Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica
IE-0323 - Digitales I
II ciclo 2023

Proyecto Final

Diseño Digital en Verilog: Comparador de Palabras de N Bits

Sebastián Bonilla Vega - C01263
Juan Pablo Díaz Matamoros - B82600
Luis Fernando Rojas Morúa - B86941

Grupo 04

Profesor: Heberth Valverde Gardela

Miércoles 29 de Noviembre, 2023

Índice

1. Introducción

Este proyecto tiene como objetivo desarrollar un circuito digital en Verilog que pueda comparar dos palabras de N bits, A y B. Donde su importancia es detectar si la palabra A es mayor a la palabra B y notificarlo a través de una señal de salida llamada Z, la cual es activa en bajo. Esto se busca realizar bajo la metodología de diseño de redes iterativas.

El proyecto se va a dividir en dos secciones, abordando en cada sección, un sentido diferente en el recorrido de las palabras de entrada; caso de izquierda a derecha en donde al realizar el código se utilizará la implementación conductual y el caso de derecha a izquierda se hará con la implementación estructural. A cada caso se le realizarán las pruebas de manera independiente para presentar la solución del problema planteado.

El trabajo llevará a la elaboración del diseño de red iterativa demostrando la formación de cada celda, posterior a ello se realiza el código bajo el lenguaje de Verilog para realizar la simulación de la red en ambos sentidos e implementaciones. Además se presentará los resultados mediante las ondas formadas por el sistema en el apartado de GTKWave.

2. Marco teórico

2.1. Conceptualización

En primera instancia se brindará una introducción, basada en los conceptos y fundamentos teóricos necesarios en el desarrollo del proyecto, con el fin de obtener una mejor compresión del mismo.

2.1.1. Sistemas Digitales

Una de las principales características que se puede destacar de los sistemas digitales, es el hecho de su capacidad para manipular elementos discretos de información, siendo estos datos aquellos que solo pueden tomar valores determinados como los números decimales, las letras del alfabeto, entre otros. De acuerdo a Morris (2003) en los sistemas digitales los elementos se representan mediante señales que comúnmente son señales eléctricas y en casi todos los sistemas, las señales emplean solo dos valores discretos, a cada dígito se le llama bit, estos sistemas reciben el nombre de binarios que utilizan los valores únicamente de 1 y 0. Con grupos de bits junto con distintas técnicas se puede representar los anteriormente mencionados.

Según Morris (2003) “Un sistema digital es un sistema que manipula elementos discretos de información representados internamente de forma binaria” (p.2). Estos elementos surgen de los datos procesados; siendo así el sistema digital una interconexión de módulos digitales generados por herramientas básicas como compuertas lógicas, circuitos combinacionales y secuenciales.

Es de suma importancia hablar sobre el HDL (hardware description language), este es un lenguaje de descripción de hardware, el cual es un lenguaje de programación que permite describir los circuitos digitales de forma textual. El cual es para simular sistemas verificando su funcionamiento antes de crearlos.

Finalmente, ”Las compuertas lógicas son circuitos electrónicos que operan con una o más señales de entrada para producir una señal de salida” (Morris 2003, p.29).

2.1.2. Circuitos Combinacional

Tal como lo sugiere su nombre, según Brunete, Herrero y Segundo (2020) en estos circuitos su estado lógico de la salida depende solamente de la combinación de sus entradas, en el momento en que se realiza la medición de la salida, por consiguiente, este tipo de circuitos no es necesario tener en cuenta la noción del tiempo así que no es posible almacenar estados y usarlos para tomar decisiones posteriores. Por ello, de acuerdo a Brunete, Herrero y Segundo (2020) estos circuitos se basan en puertas lógicas y se resuelven mediante tablas de verdad y/o álgebra booleana, donde se recogen las combinaciones de las señales de entrada para determinar la salida.

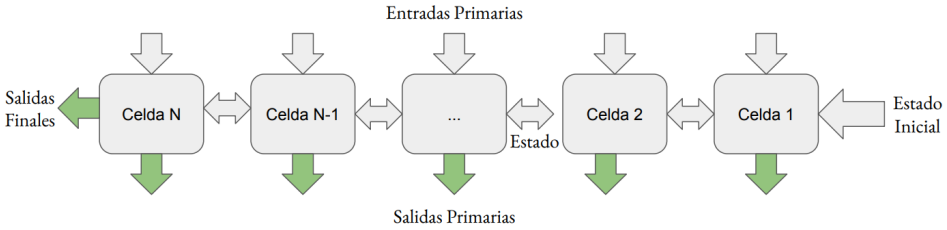


Figura 1: Estructura topológica de una red iterativa

Las compuertas lógicas son la AND, OR, NOT, NAND, NOR, INVERSORES, XOR y a partir de ellas se desarrolla la lógica necesaria para la resolución de distintas problemas planteados. Estos sistemas carecen de memoria y retroalimentación. La lógica combinacional significa la combinación de dos ó más puertas lógicas, que se obtienen mediante ecuaciones conformadas por operaciones básicas de álgebra de Boole. En este tipo de circuitos, algunas de las aplicaciones que podemos encontrar son los comparadores, multiplexores, sumadores, decodificadores, entre otra gran variedad de elementos que se pueden elaborar mediante compuertas lógicas

En síntesis basado Morris (2003) en estos circuitos están compuestos por variables de entrada, puertas lógicas y salidas; en dónde según el tipo de funcionamiento de las compuertas en donde cada variable de entrada y de salida existe físicamente como una señal binaria que representa 1 lógico y 0 lógico que al ser manipulados se obtiene una o varias salidas deseadas.

2.1.3. Redes Iterativas

Una red iterativa es un conjunto de celdas de lógica combinacional idénticas, que deben procesar una gran cantidad de bits o una desconocida cantidad de bits, en las cuales la información es trasferida de una celda a la siguiente de manera lineal, en cuanto a la igualdad de celdas está la excepción de la primera y la última celda.

De manera general y por medio de la estructura topológica podemos observar que la red iterativa se presenta de la siguiente manera:

Método de diseño de una red iterativa por transición de estados

A continuación se describirán los pasos o etapas para construir una red iterativa por este método de manera detallada, por lo general este procedimiento cuenta con 8 etapas para su elaboración:

a) Definir los estados que resuelven el problema: Es el paso más importante, los estados se obtienen del análisis de resolver el problema. Se deben definir mediante texto y no se deben utilizar símbolos. Se denota a cada estado con una letra y se utilizan los necesarios para cumplir con la solución del problema.

b) Construir la tabla de transición de estados: En esta tabla se presenta el estado que se debe propagar partiendo de todos los posibles estados en que se encuentra y ante cada una de las combinaciones de entradas que se tengan.

c) Hacer la asignación de estados, la declaración de variables de estado y variables de salida: Se le asigna a cada estado definido un código binario, se declara las variables de entrada y salida necesarias para el sistema.

d) Hacer la tabla de transición de estados codificada: Con la asignación realizada en el paso anterior, se sustituyen los códigos designados en la tabla tabla de transición de estados.

e) Dibujar el diagrama de bloques de la celda típica: En este diagrama se indican los nombres de las entradas y salidas de la celda típica; esto con el fin de identificar cada una de ellas, así como el flujo de información.

f) Diseñar la celda típica: Con la tabla de transición codificada se arman los mapas de Karnaugh de acuerdo a cada variable de entrada; posterior a ello, con técnicas de los mapas de Karnaugh de la función mínima y álgebra booleana, se determinan las ecuaciones lógicas de las salidas en donde a partir de ellas se puede realizar los esquemáticos.

g) Diseñar la celda inicial: Se decide cual es el estado de partida en la primera celda, sustituyendo esto en las ecuaciones previamente encontradas se determina las funciones lógicas de las entradas. Donde de igual forma se puede desarrollar el esquemático de esta celda.

h) Diseñar la celda final: Con la lógica previamente analizada, se encuentran los casos en donde se activa la salida final y nuevamente por medio de mapas de Karnaugh o lógica desarrollada se obtiene la ecuación de salida final.

2.1.4. Verilog

"Verilog es un lenguaje para la descripción de sistemas digitales"(Cháves 1999, p.3). Es parte de los lenguajes de HDL, que son las siglas en inglés de "Hardware Description Language". Y su principal función es modelar sistemas digitales, con el fin de verificar si el sistema funciona de manera correcta.

Estos sistemas tienen dos maneras de ser descritos, uno es a nivel estructural y otro a nivel conductual. El primero se realiza usando elementos de la librería en donde se realiza la interconexión de unos bloques con otros, mientras que el segundo se debe describir cada elemento para que se ejecute la tarea necesaria en la transferencia de información. Además, ambos modelados tienen sus ventajas y desventajas. El conductual permite implementar funciones considerablemente complejas; no es que no se pueda de forma estructural, pero el uso del modelado conductual es más rápido. El modelo estructural, al tener que definir todas las compuertas a utilizar, permite mucho más control sobre el diseño.

3. Marco Metodológico

3.1. Diseño de la Red Iteractiva

En este apartado se detallará el desarrollo necesario para obtener la red interactiva solicitada, paso a paso, explicando minuciosamente cada una de sus etapas fundamentales planteadas en el apartado anterior. Se realizará ambos análisis, primeramente en el sentido izquierda-derecha y posterior a este de derecha-izquierda; se realizara paso a paso como se menciona en el apartado anterior

Caso izquierda a derecha

Paso a: En este paso se definen los estados necesarios para resolver el problema en análisis. Y en este recorrido tres estados son suficientes para la solución de lo planteado, además se le da una codificación con una letra a cada estado seleccionado.

Estados:
u) De momento las entradas son iguales
v) El primer 1 está en A
w) El primer 1 está en B

Paso b: A continuación se diseña la tabla de transición de estado con la codificación previamente planteada.

Cuadro 1: Tabla de Transición de Estados

Estado Presente	**Estado Futuro**			
AB	00	01	10	11
U	U	W	V	U
V	V	V	V	V
W	W	W	W	W

Paso c: Se designan las dos variables de estado y de salida necesarias. Además, se designó la codificación para cada estado, dándole un valor de dos bits.

- Variables de Estado: x, y
- Variables de Salida: X, Y

Cuadro 2: Tabla de Asignación de Estados

Estado	Codificación
u	00
v	10
w	01

Paso d: Con la codificación seleccionada, se utiliza la tabla realizada en el paso b y se procede a sustituir los valores de los estados para obtener la tabla de transición de estados codificados

Cuadro 3: Tabla de Transición de Estados Codificados

Estado Presente xy	Estado Futuro AB			
	00	01	10	11
00	00	01	10	00
10	10	10	10	10
01	01	01	01	01

Paso e: Acá se diseña el diagrama de bloque de la celda típica para presentar el sistema, en donde se observa una celda típica que se compone de dos entradas primarias [A B], dos entradas secundarias [x y] y dos salidas secundarias [X Y]

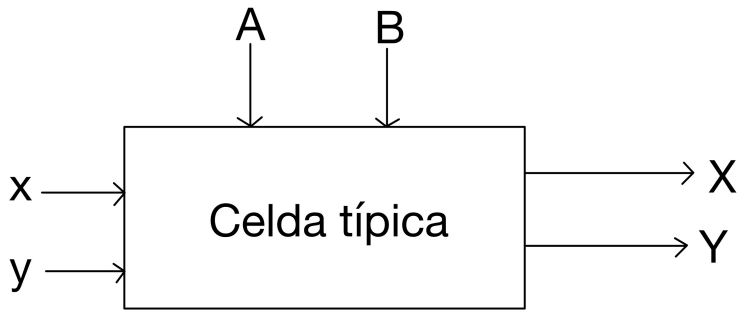
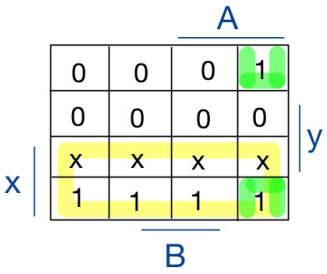


Figura 2: Celda Típica del análisis de izquierda a derecha

Paso f: En este paso se diseña la lógica interna de la celda típica, presentada en el esquemático. Esto con las ecuaciones encontradas por medio de las técnicas en los mapas de Karnaugh de 4 variables realizados con los valores de la tabla de transición de estados codificados. Se realizan dos mapas de Karnaugh porque hay dos salidas

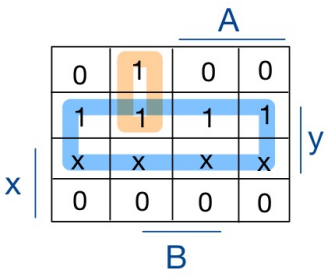
Para X	xyAB
4 variables	

X (x, y, A, B) = x + y' A B'



Para Y	xyAB
4 variables	

Y (x, y, A, B) = y + x' A' B



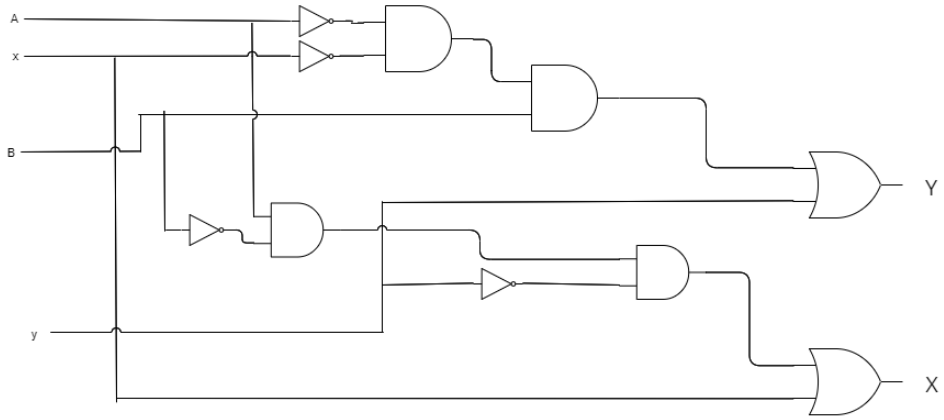


Figura 3: Diagrama esquemático de la celda típica analizada de izquierda a derecha

Paso g: Se toma el estado "u" es decir, el estado "00" como inicial y con las ecuaciones encontradas en el paso anterior de la celda típica se determina la celda inicial. Se sustituye el estado inicial en las ecuaciones y se obtiene las expresiones iniciales. Posteriormente se realiza el diagrama esquemático de dicha celda.

- Estado inicial: u = 0 por lo tanto se obtiene que X (0,0,A,B) = A B', además, Y(0,0,A,B) = A'B

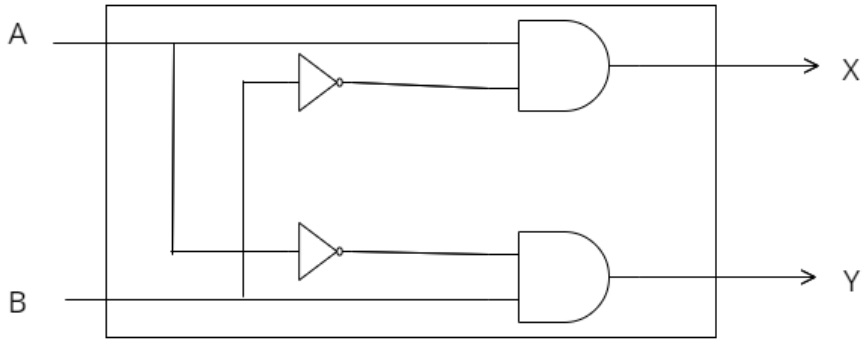


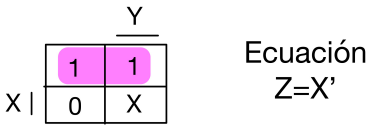
Figura 4: Celda Inicial Analizada de Izquierda a Derecha

Paso h: Por último, se diseña la celda final que se obtiene de la tabla de salidas observando los casos en que Z se activa.

Cuadro 4: Tabla de Salida

xy	z
00	1
01	1
10	0
11	X

- De la tabla construida, se construye un mapa de Karnaugh para obtener la ecuación de salida. Donde se dedujo que la ecuación de salida es Z = X'



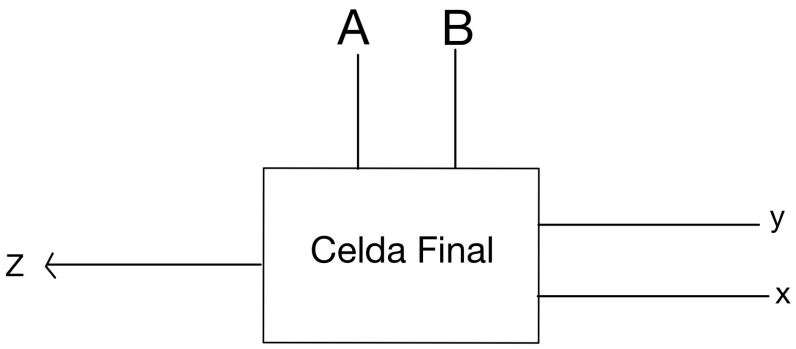


Figura 5: Diagrama de Bloques Celda Final

Caso derecha a izquierda

Paso a: En este paso se definen los estados necesarios. Y se llega a la conclusión de que en este sentido se necesitan solo dos estados y se codifican con una letra cada uno de ellos.

Estados:
x) De momento A es menor o igual a B
y) De momento A es mayor a B

Paso b: Se diseña la tabla de transición de estado con la codificación previamente mencionada.

Cuadro 5: Tabla de Transición de Estados analizando de derecha a izquierda

Estado Presente	Estado Futuro			
AB	00	01	10	11
x	x	x	y	x
y	y	x	y	y

Paso c: Se designan la variable de estado y de salida necesaria. Además, se le da la codificación para cada estado, dándole un valor de un bit en la tabla que se presenta a continuación.

- Variables de Estado: w
- Variables de Salida: W

Cuadro 6: Tabla de Asignación de Estados

Estado	Codificación
x	0
y	1

Paso d: Con la codificación seleccionada, se utiliza la tabla realizada en el paso b y se procede a sustituir los valores de los estados para obtener la tabla de transición de estados codificados

Cuadro 7: Tabla de Transición de Estados Codificada

Estado Presente	Estado Futuro			
w	AB			
	00	01	10	11
0	0	0	1	0
1	1	0	1	1

Paso e: Se diseña el diagrama de bloque de la celda típica para representar el sistema, en donde se observa una celda típica que se compone de dos entradas primarias [A B], una entrada secundaria [w] y una salida secundaria [W].

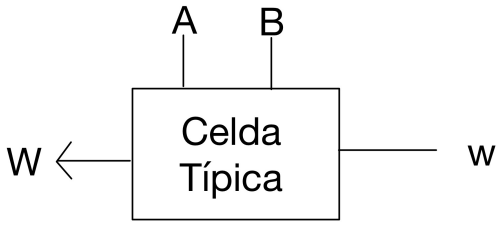
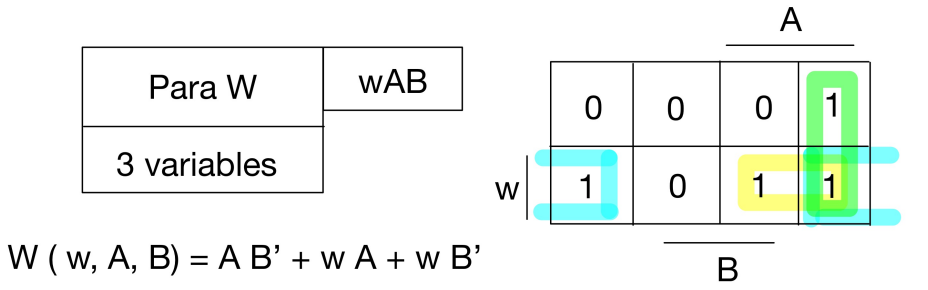


Figura 6: Celda Típica del análisis de derecha a izquierda

Paso f: En este paso se diseña la lógica interna de la celda típica, que posteriormente será presentada en un diagrama esquemático. Esto con las ecuaciones encontradas por medio de las técnicas en los mapas de Karnaugh de 3 variables realizados con los valores de la tabla de transición de estados codificados.



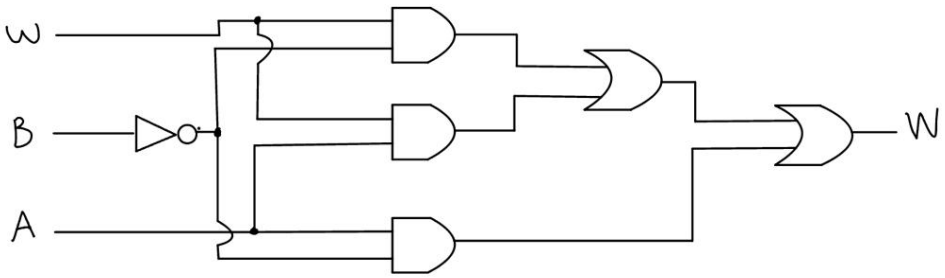


Figura 7: Diagrama esquemático de la celda típica analizada de derecha a izquierda

Paso g: Se toma el estado "x" como inicial, el cual tiene la codificación "0". Con las ecuaciones encontradas en el paso anterior de la celda típica se determina la celda inicial. Se sustituye el estado inicial en las ecuaciones y se obtiene las expresiones iniciales. Posteriormente se realiza el diagrama esquemático de dicha celda.

- Estado inicial: w = 0 por lo tanto se obtiene que W (0,A,B) = A B'

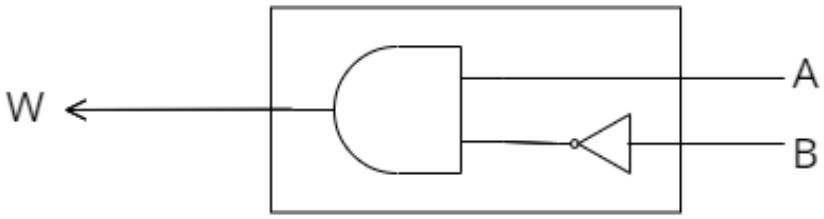


Figura 8: Celda Inicial Analizada de Derecha a Izquierda

Paso h: Finalmente, se diseña la celda final que se obtiene de la tabla de salidas observando los casos en que Z se activa.

Cuadro 8: Tabla de Salida

W	Z
0	1
1	0

- Se realiza el análisis; se tiene que Z se activa en bajo por lo tanto se observa que la salida Z se levanta en el caso de W = 1 por lo tanto la ecuación de salida se determina como Z(W) = W'. Se realiza el diagrama de bloques.

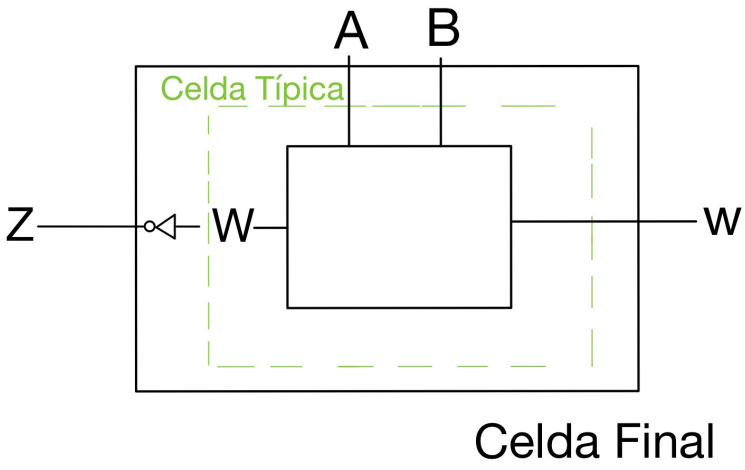


Figura 9: Diagrama de Bloques Celda Final

3.2. Código

Se van a pasar por varias etapas a la hora de diseñar el código. Es importante definir que los programas a utilizar son **Visual Studio Code**, **Icarus Verilog** y **GTK Wave**. Además, es importante hacer la anotación de que verilog se compila principalmente ejecutando comandos en la terminal de la computadora. De manera conveniente, V.S permite accesar a la terminal de la computadora. Estos comandos se ejecutan en orden, para poder ver las ondas en gtkwave. Los comandos de la terminal más utilizados son:

- **iverilog -o (nombre de archivo).vvp (nombre de archivo).v:** Este es para compilar que lo programado corra sin errores. Genera un archivo .vvp que se utilizará en el siguiente comando. Nota: Los archivos deben ser los que incluyen al test bench. (RedIterativaConductualIaD tb y Comparador tb.)

- **vvp (nombre de archivo).vvp:** Este comando fue implementado para poder generar el archivo .vcd necesario para observar las ondas en GTKWave.
- **gtkwave:** Para levantar GTKWave y cargar el .vcd.
- **Importante:** verificar que todos los archivos de código descargados se encuentren en el mismo path o directorio. De igual forma es indispensable tener la ubicación exacta del archivo al desarrollar los comandos en la terminal.

Al abrir GTKWave, se debe cargar el archivo .vcd generado por el comando **vvp**. Una vez en GTKWave, se debe hacer click en **File**, en la esquina superior izquierda. Una vez dentro de file, se hace clic en **open new tab**, y es ahí donde se selecciona el archivo .vcd, luego se hace click en la señal que esta en SST, lo que va a abrir el uut. Se selecciona el UUT, y va a mostrar las señales a visualizar. Se seleccionan las señales que se desean ver, y se hace click en el botón **Append**, en la esquina inferior Izquierda. Esto va a generar las ondas en GTKWave. Además, el léxico que se usa es el de verilog, escrito sobre Visual Studio. Bien todo se pudo haber hecho en notepad o notepad+ y ejecutarlo desde la terminal, pero Visual facilita completamente este proceso.

Caso izquierda a derecha

A la hora de implementar el caso **Izquierda a Derecha** y al analizar la cantidad de compuertas que hay que utilizar (ver figura 3), se decidió implementar un modelo **conductual** para programar la red iterativa en verilog, para el caso en el que la red se resuelva de IZQUIERDA a DERECHA. Esto es debido a que, por la cantidad de compuertas OR, AND y NOT, sería más laborioso implementar y llamar los módulos cada vez que se vayan a utilizar, por lo que es más práctico y eficiente utilizar el modelo conductual para este caso. Tenemos que implementar 4 ANDS, 2 ORS y 4 NOTs.

El modelo conductual permite usar directamente los comandos o funciones que trae verilog en su directorio, en vez de estar llamando e instanciando los módulos de las compuertas. Esto es sumamente útil, ya que ahorra tiempo y hace que el código sea más fácil de leer. Sin embargo, se pierde un poco el control que se tiene sobre el hardware, lo que puede volver las optimizaciones del mismo más laboriosas.

```
//Implementacion Conductual para las funciones X y Y, de Izq a Derecha
module celda_tipica_c
(
    input wire  A, //implementacion de las entradas A,B,x,y y una senal reset, que va ser necesaria para inducir retardos y reiniciar la senal
                   // entre prueba y prueba.
    input wire  B,
    input wire x,
    input wire y,
    input reset,
    output wire X,//Implementacion de las salidas X y Y.
    output wire Y
);
//Declaracion de variables para conectarlas al modulo, (A,B,x,y) son las entradas y (X,Y) son las salidas.
    assign X =  x | (~y & A & ~B);
    assign Y = y | (~x &  ~A & B );
//X corresponde a la funcion obtenida a mano resolviendo la red iterativa, X = x+(y')*(A)*(B') y Y = y+(x')*(A')*(B)
endmodule
```

Figura 10: Código de la Celda Típica Conductual

El primer bloque de código (Figura 10) a analizar consiste en la definición de la celda, siguiendo el proceso de diseño detallado anteriormente, lo que sería la **celda típica**. Primero, se define un modulo con su nombre **celda típica c**, donde entran **A y B**, que son las señales que causan los bits de las palabras, **x y y**, que corresponden a los estados presentes o entradas secundarias; y la señal **reset**, la cual nos va ayudar a reiniciar los valores A y B a la hora de correr múltiples pruebas. Finalmente, al módulo le salen **X y Y**. De último, se usa el comando **assign**, para propiamente, asignar las funciones lógicas para las salidas X y Y las cuales son:

$$X = x + y'AB' \tag{1}$$

$$Y = y + x'A'B \tag{2}$$

```
//Implementacion Conductual para las funciones X y Y, de Der a Izquierda
module celda_inicial_c(
    input wire  A,
    input wire  B,
    input reset,
    output wire X,
    output wire Y
);
//Declaracion de variables para conectarlas al modulo, (A,B,x,y) son las entradas y (X,Y) son las salidas.
    assign X = A & ~B;
    assign Y = ~A & B;
//X corresponde a la funcion obtenida a mano resolviendo la red iterativa, X = (A')*B y Y = 0
endmodule
```

Figura 11: Código de la Celda Inicial Conductual

El segundo bloque de código (Figura 11) a analizar consiste en la definición de la celda que siguiendo el proceso de diseño, sería la **celda inicial**. Aquí, se puede apreciar como ahora en las entradas solo están **A, B y reset**, ya que en la celda inicial, no se recibe otra cosa más que las señales dadas por los bits de A y B,

no recibe nada de otra celda ya que es la primera. Sus salidas van a ser **X y Y**, de acuerdo a la lógica que se obtuvo en la parte del diseño:

$$X = AB' \tag{3}$$

$$Y = A'B \tag{4}$$

```
//Implementacion Conductual para las funciones X y Y, de Izq a Derecha
module celda_final_c(
    input wire  x,
    input wire  y,
    input wire  X,
    input wire  Y,
    input reset,
    output wire Z,
    output wire Z_out
);
//Declaracion de variables para conectarlas al modulo, (x,y) son las entradas y (Z) son las salidas.
    assign Z =  ~X;
    assign Z_out= Z;
//X corresponde a la funcion obtenida a mano resolviendo la red iterativa, Z = (y')() y x = 0.
//Ademas, la senal Z_out es la que vamos a considerar como 'Z disparada en bajo'
endmodule
```

Figura 12: Código de la Celda Final Conductual

El tercer bloque de código (Figura 12) a analizar consiste en la **celda final**. Aquí, se puede apreciar como las entradas son **A, B, x y y**. En este caso, según el diseño a mano, entran dichas señales pero solo sale la señal x negado puesto que las demás no afectan la salida, así que en el esquemático van conectadas a tierra. Nuevamente, esta el input de reset para reiniciar la señal en cada prueba. Sus salidas van a ser **Z y Zout**. Z es la salida normal, y Zout es como queda después de aplicarle la lógica del enunciado, para que Z sea activa en bajo, y se active cuando A es mayor que B.

```
//Proyecto Final de Digitales,C01263,B82600,B86941
//Implementacion Conductual al caso IZQUIERDA a DERECHA.
`include "celda_final_c.v"
`include "celda_tipica_c.v"
`include "celda_inicial_c.v"
module RedIterativaConductualIaD #(parameter N = 7) ( //Aqui esta el parametro a escoger, para este caso es N,
                                                      //pero puede ser cualquier numero mayor a 3.
                                                      //POR CUALQUIER COSA, guardar con 'Save As' y reemplazar el archivo con el nuevo valor de N.

    input wire [N-1:0] A,
    input wire [N-1:0] B,
    output wire Z_out

    );
    wire [N:0] X_temp;//Ahora, la conexion X va a depender del N
    wire [N:0] Y_temp;//Asi como la conxion de Y, la idea es que estos van a ser los cables que conectan una celda tipica y otra.
    //Z sigue igual porque es la salida final de toda la red.

//Instanciacion de la celda inicial
    celda_inicial_c uut_inicial(
    .A(A[0]), //Esto significa que queremos conectar el primer bit de la palabra A a los puertos A,B,X y Y.
    .B(B[0]),
    .X(X_temp[0]),
    .Y(Y_temp[0])
    );
```

Figura 13: Primera Parte del Código de la Red Iterativa Conductual

El cuarto bloque de código (Figura 13) a analizar consiste en la definición del módulo en el que se conectan las 3 celdas previamente definidas. Además, en este módulo, llamado **RedIterativaConductualIaD**, se define e implementa tanto el parámetro N como el bloque **generate.** Analizando el código línea a línea, primero se incluyen los 3 archivos sobre las celdas que definimos antes. Luego, se nombra el módulo y se establece un parámetro N, puede ser cualquier valor, este parámetro ser el encargado de decirle al bloque generate **cuantas celdas típicas generar.** Por ejemplo, puedo correr todo el código con N igual a 1, pero si quiero correrlo de manera seguida con N igual 2, o cualquier otro valor, se debe cambiar en este módulo y en el del testbench, y

se debe correr la secuencia de comandos en la terminal (iverilog, vvp y gtkwave.) Además, porque no está de más, se sugiere utilizar **Save as** y reemplazar el archivo que tenga el mismo nombre.

Luego, se define lo que le entra y sale al módulo. Las **entradas son A y B**, pero definidas con wires y con su tamaño dependiente del parámetro N. La **salida es Zout.** Luego, defino las conexiones entre las celdas como **Xtemp y Ytemp.** para los estados. Importante notar que, los wires de Xtemp y YTemp también dependen del parámetro N. Finalmente, se instancia la celda inicial a través del comando **celda inicial c uut inicial**, donde **uut** es un comando que significa **unit under test.**Entonces, se instancian los puertos y conexiones a utilizar:

- **Al puerto A** se le conecta el bit menos significativo de la señal A.
- **Al puerto B** se le conecta el bit menos significativo de la señal B.
- **Al puerto Xtemp** se le conecta el bit menos significativo de la señal X.
- **Al puerto Ytemp** se le conecta el bit menos significativo de la señal Y.

```
//Instanciacion de la celda tipica con bloque generate
    genvar i;
    generate
        for (i = 0; i < N ; i=i + 1 ) begin: gen_celdas//Bucle For para generar las celdas tipicas en base al parametro N.
                                                       //i=0 inicia el bucle en 0, i<N es la condicion que debe cumplir, i=i+1
                                                       //significa que despues de cada iteracion a i se le suma 1.
                                                       //begin: gen_celdas significa que se comienza un bloque de nombre gen_celdas.
        //codigo para instanciar la celda tipica
        celda_tipica_c uut_tipica(
            .A(A[i]), //el bit i de A se conecta al puerto A, en este caso, se hace asi para que a cada celda tipica le quede un bit,
            .B(B[i]),
            .x(X_temp[i]), //el bit i-1 de X se conecta a x. Si i=0, x=0. De lo contrario, se conecta a i-1.
            .y(Y_temp[i]),
            .X(X_temp[i+1]),
            .Y(Y_temp[i+1])//el bit i+1 de Y se conecta al puerto Y.
        );
        end
    endgenerate

//Instanciacion de la celda final
    celda_final_c uut_final_c(
    .X(X_temp[N]), //Se conecta el bit mas significativo (basado en N) de la senal X al puerto x de la celda final.
    .Y(Y_temp[N]), //Se conecta el bit mas significativo (basado en N) de la senal Y al puerto y de la celda final.
    .Z(Z)
    );

    assign Z_out = (A > B) ? 0 : Z;  // Z_out va a ser el equivalente a 'Z se activa en bajo'.
                    //La logica es, si A>B, Z_out es 0 y si A==B u otro caso, Z_out esta conectado directamente a Z, esto es para garantizar
                    //que las pruebas inician desde 0.Sin esta lógica, por ejemplo, todo los resultados son como se esperan excepto el de la primera prueba.
                    //Sin embargo, al implementar esta lógica, la primera prueba si se comporta como se desea.
endmodule
```

Figura 14: Segunda Parte del Código de la Red Iterativa Conductual

En este quinto bloque de código (Figura 14), se sigue en el mismo módulo. Ahora, siguiendo la descripción línea a línea, se sigue para instanciar la celda típica. Es en esta instanciación que se define el **bloque generate.** Primero se utiliza **genvar** para generar una variable **i.** Luego, se abre el generate y definimos un bucle **for.** Dentro de este bucle for, se tiene que este bucle empieza con **i igual 0**, luego, cuando se cumple la condición que **N sea mayor que i**, a i se le va a sumar 1. Luego, se utiliza un comando **begin** para crear un bloque llamado **gen celdas**. Ahora, a través del comando **celda tipica c uut tipica**, de nuevo, con **unit under test**, se instancian todos los puertos dentro de la celda típica:

- **Al puerto A** se le conecta al bit número i de la señal A.
- **Al puerto B** se le conecta al bit número i la señal B.
- **Al puerto x** se le conecta el bit número i de la señal Xtemp.
- **Al puerto y** se le conecta el bit número i de la señal Ytemp.
- **Al puerto X** se le conecta el bit después del número i de Xtemp.
- **Al puerto Y** se le conecta el bit después del número i de la señal Ytemp.

Luego, se escriben los ends del for y del generate.

Finalmente, se instancia la **celda final** a través de **celda final c uut final c**, siempre con **unit under test.** Los puertos se conectan:

- **Al puerto x** se le conecta el bit número N de la señal Xtemp.
- **Al puerto y** se le conecta el bit número N de la señal Ytemp.
- **Al puerto Z** se le conecta Z.

Para terminar con el código de la Red Iterativa Conductual, en dirección Izquierda a Derecha, se le asigna a **Zout** la condición para que se active en bajo. La logica es, si A¿B, Zout es 0 y si A==B u otro caso, Zout se conecta directamente a Z. Esto es para garantizar que las pruebas inician desde 0.Sin esta lógica, por ejemplo, todo los resultados son como se esperan excepto el de la primera prueba. Sin embargo, al implementar esta lógica, la primera prueba si se comporta como se debe.

Caso derecha a izquierda

A la hora de implementar el caso **Derecha a Izquierda**, al analizar la cantidad de compuertas que hay que utilizar (ver figura 7), se decidió implementar un modelo **estructural** para programar la red iterativa en verilog, para el caso en el que la red se resuelva de DERECHA a IZQUIERDA. Analizando la celda típica, se tienen 3 ANDS, 2 ORs y 1 NOT. Entonces, como la cantidad de compuertas es menor, se implementa el modelo estructural. Esta implementación permite un control más directo sobre el diseño de red. Esto es ya que en el modelo estructural, se definen como módulos, todas las compuertas a utilizar teniendo mayor control sobre ellas.

```
1  module inversor (
2      input wire entrada,
3      output wire salida
4  );
5      assign salida = ~entrada;
6  endmodule
```

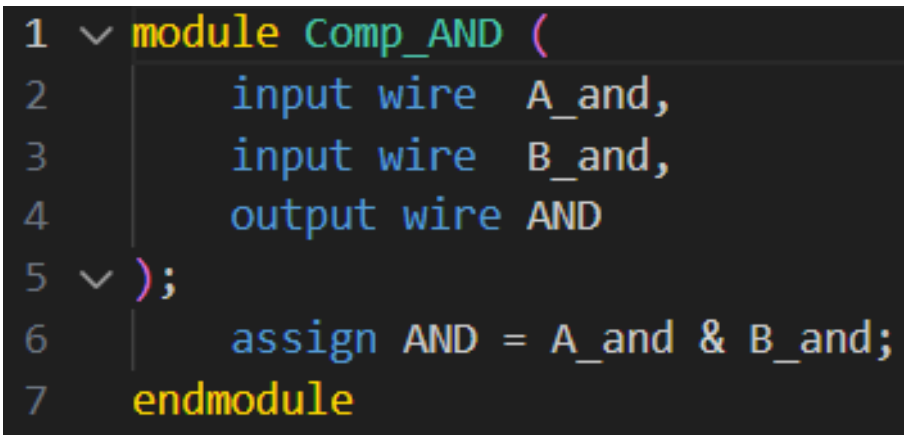
Figura 15: Inversor Para el Modelado Estructural

En la figura 15, se puede ver la definición del módulo inversor a usar para la parte estructural.En donde en este modulo se le asigna la lógica correspondiente.

```
1  module Comp_OR (
2      input wire A_or,
3      input wire B_or,
4      output wire OR
5  );
6      assign OR = A_or | B_or;
7  endmodule
```

Figura 16: Compuerta OR Para el Estructural

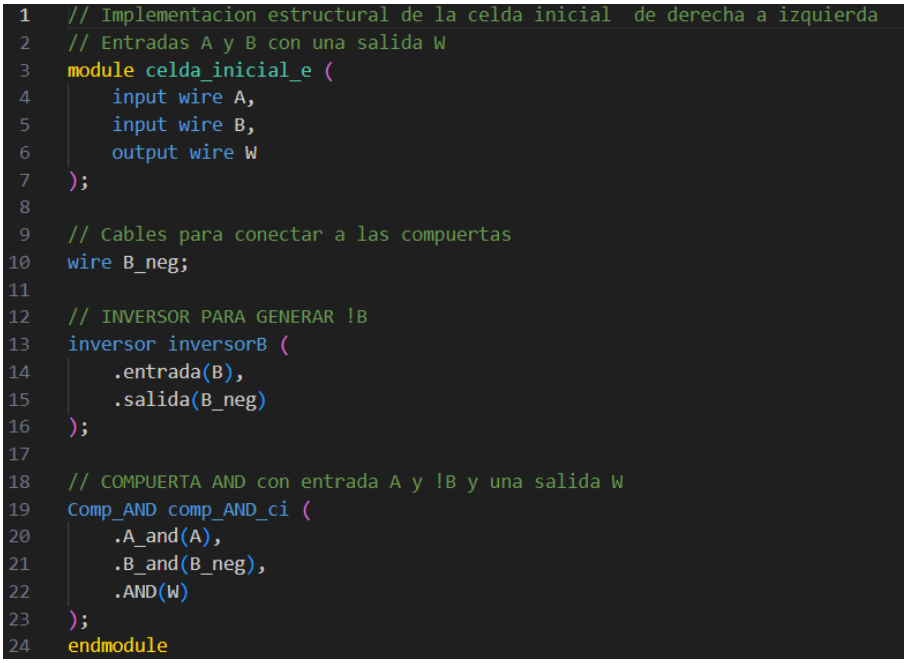
En la figura 16, se puede ver la definición del módulo Comp OR, para la compuerta OR a usar para la parte estructural. En donde en este modulo se le asigna la lógica correspondiente.



```
module Comp_AND (
    input wire  A_and,
    input wire  B_and,
    output wire AND
);
    assign AND = A_and & B_and;
endmodule
```

Figura 17: Compuerta And Para el Estructural

En la figura 17, se puede ver la definición del módulo Comp AND, para la compuerta AND a usar para la parte estructural. Definiendo la lógica correspondiente para este tipo de lógica. Entonces, ya después de haber definido los módulos de compuertas, se analizan los módulos de las celdas a utilizar.



```
// Implementacion estructural de la celda inicial  de derecha a izquierda
// Entradas A y B con una salida W
module celda_inicial_e (
    input wire A,
    input wire B,
    output wire W
);

// Cables para conectar a las compuertas
wire B_neg;

// INVERSOR PARA GENERAR !B
inversor inversorB (
    .entrada(B),
    .salida(B_neg)
);

// COMPUERTA AND con entrada A y !B y una salida W
Comp_AND comp_AND_ci (
    .A_and(A),
    .B_and(B_neg),
    .AND(W)
);
endmodule
```

Figura 18: Celda Inicial para el Estructural

En la figura 18, se define el módulo de nombre **celda inicial e**, con A y B en la parte inputs, y W como salida. Se define el cable negado **Bneg**, que es cable que sale del inversor al que se conecta B. Después, se define el módulo inversor, donde se instancia:

- **Al puerto 'entrada'** se le conecta B.
- **Al puerto 'salida'** se le conecta Bneg.

Para terminar el módulo, se instancia una compuerta AND, la cual es la representación esquemática de la celda inicial, donde A se conecta al puerto **AAnd** y Bneg al puerto **BAnd**. Para la salida **AND**, se conecta W.

```
// Implementacion estructural de la celda tipica  de derecha a izquierda
// Entradas A, B y w con una salida W
module celda_tipica_e (
    input wire A, B, w,
    output wire W
);

// Cables para conectar a las compuertas
wire B_neg;
wire wB_neg_OR;
wire wA_OR;
wire AB_neg_OR;
wire OR1;

// INVERSOR PARA GENERAR !B
inversor inversorB (
    .entrada(B),
    .salida(B_neg)
);

// COMPUERTAS AND
//Con entrada w y !B y una salida w!B que va hacia la compuerta OR 1
Comp_AND comp_AND_ct1 (
    .A_and(w),
    .B_and(B_neg),
    .AND(wB_neg_OR)
```

Figura 19: Primera Parte del Código para la Celda Típica Estuctural

Ahora, para definir la **celda típica estructural**, primero se define el módulo con el nombre celda típica e, con entradas **A,B, y w**, y salida **W**. Luego, se definen los cables a usar, como si fuera una ferretería; **Bneg, wbnegOR, wAOR, ABnegOR y OR1**, los nombres más o menos definen donde van a ser usados.

- Primero, se instancia para un inversor, al cual se le conecta la señal B en la **entrada** y Bneg para la **salida.**
- Luego, se instancia para una compuerta AND1, donde se conecta w al puerto **Aand**, Bneg al puerto **Band** y, a la salida **AND** se conecta el cable **wBnegOR.**

```
//Con entrada w y A y una salida wA que va hacia la compuerta OR 1
Comp_AND comp_AND_ct2 (
    .A_and(w),
    .B_and(A),
    .AND(wA_OR)
);
//Con entrada A y !B y una salida A!B que va hacia la compuerta OR 2
Comp_AND comp_AND_ct3 (
    .A_and(A),
    .B_and(B_neg),
    .AND(AB_neg_OR)
);
// COMPUERTAS OR
//Con entrada w!B y wA y una salida OR1 que va hacia la compuerta OR 2
Comp_OR comp_OR_ct1 (
  .A_or (wB_neg_OR),
  .B_or (wA_OR),
  .OR (OR1)
);
//Con entrada OR1 y A!B y una salida W
Comp_OR comp_OR_ct2 (
  .A_or (OR1),
  .B_or (AB_neg_OR),
  .OR (W)
);
endmodule
```

Figura 20: Segunda del Código para la Celda Típica Estuctural

Siguiendo con la instanciación:

- Ahora, se instancia para una compuerta AND2, al cual se le conecta la señal w al puerto **Aand**, la señal A al puerto **Band** y en el puerto **AND** sale **wAOR.**
- Ahora, se instancia para una compuerta AND3, al cual se le conecta la señal A al puerto **Aand**, el cable Bneg al puerto **Band** y en el puerto **AND** sale **ABnegOR.**
- Se sigue instanciando para una compuerta OR1, al cual se le conecta el cable **WBnegOR** al puerto **Aor**, el cable **wAOR** al puerto **Bor** y en el puerto **OR** sale **OR1**.
- Ahora, se instancia para una compuerta OR2, al cual se le conecta el cable OR1 al puerto **Aor**, el cable ANnegOR al puerto **Band** y en el puerto **OR** sale **W**.

```
// Implementacion estructural de la celda final  de derecha a izquierda
// Entrada W que es la entrada de la celda tipica y con una salida de Z
module celda_final_e (
    input wire W,
    input wire A,
    input wire B,
    output wire Z
);

// Cable para conectar a la compuerta
wire W_neg;

// INVERSOR PARA GENERAR !W que al final se obitne Z=!W
inversor inversorW_cf (
    .entrada(W),
    .salida(Z)
 );
endmodule
```

Figura 21: Código de la Celda Final Estructural

Para terminar con los módulos, para la celda final, se le definen **W,A y B** como entradas y **Z** como salida. Se establece un cable **Wneg**. Para instanciar el módulo del inversor, se conecta W en la entrada de este y Z en la salida. En este caso se definió el cable por redundancia, ya que no fué necesario utilizarlo.

```
//Proyecto Final de Digitales,C01263,B82600,B86941.Implementacion Estructural, caso DERECHA a IZQUIERDA.
`include "celda_inicial_e.v"
`include "celda_tipica_e.v"
`include "celda_final_e.v"
`include "Comp_AND.v"
`include "Comp_OR.v"
`include "inversor.v"
//Como es implementacion estructural, se llaman los archivos de las compuertas logicas, junto las celdas tipicas.
module Comparador #(parameter N = 5)(
    input wire [N-1:0] A, //definiendo los tamanos de A, B y W
    input wire [N-1:0] B,
    output wire  W_out);
    wire [N:0] W_temp;

    celda_inicial_e uut_inicial(
        .A(A[0]),               //Conectamos todos los puertos a los bits iniciales
        .B(B[0]),
        .W(W_temp[0])
    );
    genvar i;
    generate
        for (i = 0; i < N; i=i +1 ) begin: gen_celdas //Bucle For para generar las celdas tipicas en base al parametro N.
                                                      //i=0 inicia el bucle en 0, i<N es la condicion que debe cumplir, i=i+1
                                                      //significa que despues de cada iteracion a i se le suma 1.
                                                      //begin: gen_celdas significa que se comienza un bloque de nombre gen_celdas.
        //codigo para instanciar la celda tipica
        celda_tipica_e uut_tipica(
            .A(A[i]), //el bit i de A se conecta al puerto A, en este caso, se hace asi para que a cada celda tipica le quede un bit,
            .B(B[i]),
            .W(W_temp[i]), //el bit i-1 se conecta a w
            .W(W_temp[i+1])//el bit i+1 de Y se conecta al puerto Y.
        );
        end
    endgenerate
    celda_final_e uut_final_e( //Ya que la celda final es diferente para este diseno, conectamos los puertos A,B y W al bit numero 'N'
    .W(W_temp[N]),
    .Z(Z));
    assign W_out =  Z;//Misma logica para Z.
endmodule
```

Figura 22: Código de la Red Iterativa Estructural

Para la red iterativa, que se le puso **Comparador**, la conexión de todos los puertos es bastante similar al caso de la primera red. Primero, se incluyen las celdas típica, inicial y final, y además se incluyen los archivos de las compuertas y el inversor. Luego, se declara el módulo, junto con el parámetro N a utilizar. Luego, se

declaran A y B como wires, que dependen del valor N. Se declara Wout como salida. Además, se declara un Wtemp, que tamién depende N y va a ser usado para conectar las celdas una con otra. Primero, se instancia la celda inicial, conectando A al bit inicial de A, B con el bit inicial de B y al puerto W se le conecta el bit inicial de Wout. Después, se sigue con el bloque generate. Se delara una variable i a través de **Genvar** para el bloque Generate. Se sigue con un bucle for para generar las celdas típicas en base al parámetro N; i=0 inicia el bucle, una vez que se cumpla la condición i menor que N, a i se le suma 1.
Ahora, se procede a instanciar al celda típica:

- El bit i de A se conecta en el puerto A.
- El bit i de B se conecta en el puerto B.
- El bit i de la Wout, se conecta en el puerto w.

Ahora, se instancia la celda final, con el uut, como siempre. En la celda final:

- El bit N de Wtemp se conecta en el puerto W.
- Z se conecta directamente al puerto Z.

Finalmente, se aplica un asign a Wout, para que este sea Z.

3.3. Pruebas

Test Bench del Caso Izquierda a Derecha

Ahora, para analizar el test bench (banco de pruebas) línea a línea:

```
1  //Proyecto Final de Digitales,C01263,B82600,B86941.
2  //Implementacion Conductual al caso IZQUIERDA a DERECHA.
3  `timescale 1ns/1ns
4  `include "RedIterativaConductualIaD.v"
5  //*Importante:Para GtkWave, el valor que importa es Z_out.
6  module RedIterativaConductualIaD_tb#(parameter N = 3);//SI VAN A CAMBIAR EL PARAMETRO, TANTO ESTE COMO EL DEL OTRO MODULO DEBEN SER IGUALES!!!!
7    reg [N-1:0] A, B;
8    wire Z_out;
9    reg reset;
10
11
12   // Instanciación del módulo bajo prueba
13
14    initial begin
15          reset=1'b1;//se agrega un reinicio para garantizar que al inicio de cada prueba, las ondas empiecen en 0
16          #10//ademas, se agrega un retardo despues de cada reset.
17          reset=1'b0;
18     end
19   RedIterativaConductualIaD uut (
20     .A(A),
21     .B(B),
22     .Z_out(Z_out)
23   );
24
25   // Lógica para aplicar pruebas a la red.
26   initial begin
27     #10 reset = 1;
28     #10 reset = 0;
29     // Asigno valores iniciales a las entradas
30     $dumpfile ("RedIterativaConductualIaD_tb.vcd");//Para generar el .vcd para gtkwave
31     $dumpvars (0,uut);
```

Figura 23: Primera Parte del Código del Banco de Pruebas Conductual

De las líneas 3 a 4 (en la figura 23), se incluyen una escala de tiempo de **1 nano segundo**, que es el tiempo en el que estamos corriendo la simulación, y además, se **incluye** el archivo donde está montada la Red Iterativa que queremos probar. Luego, se define el módulo, en este caso se le puso **RedIterativaConductualIaDtb**, donde se define también el parámetro N. **IMPORTANTE, SI SE BUSCA PROBAR O GENERAR PALABRAS CON N DIFERENTE, SE DEBEN MODIFICAR AMBOS, TANTO EL N DE LA RED NORMAL COMO EL DEL BANCO DE PRUEBAS.** Luego, se definen las entradas A y B, como **registers**, de tamaño N menos 1. Luego, se define Zout como un wire y finalmente se define el reset como otro register, el cual va a reiniciar la señal de A y B para cada prueba.

Entonces, eventualmente llegamos a un bloque **Initial begin**, donde se define el valor de reset como 1, con un retardo de diez unidades de tiempo (10 nano segundos) y después se pone el reset como 0. Luego se procede a instanciar a través de **RedIterativaConductualIaD uut**, siguiendo siempre el unit under test:

- **Al puerto A** se le conecta a la señal A.
- **Al puerto B** se le conecta a la señal B.
- **Al puerto Zout** se le conecta a la salida Zout.

Ahora, a partir de este punto se implementan las pruebas a correr al modulo principal. Se inicia el reset, y se generan el **dumpfile** para generar el .vcd y el **dumpvars.**

```
33    //Se decidio aplicar pruebas con valores aleatorios. Estos valores aleatorios, se generan de un rango de 0 al parametro N.
34    //Esto significa que, por ejemplo, si N=7, A y B van a ser palabras de 7 bits, y dentro de esos 7 bits los 0s y 1s se generan
35    //de manera aleatoria. Esto garantiza probar combinaciones diferentes de bits, en vez de tener que hacerlas manualmente.
36    //El proceso es eficiente, y pone aun mas a prueba el funcionamiento de la red.
37    //Prueba 1
38    A = $urandom_range(0, (1<<N)-1);
39    B = $urandom_range(0, (1<<N)-1);
40    #1000 reset = 1;//retardo despues del reset
41    #10 reset = 0;//retardo despues del reset
42    #1000; //retardo
43    $display("Valor: A = %b, B = %b, Z_out = %b", A, B, Z_out);//esto es para que en la terminal, se impriman los valores del comparador
44                                                              //para garantizar que funcione.
45    //A partir de aca, se copia y pega las pruebas deseadas a correr. Mientras mas, mejor. Se decidio en ejecutar 10 pruebas de manera arbitraria.
46    //Prueba 2
47    A = $urandom_range(0, (1<<N)-1);
48    B = $urandom_range(0, (1<<N)-1);
49    #1000 reset = 1;
50    #10 reset = 0;
51    #1000;
52    $display("Valor: A = %b, B = %b, Z_out = %b", A, B, Z_out);
53  //Prueba 3
54    A = $urandom_range(0, (1<<N)-1);
55    B = $urandom_range(0, (1<<N)-1);
56    #1000 reset = 1;
57    #10 reset = 0;
58    #1000;
59    $display("Valor: A = %b, B = %b, Z_out = %b", A, B, Z_out);
```

Figura 24: Segunda Parte del Código del Banco de Pruebas Conductual

Como se puede ver en la Figura 24, las pruebas a implementar se definen así:

- **A** es un número aleatorio generado en un rango de 0 a N bits
- **B** es un número aleatorio generado en un rango de 0 a N bits
- Se introducen **resets y retardos** para el comportamiento de las ondas.
- Se corre un total de **10 pruebas.**

```
60  //Prueba 4
61    A = $urandom_range(0, (1<<N)-1);
62    B = $urandom_range(0, (1<<N)-1);
63    #1000 reset = 1;
64    #10 reset = 0;
65    #1000;
66    $display("Valor: A = %b, B = %b, Z_out = %b", A, B, Z_out);
67  //Prueba 5
68    A = $urandom_range(0, (1<<N)-1);
69    B = $urandom_range(0, (1<<N)-1);
70    #1000 reset = 1;
71    #10 reset = 0;
72    #1000;
73    $display("Valor: A = %b, B = %b, Z_out = %b", A, B, Z_out);
74
75  //Prueba 6
76    A = $urandom_range(0, (1<<N)-1);
77    B = $urandom_range(0, (1<<N)-1);
78    #1000 reset = 1;
79    #10 reset = 0;
80    #1000;
81    $display("Valor: A = %b, B = %b, Z_out = %b", A, B, Z_out);
82
83    //Prueba 7
84    A = $urandom_range(0, (1<<N)-1);
85    B = $urandom_range(0, (1<<N)-1);
86    #1000 reset = 1;
87    #10 reset = 0;
88    #1000;
89    $display("Valor: A = %b, B = %b, Z_out = %b", A, B, Z_out);
```

Figura 25: Tercera Parte del Código del Banco de Pruebas Conductual

Finalmente, se llega al **display** que pone 'end of test.' para indicarnos que ya finalizó el testbench. Además, se pone un display al final de cada prueba para que se nos muestre la palabra A, la palabra B, y el valor de Zout. (Ver figura 20.)

```
    //Prueba 8
    A = $urandom_range(0, (1<<N)-1);
    B = $urandom_range(0, (1<<N)-1);
    #1000 reset = 1;
    #10 reset = 0;
    #1000;
    $display("Valor: A = %b, B = %b, Z_out = %b", A, B, Z_out);

    //Prueba 9
    A = $urandom_range(0, (1<<N)-1);
    B = $urandom_range(0, (1<<N)-1);
    #1000 reset = 1;
    #10 reset = 0;
    #1000;
    $display("Valor: A = %b, B = %b, Z_out = %b", A, B, Z_out);

    //Prueba 10
    A = $urandom_range(0, (1<<N)-1);
    B = $urandom_range(0, (1<<N)-1);
    #1000 reset = 1;
    #10 reset = 0;
    #1000;
    $display("Valor: A = %b, B = %b, Z_out = %b", A, B, Z_out);
    // Mostrar resultados
    $display("end of test.");

    // Si se desea generar otro 'set' de datos random, se debe borrar el .vcd y el .vvp, y cambiar el parametro N.
  end

endmodule
```

Figura 26: Cuarta Parte del Código del Banco de Pruebas Conductual

Test Bench del Caso Derecha a Izquierda

A pesar de casos diferentes, los bancos de prueba realmente no difieren mucho entre ambos casos. Se definen cadenas de bits aleatorios que van a ser generados en base al parámetro N. Por ejemplo, si se escoge un N que es igual a 3, se van a generar un A que va a ser igual a 100 y un B que va a ser 110, si se escoge un N igual a 4, entonces las palabras serían A igual a 1001 y B igual 0000. La longitud la define el N, que a la vez va a determinar la cantidad de celdas típicas que se van a crear en el módulo, y los bits que llenan esa palabra son aleatorios. Para garantizar su aleatoriedad, **DESPUÉS DE HABER CAMBIADO EL PARÁMETRO N TANTO EN EL MÓDULO COMO EL TESTBENCH,** se recomienda guardar el archivo y nuevamente, reemplazar el archivo del test bench con el nuevo.

```
//Proyecto Final de Digitales,C01263,B82600,B86941.
//Implementacion Estructural al caso DERECHA a IZQUIERDA.
`timescale 1ns/1ns
`include "Comparador .v"

//*Importante:Para GtkWave, el valor que importa es W_out. Para efectos del proyecto, la salida Z que nos piden en este caso va a ser W, se trae asi porque asi se hizo
//diseño.
module Comparador_tb #(parameter N = 5);//SI VAN A CAMBIAR EL PARAMETRO, TANTO ESTE COMO EL DEL OTRO MODULO DEBEN SER IGUALES!!!!
  reg [N-1:0] A, B;
  wire W_out;
  reg reset;


  // Instanciación del módulo bajo prueba

  initial begin
      reset=1'b1;//se agrega un reinicio para garantizar que al inicio de cada prueba, las ondas empiecen en 0
      #10//ademas, se agrega un retardo despues de cada reset.
      reset=1'b0;
  end
Comparador uut (
  .A(A),
  .B(B),
  .W_out(W_out)
);

// Lógica para aplicar pruebas a la red.
initial begin
  #10 reset = 1;
  #10 reset = 0;
  // Asigno valores iniciales a las entradas
  $dumpfile ("Comparador_tb.vcd");//Para generar el .vcd para gtkwave
  $dumpvars (0,uut);

  //Se decidio aplicar pruebas con valores aleatorios. Estos valores aleatorios, se generan de un rango de 0 al parametro N.
  //Esto significa que, por ejemplo, si N=7, A y B van a ser palabras de 7 bits, y dentro de esos 7 bits los 0s y 1s se generan
  //de manera aleatoria. Esto garantiza probar combinaciones diferentes de bits, en vez de tener que hacerlas manualmente.
  //El proceso es eficiente, y pone aun mas a prueba el funcionamiento de la red.
```

Figura 27: Primera Parte del Código para el Testbench.

Ahora, como los códigos son virtualmente iguales se va a pasar al final del código, ya que solo se definen las pruebas en el código que no se muestra, y estos son iguales al otro testbench.

```
    //Prueba 10
    A = $urandom_range(0, (1<<N)-1);
    B = $urandom_range(0, (1<<N)-1);
    #1000 reset = 1;
    #10 reset = 0;
    #1000;
    $display("Valor: A = %b, B = %b, W_out = %b", A, B, W_out);

    // Mostrar resultados
    $display("end of test.");

    // Si se desea generar otro 'set' de datos random, se debe borrar el .vcd y el .vvp, y cambiar el parametro N.
  end
endmodule
```

Figura 28: Segunda Parte del Código para el Testbench.

Lo que se vé en la figura 28, es el código en general para cada prueba, esa es la prueba número 10, las otras 9 fueron iguales. Finalmente, se llega al .endz damos por conluida la descripción del código.

4. Análisis de resultados

Importante: Ejecutar el comando vvp en la terminal va a generar un archivo .vcd, llamado "RedIterativaConductualIaD tb.vcd" para el caso CONDUCTUAL y "Comparador tb.vcd"para el caso ESTRUCTURAL, solo que existe la posibilidad de que no aparezca el .vcd a la hora de buscar el archivo en la computadora. **En caso de que eso suceda,** lo que debe pasar es que se genere una copia de "RedIterativaConductualIaD tb" o bien "Comparador tb", uno de los dos es el .v y el otro es el .vcd. Haciendo click derecho sobre el archivo, y abriendo **Properties** se puede ver qué tipo de archivo es el que se está subiendo a GTKWave. Es una cuestión de nomenclatura, no formato.

Caso izquierda a derecha

```
Valor: A = 1, B = 0, Z = 0
Valor: A = 0, B = 0, Z = 1
Valor: A = 1, B = 1, Z = 1
Valor: A = 0, B = 0, Z = 1
Valor: A = 1, B = 1, Z = 1
Valor: A = 1, B = 1, Z = 1
Valor: A = 1, B = 1, Z = 1
Valor: A = 1, B = 0, Z = 0
Valor: A = 0, B = 0, Z = 1
Valor: A = 1, B = 0, Z = 0
end of test.
```

Figura 29: Resultados para N=1

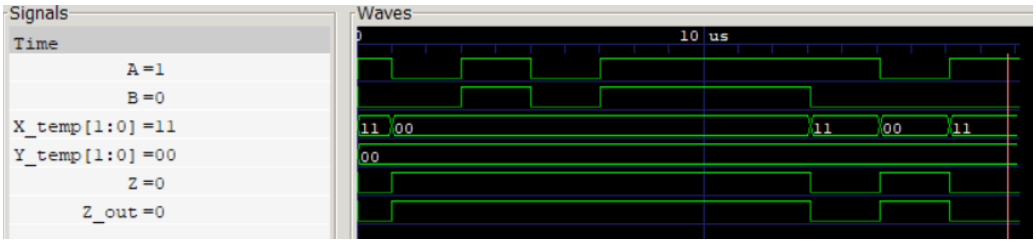


Figura 30: Ondas para Señales A y B con N=1 bits

De la **Figura 29**, se pueden ver los resultados que se decidieron imprimir (a través del comando display), a la hora de probar **A y B con 1 bit.** o sea, con **N=1.** Cuando A es 1, y B es 0, Zout es 0. Esto cumple con la lógica, ya que Z se activa con un valor de 0 cuando **A es mayor de B.** Para la siguiente prueba, A es 0 y B es 0, por lo que Zout es 1. Esto sigue cumpliendo con la lógica, ya que cuando **A y B son iguales**, Zout tiene un valor de 1. Las pruebas siguen, buscando que se cumplan todos los casos posibles, ya que hay **4** combinaciones posibles, porque es una combinación de 2 bits. La **Prueba 3**, presenta los casos A y B con valor de 1, por lo que Zout es 1 y la lógica se sigue cumpliendo. Se alterna entre los valores iguales de A y B (A y B son ambos 0, A y B son ambos 1), hasta que se llega a A con valor de 1 y B con valor de 1, los casos de la prueba 1. Para todas las pruebas, la lógica se cumple. Se pueden agregar más pruebas a gusto libre, se decidieron 10 de manera arbitraria, ya que cada prueba genera un pulso en las ondas de GTKWave, como se puede ver en la **Figura 30**.

```
Valor: A = 100, B = 010, Z = 0
Valor: A = 000, B = 001, Z = 1
Valor: A = 100, B = 110, Z = 1
Valor: A = 001, B = 000, Z = 0
Valor: A = 100, B = 100, Z = 1
Valor: A = 101, B = 100, Z = 0
Valor: A = 111, B = 110, Z = 0
Valor: A = 111, B = 011, Z = 0
Valor: A = 011, B = 010, Z = 0
Valor: A = 111, B = 001, Z = 0
end of test.
```

Figura 31: Resultados para N=3

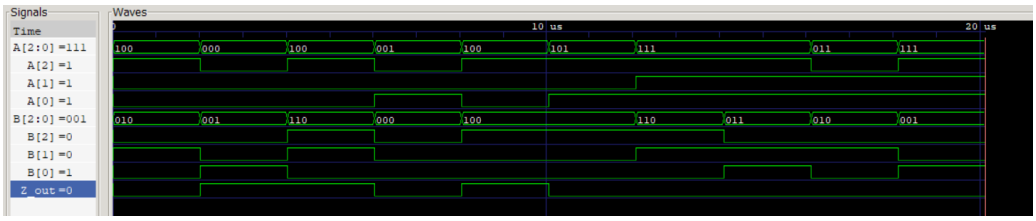


Figura 32: Ondas para Señales A y B con N=3 bits

De la **Figura 31**, se pueden ver los resultados que se decidieron imprimir (a través del comando display), a la hora de probar **A y B con 3 bits.** o sea, con **N=3.** Cuando A es 100, y B es 010, Zout es 0. Esto cumple con la lógica, ya que Z se activa con un valor de 0 cuando **A es mayor de B.** Para la siguiente prueba, A es 000 y B es 001, por lo que Zout es 1. Esto cumple con la lógica, ya que **B es mayor que A** y, Zout tiene un valor de 1 en este caso. Las pruebas siguen, buscando que se cumplan todos los casos posibles, ya que hay **8** combinaciones posibles por bits y como son dos señales, los casos aumentan a **64**. Para la **Prueba 3**, se presentan los casos A y B con valor de 100 y 110 de manera respectivamente, por lo que Zout es 1 y la lógica se sigue cumpliendo. Luego, A es 001 y B es 000. Zout es 0, el código sigue funcionando. Para la **Prueba 5**, A y B son ambos 100, entonces en el caso de que **A es igual a B**, por lo que Zout es 1. A partir de este punto, en todas las pruebas A es mayor que B, por lo que Zout es 0.Además, las ondas, al analizarlas pulso a pulso, reflejan el comportamiento esperado, ver **Figura 32**.

```
Valor: A = 10010010000101010011010100100100, B = 01000000100010010101111010000001, Z = 0
Valor: A = 00000100100001001101011000001001, B = 00110001111100000101011001100011, Z = 1
Valor: A = 10000110101110010111101100001101, B = 11000110110111111001100110001101, Z = 1
Valor: A = 00110010110000101000010001100101, B = 00001001001101110101001000010010, Z = 0
Valor: A = 10000000111100111110001100000001, B = 10000110110101111100110100001101, Z = 1
Valor: A = 10111011001000111111000101110110, B = 10011110100011011100110100111101, Z = 0
Valor: A = 11110110110101000101011111101101, B = 11000110001011011111011110001100, Z = 0
Valor: A = 11111100111111011110100111111001, B = 01100011001101110010010011000110, Z = 0
Valor: A = 01100010111101111000010011000101, B = 01010101000100111101001010101010, Z = 0
Valor: A = 11110010101011111111101111100101, B = 00111011110100100111001001110111, Z = 0
end of test.
```

Figura 33: Resultados para N=32

De la **Figura 33**, se pueden ver los resultados que se decidieron imprimir (a través del comando display), a la hora de probar **A y B con 32 bits.** o sea, con **N=32.** Ahora, como hemos verificados los casos:

- **A mayor que B**, causa que **Zout sea 0.**
- **A igual o menor que B**, causa que **Zout sea 1.**

Para este caso, no tiene sentido analizar con tanto detalle, ya que hay 2^{64} combinaciones posibles entre dos señales de 32 bits. Lo importante es que, visualmente nosotros podemos verificar de manera manual el valor de Zout. Por ejemplo, para la **Prueba 1**, para nosotros basta con ver las diferencias entre los bits más significativos, que se encuentran en la **izquierda**. Sin embargo, para la red se ocupan más estados cuando se lee de izquierda a derecha, ya que con solo que **el bit de A sea mayor que el bit de B** en su bit más significativo, Zout va a ser 0. Con solo que **el bit de A sea menor o igual al bit de B**, es suficiente para que Zout sea 1, porque es el bit más grade de ambas señales. Ahora, para el caso que **el bit de A sea igual al bit de B**, se pasa al siguiente bit, **pero eso no significa** que la señal de Zout sea 1 de manera directa. Para que Zout sea 1 a través de la condición **A igual a B**, se ocupa que **toda la cadena de bits de A y B sean iguales.** Esto significa que, mientras más bits hayan presentes, va a ser más difícil que los bits sean exactamente iguales. Entonces, es más probable que Zout sea 1 solo porque se cumple el caso de **el bit de A sea menor o igual al bit de B**.

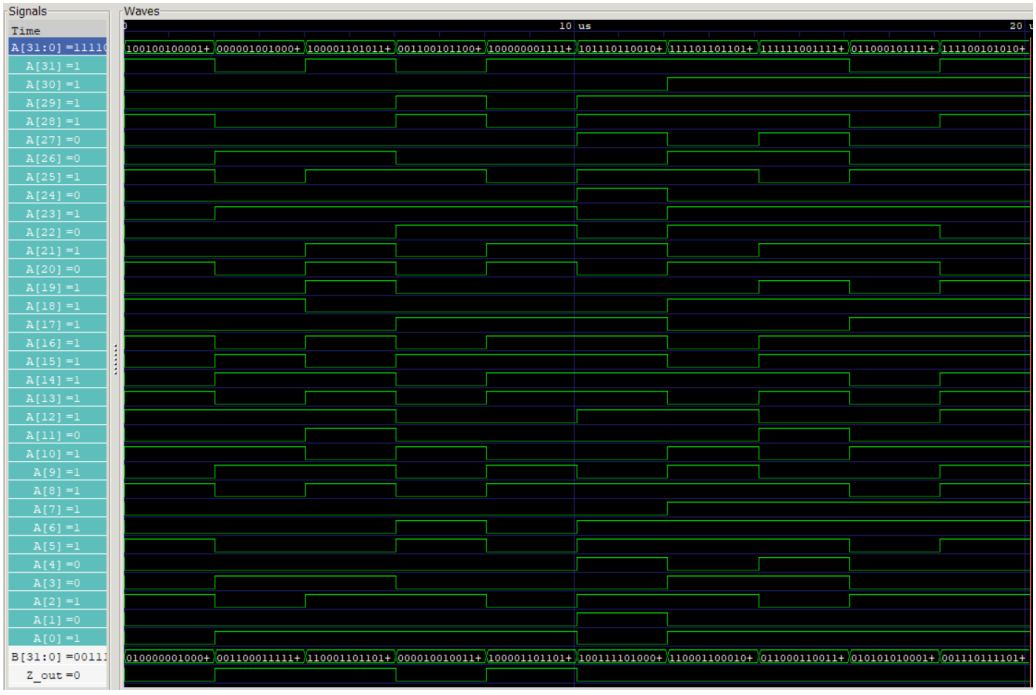


Figura 34: Ondas para Señal A con N=32 bits

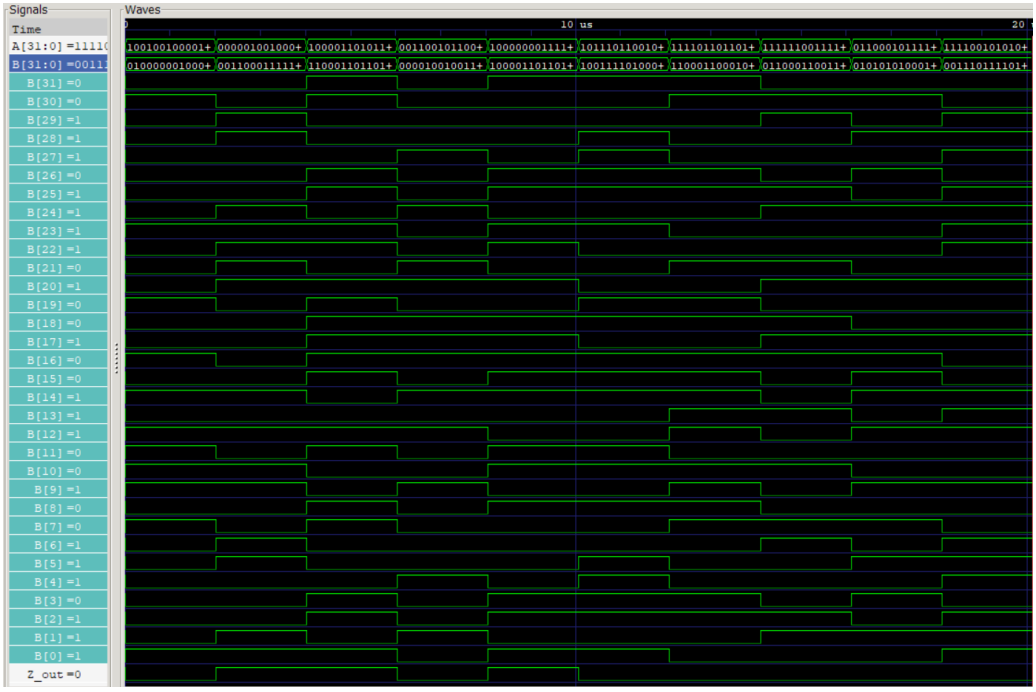


Figura 35: Ondas para Señal B con N=32 bits

Caso derecha a izquierda

```
Valor: A = 1, B = 0, W_out = 0
Valor: A = 0, B = 0, W_out = 1
Valor: A = 1, B = 1, W_out = 1
Valor: A = 0, B = 0,W_out = 1
Valor: A = 1, B = 1,W_out = 1
Valor: A = 1, B = 1, W_out = 1
Valor: A = 1, B = 1, W_out = 1
Valor: A = 1, B = 0, W_out = 0
Valor: A = 0, B = 0, W_out = 1
Valor: A = 1, B = 0, W_out  = 0
Valor: A = 0, B = 1, W_out = 1
end of test.
```

Figura 36: Resultados para N=1

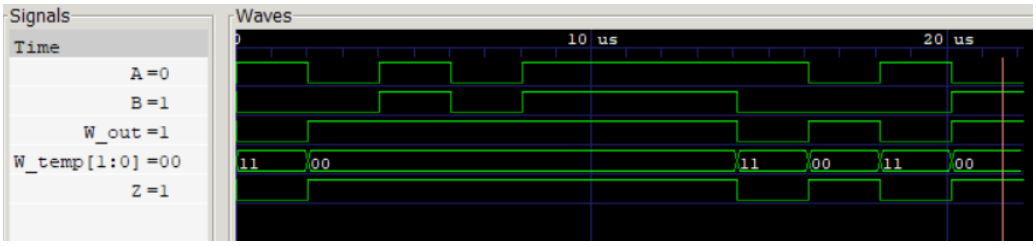


Figura 37: Ondas para Señales A y B con N=1 bits

De la **Figura 36**, se pueden ver los resultados que se decidieron imprimir (a través del comando display), a la hora de probar **A y B con 1 bit.** o sea, con **N=1.** Cuando A es 1, y B es 0, Wout es 0. Esto cumple

con la lógica, ya que Z se activa con un valor de 0 cuando **A es mayor de B.** Recordar que para este caso, W va ser nuestro Z. Para la siguiente prueba, A es 0 y B es 0, por lo que Zout es 1. Esto sigue cumpliendo con la lógica, ya que cuando **A y B son iguales**, Zout tiene un valor de 1. Las pruebas siguen, buscando que se cumplan todos los casos posibles, ya que hay **4** combinaciones posibles, porque es una combinación de 2 bits. La **Prueba 3**, presenta los casos A y B con valor de 1, por lo que Zout es 1 y la lógica se sigue cumpliendo. Se alterna entre los valores iguales de A y B (A y B son ambos 0, A y B son ambos 1), hasta que se llega a A con valor de 1 y B con valor de 1, los casos de la prueba 1. Para todas las pruebas, la lógica se cumple. Se pueden agregar más pruebas a gusto libre, se decidieron 10 de manera arbitraria, ya que cada prueba genera un pulso en las ondas de GTKWave, como se puede ver en la **Figura 37**.

```
Valor: A = 100, B = 010, W_out = 0
Valor: A = 000, B = 001, W_out = 1
Valor: A = 100, B = 110, W_out = 1
Valor: A = 001, B = 000,W_out = 0
Valor: A = 100, B = 100,W_out = 1
Valor: A = 101, B = 100, W_out = 0
Valor: A = 111, B = 110, W_out = 0
Valor: A = 111, B = 011, W_out = 0
Valor: A = 011, B = 010, W_out = 0
Valor: A = 111, B = 001, W_out  = 0
Valor: A = 000, B = 110, W_out = 1
end of test.
```

Figura 38: Resultados para N=3

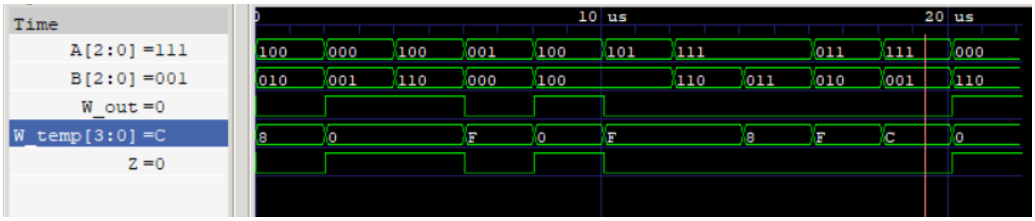


Figura 39: Resultados para N=3

De la **Figura 38**, se pueden ver los resultados que se decidieron imprimir (a través del comando display), a la hora de probar **A y B con 3 bits.** o sea, con **N=3.** Cuando A es 100, y B es 010, Zout es 0. Esto cumple con la lógica, ya que Z se activa con un valor de 0 cuando **A es mayor de B.** Para la siguiente prueba, A es 000 y B es 001, por lo que Zout es 1. Esto cumple con la lógica, ya que **B es mayor que A** y, Zout tiene un valor de 1 en este caso. Las pruebas siguen, buscando que se cumplan todos los casos posibles, ya que hay **8** combinaciones posibles por bits y como son dos señales, los casos aumentan a **64**. Para la **Prueba 3**, se presentan los casos A y B con valor de 100 y 110 de manera respectivamente, por lo que Zout es 1 y la lógica se sigue cumpliendo. Luego, A es 001 y B es 000. Zout es 0, el código sigue funcionando. Para la **Prueba 5**, A y B son ambos 100, entonces en el caso de que **A es igual a B**, por lo que Zout es 1. A partir de este punto, en todas las pruebas A es mayor que B, por lo que Zout es 0.Además, las ondas, al analizarlas pulso a pulso, reflejan el comportamiento esperado, ver **Figura 39**. Algo interesante es el comportamiento de Wout, pero como la red funciona debidamente, queda como una curiosidad.

```
Valor: A = 10010010000101010011010100100100, B = 01000000100010010101111010000001, W_out = 0
Valor: A = 00000100100001001101011000001001, B = 00110001111100000101011001100011, W_out = 1
Valor: A = 10000110101110010111101100001101, B = 11000110110111111001100110001101, W_out = 1
Valor: A = 00110010110000101000010001100101, B = 00001001001101110101001000010010,W_out = 0
Valor: A = 10000000111100111111000100000001, B = 10000110110101111100110100001101,W_out = 1
Valor: A = 10111011001000111111000101110110, B = 10011110100011011100110100111101, W_out = 0
Valor: A = 11110110110101000101011111101101, B = 11000110001011011111011110001100, W_out = 0
Valor: A = 11111100111111011110100111111001, B = 01100011001101110010010011000110, W_out = 0
Valor: A = 01100010111101111000010011000101, B = 01010101000100111101001010101010, W_out = 0
Valor: A = 11110010101011111111011111100101, B = 00111011110100100111001001110111, W_out  = 0
Valor: A = 00001001001100101101011000010010, B = 11000111111011001101101110001111, W_out = 1
```

Figura 40: Resultados para N=32

De la **Figura 40**, se pueden ver los resultados que se decidieron imprimir (a través del comando display), a la hora de probar **A y B con 32 bits.** o sea, con **N=32.** Ahora, como hemos verificados los casos:

- **A mayor que B**, causa que **Zout sea 0.**
- **A igual o menor que B**, causa que **Zout sea 1.**

Para este caso, no tiene sentido analizar con tanto detalle, ya que hay 2^{64} combinaciones posibles entre dos señales de 32 bits. Lo importante es que, visualmente nosotros podemos verificar de manera manual el valor de Zout. Por ejemplo, para la **Prueba 1**, para nosotros basta con ver las diferencias entre los bits más significativos, que se encuentran en la **izquierda**. Sin embargo, para la red se ocupan más estados cuando se lee de izquierda a derecha, ya que con solo que **el bit de A sea mayor que el bit de B** en su bit más significativo, Zout va a ser 0. Con solo que **el bit de A sea menor o igual al bit de B**, es suficiente para que Zout sea 1, porque es el bit más grade de ambas señales. Ahora, para el caso que **el bit de A sea igual al bit de B**, se pasa al siguiente bit, **pero eso no significa** que la señal de Zout sea 1 de manera directa. Para que Zout sea 1 a través de la condición **A igual a B**, se ocupa que **toda la cadena de bits de A y B sean iguales.** Esto significa que, mientras más bits hayan presentes, va a ser más difícil que los bits sean exactamente iguales. Entonces, es más probable que Zout sea 1 solo porque se cumple el caso de **el bit de A sea menor o igual al bit de B**.

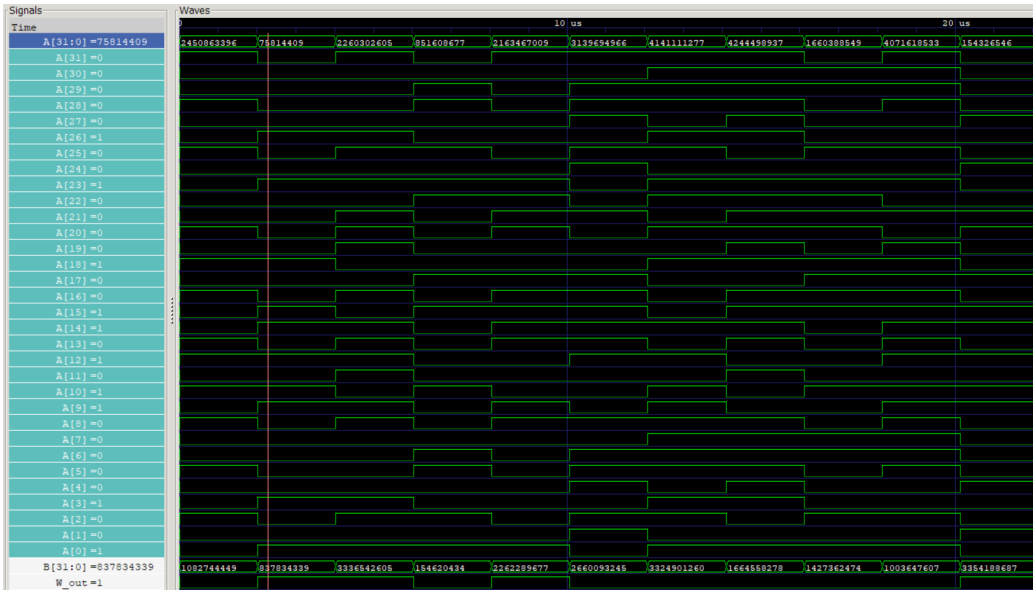


Figura 41: Ondas para Señal A con N=32 bits

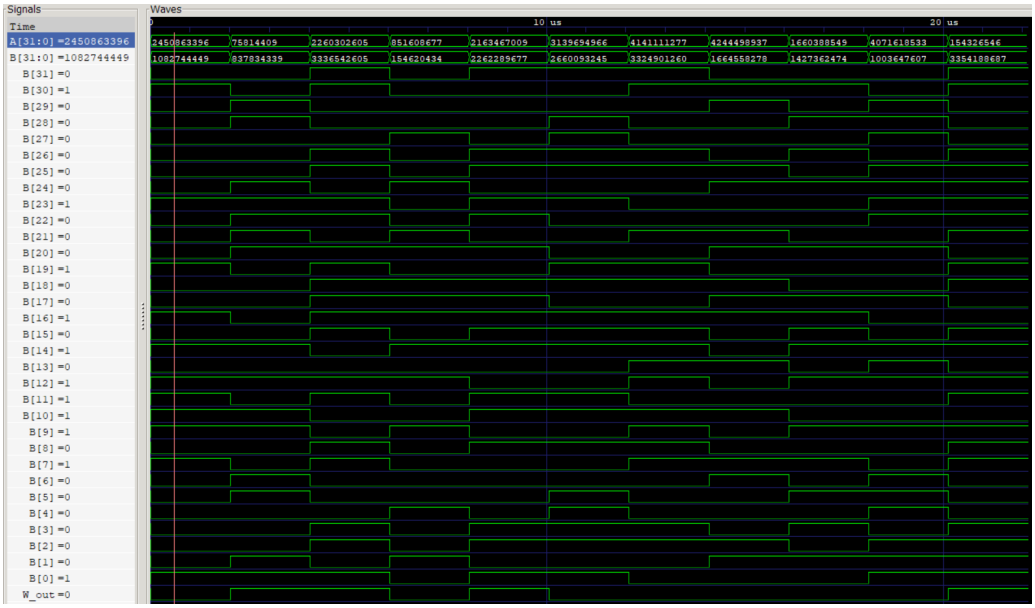


Figura 42: Ondas para Señal B con N=32 bits

5. Conclusiones

Al finalizar, la ejecución del proyecto se puede confirmar el éxito del mismo donde se consigue completar los objetivos planteados, obteniendo así una red iterativa que compara palabras aleatorias bajo a la lógica y especificaciones ya descritas en el enunciado. Para reiterarlas, Z se activa con un valor de 0 cuando la palabra A es mayor que la palabra B, y Z da un valor de 1 cuando A y B son iguales. Además, Cuando B es mayor que A, Z también presenta un valor de 1. Todo esto, se logró a tráves del diseño manual, aplicado con los métodos vistos en clase, como resolver Mapas de Karnaugh y aplicando los 8 pasos para resolver una red iterativa.

Se diseña la red de forma exitosa, en donde posteriormente se traduce al código solicitado mediante el lenguaje de Verilog obteniendo un sistema que simula el comparador de palabras demostrando que cumple con las características indicadas. Principalmente, se buscó implementar el código siguiendo la lógica obtenida después de los procedimientos a mano. Lo más importante de la parte de código, fué la implementación del bloque generate y la conexión de los módulos, donde cabe destacar la implementación conductual, y la estructural en el código y las diferencias que esas conllevas. Además, eso se hizo a través de un parámetro N, lo que permitió dinamizar la red a través de este parámetro, volviéndolo ajustable para distintos casos y situaciones.

Las demostraciones de las ondas se realizan en GTKWave,en donde se consigue la representación precisa del sistema de manera gráfica, observando el comportamiento de cada señal y cumpliendo con el objetivo principal que cuando la palabra A sea mayor a la palabra B se notifique con una salida Z que se activa en bajo. Cuándo A es igual a B y cuando B es mayor que A, la señal Z se activa con un valor de 1.

6. Referencias Bibliográficas

Chaves. J (1999). Manual de Verilog. Extraído de https://marceluda.github.io/rp_dummy/EEOF2018/verilog.pdf.

Brunete, A., San Segundo, P., Herrero, R.(2020). Introducción a la Automatización Industrial. Extraído de https://bookdown.org/alberto_brunete/intro_automatica/

Morris. M (2003). Diseño Digital, tercera edición. Extraído de https://anyflip.com/vede/hhzn/basic