ELEMENTARY LOGIC GATES AND BOOLEAN ARITHMETIC

ALEXANDER BONILLA HIGIDIO

COD: 100620011870

PRESENTADO A:

CARLOS HERNAN TOBAR ARTEAGA

INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES

UNIVERSIDAD DEL CAUCA

POPAYÁN

2023

INTRODUCCIÓN

A continuación, el presente documento tiene como finalidad dar un informe resumido respecto a la creación de los dos proyectos propuestos (Elementary logic gates y Boolean arithmetic) para la primera parte del semestre, dando a conocer un resumen de cómo se realizó cada algoritmo y los resultados obtenidos en los scripts. Además, este informe proporcionará una visión general de los logros alcanzados hasta el momento y servirá como base para futuras etapas de los proyectos y evaluaciones posteriores.

Elementary logic gates

Inicialmente, para el proyecto se requiere tener el entendimiento del significado de las "primitivas", el cuál se basa en un concepto para la creación de compuertas lógica utilizando la compuerta lógica NAND. Es entonces que se hace uso de lógica de la NAND para crear compuertas como la NOT, OR, AND y XOR

La siguiente imagen muestra el esquema básico a seguir para realizar el algoritmo de programación en VHDL.

```
 2   library <library_name>;
 3   entity <entity_name> is
 4       generic
 5       (
 6           <name>    : <type>  :=   <default_value>;
 7           <name>    : <type>  :=   <default_value>
 8       );
 9
10
11       port
12       (
13           <name>    : in  <type>;
14           <name>    : in  <type> := <default_value>;
15           <name>    : out <type>;
16           <name>    : out <type> := <default_value>
17       );
18   end <entity_name>;
19
20
21   architecture <arch_name> of <entity_name> is
22
23   begin
24
25
26
27   end <arch_name>;
```

Es entonces que se destacan principalmente la sección de las librerías y de los paquetes, la sección de la entidad y la arquitectura. Y siguiendo esta organización se procede a realizar los códigos para crear las compuertas lógicas, haciendo uso de la primitiva NAND.

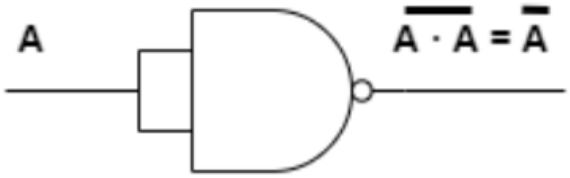
Compuerta NOT:

```
1  -- This file is part of the course "Circuitos Digitales II"
2  -- by Alexander Bonilla H
3  -- File name: project_01/NotGate.vhdl
4
5  -- Not gate:
6  -- f = not x
7
8
9  -- Library and packages
10 library  IEEE;
11 use IEEE.std_logic_1164.all;
12
13 -- Entity (Interface)
14 entity NotGate is
15    port(
16       x  :  in    std_logic;
17       f  :  out   std_logic
18    );
19 end entity;
20
21 -- Architecture (Implementation)
22 architecture arch of NotGate is
23 begin
24    F <= x nand x;
25 end architecture;
26
```

Cabe mencionar que para la creación de esta compuerta hay que comprender bien la lógica de la NAND para obtener de manera correcta las salidas deseadas. En este caso se hace uso de una solo entrada que se opera consigo misma dentro de la NAND y así negar la misma entrada. Entendiendo lo anterior se procede a realizar el algoritmo definiendo inicialmente las librerías y paquetes que se usarán (library IEEE y use IEEE.std_logic_1164.all), luego

se desarrolla parte de la entidad (entity) donde se declaran lo que son las entradas y las salidas que, para este ejercicio son solo una entrada y una salida. Finalmente, se llega a la arquitectura (Architecture) donde se determina el comportamiento que tendrán las variables que fueron declarada (para este caso a la salida 'F' se le asigna el resultado de x nand x).

A continuación se muestra el esquemático que resume la lógica utilizada:

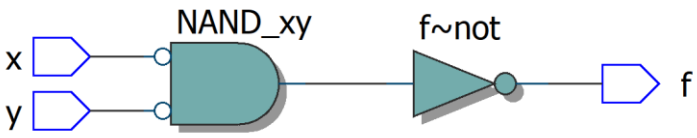


Compuerta OR:

```
4   library IEEE;
5   use IEEE.STD_LOGIC_1164.ALL;
6
7   entity OrGate is
8       Port (
9            x : in std_LOGIC;
.0           y : in  STD_LOGIC;
.1           f    : out STD_LOGIC
.2  );
.3
.4  end OrGate;
.5
.6  architecture arch of OrGate is
.7      signal NAND_xy : STD_LOGIC;
.8      signal NAND_x  : STD_LOGIC;
.9      signal NAND_y  : STD_LOGIC;
:0  begin
:1      NAND_x <= x nand x;
:2      NAND_y <= y nand y;
:3      NAND_xy <= NAND_x nand NAND_y;
:4
:5      f <= NAND_xy;
:6  end architecture;
:7
```

En este caso, se realizaron los mismos pasos anteriormente mencionados para cada sección (Librerías, Entidad y Arquitectura). La entidad presenta una entrada más llamada ‘y’ que, dentro de la arquitectura se encargará de operarse con la entrada ‘x’ y cuyo valor se le asignará a la salida ‘F’. Para la lógica de esta compuerta OR, se debe tener en cuenta también la lógica que se usó para la compuerta OR puesto que la idea inicial sería negar las entradas, es decir, ‘x nand x’ y ‘y nand y’ para que estas mismas se operen dentro de la compuerta nand que les sigue y así poder obtener las salidas de una compuerta OR normal.

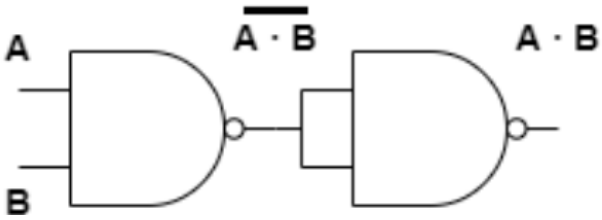
A continuación se observa el esquema gráfico que resume lo anteriormente dicho:



Compuerta AND:

```
4   --Librerias y paquetes
5   library IEEE;
6   use IEEE.std_logic_1164.all;
7
8   --Entidad
9   entity AndGate is
10          port(
11          x : in std_logic;
12          y : in std_logic;
13          f : out std_logic
14          );
15
16
17  end entity;
18
19  --Arquitectura
20  architecture AndGate_arch of AndGate is
21
22
23  signal s : std_logic;
24  begin
25
26          s <= x nand y;
27          f <= not s;
28
29  end architecture;
30
```

Para la creación de la compuerta AND, como los anteriores casos, se declaran las librerías y los paquetes, la entidad del algoritmo donde se declaran las variables ‘x,y’ como entradas y ‘f’ como salida. Luego de esto, se considera la lógica de la compuerta AND para obviarla y solo usar compuertas NAND. Esto se logra conectando las dos entradas primeramente a una compuerta NAND, luego, esa salida se conecta en los dos terminales de otra compuerta NAND que estará en serie. Es así que queda de la siguiente manera:



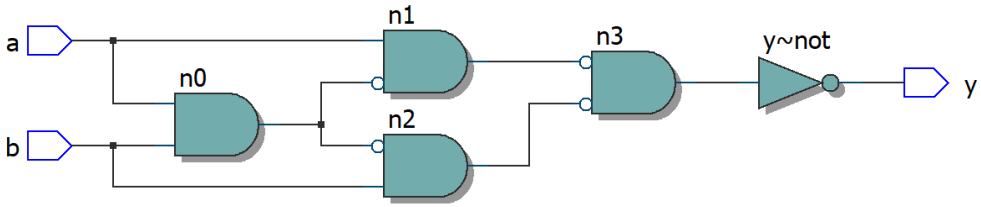
Compuerta XOR:

```
3   library IEEE;
4   use IEEE.STD_LOGIC_1164.ALL;
5
6   --definicion de entidades
7
8   entity XorGate is
9       Port (
10              a   : in std_LOGIC;
11              b   : in  STD_LOGIC;
12              y   : out STD_LOGIC);
13  end XorGate;
14
15  -- definicion de la arquitectura
16
17  architecture arch of XorGate is
18      signal n0 : STD_LOGIC;
19      signal n1 : STD_LOGIC;
20      signal n2  : STD_LOGIC;
21      signal n3 : STD_LOGIC;
22
23  begin
24     n0 <= a nand b;
25     n1 <= a nand n0;
26     n2 <= b nand n0;
27     n3 <= n1 nand n2;
28     y <= n3;
29  end Architecture;
```

Esta compuerta XOR requiere un trabajo adicional considerando que para cumplir con la lógica de esta compuerta solo se usan compuertas NAND. Es entonces que se debe analizar con mayor detenimiento la lógica booleana para dos expresiones A y B.

Dentro del código VHDL se describe una compuerta XOR (O-Exclusiva) utilizando compuertas NAND. La entidad 'XorGate' tiene dos entradas, 'a' y 'b', y una salida 'y'. La arquitectura 'arch' calcula la XOR mediante compuertas NAND intermedias: "n0" almacena a NAND b, "n1" almacena a NAND n0, 'n2' almacena b NAND n0, y 'n3' almacena n1 NAND n2. Finalmente, 'y' toma el valor de n3, que representa la salida XOR de 'a' y 'b'.

El circuito queda configurado de la siguiente manera:



Realización de los Script:

Los scripts o los test bench de los algoritmos que se realizaron también son una parte fundamental del proceso de la creación de estas compuertas, ya que permiten verificar y validar el funcionamiento de los diseños, garantizando que cumplen con los requisitos y especificaciones deseadas. Esta herramienta desempeña un papel crucial al simular diferentes situaciones y condiciones de entrada, lo que facilita la detección temprana de posibles errores o comportamientos inesperados en el código desarrollado, contribuyendo así a un proceso de desarrollo más fiable. Cabe mencionar que la compilación de todo el proyecto hecho se hace con la aplicación de ModelSim-Altera.

Por ejemplo, dentro del programa VHDL de la compuerta AND, para hacer esta prueba de verificación de resultados se recurrió al desarrollo de un código compuesto por una estructura similar a la que ya fue mencionada anteriormente. Es decir con la definición de las librerías y paquetes, una entidad (con la particularidad de que esta está vacía) y una arquitectura donde se concentra todo el proceso de testeo de dicho.

El test bench verifica el comportamiento de una compuerta lógica AND (representada por la entidad ‘AndGate’) bajo diferentes condiciones de entrada. Se utiliza una señal de estímulo para controlar las entradas ‘x’ e ‘y’ de la compuerta y se monitorea la salida ‘f’. El test bench realiza cuatro casos de prueba:

- Cuando las dos variables de entrada ‘x’ e ‘y’ son ambas '0', verifica que la salida ‘f’ sea '0'.
- Cuando ‘x’ es 0 y ‘y’ es 1, verifica que la salida ‘f’ sea '0'.
- Cuando ‘x’ es 1 y ‘y’ es 0, verifica que la salida ‘f’ sea '0'.
- Cuando las dos variables de entrada ‘x’ e ‘y’ son ambas '1', verifica que la salida ‘f’ sea '1'.

En cada caso, se utiliza la declaración "assert" para comparar la salida ‘f’ con el valor esperado y se informa si la prueba ha fallado. Además, se incluyen mensajes de informe para indicar el inicio y la finalización de la prueba, así como la detección de cualquier fallo.

Lo mismo se realiza con las demás compuertas lógicas, aplicando una misma metodología con las librerías y paquetes, la entidad vacía y una arquitectura que muestre el proceso de verificación de todas las posibles salidas cada compuerta lógica.

El proyecto 1 no solo cuenta con compuertas lógicas, también, con estos mismos códigos se pueden construir otro tipo de elementos de la electrónica digital, como lo son los Multiplexores (Mux) y demultiplexores (DMux). Donde, para el caso del multiplexor 2 a 1 se creó un código que escribe una entidad y su arquitectura en VHDL para un multiplexor de 2 entradas (a y b) controlado por una señal de selección (sel), que produce una salida (o).

- La entidad "Mux" define las entradas y salidas del multiplexor:
- "a" y "b" son las dos entradas que se seleccionarán según el valor de "sel".
- "sel" es la señal de selección que determina cuál de las dos entradas se conecta a la salida.
- "o" es la salida del multiplexor que transporta el valor de la entrada seleccionada.
- La arquitectura "arch" implementa el comportamiento del Mux:

- Se declaran dos señales internas, x1 y x2, que se utilizan para almacenar temporalmente los resultados de la operación AND entre las entradas y la señal de selección. De lo anterior se muestran las dos posibles entradas para el multiplexor.
- La señal "x1" toma el valor de ‘a’ cuando ‘sel’ es '0' (not sel) y '0' cuando ‘sel’ es '1'. Es decir, x1 representa ‘a’ cuando ‘sel’ es '0'.
- La señal x2 toma el valor de ‘b’ cuando ‘sel’ es '1' (sel) y '0' cuando ‘sel’ es '0'. Es decir, x2 representa ‘b’ cuando ‘sel’ es '1'.
- Finalmente, la salida ‘o’ toma el valor de x1 or x2, lo que significa que se selecciona "a" si "sel" es '0' y "b" si "sel" es '1'.

Caso “contrario” sucede con el demultiplexor 1 a 2, puesto que el código describe una entidad y su arquitectura en VHDL para un demultiplexor (DMux) de una entrada (a) que entrega dos salidas (x1 y x2) controladas por una señal de selección (Sel). La cual, en la entidad "DMux" define las entradas y salidas del demultiplexor:

- "a" es la entrada que se divide en dos salidas.
- "Sel" es la señal de selección que determina a cuál de las dos salidas se conecta la entrada.
- x1 y x2 son las salidas del demultiplexor, donde x1 se conecta a "a" cuando "Sel" es '0', y x2 se conecta a "a" cuando "sel" es '1'.
- La arquitectura "arch" implementa el comportamiento del demultiplexor:
- Se utiliza una asignación de señales para determinar el valor de x1 y x2 en función de las entradas "a" y "Sel".
- "x1" toma el valor de "a and (not Sel)", lo que significa que cuando "Sel" es '0', x1 será igual a "a", y cuando "Sel" es '1', x1 será igual a '0'.
- x2 toma el valor de "a and Sel", lo que significa que cuando "Sel" es '1', x2 será igual a "a", y cuando "Sel" es '0', "x2" será igual a '0'.

Anexo a lo anterior, también se ahonda en un tema interesante de la programación de VHDL, que es la creación de los vectores, una función que permite adjudicarle más bits a una entrada o una salida. Esto se comprueba, con los algoritmos de Not16 (16-bit Not),

And16 (16-bit And), Or16 (16-bit Or), Mux16 (16-bit multiplexor (Selects between two 16-bit inputs))

Los anteriores elementos tienen la particularidad de considerar más de un bit, en este caso, usan 16 bits y, por ejemplo para la compuerta Not16 (que fácilmente se puede aplicar para los demás casos de las compuertas) se tiene una entidad con su arquitectura para un inversor de 16 bits (Not16). El inversor o negador toma una entrada de 16 bits (x) y produce una salida de 16 bits (f), donde cada bit de la salida es el complemento lógico del bit correspondiente de la entrada. Aquí está la explicación del código:

La entidad "Not16" define las entradas y salidas del inversor:

- x es la entrada de 16 bits que se invertirá.
- f es la salida de 16 bits que contendrá el complemento lógico de cada bit de x.

La arquitectura "arch" implementa el comportamiento del inversor:

- Utiliza asignaciones de señales para calcular el complemento lógico de cada bit de entrada x y asignar el resultado correspondiente a la salida f.
- Cada asignación de "f(n)" se calcula como not x(n), donde "n" representa el número de bit, desde el bit 0 hasta el bit 15.

Ahora, ciertas diferencias se tienen el caso del Mux16 que considera, lo siguiente:

Un multiplexor de 16 bits (Mux16) el cual selecciona entre dos entradas de 16 bits, "a" y "b", basándose en el valor de una señal de selección "sel", y produce una salida de 16 bits "o".

La entidad Mux16 determina las entradas y salidas del multiplexor:

- "a" y "b" son las dos entradas de 16 bits que se seleccionarán.
- "sel" es la señal de selección que determina cuál de las dos entradas se conecta a la salida.
- "o" es la salida de 16 bits que transporta el valor de la entrada seleccionada.

En la arquitectura, se utiliza un algoritmo o función interna llamada Mux (el que previamente se había creado) para implementar la funcionalidad de selección bit a bit de las

entradas "a" y "b" con base en la señal "sel". El Mux toma una entrada de un solo bit y produce una salida de un solo bit.

Se instancian 16 componentes Mux para realizar la selección bit a bit de las entradas "a" y "b". Cada componente Mux selecciona un bit específico de "a" y "b" y lo dirige a la salida correspondiente de "o".

Finalmente, para concluir el proyecto se tienen en cuenta unas extensiones un tanto más "complejas" de la compuerta Or, Mux y DMux. Y es que cada una tiene un requerimiento distinto, es decir, está:

Or8Way ⇒ Or(in0,in1,...,in7)

Mux4Way16 ⇒16-bit/4-way mux (Selects between four 16-bit inputs)

Mux8Way16 ⇒ 16-bit/8-way mux (Selects between eight 16-bit inputs)

DMux4Way ⇒ 4-way demultiplexor (Channels the input to one out of four outputs)

DMux8Way ⇒ 8-way demultiplexor (Channels the input to one out of eight outputs)

En el inciso de la Or8Way, se refiere a una compuerta Or de 8 entradas con una salida y es por ello que se recurre a un código que se describe tal así:

La entidad "Or8Way" define las entradas y salidas del bloque lógico siendo "A" una entrada de 8 bits que se conecta al bloque y se realiza una operación OR en sus bits.

"S" es la salida que transporta el resultado de la operación OR de los 8 bits de entrada A.

En la arquitectura, se utilizan señales internas "I", "J", "K", "L", "M", y "N" para realizar cálculos intermedios en pasos escalonados.

Cada señal intermedia I, J, K, y L calcula la operación NAND de dos bits adyacentes en la entrada "A". Esto se hace al negar los bits correspondientes y luego realizar una operación NAND entre ellos. Estos cálculos se hacen para dividir la entrada en grupos de dos bits cada uno.

Facultad Ciencias Naturales, Exactas y de la Educación
Calle 2 No. 3N-100 Segundo Piso. Sector Tulcán Popayán - Cauca - Colombia
Teléfono: 8209842 Conmutador 8209800 Exts. 2453 – 2482

Las señales M y N calculan la operación NAND entre los resultados de los cálculos intermedios I y J, y K y L, respectivamente. Esto combina los grupos de dos bits calculados en el paso anterior en grupos de cuatro bits.

Es entonces que finalmente, la señal de salida S calcula la operación NAND entre las señales M y N, lo que resulta en realidad haciendo una operación OR de los 8 bits de la entrada A.

Otro de los ejercicios propuestos y que vale la pena de mencionar es la creación del Mux8Way16, en el cual por comodidad se usan funciones del estilo de la programación estructurada como lo es el ‘case’. En entonces que brevemente se concluye que para el código del Mux8Way16 se tiene un multiplexor de 8 vías de 16 bits caracterizado por:

una entidad con su arquitectura para un multiplexor de 8 vías de 16 bits. Este Mux tiene ocho entradas de datos (a, b, c, d, e, f, g, h), una entrada de selección de 4 bits (sel) y una salida de 16 bits (y)

- "a", "b", "c", "d", "e", "f", "g", "h" son las ocho entradas de datos, cada una de 16 bits.
- "sel" es la señal de selección de 4 bits que determina cuál de las ocho entradas se conecta a la salida.
- "y" es la salida de 16 bits que transporta los datos seleccionados por sel.
- En la arquitectura "Mux8Way16_arch", se utiliza un proceso que toma sel como entrada. En función del valor de sel, se selecciona una de las ocho entradas y se asigna a la salida "y".

Se utiliza una estructura ‘case’ para realizar la selección. Dependiendo del valor de sel, se asigna la entrada correspondiente a la salida ‘y’. Si sel tiene un valor que no coincide con ninguno de los casos especificados, se asigna un vector de 16 bits con todos sus bits en '0' a la salida y.

BOOLEAN ARITHMETIC

El objetivo de este proyecto es construir una serie de chips en VHDL que constituirán en la construcción de un chip ALU (Arithmetic Logic Unit). Los algoritmos para construir son los siguientes: HalfAdder, FullAdder, Add16, Inc16 y ALU.

HalfAdder: La idea se basa en crear un archivo VHDL para el chip 'HalfAdder'. Este medio sumador toma dos bits de entrada (a y b) y produce dos bits de salida: la suma (sum) y el acarreo (carry). Y esto se logra desarrollando una entidad halfAdder definiendo dos entradas (a y b) que son los bits que se sumarán y dos salidas (sum y carry). El resultado de la suma se coloca en sum, mientras que el acarreo generado durante la suma se coloca en carry. En la arquitectura, se implementa la lógica del "Half Adder". La suma (sum) se calcula mediante la operación XOR entre a y b. El acarreo (carry) se calcula mediante la operación AND entre a y b.

Este Half Adder constituye un bloque fundamental en la aritmética binaria y se utiliza como componente básico para construir sumadores más grandes, como el "FullAdder" o sumadores de múltiples bits (Add16 y Inc16).

FullAdder: Construcción de un archivo VHDL para el chip del "FullAdder". Este código toma tres bits de entrada (a, b y c) y produce dos bits de salida: la suma (sum) y el acarreo (carry). Es decir, tener una entidad FullAdder define tres entradas (a, b, y c) que son los bits que se sumarán, y dos salidas (sum y carry). El resultado de la suma se coloca en sum, y el acarreo generado durante la suma se coloca en carry.

En la arquitectura (architecture arch of FullAdder), se implementa la lógica del "Full Adder". Se utilizan dos componentes halfAdder para realizar la suma de forma incremental.

- halfAdder1 suma los bits a y b y produce sum1 y carry1 como resultado. sum1 es la suma de a y b, y carry1 es el acarreo generado en esta suma.
- halfAdder2 suma sum1 (que es el resultado de la primera suma) y c (el acarreo de la suma anterior) para obtener la suma final (sum) y un acarreo adicional (carry2).
- Luego, se calcula el acarreo final (carry) como la combinación OR de carry1 y carry2. Esto asegura que el resultado sea un sumador completo.

Concluyendo así un componente fundamental en la aritmética booleana y se utiliza como bloque básico para construir sumadores de múltiples bits.

Add16: Creación de un archivo VHDL para el chip Add16. El Add16 realiza la suma de dos valores de 16 bits (a y b). Debes usar FullAdder para construir este chip. Este ejercicio se ignora el bit de acarreo más significativo.

El Add16 realiza la suma de dos valores de 16 bits (a y b) usando componentes FullAdder anteriormente hecho. El código Add16 tiene cuatro puertos en total: dos entradas de 16 bits (a y b), una salida de suma de 16 bits (sum) que contiene el resultado y una salida que indica si hubo un acarreo en la suma (carry). Se utilizan 16 casos del componente FullAdder para sumar cada bit de las entradas, propagando los acarreos de un bit al siguiente binario.

Inc16: Creación de un algoritmo VHDL para el chip Inc16. El Inc16 incrementa un valor de 16 bits (in) en 1. Debes usar HalfAdder para construir este chip.

La explicación del código consiste en lo siguiente. Se trata de un código VHDL que implementa un "incrementador" de 16 bits, es decir, un circuito que suma '1' al valor de entrada de 16 bits y regresa el resultado en la salida de 16 bits. El código particularmente usa una componente llamada HalfAdder, que es un sumador de medio bit, es decir, un circuito que suma dos bits y devuelve la suma y el acarreo. El código declara una señal interna llamada carryOut, que es un vector de 17 bits que almacena los acarreos generados por los 'halfAdders' (función la cual anteriormente fue construida). El código inicializa el primer bit de carryOut en '1', que es el valor que se quiere sumar al valor de entrada. Luego, el código usa un bucle 'for generate' para instanciar 16 HalfAdders, uno para cada bit del valor de entrada. Cada HalfAdder recibe como entradas un bit del valor de entrada (x(i)) y un bit del vector de acarreos (carryOut(i)). Cada HalfAdder devuelve como salidas un bit del valor de salida (y(i)) y el siguiente bit del vector de acarreos (carryOut (i + 1)). De esta forma, el código implementa la operación de incrementar el valor de entrada en 1 usando sumadores de medio bit.

ALU (Arithmetic Logic Unit): Finalmente, se considera la creación de un algoritmo para el chip ALU. El ALU es un código más complejo y realiza varias operaciones, incluyendo

la suma de dos valores de 16 bits, operaciones lógicas (AND) y operaciones de negación. Debes usar los chips que has construido anteriormente (HalfAdder, FullAdder, Add16, Inc16) para construir el ALU. Además, el ALU también tiene dos salidas adicionales: una que indica si la salida es igual a 0 (zr) y otra que indica si la salida es negativa (ng).

Este código pretende realiza diversas operaciones en dos entradas de 16 bits, x e y, de acuerdo con las señales de control y genera una salida de 16 bits, sal, junto con señales de estado zr (bandera de cero) y ng (bandera de negativo).

Brevemente se describe así el proceso:

- La entidad ALU define la interfaz de la unidad ALU, especificando las entradas (x, y, señales de control) y las salidas (sal, zr, ng).

- En la arquitectura se declaran señales internas que se utilizarán para realizar cálculos intermedios y mantener el estado de la ALU.

- Luego, instancia varios componentes, lo cuales fueron desarrollados a lo largo de estos dos proyectos (HalfAdder, Inc16, Add16, Not16) que se utilizan para realizar operaciones en las entradas x e y.

- Process Mux16X1 y Mux16X2: Estos procesos implementan dos multiplexores de 16 bits, uno para la entrada x y otro para la entrada y.

- Los multiplexores seleccionan entre diferentes valores de entrada (x, notX, 0) según las señales de control (zx, nx, zy, ny) y generan las salidas x1 y y1.

- En los componentes del sumador Add16 se utilizan para realizar sumas de 16 bits, y se instancian dos veces: FullAdderX suma x1 y y1, y FullAdderY suma notX y notY.

- Process Mux16Out1: Este implementa otro multiplexor de 16 bits que selecciona entre las salidas de FullAdderX y FullAdderY según la señal de control f. La salida se almacena en out1.

- Proceso Mux16Out2: Este procedimiento implementa otro multiplexor que selecciona entre out1 y notOut1 según la señal de control no. La salida final se asigna a sal.

- Asignación de banderas: “zr” se establece en '1' cuando sal es igual a "0000000000000000" (todas las salidas en 0) y en '0' en caso contrario. Mientras que “ng” se le asigna al valor del bit más significativo de sal (bit 15), que indica si el resultado es negativo.

Lamentable el código respecto al ALÚ no pudo compilarse exitosamente debido a errores de sintaxis y otros problemas de lógica y funciones. Los errores encontrados en este código, fueron más de los que fácilmente eran solucionables, sin embargo la idea será mejorar y analizar más el código para realizar los cambios respectivos y así poder realizarle también el Test Bench.