Tarea 3 INF245:
Implementación de ALU mediante el lenguaje de descripción de hardware Verilog

20 de junio de 2022

Integrantes:
Iván Cano ROL: 202073543-2
Claudio Inal ROL: 201873060-2

1. Resumen

En este informe se detallará el procedimiento para la creación de una ALU con diferentes operaciones controlada por una lógica secuencial que será capaz de entregar resultados según operaciones definidas en el enunciado de esta tarea. Tanto las operaciones como la ALU final serán descritas mediante una HDL llamada Verilog, y simuladas en la plataforma EDA Playground con diferentes Testbenchs (pruebas).

Los resultados de estas simulaciones se muestran a continuación:

$$\frac{Resultados\ exitosos}{Resultados\ totales} = \frac{16}{16}$$

$$\frac{Flags\ exitosos}{Flags\ totales} = \frac{16}{16}$$

2. Introducción

Según lo descrito en el enunciado de la tarea, debemos construir una ALU que sea capaz de calcular diferentes operaciones, por lo que para empezar describiremos estas operaciones con Verilog mediante módulos que tendrán la lógica capaz de satisfacer esa demanda.

Luego de definir dichas operaciones, deberemos crear contadores, un controlador de una ROM y multiplexores para poder obtener la lógica que iterará sobre unas instrucciones guardadas en un archivo que simulará nuestra ROM.

Hecho lo anterior podemos ensamblar nuestra ALU en forma de un MUX de 16 opciones, con inputs que serán las operaciones ya calculadas y variables de control que guiarán el comportamiento del MUX, y como output tendrá ciertas flags que determinarán el estado de la ALU.

Para finalizar se diseñará una Testbench para simular el funcionamiento de esta implementación, con diferentes test que ultimamente definirá la validez de esta implementación.

3. Desarrollo

Comenzaremos por implementar las operaciones que serán parte de una ALU, para luego implementar esta ALU mediante el uso de multiplexores, contadores, ROM y un reloj.

3.1. Diseño

3.1.1. Operaciones aritméticas

Antes de proceder con la elaboración de las operaciones, necesitamos unos módulos que serán auxiliares para la construcción de los módulos mas complejos, como lo son los sumadores completos de 2 bits (Full Adder), la lógica del Carry Look-ahead para 4 bits e incorporar esa lógica a un sumador CLA de dos entradas de 4 bits:

Como se muestra en la figura 1, tenemos el full adder para un par de entradas de 1 bit, junto con el carry de entrada, la salida G corresponde al carry Generate, P corresponde al carry Propagate, y cout es el carry de salida.
En la figura 2 se muestra la lógica para calcular los carry de salida de cada módulo full adder, notar que estos dependen exclusivamente de las entradas originales, y son calculadas con la ecuación de recurrencia $C_i = G_i + P_i \times C_{i-1}$, calculada hasta el cuarto bit.
En la figura 3 tenemos un adder de 4 bits, en donde se hace uso del full adder en conjunto con el CLA de 4 bits. Las entradas corresponden al par de números de 4 bits y el carry de entrada, las salidas corresponden al resultado de la suma junto con el carry de salida.

```
module FullAdder(
    input logic a, b, cin,
    output logic g, p, s,cout);

    assign g = a & b;
    assign p = (a ^ b) & cin;
    assign s = cin ^ (a ^ b);
    assign cout = g|p;

endmodule
```

Figura 1: Full adder

```
module FourLALogic(
    input logic cin,
    input logic [3:0] g, p,
    output logic [3:0] c);

    assign c[0] = g[0] | (p[0] & cin);
    assign c[1] = g[1] | (p[1] & g[0]) |
      (p[1] & p[0] & cin);
    assign c[2] =
      g[2] | (p[2] & g[1]) |
      (p[2] & p[1] & g[0]) |
      (p[2] & p[1] & p[0] & cin);
    assign c[3] =
      g[3] | (p[3] & g[2]) |
      (p[3] & p[2] & g[1]) |
      (p[3] & p[2] & p[1] & g[0]) |
      (p[3] & p[2] & p[1] & p[0] & cin);

endmodule
```

Figura 2: CLA 4 bits

```
module FourLA(
    input logic [3:0] a, b,
    input logic cin,
    output logic [3:0] s,
    output logic cout);

    logic [3:0] g, p,c;

    FullAdder fa0(a[0], b[0], cin, g[0], p[0], s[0]);
    FullAdder fa1(a[1], b[1], c[0], g[1], p[1], s[1]);
    FullAdder fa2(a[2], b[2], c[1], g[2], p[2], s[2]);
    FullAdder fa3(a[3], b[3], c[2], g[3], p[3], s[3]);
    FourLALogic cla(cin, g[3:0], p[3:0], c[3:0]);
    assign cout = c[3];

endmodule
```

Figura 3: Adder 4 bits

- Suma con Carry Look-ahead Adder (8 bits): Para implementar esta operación, haremos uso de 2 adder CLA de 4 bits, en donde el primero se encargará de los primeros 4 bits, y el segundo de los últimos 4. Las salidas corresponden al resultado de la suma, el carry de salida y una flag que indica si hay overflow.

```
module LookAheadAdder8(
    input logic [7:0] a, b,
    input logic cin,
    output logic [7:0] s,
    output logic cout,v);

    logic cout1;
    FourLA LA30(a[3:0],b[3:0],cin,s[3:0],cout1);
    FourLA LA74(a[7:4],b[7:4],cout1,s[7:4],cout);
    assign v =~a[7]&~b[7]&s[7]|a[7]&b[7]&~s[7]|a[7]&~b[7]&~s[7]|~a[7]&b[7]&s[7];

endmodule
```

Figura 4: Suma CLA 8 bits

- Resta con Ripple-carry Adder: Para implementar esta operación, haremos uso de los módulos Full Adder y Bitwise NOT (definido más adelante). Tenemos las entradas a y b de 8 bits cada una, esta operación calculará $a - b = a + \overline{b} + 1$, por lo que aplicaremos Bitwise NOT (complemento 1) a b y sumaremos 1 para obtener el resultado. Las salidas corresponden a el resultado, el signo y la flag.

```
module RestaRCA(
  input logic [7:0]a,b,
  input logic cin,
  output logic [7:0] s,
  output logic cout,v);

  logic [7:0] c,x;
  bitwiseNot Nt(b,x);
  FullAdder R0(a[0],x[0],cin,,,s[0],c[0]);
  FullAdder R[7:1](a[7:1],x[7:1],c[6:0],,,s[7:1],c[7:1]);
  assign cout=c[7];
  assign v =~a[7]&~b[7]&s[7]|a[7]&b[7]&~s[7];

endmodule
```

Figura 5: Resta RCA 8 bits

- Multiplicación: Para implementar esta operación, haremos uso del módulo FourLA (sumador con carry Look-ahead de 4 bits). Simulando el algoritmo de multiplicación común, iremos iterando por cada bit de una de las entradas, multiplicando al número binario de la otra entrada, haremos Left logical shift por cada resultado obtenido, y sumaremos las respuestas para obtener el resultado final.

```
module Multiplier(
  input logic [7:0] a,b,
  output logic [7:0] x,
  output logic V);

  logic[3:0] t,t1,t2,t3,p,p1,p2;
  logic[3:0] ab2,ab3;
  logic cin;
  assign cin=0;

  assign V=a[7]|a[6]|a[5]|b[4]|b[7]|b[6]|b[5]|b[4];

  assign t[3]=0;
  assign t[2]=a[3]&b[0];
  assign t[1]=a[2]&b[0];
  assign t[0]=a[1]&b[0];

  assign t1[3]=a[3]&b[1];
  assign t1[2]=a[2]&b[1];
  assign t1[1]=a[1]&b[1];
  assign t1[0]=a[0]&b[1];

  FourLA f1(t,t1,cin,p,t2[3]);
  assign t2[2]=p[3];
  assign t2[1]=p[2];
  assign t2[0]=p[1];
```

Figura 6: Multiplicación (1)

```
  FourLA f2(ab2,t2,cin,p1,t3[3]);

  assign t3[2]=p1[3];
  assign t3[1]=p1[2];
  assign t3[0]=p1[1];
  assign ab3[3]=a[3]&b[3];
  assign ab3[2]=a[2]&b[3];
  assign ab3[1]=a[1]&b[3];
  assign ab3[0]=a[0]&b[3];
  FourLA f3(ab3,t3,cin,p2,x[7]);

  assign x[0]=a[0]&b[0];
  assign x[1]=p[0];
  assign x[2]=p1[0];
  assign x[3]=p2[0];
  assign x[4]=p2[1];
  assign x[5]=p2[2];
  assign x[6]=p2[3];

endmodule
```

Figura 7: Multiplicación (2)

3.1.2. Operaciones lógicas

- Bitwise NOT: Recibe una entrada de 8 bits, y retorna la negación bit a bit de la entrada original.
- Bitwise OR: Recibe dos entradas de 8 bits, y retorna la operación OR aplicada a cada par de bits de la entrada original en la posición respectiva.
- Bitwise AND: Recibe dos entradas de 8 bits, y retorna la operación AND aplicada a cada par de bits de la entrada original en la posición respectiva.
- Bitwise XOR: Recibe dos entradas de 8 bits, y retorna la operación XOR aplicada a cada par de bits de la entrada original en la posición respectiva.

```
module bitwiseNot(
  input logic[7:0] a,
  output logic[7:0] x);
  assign x=~a;
endmodule
```

Figura 8: NOT

```
module bitwiseOr(
  input logic[7:0] a,b,
  output logic[7:0] x);
  assign x=a|b;
endmodule
```

Figura 9: OR

```
module bitwiseAnd(
  input logic[7:0] a,b,
  output logic[7:0] x);
  assign x=a&b;
endmodule
```

Figura 10: AND

```
module bitwiseXor(
  input logic[7:0] a,b,
  output logic[7:0] x);
  assign x=a^b;
endmodule
```

Figura 11: XOR

3.1.3. Operaciones de transposición

Para implementar estas operaciones haremos uso de un módulo auxiliar, el shifter:

```
module Shifter(
  input logic s,x,x1,
  output logic y);
  assign y=(~s&x)|(s&x1);
endmodule
```

Figura 12: Shifter

- Logical Shift Left: las entradas son a y b en binario, el módulo mueve o shiftea los bits de a hacia la izquierda b veces, llenando con ceros los bits menos significativos y devolviendo ese resultado.

```
module LogicShiftL(
  input logic [7:0] a,b,
  output logic [7:0] y);
  logic [7:0] t,t1,t2,t3;
  logic s1,s2,s3,s4,zero;
  assign zero=0;
  assign s1=b[0];
  assign s2=b[1];
  assign s3=b[2];
  assign s4=b[3]|b[4]|b[5]|b[6]|b[7];

  Shifter zs0(s1,a[0],zero,t[0]);
  Shifter zs[7:1](s1,a[7:1],a[6:0],t[7:1]);

  Shifter os0(s2,t[0],zero,t1[0]);
  Shifter os1(s2,t[1],zero,t1[1]);
  Shifter os[7:2](s2,t[7:2],t[5:0],t1[7:2]);
```

Figura 13: LSL (1)

```
  Shifter ts0(s3,t1[0],zero,t2[0]);
  Shifter ts1(s3,t1[1],zero,t2[1]);
  Shifter ts2(s3,t1[2],zero,t2[2]);
  Shifter ts3(s3,t1[3],zero,t2[3]);
  Shifter ts[7:4](s3,t1[7:4],t1[3:0],t2[7:4]);

  assign t3[7]=t2[7]&~s4;
  assign t3[6]=t2[6]&~s4;
  assign t3[5]=t2[5]&~s4;
  assign t3[4]=t2[4]&~s4;
  assign t3[3]=t2[3]&~s4;
  assign t3[2]=t2[2]&~s4;
  assign t3[1]=t2[1]&~s4;
  assign t3[0]=t2[0]&~s4;

  assign y=t3;
endmodule
```

Figura 14: LSL (2)

- Logical Shift Right: las entradas son a y b en binario, el módulo mueve o shiftea los bits de a hacia la derecha b veces, llenando con ceros los bits mas significativos y devolviendo ese resultado.

```
module LogicShiftR(
  input logic [7:0] a,b,
  output logic [7:0] y);

  logic [7:0] t,t1,t2,t3;
  logic s1,s2,s3,s4;

  assign s1=b[0];
  assign s2=b[1];
  assign s3=b[2];
  assign s4=b[3]|b[4]|b[5]|b[6]|b[7];

  Shifter zs7(s1,a[7],0,t[7]);
  Shifter zs[6:0](s1,a[6:0],a[7:1],t[6:0]);

  Shifter os7(s2,t[7],0,t1[7]);
  Shifter os6(s2,t[6],0,t1[6]);
  Shifter os[5:0](s2,t[5:0],t[7:2],t1[5:0]);
```

Figura 15: LSR (1)

```
  Shifter os7(s2,t[7],0,t1[7]);
  Shifter os6(s2,t[6],0,t1[6]);
  Shifter os[5:0](s2,t[5:0],t[7:2],t1[5:0]);

  Shifter ts7(s3,t1[7],0,t2[7]);
  Shifter ts6(s3,t1[6],0,t2[6]);
  Shifter ts5(s3,t1[5],0,t2[5]);
  Shifter ts4(s3,t1[4],0,t2[4]);
  Shifter ts[3:0](s3,t1[3:0],t1[7:4],t2[3:0]);

  assign t3[7]=t2[7]&~s4;
  assign t3[6]=t2[6]&~s4;
  assign t3[5]=t2[5]&~s4;
  assign t3[4]=t2[4]&~s4;
  assign t3[3]=t2[3]&~s4;
  assign t3[2]=t2[2]&~s4;
  assign t3[1]=t2[1]&~s4;
  assign t3[0]=t2[0]&~s4;

  assign y=t3;

endmodule
```

Figura 16: LSR (2)

- Arithmetic Shift Right: Similar al LSR, solo que esta vez el bit más significativo de a se mantiene en la salida.

```
module ArShiftR(
  input logic [7:0] a,b,
  output logic [7:0] y);

  logic [7:0] t,t1,t2,t3;
  logic s1,s2,s3,s4;

  assign s1=b[0];
  assign s2=b[1];
  assign s3=b[2];
  assign s4=b[3]|b[4]|b[5]|b[6]|b[7];

  Shifter zs7(s1,a[7],a[7],t[7]);
  Shifter zs[6:0](s1,a[6:0],a[7:1],t[6:0]);

  Shifter os7(s2,t[7],t[7],t1[7]);
  Shifter os6(s2,t[6],t[7],t1[6]);
  Shifter os[5:0](s2,t[5:0],t[7:2],t1[5:0]);
```

Figura 17: ARS (1)

```
  Shifter ts7(s3,t1[7],t1[7],t2[7]);
  Shifter ts6(s3,t1[6],t1[7],t2[6]);
  Shifter ts5(s3,t1[5],t1[7],t2[5]);
  Shifter ts4(s3,t1[4],t1[7],t2[4]);
  Shifter ts[3:0](s3,t1[3:0],t1[7:4],t2[3:0]);

  assign t3[7]=t2[7]&~s4;
  assign t3[6]=t2[6]&~s4;
  assign t3[5]=t2[5]&~s4;
  assign t3[4]=t2[4]&~s4;
  assign t3[3]=t2[3]&~s4;
  assign t3[2]=t2[2]&~s4;
  assign t3[1]=t2[1]&~s4;
  assign t3[0]=t2[0]&~s4;

  assign y=t3;
endmodule
```

Figura 18: ARS (2)

- Rotate left: Similar al LSL, pero no se pierden los bits que quedan a la izquierda, si no que por cada shift, el bit más significativo toma la posición del bit menos significativo, devolviendo ese resultado.

```
module RotateL(
  input logic[7:0] a,b,
  output logic[7:0] y);

  logic [7:0] t,t1,t2;
  logic s1,s2,s3,s4;

  assign s1=b[0];
  assign s2=b[1];
  assign s3=b[2];

  Shifter sh0(s1,a[0],a[7],t[0]);
  Shifter sh1(s1,a[1],a[0],t[1]);
  Shifter sh2(s1,a[2],a[1],t[2]);
  Shifter sh3(s1,a[3],a[2],t[3]);
  Shifter sh4(s1,a[4],a[3],t[4]);
  Shifter sh5(s1,a[5],a[4],t[5]);
  Shifter sh6(s1,a[6],a[5],t[6]);
  Shifter sh7(s1,a[7],a[6],t[7]);
```

Figura 19: RL (1)

```
  Shifter st0(s2,t[0],t[6],t1[0]);
  Shifter st1(s2,t[1],t[7],t1[1]);
  Shifter st2(s2,t[2],t[0],t1[2]);
  Shifter st3(s2,t[3],t[1],t1[3]);
  Shifter st4(s2,t[4],t[2],t1[4]);
  Shifter st5(s2,t[5],t[3],t1[5]);
  Shifter st6(s2,t[6],t[4],t1[6]);
  Shifter st7(s2,t[7],t[5],t1[7]);

  Shifter sz0(s3,t1[0],t1[4],t2[0]);
  Shifter sz1(s3,t1[1],t1[5],t2[1]);
  Shifter sz2(s3,t1[2],t1[6],t2[2]);
  Shifter sz3(s3,t1[3],t1[7],t2[3]);
  Shifter sz4(s3,t1[4],t1[0],t2[4]);
  Shifter sz5(s3,t1[5],t1[1],t2[5]);
  Shifter sz6(s3,t1[6],t1[2],t2[6]);
  Shifter sz7(s3,t1[7],t1[3],t2[7]);

  assign y=t2;

endmodule
```

Figura 20: RL (2)

- Rotate right: Similar al LSR, pero no se pierden los bits que quedan a la derecha, si no que por cada shift, el bit menos significativo toma la posición del bit más significativo, devolviendo ese resultado.

```
module RotateR(
  input logic[7:0] a,b,
  output logic[7:0] y);

  logic [7:0] t,t1,t2;
  logic s1,s2,s3,s4;

  assign s1=b[0];
  assign s2=b[1];
  assign s3=b[2];

  Shifter sh0(s1,a[0],a[1],t[0]);
  Shifter sh1(s1,a[1],a[2],t[1]);
  Shifter sh2(s1,a[2],a[3],t[2]);
  Shifter sh3(s1,a[3],a[4],t[3]);
  Shifter sh4(s1,a[4],a[5],t[4]);
  Shifter sh5(s1,a[5],a[6],t[5]);
  Shifter sh6(s1,a[6],a[7],t[6]);
  Shifter sh7(s1,a[7],a[0],t[7]);
```

Figura 21: RR (1)

```
  Shifter st0(s2,t[0],t[2],t1[0]);
  Shifter st1(s2,t[1],t[3],t1[1]);
  Shifter st2(s2,t[2],t[4],t1[2]);
  Shifter st3(s2,t[3],t[5],t1[3]);
  Shifter st4(s2,t[4],t[6],t1[4]);
  Shifter st5(s2,t[5],t[7],t1[5]);
  Shifter st6(s2,t[6],t[0],t1[6]);
  Shifter st7(s2,t[7],t[1],t1[7]);

  Shifter sz0(s3,t1[0],t1[4],t2[0]);
  Shifter sz1(s3,t1[1],t1[5],t2[1]);
  Shifter sz2(s3,t1[2],t1[6],t2[2]);
  Shifter sz3(s3,t1[3],t1[7],t2[3]);
  Shifter sz4(s3,t1[4],t1[0],t2[4]);
  Shifter sz5(s3,t1[5],t1[1],t2[5]);
  Shifter sz6(s3,t1[6],t1[2],t2[6]);
  Shifter sz7(s3,t1[7],t1[3],t2[7]);

  assign y=t2;

endmodule
```

Figura 22: RR (2)

3.1.4. Operaciones del torneo

Para implementar los módulos de esta sección es necesario tener unos módulos auxiliares que no definiremos aquí para mantener orden en el informe, los cuales corresponden a registros (flip-flops), MUX, MUX5 y contadores. La utilidad de estos módulos viene dada por la forma en que se ingresan los datos de cada participantes, la cual es secuencial y dependiendo del ciclo en el que estemos, deberemos ingresar un dato distinto. Una vez terminen los ciclos se entregará el resultado del módulo correspondiente.

Estos módulos determinarán si el competidor A supera en poder al competidor B, de ocurrir lo primero entregará 0x00, en otro caso entregará 0xff. Esto se puede determinar restando el poder de A con el del B, llenando la salida de 8 bits con el bit más significativo, la cual determina el signo del resultado.

Cada módulo tiene un reset para reinicializar las iteraciones.

- A y B son peso ligero:

```
module PesoLigero(
  input logic [7:0] a,b,
  input logic clk,rst,
  output logic [7:0] w,poder1,poder2,agi1,agi2,est1,est2,
  output logic c);

  reg[7:0] wr,o0,o1;
  counter1 con(clk,rst,c);
  deMux dmx(a,b,c,o0,o1,agi1,agi2);
  register regi(o0,o1,~c,clk,,est1,est2);

  assign poder1=((est1/agi1) + (100/est1) + agi1);
  assign poder2=((est2/agi2) + (100/est2) + agi2);
```

Figura 23: Cálculo peso ligero (1)

```
  assign poder1=((est1/agi1) + (100/est1) + agi1);
  assign poder2=((est2/agi2) + (100/est2) + agi2);

  assign wr=(poder2-poder1);

  assign w[7]= ~wr[7]&c;
  assign w[6]= ~wr[7]&c;
  assign w[5]= ~wr[7]&c;
  assign w[4]= ~wr[7]&c;
  assign w[3]= ~wr[7]&c;
  assign w[2]= ~wr[7]&c;
  assign w[1]= ~wr[7]&c;
  assign w[0]= ~wr[7]&c;

endmodule
```

Figura 24: Cálculo peso ligero (2)

- A y B son peso pesado:

```
module PesoPesado(
input logic [7:0] a,b,
  input logic clk,rst,
  output logic [7:0] w,poder1,poder2,res1,res2,peso1,peso2,

  output logic c);

  reg[7:0] wr,o0,o1;
  counter1 con(clk,rst,c);
  deMux dmx(a,b,c,o0,o1,res1,res2);
  register regi(o0,o1,~c,clk,,peso1,peso2);

  //logic [7:0]c
  assign poder1=(5*peso1 + 2*res1);
  assign poder2=(5*peso2 + 2*res2);
```

Figura 25: Cálculo peso pesado (1)

```
  assign wr=(poder2-poder1);

  assign w[7]= ~wr[7]&c;
  assign w[6]= ~wr[7]&c;
  assign w[5]= ~wr[7]&c;
  assign w[4]= ~wr[7]&c;
  assign w[3]= ~wr[7]&c;
  assign w[2]= ~wr[7]&c;
  assign w[1]= ~wr[7]&c;
  assign w[0]= ~wr[7]&c;

endmodule
```

Figura 26: Cálculo peso pesado (2)

- A y B son pesos distintos:

```
module PesoMixto(
input logic [7:0] a,b,
  input logic clk,rst,
  output logic [7:0] w,poder1,poder2,est1,est2,agi1,agi2,peso1,peso2,str1,str2,res1,res2,

  output logic[2:0] c,
    output logic s);

  reg[7:0] wr,o0,o1,o2,o3,o4,o5,o6,o7;
  reg s1,s2,s3,clr;
  assign clr=0;
  assign s=~c[2]&~c[1]&~c[0];
  assign s1=~c[2]&~c[1]&c[0];
  assign s2=~c[2]&c[1]&~c[0];
  assign s3=~c[2]&c[1]&c[0];

  counter3 con(clk,rst,c);
  deMux5 dmx(a,b,c,o0,o1,o2,o3,o4,o5,o6,o7,res1,res2);
```

Figura 27: Cálculo peso mixto (1)

```
  assign poder1=((est1/agi1) + 3*peso1+((str1+agi1+res1)/3));
  assign poder2=((est2/agi2) + 3*peso2+((str2+agi2+res2)/3));


  assign wr=(poder2-poder1);

  assign w[7]= ~wr[7]&c[2];
  assign w[6]= ~wr[7]&c[2];
  assign w[5]= ~wr[7]&c[2];
  assign w[4]= ~wr[7]&c[2];
  assign w[3]= ~wr[7]&c[2];
  assign w[2]= ~wr[7]&c[2];
  assign w[1]= ~wr[7]&c[2];
  assign w[0]= ~wr[7]&c[2];

endmodule
```

Figura 28: Cálculo peso mixto (2)

3.1.5. ALU

Ahora podemos definir lo que será el control del ALU mediante un módulo que funciona como un multiplexor con 16 opciones, controlados con una entrada de 4 bits (las cuales representan la operación a realizar según fueron definidas en el enunciado de la tarea).

Entrega el resultado en formato 8 bits, y una serie de flags como se definieron en el enunciado de la tarea en un output de 12 bits (los bits 11, 10 y 9 no se utilizan).

```
module MuxAluContrl(
  input logic[7:0] suma,resta,mult,div,
  notv,orv,andv,xorv,lsl,lsr,asr,rol,
  ror,pesol,pesop,pesom,
  input logic[3:0] ctrl,
  input logic carry,over,E,
  output logic[7:0] S,
  output logic[11:0] flags);

  logic[7:0]ctrl3,ctrl2,ctrl1,ctrl0;
  assign ctrl3[7]=ctrl[3];
  assign ctrl3[6]=ctrl[3];
  assign ctrl3[5]=ctrl[3];
  assign ctrl3[4]=ctrl[3];
  assign ctrl3[3]=ctrl[3];
  assign ctrl3[2]=ctrl[3];
  assign ctrl3[1]=ctrl[3];
  assign ctrl3[0]=ctrl[3];
  assign ctrl2[7]=ctrl[2];
  assign ctrl2[6]=ctrl[2];
  assign ctrl2[5]=ctrl[2];
  assign ctrl2[4]=ctrl[2];
  assign ctrl2[3]=ctrl[2];
  assign ctrl2[2]=ctrl[2];
  assign ctrl2[1]=ctrl[2];
  assign ctrl2[0]=ctrl[2];
  assign ctrl1[7]=ctrl[1];
  assign ctrl1[6]=ctrl[1];
  assign ctrl1[5]=ctrl[1];
  assign ctrl1[4]=ctrl[1];
  assign ctrl1[3]=ctrl[1];
  assign ctrl1[2]=ctrl[1];
  assign ctrl1[1]=ctrl[1];
  assign ctrl1[0]=ctrl[1];
  assign ctrl0[7]=ctrl[0];
  assign ctrl0[6]=ctrl[0];
  assign ctrl0[5]=ctrl[0];
  assign ctrl0[4]=ctrl[0];
  assign ctrl0[3]=ctrl[0];
  assign ctrl0[2]=ctrl[0];
  assign ctrl0[1]=ctrl[0];
  assign ctrl0[0]=ctrl[0];
```

Figura 29: MUX ALU (1)

```
  assign S= (suma& ~ctrl3&~ctrl2&~ctrl1&~ctrl0)|
    (resta& ~ctrl3&~ctrl2&~ctrl1&ctrl0)|
    (mult& ~ctrl3&~ctrl2&ctrl1&~ctrl0)|
    (div& ~ctrl3&~ctrl2&ctrl1&ctrl0)|
    (notv& ~ctrl3&ctrl2&~ctrl1&~ctrl0)|
    (orv& ~ctrl3&ctrl2&~ctrl1&ctrl0)|
    (andv& ~ctrl3&ctrl2&ctrl1&~ctrl0)|
    (xorv& ~ctrl3&ctrl2&ctrl1&ctrl0)|
    (lsl& ctrl3&~ctrl2&~ctrl1&~ctrl0)|
    (lsr& ctrl3&~ctrl2&~ctrl1&ctrl0)|
    (asr& ctrl3&~ctrl2&ctrl1&~ctrl0)|
    (rol& ctrl3&~ctrl2&ctrl1&ctrl0)|
    (ror& ctrl3&ctrl2&~ctrl1&~ctrl0)|
    (pesol& ctrl3&ctrl2&~ctrl1&ctrl0)|
    (pesop& ctrl3&ctrl2&ctrl1&~ctrl0)|
    (pesom& ctrl3&ctrl2&ctrl1&ctrl0);

  assign flags[11]=0;
  assign flags[10]=0;
  assign flags[9]=0;
  assign flags[8]= (~ctrl3&~ctrl2&~ctrl1&~ctrl0)|
    (~ctrl3&~ctrl2&~ctrl1&ctrl0)|
    (~ctrl3&~ctrl2&ctrl1&~ctrl0)|
    (~ctrl3&~ctrl2&ctrl1&ctrl0);
  assign flags[7]=(~ctrl3&ctrl2&~ctrl1&~ctrl0)|
    (~ctrl3&ctrl2&~ctrl1&ctrl0)|
    (~ctrl3&ctrl2&ctrl1&~ctrl0)|
  (~ctrl3&ctrl2&ctrl1&ctrl0);
  assign flags[6]=(ctrl3&~ctrl2&~ctrl1&~ctrl0)|
    (ctrl3&~ctrl2&~ctrl1&ctrl0)|
    (ctrl3&~ctrl2&ctrl1&~ctrl0)|
    (ctrl3&~ctrl2&ctrl1&ctrl0)|
    (ctrl3&ctrl2&~ctrl1&~ctrl0);
  assign flags[5]=(ctrl3&ctrl2&~ctrl1&ctrl0)|
    (ctrl3&ctrl2&ctrl1&~ctrl0)|
    (ctrl3&ctrl2&ctrl1&ctrl0);
  assign flags[4]= ~S[7]&~S[6]&~S[5]&~S[4]&~S[3]&~S[2]&~S[1]&~S[0];
  assign flags[3]=S[7];
  assign flags[2]=carry;
  assign flags[1]=over;
  assign flags[0]=E&~ctrl[3]&~ctrl[2]&~ctrl[1]&ctrl[0];

endmodule
```

Figura 30: MUX ALU (2)

3.2. Testbench

Antes de definir el testbench necesitamos mencionar unos módulos que nos serán de ayuda tanto para controlar la ALU como para poder tener un iterador sobre operaciones desde memoria de lectura (ROM), y otros módulos auxiliares que determinarán las flags en algunos casos:

- PC: iterador sobre la ROM.
- ROM: leerá las instrucciones desde un archivo .dat y por cada iteración entregará la instrucción correspondiente a la ALU.
- RstPL, RstPP, RstPM: resets de control para los módulos peso ligero, peso pesado y peso mixto respectivamente.
- Over: determina si al evaluar las operaciones, estas generan overflow.
- Carry: determina si al evaluar las operaciones, estas generan un carry de salida.

Teniendo estos módulos pasamos a definirla en Verilog. Empezamos definiendo todas las variables involucradas: flags, variables de input a y b, resultado, operaciones disponibles, control de multiplexor, carry, etc. Luego inicializamos un reloj que irá iterando entre subidas y bajadas, un contador y la ROM donde iterará el contador. Calculamos los resets correspondientes a peso ligero, peso pesado y peso mixto según la operación seleccionada. Luego calculamos todas las operaciones disponibles.
Con todas las variables establecidas podemos ingresar la información en la ALU para obtener tanto los resultados como las flags, como se define a continuación:

```
module Testbench();

  logic[11:0]flags;
  logic[7:0] a,b,s,suma,resta,mult,div,notv,
  orv,andv,xorv,lslv,lsrv,asrv,rolv,rorv,
  pesol,pesop,pesom;
  logic[3:0]ctrl,c;
  logic clk,cin,reset,resetpl,resetpp,resetpm,
  csum,cres,vsum,vres,vmul,carry,over,E;

  always
    begin
    #5 clk = 1;
    #5 clk = 0;
  end

  PC pc(clk,reset,c);
  ROM rm(c,clk,ctrl);

  //resets
  RstPL rstpl(ctrl,resetpl);
  RstPP rstpp(ctrl,resetpp);
  RstPM rstpm(ctrl,resetpm);
```

Figura 31: Testbench (1)

```
//aritmetica
LookAheadAdder8 LAA8(a,b,cin,suma,csum,vsum);
RestaRCA rest(a,b,cin,resta,cres,vres);
Multiplier multi(a,b,mult,vmul);
Divider divi(a,b,div,E);
//logica
bitwiseNot bnot(a,notv);
bitwiseOr bor(a,b,orv);
bitwiseAnd band(a,b,andv);
bitwiseXor bxor(a,b,xorv);
//traslacion
LogicShiftL lsl(a,b,lslv);
LogicShiftR lsr(a,b,lsrv);
ArShiftR asr(a,b,asrv);
RotateL rol(a,b,rolv);
RotateR ror(a,b,rorv);
//torneo
PesoLigero pl(a,b,clk,resetpl,pesol);
PesoPesado pp(a,b,clk,resetpp,pesop);
PesoMixto pm(a,b,clk,resetpm,pesom);
Carry carr(csum,cres,ctrl,carry);
Over ovr(vmul,vsum,vres,ctrl,over);
//ALU
MuxAluContrl alu(suma,resta,mult,div,notv,andv,
                 xorv,lslv,lsrv,asrv,rolv,
                 rorv,pesol,pesop,pesom,
                 ctrl,carry,over,E,s,flags);
```

Figura 32: Testbench (2)

Cabe mencionar que se omitió la parte del ingreso de variables y display de resultados, ya que estos pertenecen a la sección siguiente.

4. Resultados

Las pruebas realizadas son las mismas que las del ejemplo, con sus respectivos resultados tanto de output como de flags. A continuación se muestran los resultados en la tabla:

ROM	Operación	Input A	Input B	Output S	Flags
0	Suma	7f	2f	ae	10a
1	Suma	ff	01	00	116
2	División INT	11	04	04	100
3	Bitwise NOT	55	–	aa	088
4	Peso ligero	64	60	–	–
5	Peso ligero	07	04	ff	028
6	Peso mixto	3c	53	–	–
7	Peso mixto	04	04	–	–
8	Peso mixto	0f	15	–	–
9	Peso mixto	06	01	–	–
10	Peso mixto	05	05	00	030
11	Multiplicación	03	05	0f	100
12	Arithmetic shift right	ff	02	ff	048
13	Logical shift right	ff	02	3f	040
14	Rotate left	aa	03	55	040
15	División INT	3b	00	00	110

5. Análisis

Podemos notar que hay algunas discrepancias con los resultados en el ejemplo de la tarea, pero algunas de estas operaciones realizadas a mano al final fueron correctas y concordaban con los resultados obtenidos en la sección anterior.

Solo tomando en cuenta los outputs y las flags del ejemplo del enunciado podemos contar un total de 32 casos, 16 para cada uno, de los cuales la ALU fue capaz de acertar en 32, entonces:

$$\frac{Resultados\ exitosos}{Resultados\ totales} = \frac{16}{16}$$

$$\frac{Flags\ exitosos}{Flags\ totales} = \frac{16}{16}$$

6. Conclusión

Los HDL nos ayudan a abaratar tiempo y costo en la implementación de operaciones con lógica booleana, siendo capaces de encapsular el funcionamiento a las operaciones que lo ameriten, dando la oportunidad de hacer reuso de módulos casi sin esfuerzo una vez se tiene dominio del lenguaje.

Con respecto a nuestra ALU, esta fue capaz de resolver la mayoría de las pruebas del testbench. Algunas operaciones como la de división no pudieron ser definidas por falta de tiempo, por lo que esto resta a la completitud del informe y la tarea en general, pero aún así se supo lidiar con el problema simplemente usando opción de Verilog de usar división en otras bases aparte de la binaria. Además se omitieron algunas implementaciones que consideramos poco adecuadas para mantener la pulcritud del informe.

En nuestra opinión este informe está un 95 % completo.