PRÁCTICA 4: DISEÑO DE UNA ALU DE 32 BITS PARA EL PROCESADOR ICAI-RISC-V

Alumnos: Pablo Menéndez Ruiz de Azúa y Carles Olucha Royo
Fecha de realización: 17/03/2021
Fecha de entrega: 15/04/2021
Asignatura: Sistemas digitales II
Profesor: Pedro Olmos González

Índice

Objetivos

El objetivo de esta práctica es el diseño de una ALU de 32 bits, la cual será utilizada en la ruta de datos del procesador ICAI–RISC–V.

Introducción

En esta práctica se va a implementar una ALU similar a la que se realizó en el curso de Sistemas Digitales I. La principal diferencia es que esta ALU está pensada para ser usada en la implantación del procesador ICAI–RISC–V, por lo que sus dos entradas de datos y su salida serán de 32 bits. Además, las operaciones a realizar serán las marcadas por la arquitectura del procesador, las cuales se resumen en la tabla 1.

Operación	Codificación	Explicación
ADD	0000	Suma a+b
SUB	1000	Resta a-b
SLT	0010	Set on Less Than (salida = 1 si a <b)
SLTU	0010	Set on Less Than Unsigned (salida = 1 si a <b, comparación sin signo)
SLL	0001	Shift Left Logical
SRL	0101	Shift Right Logical
SRA	1101	Shift Right Arithmetic
XOR	0100	Operación XOR bit a bit
OR	0110	Operación OR bit a bit
AND	0111	Operación AND bit a bit

Tabla 1 – Operaciones de la ALU

Diagrama de bloques

Antes de empezar a programar cada uno de los componentes necesarios para diseñar nuestro display, vamos a hacer un diagrama de bloques con cada componente del circuito.

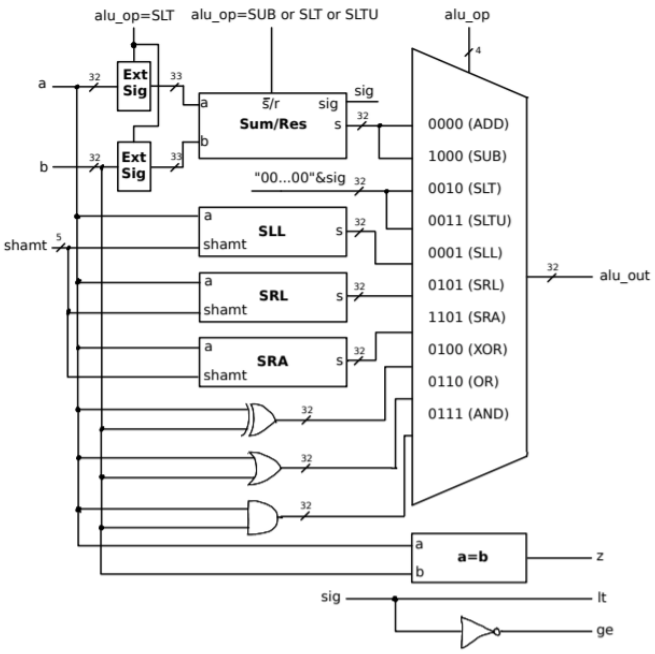

Figura 1 – Diagrama de bloques de la ALU

Como podemos ver, los bloques son bastante sencillos, por lo que, en esta práctica, únicamente diseñaremos a parte de la ALU, el sumador/restador.

Sumador/Restador

Este componente es bastante sencillo, ya que lo único que va a hacer es sumar o restar dos números que le entran, dependiendo de otra señal de entrada s_r. Por último, además de obtener la solución de la operación de suma o diferencia, obtenemos un acarreo, para ver si ha habido desbordamiento.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity Sum_res is
6      generic(
7          generic_length : integer
8      );
9     port(
10       a,b       : in std_logic_vector(generic_length downto 0); --Entrada
11       s      : out std_logic_vector(generic_length-1 downto 0); --Salida
12       s_r       : in std_logic_vector(3 downto 0);
13       sig      : out std_logic); --Acarreo de salida
14 end Sum_res;
15
16 architecture behavioral of Sum_res is
17     signal salida : unsigned(generic_length downto 0);
18
19 begin
20     salida <= unsigned(a) + unsigned(b) when s_r = "0000" else unsigned(a) - unsigned(b);
21
22     sig <= salida(generic_length);
23     s <= std_logic_vector(salida(generic_length -1 downto 0));
24 end behavioral;
```

Figura 2 – Código VHDL del sumador/restador

ALU

Una vez, que hemos diseñador el sumador/restador, ya podemos diseñar nuestro componente principal que es la ALU.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALU is
    generic(
        generic_length : integer :=4
    );
    port(
        a,b : in std_logic_vector(generic_length -1 downto 0);
        alu_op : in std_logic_vector(3 downto 0);
        shamt   : in std_logic_vector(1 downto 0); --normalmente (4 downto 0) pero para implantar lo hacemos con 4 bits
        alu_out : out std_logic_vector(generic_length -1 downto 0);
        z,lt,ge : out std_logic
    );
end ALU;

architecture behavioral of ALU is
    signal a_ext, b_ext : std_logic_vector(generic_length downto 0);
    signal sr_out, sll_out, srl_out, sra_out,and_out,or_out,xor_out : std_logic_vector(generic_length -1 downto 0);
    signal sig : std_logic;
    signal ceros : std_logic_vector(generic_length -2 downto 0);

    component Sum_res
        generic(
            generic_length : integer
        );
        port(
            a,b       : in std_logic_vector(generic_length downto 0); --Entrada
            s     : out std_logic_vector(generic_length-1 downto 0); --Salida
            s_r       : in std_logic_vector(3 downto 0);
            sig      : out std_logic); --Acarreo de salida
    end component;
begin

    -- Extensión de signo
    a_ext <= a(generic_length -1) & a when alu_op = "0010" else '0' & a;
    b_ext <= b(generic_length -1) & b when alu_op = "0010" else '0' & b;

    -- vector de ceros para el slt
    ceros <= (others => '0');

    -- Sumador/Restador
    il_sr : Sum_res
        generic map(
            generic_length => generic_length
        )
        port map(
            a => a_ext,
            b => b_ext,
            s => sr_out,
            s_r => alu_op,
            sig => sig
        );

    -- Desplazamientos
    sll_out <= std_logic_vector(shift_left(unsigned(a),to_integer(unsigned(shamt))));
    srl_out <= std_logic_vector(shift_right(unsigned(a),to_integer(unsigned(shamt))));
    sra_out <= std_logic_vector(shift_right(signed(a),to_integer(unsigned(shamt))));

    -- Funciones lógicas
    Genlog: for i in 0 to generic_length -1 generate

        and_out(i) <= a(i) and b(i);
        or_out(i) <= a(i) or b(i);
        xor_out(i) <= a(i) xor b(i);
    end generate Genlog;

    -- Salidas de comparacion
    z <= '1' when a = b else '0';
    lt <= sig;
    ge <= not sig;

    -- Multiplexor
    with alu_op select
        alu_out <=
            sr_out when "0000",
            sr_out when "1000",
         ceros & sig when "0010",
         ceros & sig when "0011",
            sll_out when "0001",
            srl_out when "0101",
            sra_out when "1101",
            xor_out when "0100",
            or_out when "0110",
            and_out when "0111",
```

Figura 3 – Código VHDL de la ALU

Una vez realizado el diseño de la ALU, vamos a comprobar que funciona correctamente. Para ello, realizaremos un testbench, con únicamente 4 bits en vez de 32 para que no dure mucho la simulación y no consumamos toda la RAM del ordenador.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALU_vhd_tst is
end ALU_vhd_tst;

architecture tb of ALU_vhd_tst is
    constant generic_length : integer :=4;
    signal a,b :std_logic_vector(generic_length -1 downto 0);
    signal alu_op : std_logic_vector(3 downto 0);
    signal  shamt   : std_logic_vector(2 downto 0); -- sería logaritmo de generic_length menos 1
    signal  alu_out : std_logic_vector(generic_length -1 downto 0);
    signal  z,lt,ge : std_logic;
    signal a_and_b, a_or_b, a_xor_b : std_logic_vector(generic_length -1 downto 0);
    --signal sll_out,srl_out,sra_out : std_logic_vector(generic_length -1 downto 0);
    signal shamt_i : unsigned(2 downto 0);
    constant max_value: integer := (2**generic_length-1);

    constant c_time : time :=1 ns;

begin

   i1 : entity work.ALU
      generic map(
         generic_length => generic_length)
      port map(
         a => a,
         b => b,
         alu_op => alu_op,
         shamt => shamt,
         alu_out => alu_out,
         z => z,
         lt => lt,
         ge => ge
      );


   p_stim: process
   --inicializamos las entradas
   begin
      a <= (others => '0');
      b <= (others => '0');
      alu_op <= "1000"; -- Inicializo a esta porque no es ninguna de las del mux
      shamt <= (others => '0');

      a_and_b <= a and b;
      a_or_b <= a or b;
      a_xor_b <= a xor b;

      wait for 100 ns;

      -- Hacemos el bucle
      for i in 0 to max_value loop
         a <= std_logic_vector(to_unsigned(i,generic_length));
         for j in 0 to max_value loop
            b <= std_logic_vector(to_unsigned(j,generic_length));
            -- ya tenemos las señales, ahora probamos la suma
            wait for c_time;
            alu_op <= "0000";
            --esperamos un tiempo y comprobamos la suma
            wait for c_time;

            assert alu_out = std_logic_vector(unsigned(a)+unsigned(b))
               report "No se suman bien"
               severity failure;

            wait for c_time;

            alu_op <= "1000";

            wait for c_time;
            assert alu_out = std_logic_vector(unsigned(a)-unsigned(b))
               report "No se restan bien"
               severity failure;
            wait for c_time;

            --Revisar slt y sltu (habría que extender signo)
            alu_op <= "0010";
            wait for c_time;
            if signed(a)<signed(b) then
               assert alu_out(0) = '1'
                  report "slt no funciona"
                  severity failure;
            end if;
            wait for c_time;

            alu_op <= "0011";
            wait for c_time;
            if unsigned(a)<unsigned(b) then
               assert alu_out(0) = '1'
                  report "sltu no funciona"
```

```
99            -- Funciones lógicas
100           --xor
101           a_and_b <= a and b;
102           a_or_b <= a or b;
103           a_xor_b <= a xor b;
104
105           alu_op <= "0100";
106           wait for c_time;
107           assert alu_out = a_xor_b
108              report "Fallo en el xor"
109              severity failure;
110           wait for c_time;
111
112           --or
113           alu_op <= "0110";
114           wait for c_time;
115           assert alu_out = a_or_b
116              report "Fallo en el xor"
117              severity failure;
118           wait for c_time;
119
120           --and
121           alu_op <= "0111";
122           wait for c_time;
123           assert alu_out = a_and_b
124              report "Fallo en el xor"
125              severity failure;
126           wait for c_time;
127           --Comparaciones
128           if a = b then
129              wait for c_time;
130              assert z = '1'
131                 report "La comparacion de igualdad no funciona"
132                 severity failure;
133           end if;
134
135           if a >= b then
136              wait for c_time;
137              assert ge = '1'
138                 report "La comparacion de igualdad no funciona"
139                 severity failure;
140           end if;
141
142           if a < b then
143              wait for c_time;
144              assert lt = '1'
145                 report "La comparacion de igualdad no funciona"
146                 severity failure;
147           end if;
148        end loop;
149
150        for k in 0 to 4 loop
151           wait for c_time;
152           shamt <= std_logic_vector(to_unsigned(k,3));
153           --shamt_i <= to_unsigned(k,3);
154           wait for c_time;
155           shamt_i <= unsigned(shamt);
156           wait for c_time;
157
158           -- shift logic left
159           alu_op <= "0001";
160           wait for c_time;
161           assert alu_out = std_logic_vector(shift_left(unsigned(a),to_integer(shamt_i)))
162              report "Fallo en el desplazamiento lógico a la izquierda"
163              severity failure;
164
165           wait for c_time;
166           alu_op <= "0101";
167           wait for c_time;
168           assert alu_out = std_logic_vector(shift_right(unsigned(a),to_integer(shamt_i)))
169              report "Fallo en el desplazamiento lógico a la derecho"
170              severity failure;
171
172           wait for c_time;
173           alu_op <= "1101";
174           wait for c_time;
175           assert alu_out = std_logic_vector(shift_right(signed(a),to_integer(shamt_i)))
176              report "Fallo en el desplazamiento lógico a la izquierda"
177              severity failure;
178
179        end loop;
180
181     end loop;
182
183
184     assert false
185     report "Fin de la simulación"
186     severity failure;
187  end process;
```

Figura 4- Testbench de la ALU

Una vez escrito el testbench y corregidos todos los errores, vamos a simular nuestro componente para comprobar que todo funciona como debería.

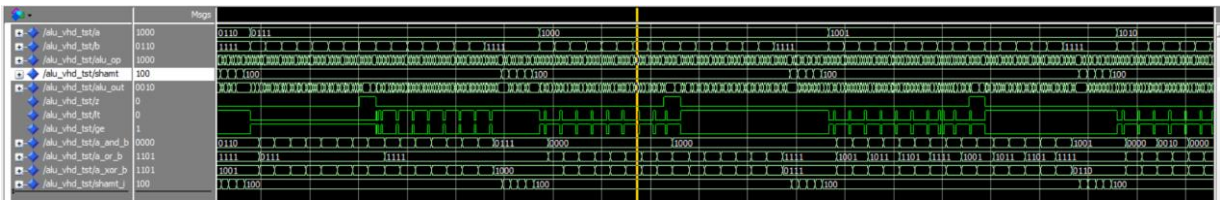

Figura 5 – Trozo de la simulación => Operación 8 -6 = 2

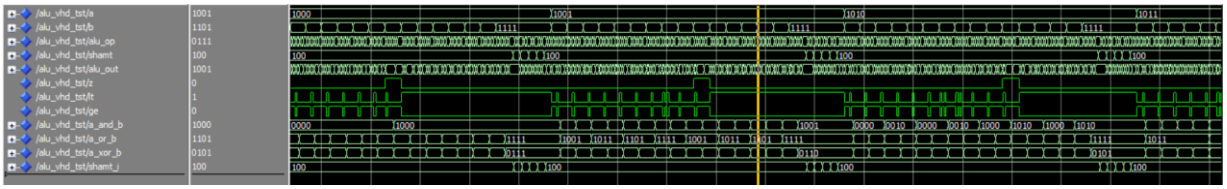

Figura 6 – Trozo de la simulación => Operación 1001 and 1101 = 1001

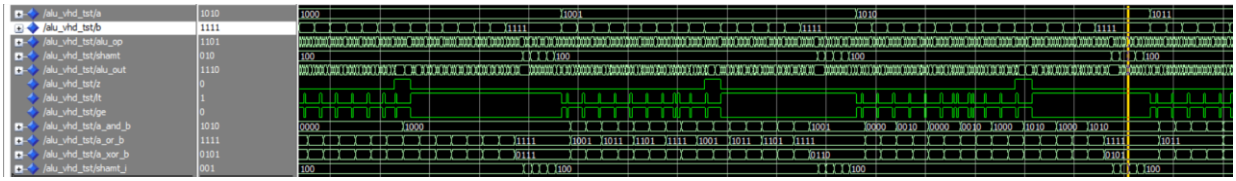

Figura 7 – Trozo de la simulación => Operación 1010 sra 010 = 1110

Implantación física

Finalmente, vamos a implantar nuestra ALU en la placa FPGA, para comprobar si no solo funciona en la simulación, sino también cuando lo implantamos. Para realizar esta implantación, hemos tenido en cuenta que los 4 interruptores de la izquierda son a, los 4 siguientes son b y los dos últimos son el inmediato. Los 4 botones son el alu_op y los 4 leds rojos de la derecha son la salida. Además, hemos incluido z, lt y ge en los leds verde en la parte derecha.

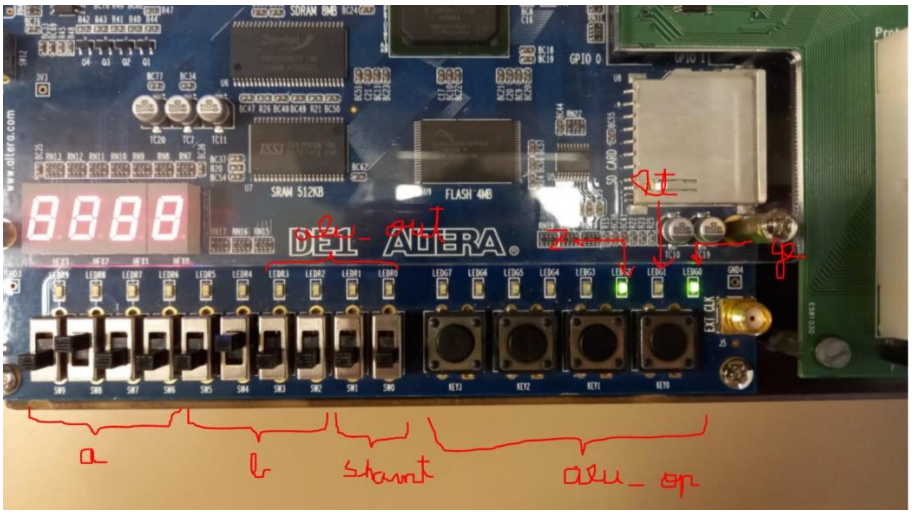

Figura 8 – Asignación de pines FPGA

Una vez que asignamos todos los pines a un periférico de la FPGA, nos dispusimos a realizar las pruebas

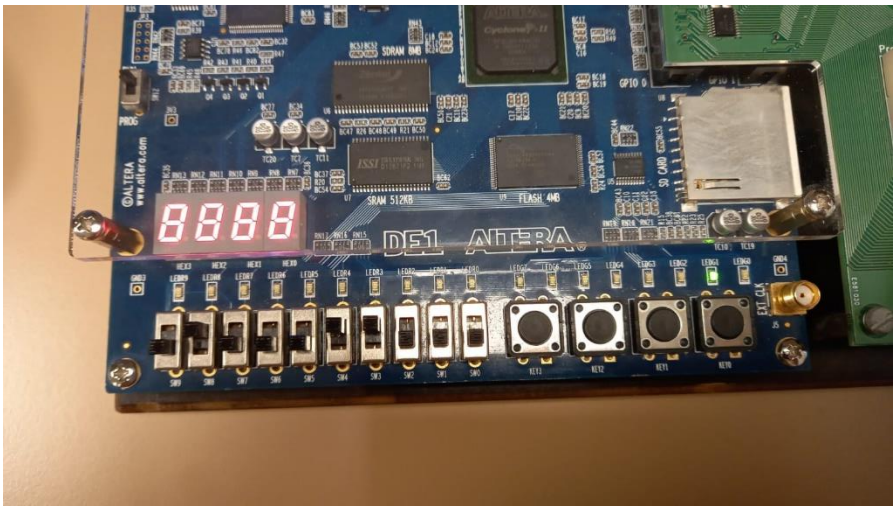

Figura 9 – a (4) < b(6) => lt = 1

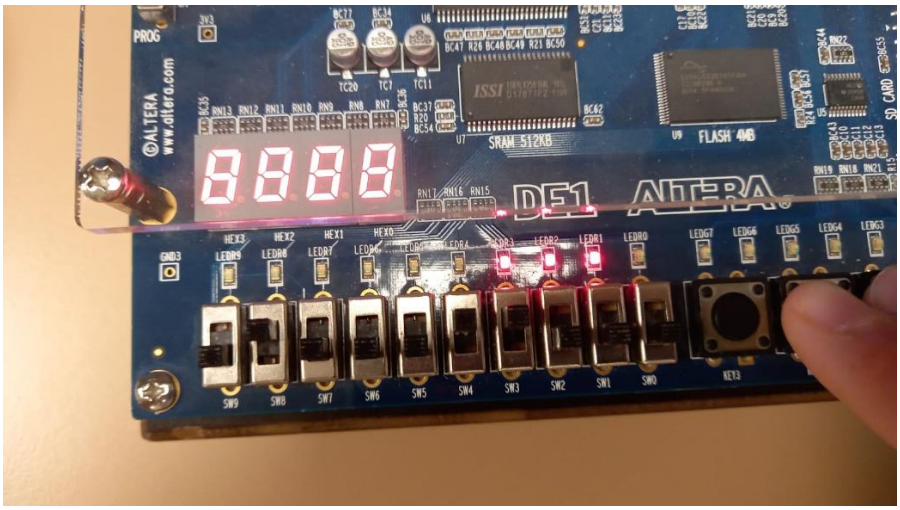

Figura 10 – a (4) – b(6) = -2 (1110)

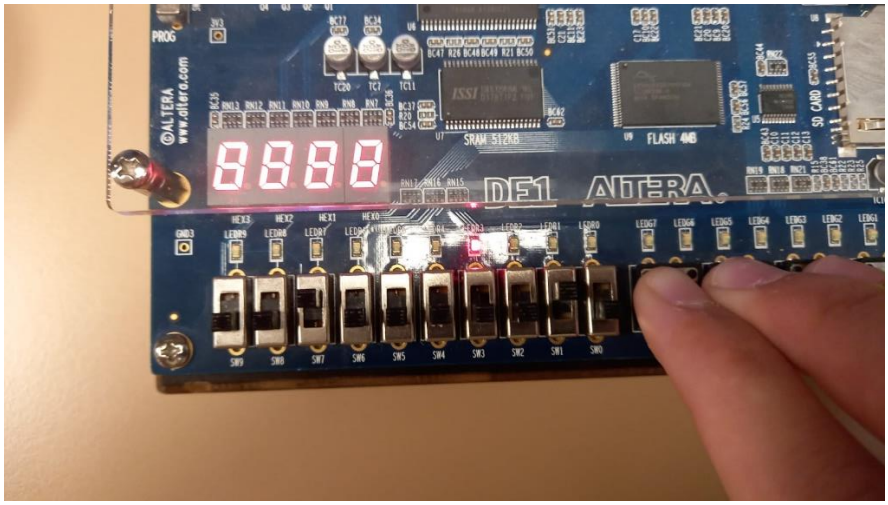

Figura 11 – sll a = 2, shamt = 2, salida = 8

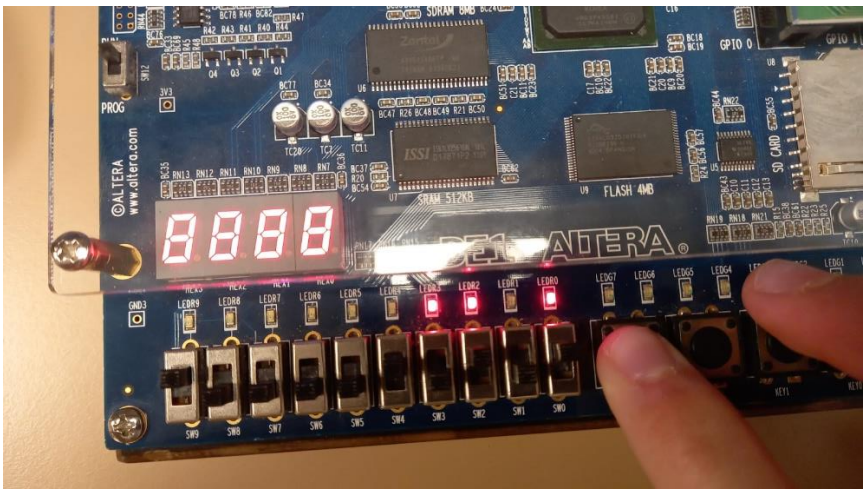

Figura 12 – 1000 or 0101 = 1101

Conclusiones y análisis de los resultados

Nuestro objetivo principal al comienzo de esta práctica era el diseño de una ALU de 32 bits, para la posterior implantación en un microprocesador ICAI-RISC-V. Después de haber diseñado en VHDL el componente, haber hecho una simulación exhaustiva y, finalmente, haberlo implantado en la placa FPGA, hemos comprobado que nuestra ALU funciona perfectamente. Por lo tanto, podemos concluir que hemos cumplido nuestro objetivo inicial y que podemos implantar la ALU en el microprocesador del ICAI-RISC-V que diseñaremos en la próxima práctica-