Universidad Técnica Federico Santa María

"*Diseño de una ALU utilizando la herramienta EDA Playgrond*"

Bayron Valenzuela, ROL: 202073580-8
Sofía Riquelme, ROL: 202073615-4

Fecha de entrega: 19 de Junio de 2022

1. Resumen

En el presente informe se describe el desarrollo de la tarea 3 de arquitectura de computadores, en la cual se diseñó una ALU al igual que un testbench a través del lenguaje de descripción de hardware Verilog que comprueba el funcionamiento de esta. La ALU diseñada puede realizar operaciones aritméticas, lógicas, de transposición y del torneo. Las operaciones del torneo se refieren a las de la tarea 1, donde se ingresan dos jugadores y se retorna el ganador. Las entradas de la ALU corresponden a dos inputs de 8 bits cada uno, una ROM con las instrucciones a ejecutar (las cuales son de 4 bits) y un PC que itera sobre la ROM. La salida depende de la instrucción de la ROM y también se dispone de un flag de 9 bits con una serie de indicadores del resultado de la operación. Se obtuvieron los siguientes resultados.

2. Introducción

El objetivo de esta tarea era entender cómo funcionan los lenguajes de descripción de hardware, y utilizarlo para construir una ALU funcional, la cual fuera capaz de realizar 16 operaciones distintas, las cuales son:

1. Suma
2. Resta
3. Multiplicación
4. División entera
5. Bitwise NOT
6. Bitwise OR
7. Bitwise AND
8. Bitwise XOR
9. Logical shift left
10. Logical shift right
11. Arithmetic shift right
12. Rotate left
13. Rotate right
14. Peso ligero
15. Peso Pesado
16. Peso mixto

También, la flag está hecha de la siguiente manera:

```
[9] [8] [7] [6] [5] [4] [3] [2] [1] [0]
[0] [T] [L] [T] [R] [Z] [N] [C] [V] [E]
```

Donde los números son las posiciones de los bits, y los flags están codificadas de la siguiente manera:

Flag	Condición
A	Si se ejecuta una operación aritmética
L	Si se ejecuta una operación lógica
T	Si se ejecuta una operación transposición
R	Si se ejecuta una operación del torneo
Z	Si el resultado es 0
N	Si el resultado es negativo
C	Si el resultado produce carry
V	Si el resultado produce overflow
E	Si el resultado produce un error

Tabla 1: Codificación de flags

Para el desarrollo de la tarea se utilizaron varias de las operaciones ya implementadas de Verilog, y también implementamos varias a mano.

3. Desarrollo

Para el desarrollo de la tarea, se comenzó implementando las operaciones una por una. Se comenzó por las lógicas, las cuales corresponden a Bitwise NOT, Bitwise OR, Bitwise AND y Bitwise XOR. Ya que estas operaciones son bitwise, se deben realizar bit a bit, por lo que para cada una de estas operaciones se realizó la comparación correspondiente ya implementada por logisim para cada bit, lo cual se muestra en más detalle a continuación:

```
module bitwise_not(A, B, out, flag);
    input logic [7:0] A;
    output logic [7:0] out;
    output logic [9:0] flag;

    assign out[0]= ~ A[0];
    assign out[1]= ~ A[1];
    assign out[2]= ~ A[2];
    assign out[3]= ~ A[3];
    assign out[4]= ~ A[4];
    assign out[5]= ~ A[5];
    assign out[6]= ~ A[6];
    assign out[7]= ~ A[7];

    assign flag = {{1'b0},{1'b1},{2{1'b0}},{
    out==0},{out[7]},{3{1'b0}}};
endmodule: bitwise_not
```

Listing 1: Operación Bitwise NOT

```
module bitwise_or(A, B, out, flag);
    input logic [7:0] A;
    input logic [7:0] B;
    output logic [7:0] out;
    output logic [9:0] flag;

    assign out[0]= A[0] | B[0];
    assign out[1]= A[1] | B[1];
    assign out[2]= A[2] | B[2];
    assign out[3]= A[3] | B[3];
    assign out[4]= A[4] | B[4];
    assign out[5]= A[5] | B[5];
    assign out[6]= A[6] | B[6];
    assign out[7]= A[7] | B[7];

    assign flag = {{1'b0},{1'b1},{2{1'b0}},{
    out==0},{out[7]},{3{1'b0}}};
endmodule: bitwise_or
```

Listing 2: Operación Bitwise OR

```
module bitwise_and(A, B, out, flag);
    input logic [7:0] A;
    input logic [7:0] B;
    output logic [7:0] out;
    output logic [9:0] flag;

    assign out[0]= A[0] & B[0];
    assign out[1]= A[1] & B[1];
    assign out[2]= A[2] & B[2];
    assign out[3]= A[3] & B[3];
    assign out[4]= A[4] & B[4];
    assign out[5]= A[5] & B[5];
    assign out[6]= A[6] & B[6];
    assign out[7]= A[7] & B[7];

    assign flag = {{1'b0},{1'b1},{2{1'b0}},{
    out==0},{out[7]},{3{1'b0}}};
endmodule: bitwise_and
```

Listing 3: Operación Bitwise AND

```
module bitwise_xor(A, B, out, flag);
    input logic [7:0] A;
    input logic [7:0] B;
    output logic [7:0] out;
    output logic [9:0] flag;

    assign out[0]= A[0] ∧ B[0];
    assign out[1]= A[1] ∧ B[1];
    assign out[2]= A[2] ∧ B[2];
    assign out[3]= A[3] ∧ B[3];
    assign out[4]= A[4] ∧ B[4];
    assign out[5]= A[5] ∧ B[5];
    assign out[6]= A[6] ∧ B[6];
    assign out[7]= A[7] ∧ B[7];

    assign flag = {{1'b0},{1'b1},{2{1'b0}},{
    out==0},{out[7]},{3{1'b0}}};
endmodule: bitwise_xor
```

Listing 4: Operación Bitwise XOR

Luego, como las operaciones de transposición no son bitwise, todas las operaciones de shift se realizaron con las ya implementadas por Verilog:

```
module lsl(A, B, out, flag);
    input logic [7:0] A;
    input logic [7:0] B;
    output logic [7:0] out;
    output logic [9:0] flag;

    assign flag = {{2{1'b0}},{1'b1},{1'b0},{
    out==0},{out[7]},{3{1'b0}}};
    assign out = A >> B;
endmodule: lsl
```

Listing 5: Operación Logical Shift Left

```
module lsr(A, B, out, flag);
    input logic [7:0] A;
    input logic [7:0] B;
    output logic [7:0] out;
    output logic [9:0] flag;

    assign flag[0] = 0;
    assign flag[1] = 0;
    assign flag[2] = 0;
    assign flag[3] = out[7];
    assign flag[4] = 0;
    assign flag[5] = 0;
    assign flag[6] = 1;
    assign flag[7] = 0;
    assign flag[8] = 0;

    assign flag = {{2{1'b0}},{1'b1},{1'b0},{
    out==0},{out[7]},{3{1'b0}}};
    assign out = A << B;
endmodule: lsr
```

Listing 6: Operación Logical Shift Right

```
module ror(A, B, out, flag);
    input logic[7:0] A;
    input logic[7:0]B;
    output logic [7:0] out;
    output logic[9:0] flag;

    assign out = (A << ('h8-B)) | (A >> 'h8-('h8-B));
  assign flag = {{2{1'b0}},{1'b1},{1'b0},{out==0},{out[7]},{3{1'b0}}};
endmodule: ror
```

Listing 7: Operación Rotate Right

```
module rol(A, B, out, flag);
input logic[7:0] A;
    input logic[7:0]B;
    output logic [7:0] out;
    output logic[9:0] flag;

    assign out = (A << B) | (A >> ('h8-B));
  assign flag = {{2{1'b0}},{1'b1},{1'b0},{out
    ==0},{out[7]},{3{1'b0}}};
```

Listing 8: Operación Rotate Left

```
module asr(A, B, out, flag);
    input signed [7:0] A;
    input logic [7:0] B;
    output signed [7:0] out;
    output logic [9:0] flag;

    assign flag = {{2{1'b0}},{1'b1},{1'b0},{
    out==0},{out[7]},{3{1'b0}}};
    assign out = $ signed(A>>>B);
endmodule: asr
```

Listing 9: Operación Arithmetic Shift Right

Posteriormente, para las operaciones aritméticas también se utilizaron las implementadas por Verilog, a excepción de la suma con carry look ahead. El detalle se puede ver a continuación:

```
module amult(A, B, out, flag);
    input logic [7:0] bin1;
    input logic [7:0] bin2;
    output logic [7:0] out;
    output logic [8:0] flag;

    assign out = bin1*bin2;
    assign flag = {{1'b1},{{3'b0}},{out==0},
    out[7],{1'b0},{1'b1},{1'b0}};
endmodule: amult
```

Listing 10: Operación Multiplicación

```
module division(A, B, out, flag);
    input logic [7:0] A, B;
    output logic [7:0] out;
    output logic [9:0] flag;

    always @B begin
        if (!B) begin
          assign out = 8'b0;
        end else begin
          assign out = A/B;
        end
    end
    assign flag = {{1'b1}, {3{1'b0 }}, {(&B)
    &(out==0)},out[7] , {1'b0} ,{(A[7]&!(&A
    [6:0]))&(B[0]&!(&B[7:1]))}, !(&B)};
endmodule:divsion
```

Listing 11: Operación División

Por otro lado, la sumna con carry lookahead se implementó en base a lo que dice el texto guía [1]

```
module CLA4Bit (A, B, cin, sum, cout);
    input [3:0] A, B;
  input cin;
  output logic [3:0] sum;
  output logic cout;
  wire p0,p1,p2,p3,g0,g1,g2,g3,c1,c2,c3,c4,c0;

    assign p0=(A[0]^B[0]),
    p1=(A[1]^B[1]),
    p2=(A[2]^B[2]),
    p3=(A[3]^B[3]);

    assign g0=(A[0]&B[0]),
    g1=(A[1]&B[1]),
    g2=(A[2]&B[2]),
    g3=(A[3]&B[3]);

  assign c0=cin,
       c1=g0|(p0&cin),
       c2=g1|(p1&g0)|(p1&p0&cin),
      c3=g2|(p2&g1)|(p2&p1&g0)|(p2&p1&p0&cin),
       c4=g3|(p3&g2)|(p3&p2&g1)|(p3&p2&p1&g0)|(p3&p2&p1&p0&cin);

    assign sum[0] = A[0] ^ B[0] ^ cin,
    sum[1] = A[1] ^ B[1] ^ c1,
    sum[2] = A[2] ^ B[2] ^ c2,
    sum[3] = A[3] ^ B[3] ^ c3;

  assign cout=c4;
endmodule: CLA4Bit

module cla_adder(A, B, sum, flag);
    input [7:0] A, B;
    reg [7:0] cin;
    reg cout;
    output logic [7:0] sum;
    output logic [9:0] flag;
    wire c1;
    assign cin = 0;
    CLA4Bit c11(A[3:0],B[3:0],cin,sum[3:0],c1);
    CLA4Bit c22(A[7:4],B[7:4],c1,sum[7:4],cout);

  assign flag = {{1'b1}, {3{1'b0 }},{sum==0}, sum[7] , cout ,{(A[7]==B[7])^cout}, {1'b0}};
endmodule: cla_adder
```

Listing 12: Suma con Carry Lookahead

[1] Sarah Harris and David Harris. 2015. Digital Design and Computer Architecture: ARM Edition (1st. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. Section 5.2.1, Page 241.

Para la resta con ripple carry se ajustó un sumador con ripple carry, el cual utiliza un sumador completo de manera auxiliar:

```
module ripple_carry_subtractor(A, B, out, flag);
  output [7:0] out;
  output logic [8:0] flag;
  input [7:0] A;
  input [7:0] B;
  reg Op = 1;
  reg C, V;
  wire C0,C1,C2,C3,C4,C5,C6,C7;
    wire B0,B1,B2,B3,B4,B5,B6,B7;

    xor(B0, B[0], Op);
    xor(B1, B[1], Op);
    xor(B2, B[2], Op);
    xor(B3, B[3], Op);
    xor(B4, B[4], Op);
    xor(B5, B[5], Op);
    xor(B6, B[6], Op);
    xor(B7, B[7], Op);
    xor(C, !C3, Op);
    xor(V, !C3, C2);

  full_adder fa0(out[0], C0, A[0], B0, Op);
  full_adder fa1(out[1], C1, A[1], B1, C0);
  full_adder fa2(out[2], C2, A[2], B2, C1);
  full_adder fa3(out[3], C3, A[3], B3, C2);
  full_adder fa4(out[4], C4, A[4], B4, C3);
  full_adder fa5(out[5], C5, A[5], B5, C4);
  full_adder fa6(out[6], C6, A[6], B3, C5);
  full_adder fa7(out[7], C7, A[7], B7, C6);

  assign flag = {{1'b1}, {3{1'b0 }},!(&out), out[7] , C ,V, {1'b0 }};
endmodule: ripple_carry_subtractor

module full_adder(S, Cout, A, B, Cin);
   output S;
   output Cout;
   input  A;
   input  B;
   input  Cin;
   wire   w1,w2,w3,w4;
   xor(w1, A, B);
   xor(S, Cin, w1);
   and(w2, A, B);
   and(w3, A, Cin);
   and(w4, B, Cin);
   or(Cout, w2, w3, w4);
endmodule:full_adder
```

Listing 13: Resta con Ripple Carry

Por último, para las operaciones del torneo se utilizaron las operaciones ya implementadas por verilog para así calcular el poder absoluto de cada jugador:

```
module pLigero(height_A, AGI_A, height_B, AGI_B, winner, flag);
  input [7:0] height_A, AGI_A, height_B, AGI_B;
  output [7:0] winner;
  output [8:0] flag;

  assign winner = ((height_A/AGI_A) + (100/height_A) + AGI_A) > ((height_B/AGI_B) + (100/
    height_B) + AGI_B) ? 8'h00 : 9'hff;

  assign flag = winner[0] == 1'b1 ? 9'h028 : 9'h020;


endmodule : pLigero


module pPesado(weight_A, RES_A, weight_B, RES_B, winner, flag);
  input [7:0] weight_A, RES_A, weight_B, RES_B;
  output [7:0] winner;
  output [8:0] flag;

  assign winner = ((5*weight_A) + (2*RES_A)) > ((5*weight_B) + (2*RES_B)) ? 8'h00 : 9'hff;
  assign flag = winner[0] == 1'b1 ? 9'h028 : 9'h020;

endmodule : pPesado

module pMixto(height_A, AGI_A, weight_A, STR_A, RES_A, height_B, AGI_B, weight_B, STR_B,
    RES_B, winner, flag);
  input [7:0] height_A, AGI_A, weight_A, STR_A, RES_A, height_B, AGI_B, weight_B, STR_B,
    RES_B;
  output [7:0] winner;
  output [8:0] flag;

  assign winner = ((height_A/AGI_A) + (3*weight_A) + ((STR_A+AGI_A+RES_A)/3)) > ((height_B/
    AGI_B) + (3*weight_B) + ((STR_B+AGI_B+RES_B)/3)) ? 8'h00 : 9'hff;
  assign flag = winner[0] == 1'b1 ? 9'h028 : 9'h020;

endmodule : pMixto
```

Listing 14: Operaciones del torneo

Posteriormente a las operaciones, se implementó la ROM. Dado que la entrada de datos a la ALU se hará cargando la ROM, y esta puede tener a lo más 16 operaciones, se diseñó una ROM que pudiera almacenar 16 registros de 4 bits cada uno, correspondientes a las instrucciones.

Luego se realizó el Program Counter (PC) lo cual es una especie de contador que itera sobre la rom y así acceder a las instrucciones. El detalle se muestra a continuación:

```
module rom(cs,addr,data);
  input[3:0]addr;
  input cs;
  output reg[3:0]data;
  logic [3:0] mem [15:0];

  initial begin
    mem[0] = 4'h0;
    mem[1] = 4'hf;
    mem[2] = 4'h3;
    mem[3] = 4'h4;
    mem[4] = 4'hd;
    mem[5] = 4'hd;
    mem[6] = 4'hf;
    mem[7] = 4'hf;
    mem[8] = 4'hf;
    mem[9] = 4'hf;
    mem[10] = 4'hf;
    mem[11] = 4'h2;
    mem[12] = 4'ha;
    mem[13] = 4'h9;
    mem[14] = 4'hb;
    mem[15] = 4'h3;
  end
always@(cs or addr)
begin
  data = mem[addr];
end
endmodule :rom

module PC(clk,reset,count);
  input clk,reset;
  output reg [3:0] count;
  always@(posedge clk)
  begin
    if(reset)
      count <= 0;
    else
      count <= count + 1;
  end
endmodule :PC
```

Listing 15: Gestión de Memoria

4. Resultados

A continuación se muestran diversos resultados del testbench:

- Prueba General

    1. Suma
    2. Resta
    3. División entera
    4. Bitwise NOT
    5. Peso ligero
    6. Peso mixto
    7. Peso mixto
    8. Peso mixto
    9. Peso mixto
    10. Peso mixto
    11. Multiplicación
    12. Arithemetic shift right
    13. Logical shift right
    14. Rotate left
    15. División entera

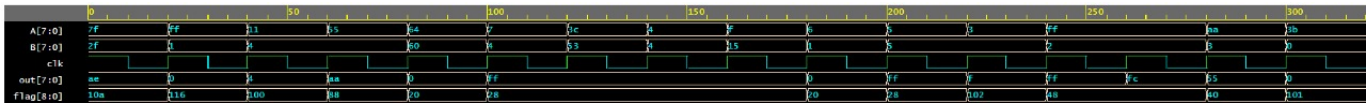

Figura 1: Prueba

5. Análisis

A continuación se discuten más en detalle los resultados presentados en la sección anterior:

- Prueba General:

1. Suma: se puede ver que el resultado es correcto dado que ese el resultado de la suma, y el flag tiene activado que es una operación aritmética, que el resultado es negativo y que tiene overflow, lo que corresponde con el resultado obtenido.
2. Suma: se puede ver que el resultado es correcto dado que ese el resultado de la suma, y el flag tiene activado que es una operación aritmética, que el resultado es negativo y que tiene carry y que tiene overflow, lo que corresponde con el resultado obtenido.
3. División entera: se puede ver que el resultado es correcto dado que ese el resultado de la división considerando que es solo la parte entera, y el flag tiene activado solo que es una operación aritmética, lo que corresponde con el resultado obtenido.
4. Bitwise not: se puede ver que el resultado es correcto dado que ese el resultado de negar todos los bits, y el flag tiene activado que es una operación lógica y que el resultado es negativo, lo cual corresponde con el resultado.
5. Peso ligero: se puede ver que el poder absoluto del segundo jugador es mayor por lo que el resultado es ff, y la flag tiene activado que es una operación de torneo es un resultado negativo, lo cual corresponde con el resultado
6. Multiplicación: se puede ver que el resultado es correcto dado que ese el resultado de la multiplicación, y el flag tiene activado que es una operación aritmética y que genera overflow, lo que corresponde al resultado.
7. Arithmetic shift right: se puede ver que el resultado es correcto dado que ese el resultado de un shift conservando el signo, y el flag tiene activado que es una operación lógica y que el resultado es negativo, lo que corresponde al resultado
8. Logical shift right: el resultado está incorrecto
9. Rotate left: se puede ver que el resultado es correcto dado que ese el resultado de una rotación a la izquierda, y el flag tiene activado que es una operación lógica lo que corresponde al resultado.
10. División entera: se puede ver que el resultado es correcto dado que al dividir por 0 el output debiese ser 0, y el flag tiene activado que es una operación aritmética y que genera error por división por 0, lo que corresponde al resultado.

6. Conclusión

En esta tarea nos enfrentamos con varias dificultades, pero la principal de todas fue que no conociamos como funcionaba de manera correcta (o incluso incorrecta) system verilog, por lo que toda informacion tuvo que ser buscada, junto con eso buscar el funcionamiento de PC y ROM, y como estos se usan en system verilog. El grado de completitud de esta tarea no fue el 100 % ya que el tema del carry vs overflow nos complico, cosa que se preguntó en el foro el miércoles pero no hubo respuesta, por lo que asumimos cosas sobre este. Además, el módulo lsr (logical shift right) funciona bien por si solo, mas no del todo en la simulacion. Junto con eso, los resultados de las peleas los daba de manera correcta, utilizando el clock dentro de cada uno de esos módulos.