. div_a_b_random:

Genera valores aleatorios para A y B y configura sel para realizar una operación de división (sel[i] = 3'b101).

8. add_a_b_zero:

Asigna valores cero a A y B y configura sel para realizar una operación de suma (sel[i] = 3'b000).

9. sub_a_b_zero:

Asigna valores cero a A y B y configura sel para realizar una operación de resta (sel[i] = 3'b001).

10. mul_a_b_zero:

Asigna valores cero a A y B y configura sel para realizar una operación de multiplicación (sel[i] = 3'b100).

11. a_zero_b_op_random:

Asigna valores cero a A, genera valores aleatorios para B y sel para operaciones aleatorias.

12. b_zero_a_op_random:

Asigna valores cero a B, genera valores aleatorios para A y sel para operaciones aleatorias.

```
interface alu_if #(parameter N = 4, WIDTH = 8);

    logic clk;
    logic arst;
    logic [WIDTH-1:0] A [N-1:0];
    logic [WIDTH-1:0] B [N-1:0];
    logic [2:0] sel [N-1:0];
    logic [N-1:0] enable;
    logic [WIDTH*2-1:0] Z [N-1:0];

    task automatic generate_random_stimuli();...
    task automatic add_a_b_random();...
    task automatic sub_a_b_random();...
    task automatic and_a_b_random();...
    task automatic or_a_b_random();...
    task automatic mul_a_b_random();...
    task automatic div_a_b_random();...

    task automatic add_a_b_zero();...

    // Task para operación de resta con A y B en cero
    task automatic sub_a_b_zero();...

    // Task para operación de multiplicación con A y B en cero
    task automatic mul_a_b_zero();...

    // Task para operación con A en cero y B y opcode aleatorios
    task automatic a_zero_b_op_random();...

    // Task para operación con B en cero y A y opcode aleatorios
    task automatic b_zero_a_op_random();...
endinterface
```

Ilustración 5 Interfaz de la ALU vectorial

La elección de realizar estas pruebas, tanto con estímulos aleatorios como con valores cero, tiene como objetivo cubrir una variedad de escenarios que permiten evaluar el comportamiento y la eficacia del diseño bajo diferentes condiciones:

Pruebas con valores aleatorios: Se busca garantizar que el módulo funcione correctamente para un rango amplio de entradas posibles. Al randomizar las entradas, aseguramos que el sistema puede manejar cualquier combinación de datos sin errores, lo que simula condiciones más realistas en un entorno de funcionamiento real. Además, al realizar estas pruebas para las señales de entrada como A, B, sel y enable, se verifica si el diseño es flexible y adecuado para manejar una gama diversa de operaciones.

Pruebas con valores cero: Verifican el comportamiento del diseño en casos extremos o límites. Evalúa cómo el sistema maneja situaciones en las que las entradas no contienen valores significativos y así comprobar la correcta implementación de ciertas operaciones como la multiplicación por cero o la división por cero. Al incluir estos casos, también aseguramos que el sistema no genere resultados inesperados o errores, como división por cero.

Pruebas con configuraciones específicas: Configurar las operaciones específicas en diferentes momentos (como suma, resta, multiplicación, etc.) asegura que el sistema pueda realizar correctamente cada tipo de operación sin interferencia entre ellas. Estas pruebas son importantes para validar la correcta implementación de cada operación individualmente, verificando que los valores de las señales de control (sel) seleccionen las operaciones correctas.

Testscases

Esta serie de tasks se llaman en un módulo tb_vector_alu por medio de una instancia a la interfaz y un if que se llama a elección del tester y que ejecuta determinado número de veces las pruebas y por lo tanto las tasks deseadas, se muestran los TESTS del testbench:

```
`ifdef TEST1
  `CREATE_COVERGROUP_BASIC(cg_inputs)
  initial begin
    cg_inputs cg_inputs_inst = new();
    // Reset
    alu_if_inst.arst = 1;
    #1ns;
    alu_if_inst.arst = 0;

    repeat (2000) begin
      @(posedge alu_if_inst.clk);
      alu_if_inst.generate_random_stimuli();
      cg_inputs_inst.sample();
    end
  end
`endif
```

Ilustración 6 TEST1

```

initial begin
    cg_add_inputs cg_add_inputs_i = new();
    cg_sub_inputs cg_sub_inputs_i = new();
    cg_and_inputs cg_and_inputs_i = new();
    cg_or_inputs cg_or_inputs_i = new();
    cg_mul_inputs cg_mul_inputs_i = new();
    cg_div_inputs cg_div_inputs_i = new();

    alu_if_inst.arst = 1;
    #ins;
    alu_if_inst.arst = 0;

    repeat (2000) begin
        @(posedge alu_if_inst.clk);
        alu_if_inst.add_a_b_random();
        cg_add_inputs_i.sample();
    end
    repeat (2000) begin
        @(posedge alu_if_inst.clk);
        alu_if_inst.sub_a_b_random();
        cg_sub_inputs_i.sample();
    end
    repeat (2000) begin
        @(posedge alu_if_inst.clk);
        alu_if_inst.and_a_b_random();
        cg_and_inputs_i.sample();
    end
    repeat (2000) begin
        @(posedge alu_if_inst.clk);
        alu_if_inst.or_a_b_random();
        cg_or_inputs_i.sample();
    end
    repeat (2000) begin
        @(posedge alu_if_inst.clk);
        alu_if_inst.mul_a_b_random();
    end
end

`elsif TEST3
    initial begin
        ////////////////////////////////////////////////////
        // 1. Generar 20 sumas con zeros //
        // 2. Generar 20 restas con zeros //
        // 3. Generar 20 multiplicaciones con zeros //
        ////////////////////////////////////////////////////
        repeat (20) begin
            @(posedge alu_if_inst.clk);
            alu_if_inst.add_a_b_zero();
        end
        repeat (20) begin
            @(posedge alu_if_inst.clk);
            alu_if_inst.sub_a_b_zero();
        end
        repeat (20) begin
            @(posedge alu_if_inst.clk);
            alu_if_inst.mul_a_b_zero();
        end
    end
end

```

Ilustración 7 TEST2

Ilustración 8 TEST2 pt. 1

```

        cg_mul_inputs_i.sample();
    end
    repeat (2000) begin
        @(posedge alu_if_inst.clk);
        alu_if_inst.div_a_b_random();
        cg_div_inputs_i.sample();
    end
end

```

Ilustración 9 TEST2 pt. 2

```

`elsif TEST4
    // Prueba 4 -
    ////////////////////////////////////////////////////
    // 1. Generar operaciones con a = 0 //
    // 2. Generar operaciones con b = 0 //
    ////////////////////////////////////////////////////
    `CREATE_COVERGROUP_WITH_SEL(cg_a_zero_b_op_random)
    `CREATE_COVERGROUP_WITH_SEL(cg_b_zero_a_op_random)

    initial begin
        cg_a_zero_b_op_random cg_a_zero_b_op_random_i = new();
        cg_b_zero_a_op_random cg_b_zero_a_op_random_i = new();

        repeat (2000) begin
            @(posedge alu_if_inst.clk);
            alu_if_inst.a_zero_b_op_random();
            cg_a_zero_b_op_random_i.sample();
        end
        repeat (2000) begin
            @(posedge alu_if_inst.clk);
            alu_if_inst.b_zero_a_op_random();
            cg_b_zero_a_op_random_i.sample();
        end
    end
end
`endif

```

Ilustración 10 TEST4

Como se puede observar estos test llaman a algunas de las tasks que se definieron previamente en la interfaz múltiples veces para así generar estímulos con valores que cubran todo el rango posible.

Covergroups

```
`define CREATE_COVERGROUP_BASIC(name) \  
    covergroup name; \  
        cp_a0 : coverpoint alu_if_inst.A[0] { option.auto_bin_max = bins_nums; } \  
        cp_a1 : coverpoint alu_if_inst.A[1] { option.auto_bin_max = bins_nums; } \  
        cp_a2 : coverpoint alu_if_inst.A[2] { option.auto_bin_max = bins_nums; } \  
        cp_a3 : coverpoint alu_if_inst.A[3] { option.auto_bin_max = bins_nums; } \  
        cp_b0 : coverpoint alu_if_inst.B[0] { option.auto_bin_max = bins_nums; } \  
        cp_b1 : coverpoint alu_if_inst.B[1] { option.auto_bin_max = bins_nums; } \  
        cp_b2 : coverpoint alu_if_inst.B[2] { option.auto_bin_max = bins_nums; } \  
        cp_b3 : coverpoint alu_if_inst.B[3] { option.auto_bin_max = bins_nums; } \  
    endgroup  
  
`define CREATE_COVERGROUP_WITH_SEL(name) \  
    covergroup name; \  
        cp_a0 : coverpoint alu_if_inst.A[0] { option.auto_bin_max = bins_nums; } \  
        cp_a1 : coverpoint alu_if_inst.A[1] { option.auto_bin_max = bins_nums; } \  
        cp_a2 : coverpoint alu_if_inst.A[2] { option.auto_bin_max = bins_nums; } \  
        cp_a3 : coverpoint alu_if_inst.A[3] { option.auto_bin_max = bins_nums; } \  
        cp_b0 : coverpoint alu_if_inst.B[0] { option.auto_bin_max = bins_nums; } \  
        cp_b1 : coverpoint alu_if_inst.B[1] { option.auto_bin_max = bins_nums; } \  
        cp_b2 : coverpoint alu_if_inst.B[2] { option.auto_bin_max = bins_nums; } \  
        cp_b3 : coverpoint alu_if_inst.B[3] { option.auto_bin_max = bins_nums; } \  
        cp_sel0 : coverpoint alu_if_inst.sel[0] { option.auto_bin_max = bins_nums; } \  
        cp_sel1 : coverpoint alu_if_inst.sel[1] { option.auto_bin_max = bins_nums; } \  
        cp_sel2 : coverpoint alu_if_inst.sel[2] { option.auto_bin_max = bins_nums; } \  
        cp_sel3 : coverpoint alu_if_inst.sel[3] { option.auto_bin_max = bins_nums; } \  
    endgroup
```

Para organizar el listado de los valores que tomaron las entradas del DUT se formaron macros para los covergroups, en las que se generaron coverpoints para cada una de las instancias del DUT:

Como se puede observar se tiene un valor máximo para los bins, el cuál se definió dentro el módulo siendo igual a 256, por el número de valores que pueden tomar las

entradas del DUT.

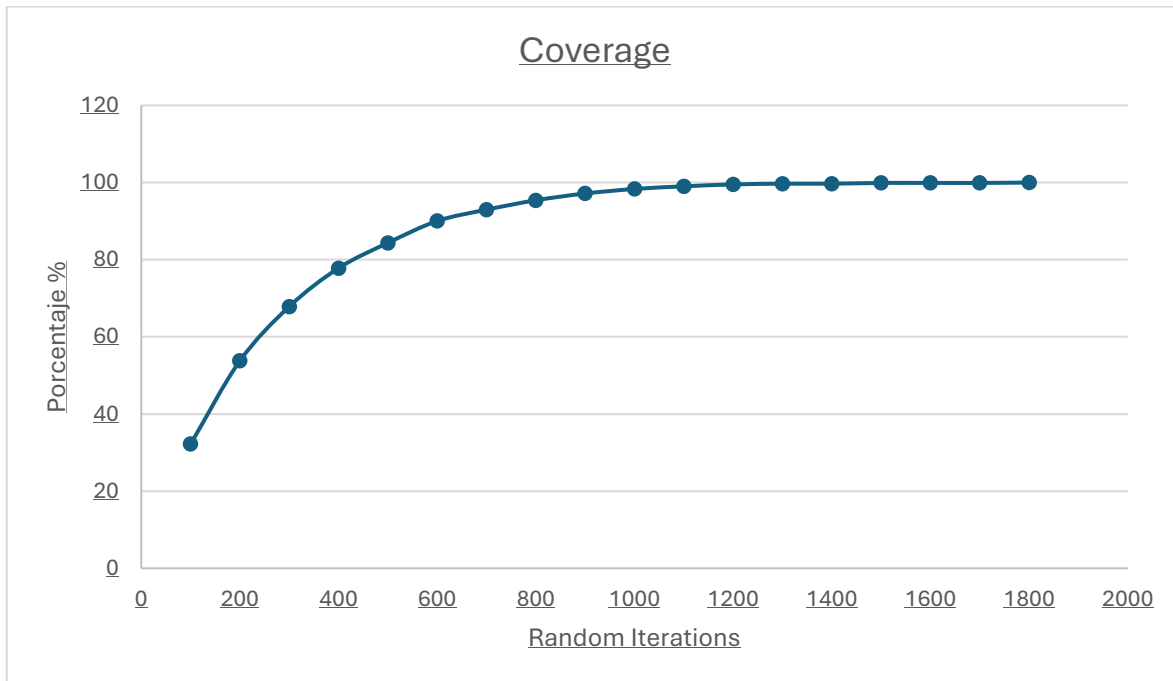
Observando las ilustraciones anteriores se puede observar que se generan covergroups cuando se llaman las tasks a través de estas macros, que entonces al un task dar un valor a una entrada se registrará en su respectivo coverpoint y en el covergroup en general.

Definir macros permite reutilizar fácilmente configuraciones de cobertura en diferentes contextos o configuraciones de pruebas.

- **Facilita la consistencia:** Cada covergroup sigue un formato estándar.
- **Permite la modularidad:** Se puede crear y extender cobertura específica sin duplicar código.
- **Simplifica los cambios:** Si se necesita modificar la configuración (por ejemplo, cambiar bins_nums), solo se ajusta en una vez.

Los coverpoint y covergroup se manejan así porque simplifica el monitoreo de los valores que tienen las entradas, de esta manera a un solo clic se tiene el covergroup de todas las entradas.

Para obtener el numero de iteraciones optimo para tener el 100% del coverage de entradas de 8 bits se tuvo que tomar mediciones incrementando cada 100 iteraciones , desde 500 hasta poder obtener el 100% del coverage, aquí se puede observar que a las 1800 iteraciones obtenemos el 100%, entonces 1800 es el numero optimo para todas las pruebas.



El task random fue efectivo, ya que si generó todos los posibles valores de las entradas A y B, dando una efectividad del 100%, como se puede observar en el siguiente reporte:

Group - tb_vector_alu::cg_inputs

Group Info : cg_inputs

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Crvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	0	64	0	

Source File(s) : [C:\Users\jessie\Documents\Circuitr\Digital_projects\ALU\YinYao\project_1\project_1_area\sim_1\area\vector_alu_tb.v](#)

Group Instance(s) Info : 1 (Total)

Name (%)	Score (%)	Weight	Goal	At Least	Auto Bin Max	Print Missing	Comment
cg_instpu...	100	1	100	1	64	0	

Instance : cg_inputs_inst

Instance Info : cg_inputs_inst

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	In Instantized	Comment
100	1	100	1	64	0	1	

Instance Variable(s) Summary : cg_inputs_inst


Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
8	Variables	2048	0	2048	100

Instance Cover Point(s) Details : cg_inputs_inst

Name (%)	Expected	Uncovered	Covered	Percent (%)	Goal	Weight	At Least	Auto Bin Max	Comment
cg_a0	256	0	256	100	100	1	1	256	
cg_a1	256	0	256	100	100	1	1	256	
cg_a2	256	0	256	100	100	1	1	256	
cg_a3	256	0	256	100	100	1	1	256	
cg_b0	256	0	256	100	100	1	1	256	
cg_b1	256	0	256	100	100	1	1	256	
cg_b2	256	0	256	100	100	1	1	256	
cg_b3	256	0	256	100	100	1	1	256	

Ilustración 11 Covergroup de TEST2

Por otra parte TEST2 generó una efectividad del 99.5%, bastante bien, considerando que dentro de esta prueba se consideran los valores de A y B pero considerando que se cumpla en cada una de las seis operaciones.



Groups Coverage Summary

Score	Inst Score
99.9512	99.9512

Total groups in report: 6

Name	Score	Num Instances	Avg Instances Score	Weight	Goal	Merge Instances	Get Inst Coverage	Per Instance	Auto Bin Max	Comment
tb_vector_alu::cg_add_inputs	100	1	100	1	100	0	0	0	64	
tb_vector_alu::cg_sub_inputs	99.9023	1	99.9023	1	100	0	0	0	64	
tb_vector_alu::cg_and_inputs	100	1	100	1	100	0	0	0	64	
tb_vector_alu::cg_or_inputs	99.9512	1	99.9512	1	100	0	0	0	64	
tb_vector_alu::cg_mul_inputs	99.9512	1	99.9512	1	100	0	0	0	64	
tb_vector_alu::cg_div_inputs	99.9023	1	99.9023	1	100	0	0	0	64	

Ilustración 12 Covergroup de TEST2

Finalmente se realiza la prueba 4, que involucra dar un valor fijo a una variable y luego invertir el papel de éstas dos, por lo que se puede esperar que la efectividad sea menor a las anteriores, y ese fue el resultado.

Nota: la prueba 3 no se realizó ya que es similar a la 4.

AMD

Groups Coverage Summary

Score	Inst Score
66.7806	66.7806

Total groups in report: 2

Name	Score	Num Instances	Avg Instances Score	Weight	Goal	Merge Instances	Get Inst Coverage	Per Instance	Auto Bin Max	Comment
tb_vector_alu::cg_a_zero_b_op_random	66.7969	1	66.7969	1	100	0	0	0	64	
tb_vector_alu::cg_b_zero_a_op_random	66.7643	1	66.7643	1	100	0	0	0	64	

Ilustración 13 Covergroup de TEST4

Estas pruebas se realizaron con el objetivo de comprobar la viabilidad de la randomización de los valores dentro de las task del testbench, podemos concluir que en general si se cubren todas las posibilidades esperadas.

Asserts

En Vivado, los asserts (afirmaciones) son una herramienta utilizada en la simulación de SystemVerilog para verificar el comportamiento esperado de un diseño. Actúan como **chequeos automáticos** que validan si ciertas condiciones se cumplen durante la simulación, ayudando a detectar errores temprano en el ciclo de desarrollo.

En Test 1 Asegura que se lances estímulos aleatorios con la idea de que cubran un rango ampli para el test, por lo que el assert:

```
assert(cg_inputs_inst.cp_a0.get_coverage() == 100)
    else $error("Error: Cobertura insuficiente para A[0].");
```

Lo que hace es asegurarse que si saque todos los valores random posible revisando que su coverage sea de 100%.

En Test 2 se asegura que A y B no salgan de los valores de 0 a 256, en suma, resta multiplicación.

En este código, se utiliza la estructura foreach porque las señales alu_if_inst.A y alu_if_inst.B son arreglos. La instrucción foreach en SystemVerilog permite recorrer de manera eficiente cada elemento de un arreglo, aplicando una operación específica a cada uno. Esto es particularmente útil cuando se necesita realizar verificaciones o aplicar lógica de manera uniforme a todos los elementos del arreglo.

En este caso, se utiliza para recorrer cada índice del arreglo y verificar que el valor firmado de cada elemento sea igual a cero. Si la condición no se cumple, se genera un error personalizado que incluye información sobre el índice y el valor que no satisface la verificación.

El uso de foreach ofrece varias ventajas respecto a un bucle for tradicional. En primer lugar, mejora la simplicidad y legibilidad del código al eliminar la necesidad de especificar manualmente los límites del arreglo. En lugar de manejar manualmente los límites del arreglo, foreach se adapta automáticamente al tamaño. Por ejemplo, si A tiene tamaño [0:3], el foreach automáticamente iterará por los índices 0, 1, 2, 3; esto permite que el código sea más fácil de entender y mantener. Finalmente, reduce la probabilidad de errores al evitar cálculos manuales de índices, proporcionando una solución más limpia y directa para iterar sobre los elementos.

```
foreach (alu_if_inst.A[i])
    assert ($signed(alu_if_inst.A[i]) >= 0 &&
$signed(alu_if_inst.A[i]) < 256)
        else $error("Entrada A[%0d] fuera de rango: %0d", i,
alu_if_inst.A[i]);

    foreach (alu_if_inst.B[i])
        assert ($signed(alu_if_inst.B[i]) >= 0 &&
$signed(alu_if_inst.B[i]) < 256)
            else $error("Entrada B[%0d] fuera de rango: %0d", i,
alu_if_inst.B[i]);
```

En división es similar solo que ahora en vez de asegurar que tenga cero, se asegura la cuenta desde el 1

```
foreach (alu_if_inst.A[i])
    assert ($signed(alu_if_inst.A[i]) >= 1 &&
$signed(alu_if_inst.A[i]) < 256)
    else $error("Entrada A[%0d] fuera de rango: %0d", i,
alu_if_inst.A[i]);
    foreach (alu_if_inst.B[i])
        assert ($signed(alu_if_inst.B[i]) >= 1 &&
$signed(alu_if_inst.B[i]) < 256)
        else $error("Entrada B[%0d] fuera de rango: %0d", i,
alu_if_inst.B[i]);
```

En Test 3 Se utiliza sencillamente que los valores sean 0 en sus sumas si solo A esta sumando 0 se pone la condición al momento de aparecer un cero arroje un error, lo mismo con B

```
foreach (alu_if_inst.A[i])
    assert ($signed(alu_if_inst.A[i]) == 0)
    else $error("A no es cero", i, alu_if_inst.A[i]);
    foreach (alu_if_inst.B[i])
        assert ($signed(alu_if_inst.B[i]) == 0)
        else $error("B no es cero", i, alu_if_inst.B[i]);
```

En Test 4 se utiliza los mismos asserts que en Test 3.

Conclusiones

A pesar de que para el enfoque que se le dio a esta cobertura parece ser un éxito, podría ser interpretado de diferente manera si se busca un diferente objetivo, por ejemplo saber si se cubren todos los resultados posibles de una operación o todos los valores mayores a 255, etc. Por lo que puede ser trabajo de optimización a futuro.

Este trabajo ha sido un gran aprendizaje para todo el equipo, ya que es un primer acercamiento al mutuo trabajo y retroalimentación que se tiene en el proceso de diseño y verificación de un proyecto.

