Cátedra de Arquitectura de Computadoras

Trabajo Práctico N° I
ALU

Carrizo, Aixa Mariel
Piñero, Tomás Santiago
15 de Octubre de 2020

Índice

1. Enunciado

El objetivo de este trabajo es implementar en FPGA una ALU, cuyos requerimientos son los siguientes:

- Debe ser parametrizable (bus de datos) para poder ser utilizada posteriormente en el trabajo final;
- Validar el desarrollo por medio de Test Bench;
- Las operaciones de la ALU deben ser las siguientes:
  - `ADD: 100000`
  - `SUB: 100010`
  - `AND: 100100`
  - `OR: 100101`
  - `XOR: 100110`
  - `SRA: 000011`
  - `SRL: 000010`
  - `NOR: 100111`

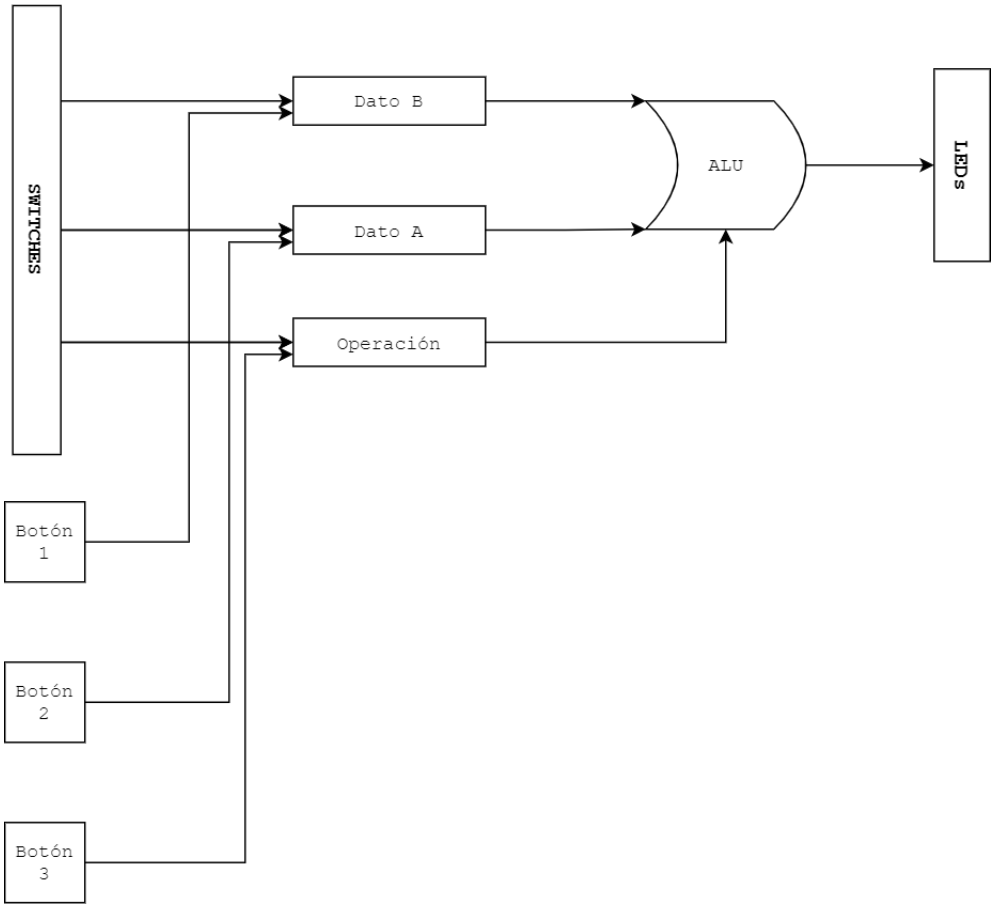

Figura 1: Esquema del proyecto.

2. Desarrollo

Lo primero que se realizó fue el diseño de la ALU, que es un circuito puramente combinacional. Luego, se realizó el diseño de un módulo *top*, en el que se utiliza un *clock* y se realiza el ingreso de los valores a las entradas de la ALU.

2.1. ALU

2.1.1. Diseño

En este diseño, la ALU tiene tres entradas y una única salida. Las dos entradas que corresponden a los datos y la salida son de tamaño parametrizable: `N_BITS`, mientras que la entrada correspondiente a la operación a realizar es fija (6 bits).

Las entradas son de tipo *wire*, ya que se ingresan mediante el *switch*. La salida, por otro lado, es de tipo *reg* para almacenar su valor.

Cada vez que se detecte un cambio, se comprueba el código de operación y se la ejecuta. En caso de que el número de operación no sea correcto, se pone la salida en 0.

Código 1: Código fuente de la ALU.

```
1  `timescale 1ns / 1ps
2
3  module alu#(
4    //Parameters
5    parameter    N_BITS          = 8
6  )
7  (
8    //inputs
9    input wire [N_BITS-1:0]     i_dato_A,
10   input wire [N_BITS-1:0]     i_dato_B,
11   input wire [5:0]            i_operacion,
12
13   //output
14   output reg [N_BITS-1:0]     o_alu
15 );
16
17     always@(*) begin:alu
18         case(i_operacion)
19         6'b100000:  o_alu = i_dato_A + i_dato_B;     //suma
20         6'b100010:  o_alu = i_dato_A - i_dato_B;     //resta
21         6'b100100:  o_alu = i_dato_A & i_dato_B;     //and
22         6'b100101:  o_alu = i_dato_A | i_dato_B;     //or
23         6'b100110:  o_alu = i_dato_A ^ i_dato_B;     //xor
24         6'b000011:  o_alu = i_dato_A >>> i_dato_B;   //SRA (arithmetic):
               extiende el signo
25         6'b000010:  o_alu = i_dato_A >> i_dato_B;    //SRL (logic): insterta
               0
26         6'b100111:  o_alu = ~(i_dato_A | i_dato_B); //nor
27         default: o_alu = {N_BITS{1'b0}};            //default = 0
28         endcase
29     end
30 endmodule
```

El esquema RTL resultante es el siguiente:

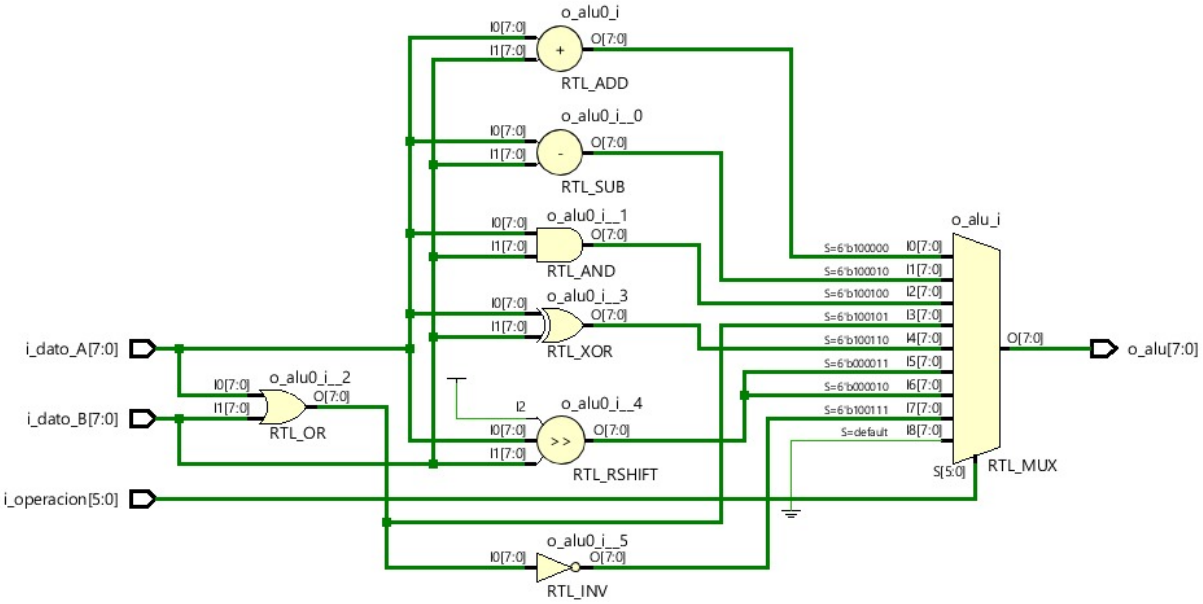

Figura 2: Esquema RTL del módulo *ALU* en Vivado.

2.1.2. *Testbench*

En el *testbench*, se generan entradas aleatorias con cada flanco de subida del *clock* y se realiza una operación tras otra hasta que se termine el tiempo de ejecución del *test* (en este caso 1000 [ns]).

Código 2: *Testbench* para la ALU.

```
1  `timescale 1ns / 1ps
2
3  module tb_alu();
4
5    localparam N_BITS          = 8;
6
7    //operaciones de la alu
8    localparam ADD = 6'b100000;
9    localparam SUB = 6'b100010;
10   localparam AND = 6'b100100;
11   localparam OR  = 6'b100101;
12   localparam XOR = 6'b100110;
13   localparam SRA = 6'b000011;
14   localparam SRL = 6'b000010;
15   localparam NOR = 6'b100111;
16
17   // señales para el testbench
18   reg  [N_BITS-1:0]          dato_a;
19   reg  [N_BITS-1:0]          dato_b;
20   reg  [N_BITS-3:0]          operacion;
21   wire [N_BITS-1:0]          resul;
22   reg  [N_BITS-1:0]          resul_esperado;
23   reg  [3:0]                 id_op; //para realizar las operaciones
24   reg                        clk;
25   reg                        test_start;
```

```
26
27
28  initial begin
29    //inicializacion de variables en 0
30    id_op  = 4'b0;
31    dato_a = 8'b0;
32    dato_b = 8'b0;
33    operacion = 6'b0;
34    resul_esperado = 8'b0;
35    test_start = 1'b0;
36    clk = 1'b0;
37
38    #50
39    test_start = 1'b1;
40    #1000
41
42    $display ("-----TEST OK!-----");
43    $finish();
44  end
45
46
47  always@(posedge clk)
48  begin
49    dato_a <= $random();
50    dato_b <= $urandom();
51  end
52
53
54  always @(posedge clk)
55  begin
56    case(id_op)
57    4'd0:
58        begin
59            operacion <= ADD;
60            resul_esperado <= dato_a + dato_b;
61        end
62    4'd1:
63        begin
64            operacion <= SUB;
65            resul_esperado <= dato_a - dato_b;
66        end
67    4'd2:
68        begin
69            operacion <= AND;
70            resul_esperado <= dato_a & dato_b;
71        end
72    4'd3:
73        begin
74            operacion <= OR;
75            resul_esperado <= dato_a | dato_b;
76        end
77    4'd4:
78        begin
79            operacion <= XOR;
80            resul_esperado <= dato_a ^ dato_b;
81        end
82    4'd5:
83        begin
```

```
84              operacion <= SRA;
85              resul_esperado <= dato_a >>> dato_b;
86          end
87      4'd6:
88          begin
89              operacion <= SRL;
90              resul_esperado <= dato_a >> dato_b;
91          end
92      4'd7:
93          begin
94              operacion <= NOR;
95              resul_esperado <= ~(dato_a | dato_b);
96          end
97      default: $finish();
98      endcase
99    end
100
101   always #10 clk = ~clk;
102
103   always @(posedge clk)
104   begin
105     if (test_start)
106      begin
107         if(resul != resul_esperado)
108             begin
109                 $display("----- TEST FALLO -----");
110                 $display(id_op);
111                 $display(resul);
112                 $display(resul_esperado);
113                 $finish();
114             end //end if
115         id_op = (id_op + 4'b1) % 8;
116      end
117   end
118
119   alu
120   #
121   (
122         .N_BITS(N_BITS)
123   )
124   my_alu
125   (
126         .i_dato_A(dato_a),
127         .i_dato_B(dato_b),
128         .i_operacion(operacion),
129         .o_alu(resul)
130    );
131 endmodule
```

2.1.3. Simulación

Primero se realizó la simulación de comportamiento del módulo diseñado. El resultado fue el siguiente:

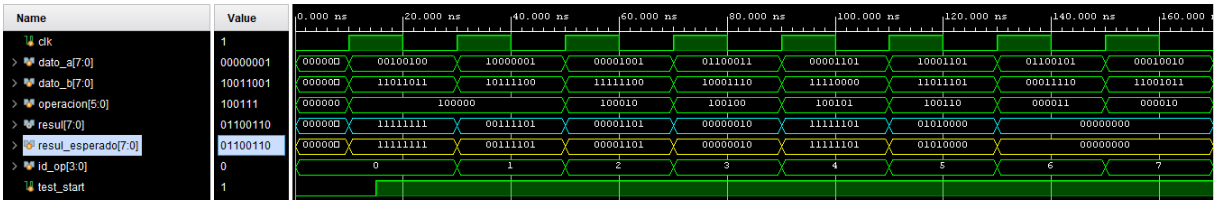

Figura 3: Simulación de comportamiento.

En la Fig. **3** se puede observar que la ALU funciona correctamente. En *cyan* está el resultado de la operación realizada y en amarillo el resultado esperado. Cabe mencionar que las operaciones solamente se realizan si la variable *test_start* se encuentra en 1.

Luego, se realizó la simulación post-síntesis con *timing*. Su resultado fue:

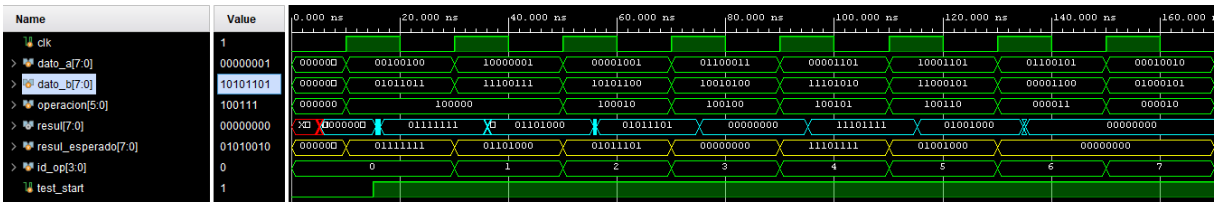

Figura 4: Simulación post-síntesis con *timing*.

En la Fig. **4** se pueden ver la presencia de retardos entre los resultados de las las operaciones, dado que los cambios en la salida del módulo ya no coinciden con los flancos positivos del *clock*. Esto se debe a que, al momento de implementar el diseño en la FPGA, existe un retardo propio del hardware que se simula.

2.2. Top

2.2.1. Diseño

En el módulo '*top*' se implementaron el *switch*, los botones y los LEDs. Los dos primeros como entradas y el último como salida, todos tipo *wire*.

Este módulo instancia un módulo ALU llamado '*my_alu*', pasando como parámetros la cantidad de bits de las entradas, el valor de las mismas (a través del *switch*) y asigna el resultado de la ALU a los LEDs.

Cada vez que se detecte un flanco de subida del *clock*, se verifica qué entrada se ingresó: el dato A, el dato B o la operación a realizar.

A continuación se muestra el código fuente.

Código 3: Código fuente del sistema.

```
1 `timescale 1ns / 1ps
2
3 module top
4 #
5 (
6     parameter    N_BITS = 8
7 )
```

```
(
    input  wire [N_BITS-1:0] i_switch,
    input  wire [2:0]        i_boton,
    input  wire              i_clock,
    output wire [N_BITS-1:0] o_leds
);
    reg [N_BITS-1:0] dato_a;
    reg [N_BITS-1:0] dato_b;
    reg [5:0]      operacion;

    //instancia el modulo ALU
    alu
    #
    (
        .N_BITS(N_BITS)
    )
    my_alu
    (
        .i_dato_A(dato_a),
        .i_dato_B(dato_b),
        .i_operacion(operacion),
        .o_alu(o_leds)
    );

    //asigna los numeros ingresados segun los
    //botones
    always@(posedge i_clock) begin:inputs

        case(i_boton)
        3'b001: dato_a <= i_switch;
        3'b010: dato_b <= i_switch;
        3'b100: operacion <= i_switch;
        endcase

    end//inputs

endmodule
```

El esquema RTL resultante es el que se muestra a continuación.

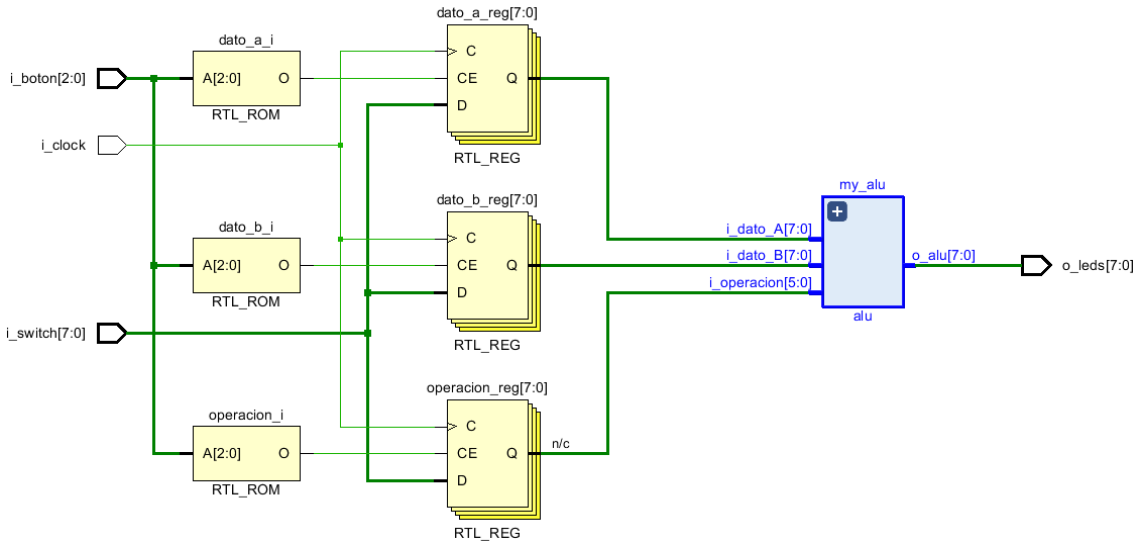

Figura 5: Esquema RTL del módulo *top* en Vivado.

2.2.2. *Testbench*

El *testbench* del módulo top es muy simple: inicia los valores de *clock*, botones, *switch* e "id_op" en 0. Luego, se ingresan los valores de los datos, presionando sus botones correspondientes y finalmente, en un bucle *for*, se realizan todas las operaciones con dichos valores.

A continuación se muestra el código utilizado:

Código 4: *Testbench* para el sistema.

```
1  `timescale 1ns / 1ps
2
3  module tb_top();
4
5    localparam N_BITS        = 8;
6    localparam N_OPERACIONES = 8;
7
8    //operaciones de la alu
9    localparam ADD = 6'b100000;
10   localparam SUB = 6'b100010;
11   localparam AND = 6'b100100;
12   localparam OR  = 6'b100101;
13   localparam XOR = 6'b100110;
14   localparam SRA = 6'b000011;
15   localparam SRL = 6'b000010;
16   localparam NOR = 6'b100111;
17
18   // señales para el testbench
19   reg  [N_BITS-1:0] switch;
20   reg  [2:0]        boton;
21   reg                clk;
22   wire [N_BITS-1:0] leds;
23
24   reg  [3:0]        id_op; //para realizar las operaciones
25
```

```
26  initial begin
27    //inicializacion de variables en 0
28    clk    = 1'b0;
29    boton  = 3'b0;
30    switch = {N_BITS {1'b0}};
31    id_op  = 4'b0;
32
33    //asigna un valor a i_dato_A
34    #100
35    switch = {$urandom()};
36    #50
37    boton = 3'b001;
38    #50
39    boton = 3'b000;
40
41    //asigna un valor a i_dato_B
42    #50
43    switch = {$urandom()};
44    #50
45    boton = 3'b010;
46    #50
47    boton = 3'b000;
48
49    //realiza todas las operaciones de la alu con
50    //los datos dados
51    for(id_op = 0; id_op < N_OPERACIONES; id_op = id_op + 1)
52    begin
53
54     #50
55     boton = 3'b100;
56     case(id_op)
57     4'd0: switch = ADD;
58     4'd1: switch = SUB;
59     4'd2: switch = AND;
60     4'd3: switch = OR;
61     4'd4: switch = XOR;
62     4'd5: switch = SRA;
63     4'd6: switch = SRL;
64     4'd7: switch = NOR;
65     endcase
66
67     #50
68       boton = 3'b000;
69    end
70
71    #1200;
72    $display("---------- OK! ----------");
73    $finish();
74  end
75
76  //inicio el modulo top que, asu vez,
77  //instancia la ALU
78  top
79  #(
80    .N_BITS(N_BITS)
81  )
82  u_top
83  (
```

```
84      .i_switch  (switch),
85      .i_boton   (boton),
86      .i_clock     (clk),
87      .o_leds      (leds)
88    );
89
90    //genera el clock
91    always #10 clk = ~clk;
92  endmodule
```

2.2.3. Simulación

En este caso se realizó lo mismo que en la sección **2.1.3**.

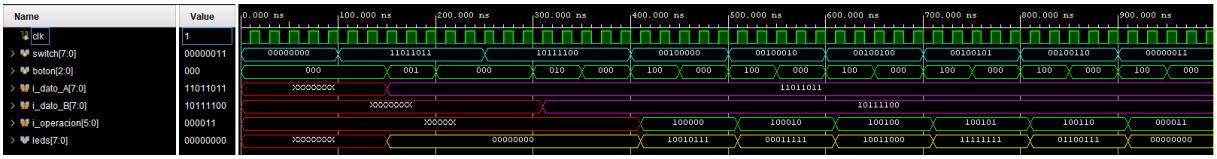

Figura 6: Simulación de comportamiento: 'top'.

En la Fig. **6** se observa que todos los cambios de valor en la salida '*o_leds*' ocurren con el flanco positivo del *clock*.

Los símbolos '`XXXXXXXX`' indican que al iniciar la ejecución, los valores son desconocidos, y por lo tanto, inválidos.

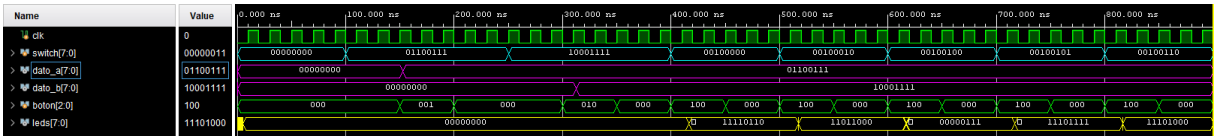

Figura 7: Simulación post-síntesis con *timing*: 'top'.

La Fig. **7** muestra el resultado de la simulación post-síntesis con *timing*, donde los valores desconocidos, si bien siguen presentes, permanecen durante un tiempo menor en todas las variables, en comparación a la simulación de comportamiento realizada anteriormente. También se ven los retardos generados en la salida, de forma que los cambios en la misma ya no coinciden con el flanco positivo del *clock*.

3. Cálculo de frecuencia máxima

Para obtener el valor de la máxima frecuencia a la que puede trabajar el circuito se tomaron los tiempos de *setup* para cada uno de los botones. Los valores son:

- Tiempo de retardo entre que se presiona el primer botón y el dato está disponible en la ALU:
$$t_{setup_A} = 2{,}52\,[ns]$$
- Tiempo de retardo entre que se presiona el segundo botón y el dato está disponible en la ALU:
$$t_{setup_B} = 12{,}52\,[ns]$$
- Tiempo de retardo entre que se presiona el tercer botón y el resultado está disponible:
$$t_{setup_R} = 16{,}662\,[ns]$$

Sumando los tiempos mencionados, se obtiene el tiempo del *clock*:

$$\begin{aligned} t_{clock} &= t_{setup_A} + t_{setup_B} + t_{setup_R} \\ t_{clock} &= 2{,}52 + 12{,}52 + 16{,}662 \\ t_{clock} &= 31{,}702\,[ns] \end{aligned}$$

Finalmente, el periodo del *clock* es:

$$T_{clock} = 31{,}54\,[MHz] \tag{1}$$

4. Conclusiones

El diseño de los módulos no resultó complejo, pero se tuvo problemas con las simulaciones del *testbench* de la ALU, que en un principio no tenían diferencia, ya que todos los cambios de variable ocurrían simultáneamente con el flanco positivo del *clock*. Por este motivo, se cambió la placa '*Basys 3*' por una '*Spartan 7*'. Finalmente, el problema resultó ser el diseño que se estaba utilizando.