Universidad de Costa Rica

Sede Central. Rodrigo Facio.

Escuela ingeniería Eléctrica.

IE0323 Circuitos Digitales I

Proyecto final de Verilog.

Estudiantes:

Jostin Cubero Torres B 92491

César Andrés Fonseca Villegas C12959

Marvin López Valverde C14324

Profesor:

Ing. Rafael Esteban Badilla Alvarado, Lic.

II ciclo 2023

Índice

1 - Introducción

En el ámbito de los circuitos digitales, las redes iterativas juegan un papel crítico. Su capacidad para realizar operaciones repetitivas y ajustar parámetros de forma iterativa es esencial en el diseño y funcionamiento eficiente de sistemas digitales complejos. En este proyecto, nos sumergimos en el diseño de un circuito digital implementado en Verilog, con un objetivo central: la comparación de dos palabras de longitud variable mediante el uso de estas redes iterativas.

El uso de Verilog como lenguaje de descripción de hardware nos proporciona las herramientas necesarias para la implementación de estas redes iterativas. Este proyecto se enfoca en no solo desarrollar un circuito funcional para la comparación de palabras, sino también en comprender y aplicar las redes iterativas en el diseño de sistemas digitales

2 - Marco teórico

En esta sección se presentan varios conceptos fundamentales para comprender el el proposito y el funcionamiento del proyecto que se explica en este reporte.

2.1 - Redes iterativas

Las redes iterativas son sistemas de interconexión donde la información o señales se propagan a través de iteraciones, y cada iteración puede influir en la siguiente. Al mismo tiempo la metodología de diseño mediante redes iterativas se emplea en la creación de circuitos digitales combinacionales destinados al procesamiento de un extenso número de bits o una cantidad no determinada de bits. En este enfoque, una red iterativa consiste en un conjunto de celdas de lógica combinacional idénticas, donde la información se transfiere de una celda a la siguiente de forma secuencial y lineal. En donde cada iteración puede actualizar el estado de los nodos en la red basándose en la información proveniente de las iteraciones anteriores. Su conectividad puede variar desde estructuras simples hasta redes más complejas, como redes neuronales o algoritmos de optimización distribuida.

Estas redes encuentran aplicación en diversos campos, como ciencia de la computación, teoría de la información, inteligencia artificial y procesamiento de señales, entre otros.

Las redes iterativas también se utilizan en problemas de optimización, donde se busca encontrar soluciones que minimicen o maximicen ciertas funciones objetivo. Algunos algoritmos como el método del gradiente descendente son ejemplos de enfoques iterativos utilizados en optimización.

En el procesamiento de señales, las redes iterativas pueden utilizarse para mejorar la calidad de la señal o realizar tareas específicas como la reducción de ruido. Otros algoritmos como el filtro adaptativo LMS (Least Mean Squares) son iterativos y se aplican en este tipo de procesamiento.

Por otro lado, las redes neuronales, son un subconjunto de redes iterativas, estas son ampliamente utilizadas en aprendizaje automático para tareas como clasificación, regresión y reconocimiento de patrones. Los algoritmos de

entrenamiento de redes neuronales a menudo implican iteraciones para ajustar los pesos y mejorar el rendimiento del modelo.

Algunos problemas y/o consideraciones es la convergencia, un aspecto crucial en las redes iterativas. Es importante garantizar que el proceso iterativo alcance un estado estable o una solución óptima. El diseño eficiente de algoritmos iterativos es esencial para garantizar un procesamiento rápido y eficiente de la información.

2.2 - Verilog

Verilog es un lenguaje de descripción de hardware (HDL) utilizado en el diseño y la verificación de circuitos digitales. Se utiliza para modelar, simular y sintetizar circuitos digitales, desde simples componentes hasta sistemas complejos en chips (SoCs).

Verilog también proporciona una abstracción de hardware que permite a los diseñadores describir la funcionalidad de los circuitos digitales a un nivel más alto que el código RTL (Register-Transfer Level). Permite el modelado de comportamiento y estructura. El nivel de abstracción puede variar desde la descripción del comportamiento del sistema hasta la especificación detallada de la estructura del hardware.

Los circuitos se describen en términos de módulos, lo que facilita la jerarquización del diseño y la reutilización de componentes. Verilog se utiliza ampliamente en simulación para verificar el comportamiento y la funcionalidad del diseño antes de la implementación física. Además, permite la creación de bancos de pruebas para verificar el correcto funcionamiento de los circuitos.

Verilog es compatible con herramientas de síntesis lógica que traducen la descripción de hardware en un diseño netlist que puede ser implementado en un FPGA o ASIC.

Verilog incluye una variedad de operadores para realizar operaciones lógicas y aritméticas. Entre ellas, las estructuras de control como if, else, for y case, las cuales facilitan la implementación de lógica más compleja, incluye tipos de datos como reg, wire, integer, real, entre otros, para representar diferentes tipos de

información en un diseño, y también se puede utilizar procesos como always e initial, los cuales son bloques que definen procesos en Verilog. Y por último, la sensibilidad a eventos como posedge y negedge se utiliza para controlar la ejecución de bloques de código en respuesta a cambios de señales.

2.3 - Módulos

Un módulo en Verilog es un bloque de código que implementa una cierta funcionalidad. Los módulos pueden ser utilizados para representar componentes individuales de un diseño de hardware digital, o para representar sistemas completos.

Los módulos se utilizan para organizar el código de Verilog en unidades más pequeñas y manejables. Esto hace que el código sea más fácil de leer, mantener y depurar.

Los módulos también se utilizan para facilitar la reutilización del código. Un módulo puede ser utilizado en varios diseños diferentes, lo que ahorra tiempo y esfuerzo. Para crear un módulo en Verilog, se utiliza la palabra clave module. La declaración de un módulo incluye el nombre del módulo, los puertos de entrada y salida, y el código que implementa la funcionalidad del módulo.

Los puertos de un módulo se utilizan para comunicar el módulo con el mundo exterior. Los puertos de entrada reciben datos del mundo exterior, y los puertos de salida envían datos al mundo exterior.

El código que implementa la funcionalidad del módulo se puede escribir utilizando cualquier construcción de lenguaje de Verilog. Los módulos pueden utilizar declaraciones de variables, expresiones, operadores, bucles y condicionales.

2.4 - Testbench

Un testbench en Verilog es un conjunto de módulos de Verilog que se utilizan para probar un módulo de diseño, también conocido como módulo de lógica. El testbench

proporciona entradas al módulo de diseño y verifica sus salidas para asegurarse de que cumple con sus especificaciones.
Los testbenches son una herramienta esencial para el diseño de hardware digital. Permiten probar el funcionamiento de un módulo de diseño sin tener que construirlo físicamente. Esto puede ahorrar tiempo y dinero, y también puede ayudar a identificar errores en el diseño antes de que se implementen en hardware real.
Cómo se hace un testbench en Verilog
Un testbench en Verilog se compone de dos partes principales:

- Un módulo de interfaz: Este módulo proporciona una interfaz entre el testbench y el módulo de diseño. El módulo de interfaz define las señales de entrada y salida del módulo de diseño.
- Un módulo de prueba: Este módulo genera las entradas al módulo de diseño y verifica sus salidas. El módulo de prueba puede ejecutar una serie de pruebas para verificar el funcionamiento del módulo de diseño.

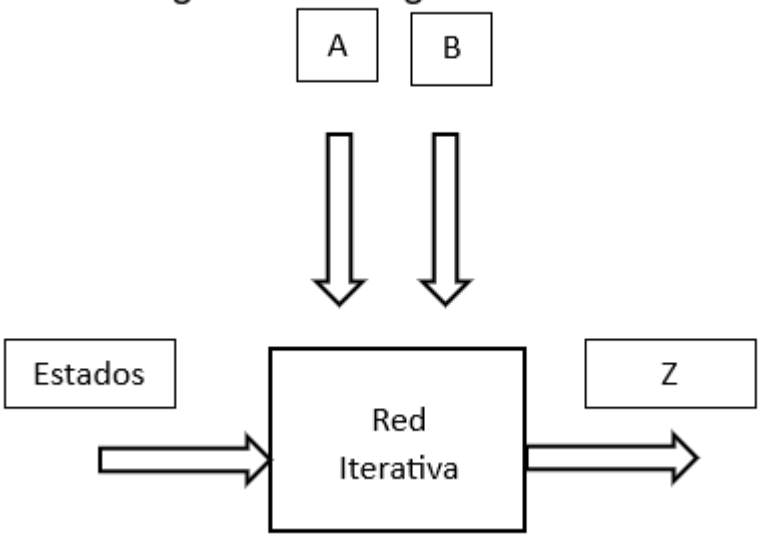
El módulo de interfaz define las señales de entrada y salida del módulo de diseño. Las señales de entrada se utilizan para proporcionar entradas al módulo de diseño, y las señales de salida se utilizan para verificar las salidas del módulo de diseño.
El módulo de interfaz puede generar las señales de entrada de forma aleatoria, o puede generarlas de acuerdo con un conjunto de casos de prueba.
El módulo de prueba genera las entradas al módulo de diseño y verifica sus salidas, puede ejecutar una serie de pruebas para verificar el funcionamiento del módulo de diseño y puede verificar las salidas del módulo de diseño de varias maneras. Puede comparar las salidas del módulo de diseño con los resultados esperados, o puede utilizar una herramienta de verificación formal para verificar las propiedades del módulo de diseño.

3 - Desarrollo

3.1 - Diseño de la red iterativa

En este apartado se describirá cada paso tomado, y la justificación de estos para realizar el diseño de las redes.

Figura 1. Vista general de la red iterativa.



1. **Primer caso: Dirección de izquierda a derecha.**

Realizamos la declaración de estados:

K) Actualmente la palabra A y la palabra B son iguales.

L) La palabra A es mayor que la palabra B.

M) La palabra A es menor que la palabra B.

En este caso la cantidad mínima de estados necesarios para realizar el trayecto de izquierda a derecha serían tres estados, debido a que se necesita, un estado capaz de detectar la igualdad de las palabras, y debido a que de esta manera se está analizando cada palabra del bit más significativo al menos significativo, por lo que apenas se encuentre que en una comparación de bit se detecte que un bit es mayor o menor que otro, automáticamente este será un estado absorbente, debido a que la comparación de los siguientes bits menos significativos no tendrán importancia.

Tabla 1. Tabla de transición de estados.

Estado Presente	**Estado Futuro**

	AB			
	00	**01**	**10**	**11**
K	K	M	L	K
L	L	L	L	L
M	M	M	M	M

Tabla 2. Asignación de estados.

	yp
K	01
L	10
m	11

Tabla 3. Tabla de transición de estados codificada.

Estado Presente	**Estado Futuro**			
yp	**AB**			
	00	**01**	**10**	**11**
01	01	11	10	01
10	10	10	10	10
11	11	11	11	11

	YP

Realizamos el diseño de la celda típica.

Figura 2. Vista general de la celda típica.

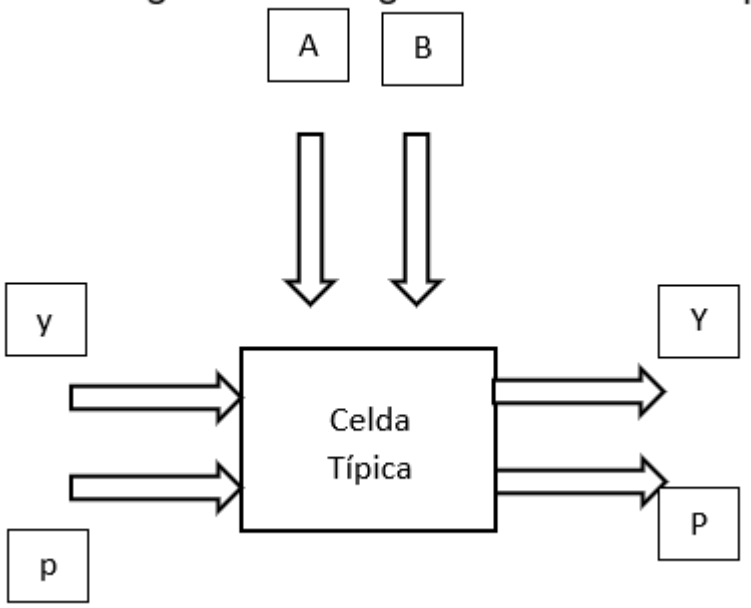
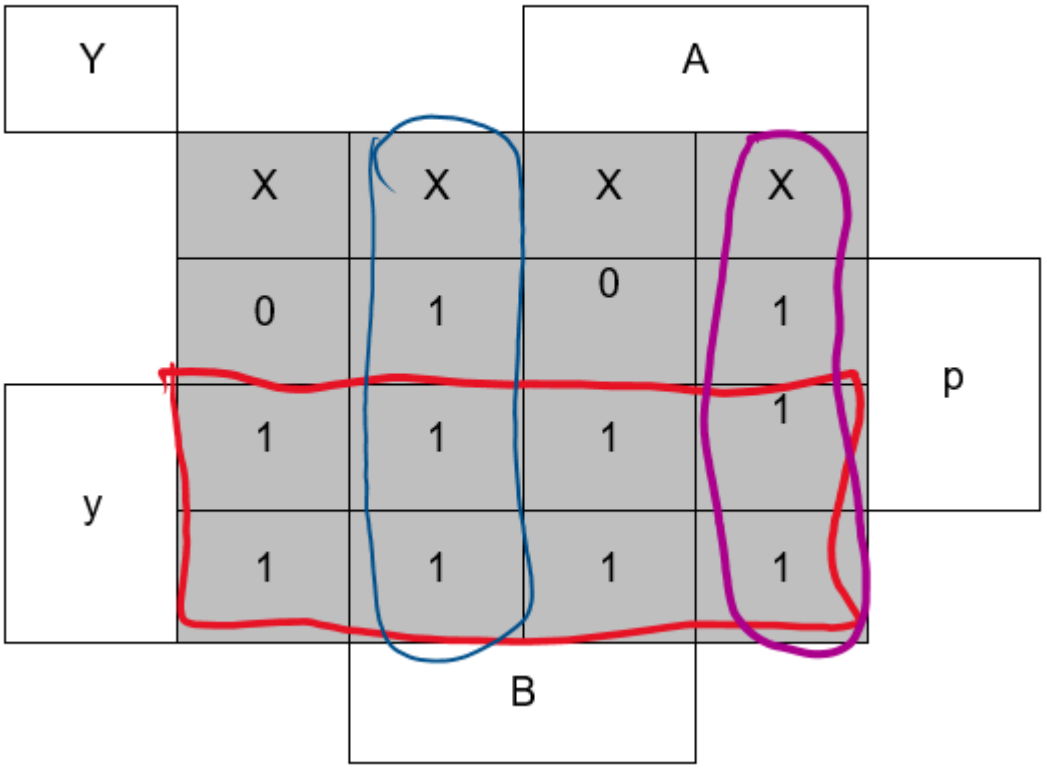


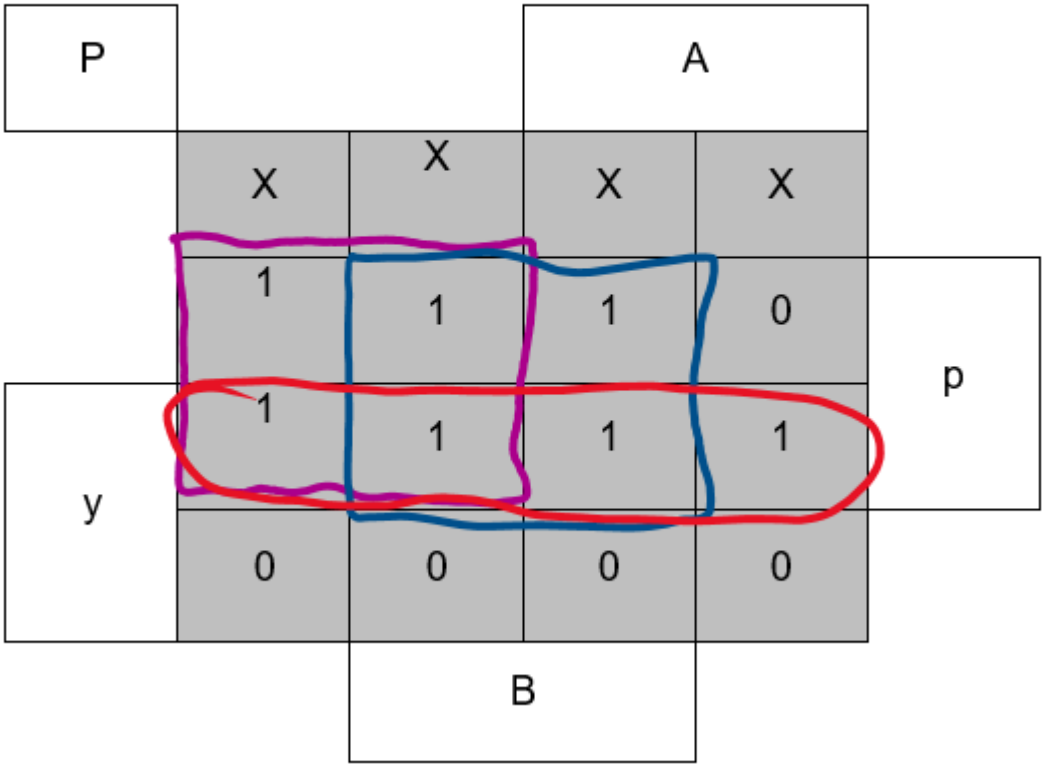
Tabla 4. MK del estado futuro Y.



Por lo tanto, la función de Y es:

Y(y,p,A,B)=y+A`B+AB`

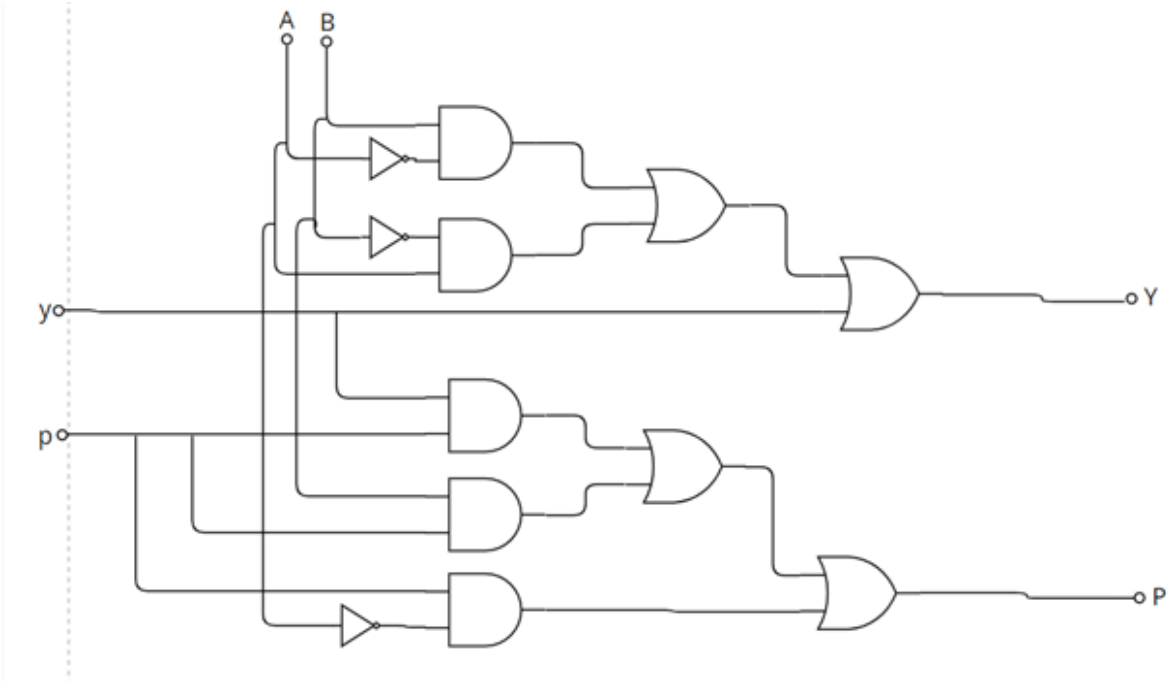
Tabla 5. MK del estado futuro P.



Por lo tanto, la función de P es:

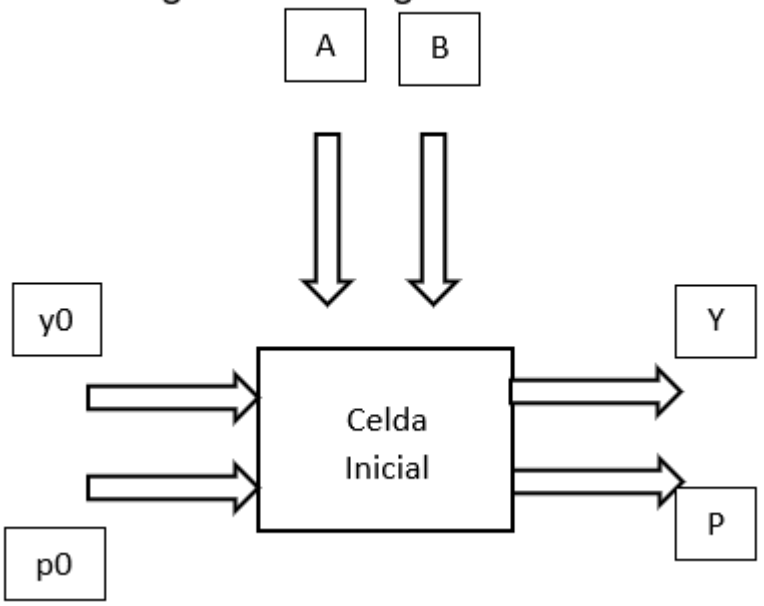
P(y,p,A,B)=yp+pA`+pB

Figura 3. Esquemático de celda típica.



Realizando el diseño de la celda inicial, denotamos el estado inicial como K (yp=01), debido a que decir que inicialmente las palabras son iguales no afectaría el siguiente estado, ya que este es un estado neutro.

Figura 4. Vista general de la celda inicial.



Por lo tanto, la función de Y es:

Y(y,p,A,B)=y+A`B+AB`

Y(0,1,A,B)=A`B+AB

Por lo tanto, la función de P es:

P(y,p,A,B)=yp+pA`+pB

P(0,1,A,B)=A`+B

Figura 5. Esquemático de celda inicial.

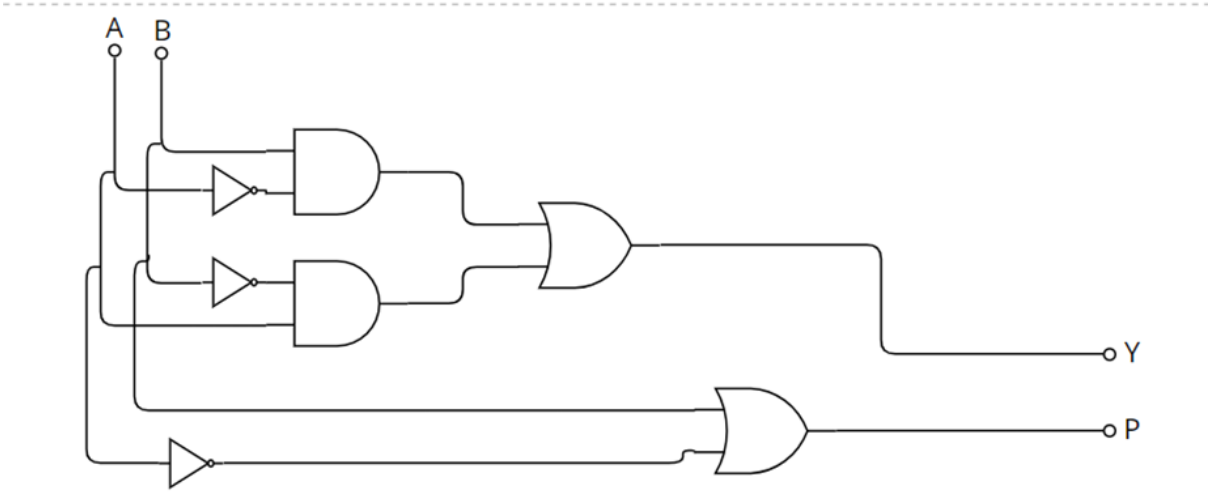


Figura 6. Vista general de la celda final.

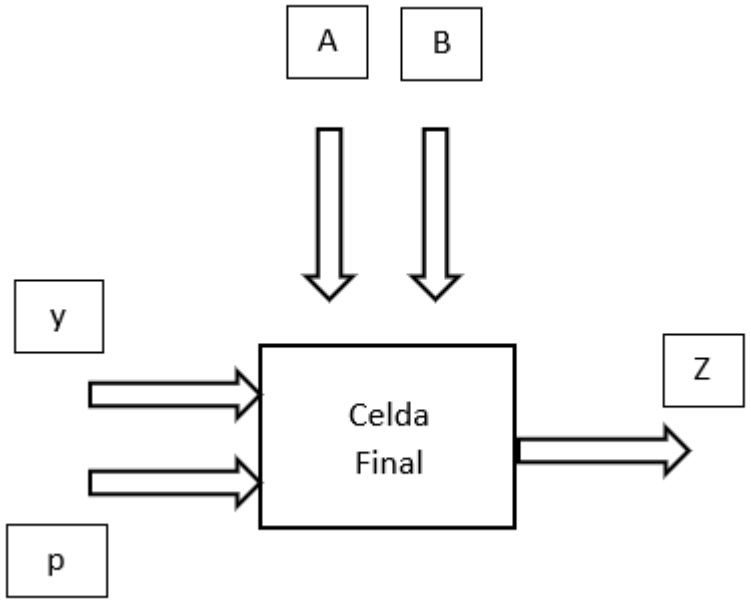
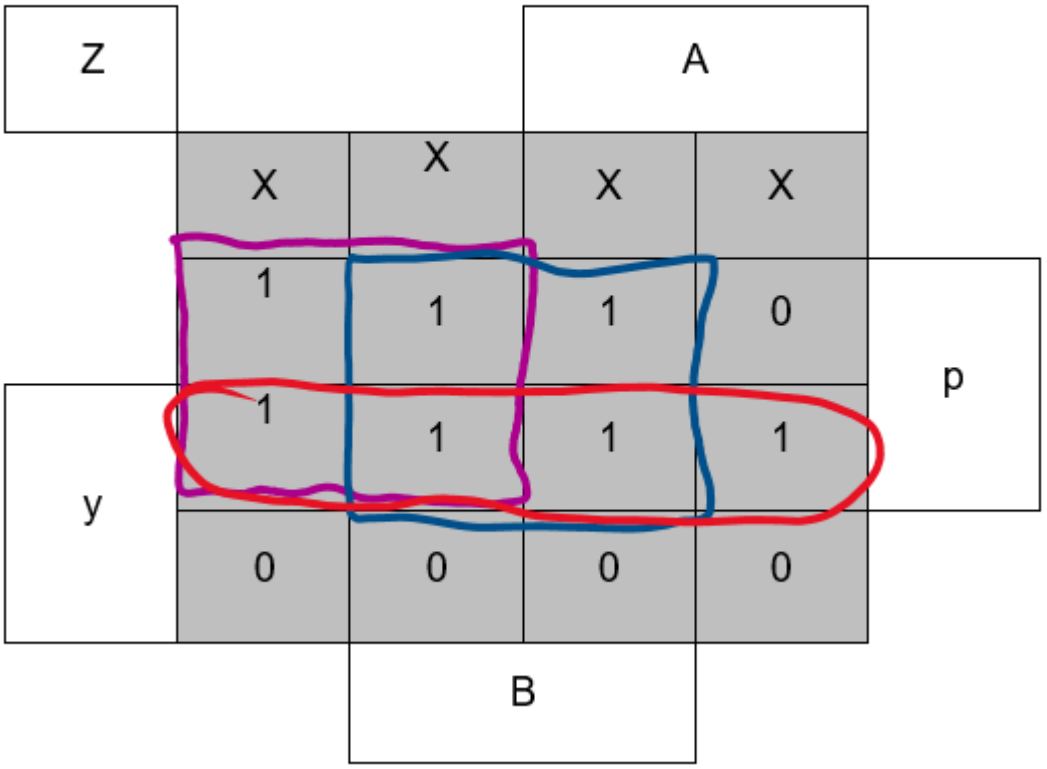


Tabla 6.

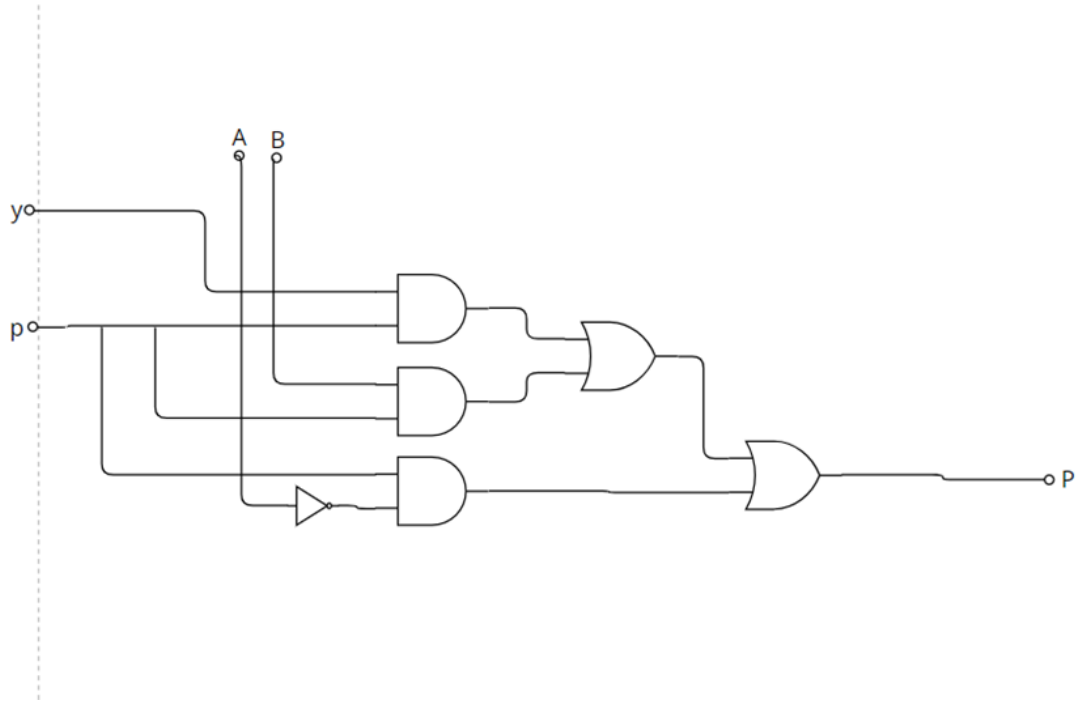
Estado Presente	**Salida Z (activa en bajo)**			
yp	**AB**			
	00	**01**	**10**	**11**
01	1	1	0	1
10	0	0	0	0
11	1	1	1	1
	Z			

Tabla 7. MK de salida Z.



Z(y,p,A,B)=yp+pA`+pB

Figura 7. Esquemático de celda final.



2. **Segundo caso: Dirección de derecha a izquierda.**

Realizamos la declaración de estados:

K) Actualmente la palabra A es menor o igual a la palabra B.

L) Actualmente la palabra A es mayor que la palabra B.

En este caso la cantidad mínima de estados necesarios para realizar el trayecto de derecha a izquierda serían dos estados, debido a que en este caso a diferencia de sentido anterior no se necesita un estado exclusivo para que sea capaz de detectar la igualdad de las palabras, y debido a que de esta manera se está analizando cada palabra del bit menos significativo al más significativo, en este caso no existirían estados absorbentes ya que reiteradamente estaría comparando un bit cada vez más significativo que el anterior.

Tabla 8. Tabla de transición de estados.

Estado Presente	**Estado Futuro**
	AB

	00	01	10	11
K	K	K	L	K
L	L	K	L	L

Tabla 9. Asignación de estados.

	p
K	0
L	1

Tabla 10. Tabla de transición de estados codificada.

Estado Presente	**Estado Futuro**			
p	**AB**			
	00	**01**	**10**	**11**
0	0	0	1	0
1	1	0	1	1
	P			

Realizamos el diseño de la celda típica.

Figura 8. Vista general de la celda típica.

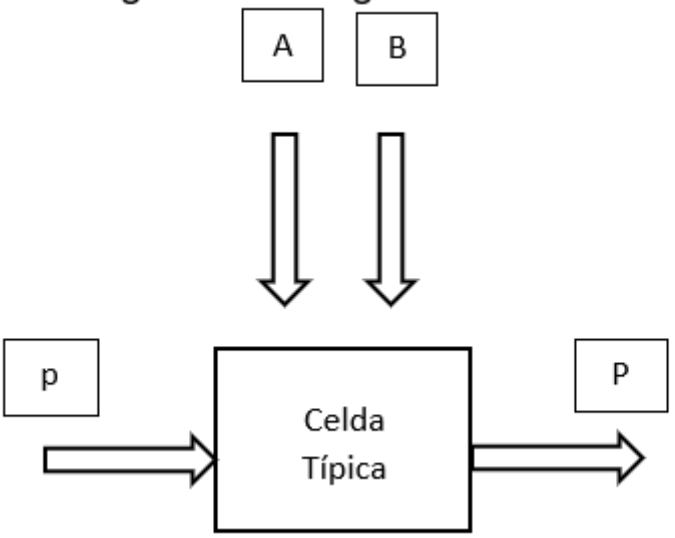


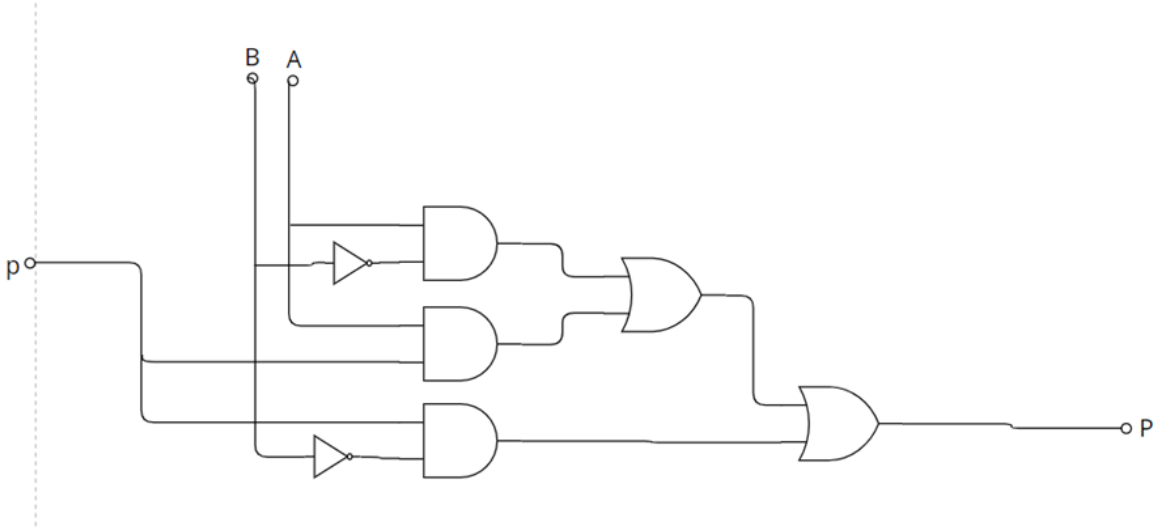
Tabla 11. MK del estado futuro P.

P	A			
	0	0	0	1
p	1	0	1	1
		B		

Por lo tanto, la función de P es:

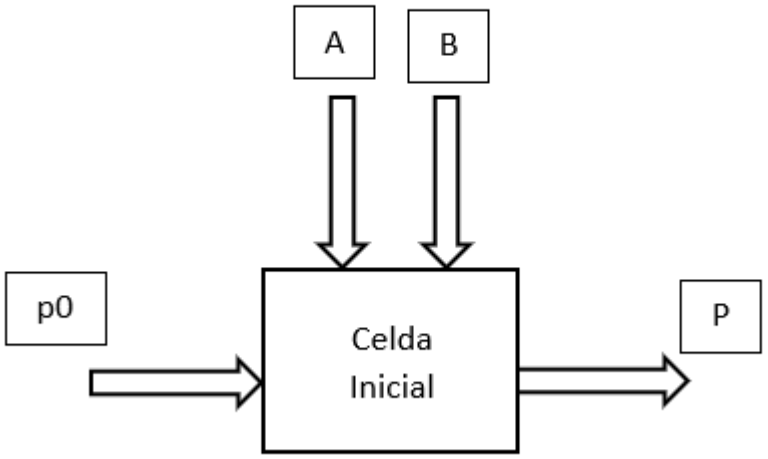
P(p,A,B)=AB`+pA+pB`

Figura 9. Esquemático de celda típica.



Realizando el diseño de la celda inicial, denotamos el estado inicial como K (p=0), debido a que decir que inicialmente la palabra A es menor o igual a la palabra B, no afectaría el siguiente estado, ya que este es un estado neutro.

Figura 10. Vista general de la celda inicial.



Por lo tanto, la función de P es:

P(p,A,B)=AB`+pA+pB`

P(0,A,B)=AB`

Figura 11. Esquemático de celda inicial.

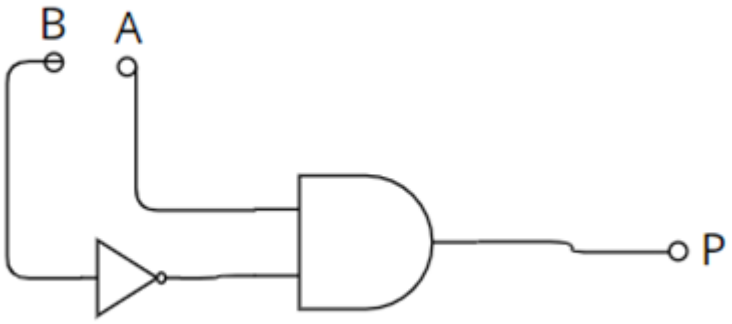


Figura 12. Vista general de la celda final.

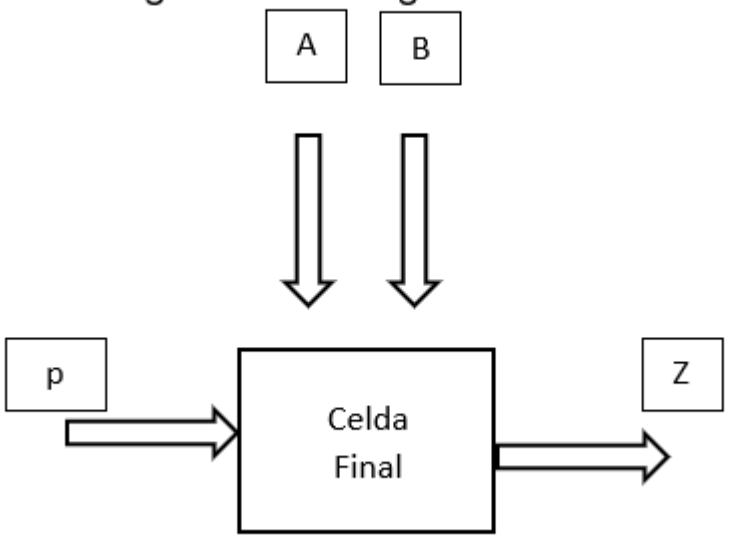


Tabla 11.

Estado Presente	**Salida Z (activa en bajo)**			
p	**AB**			
	00	**01**	**10**	**11**
0	1	1	0	1
1	0	1	0	0
	Z			

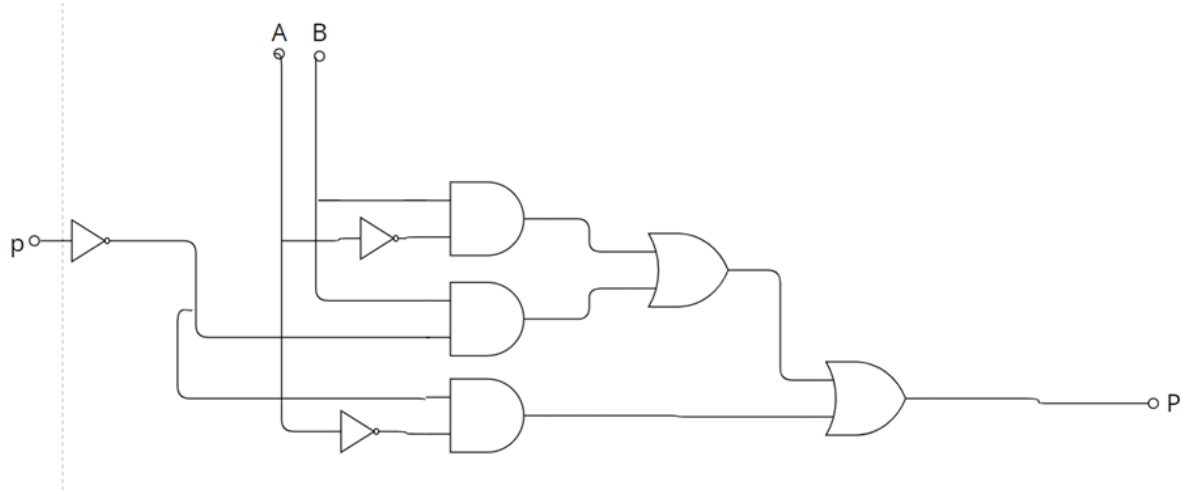
Tabla 12. MK de la salida Z.

Z	A			
	1	1	1	0
p	0	1	0	0
		B		

Por lo tanto, la función de Z es:

Z(p,A,B)=A`B+p`A`+p`B

Figura 13. Esquemático de celda final.



3.2 - Código

Modelo Conductual de izquierda a derecha:

En este modelo se tomó la red iterativa planteada en la sección 3.1 planteada para resolverse de izquierda a derecha, donde se presentaron tres posibles casos, y se buscó plasmarlo en codigo mediante el modelo conductual.

Primeramente se diseñaron las celdas, tanto inicial, como tipica, como final, así que primero se explicará el modelado y funcionamiento del codigo de la celda inicial:

Figura X:

```
1   module celda_inicial(
2       input [2:0] A, B,
3       output f_in, g_in
4   );
5
6   //todas las salidas de Z
7   assign f_in = (~A|B);
8
9   //todas las salidas de la funcion Y
10  assign g_in = ((~A|B) & (A|~B));
11
12  endmodule
```

Modelo conductual de celda inicial de izquierda a derecha

Como se muestra en el codigo de la figura, se escribió un modulo llamado “Celda inicial”, este modulo recibe dos vectores A y B, cada uno de 3 bits (esto debido a que las pruebas principales se hicieron con esta cantidad de bits), y tiene dos salidas llamadas f_in y g_in respectivamente, estas salidas se plantearon de manera conductual siguiendo las siguientes ecuaciones:

$$Y(0,1,A,B) = A'B + AB'$$

$$P(0,1,A,B) = A' + B$$

Las cuales fueron transcritas de manera logica en el codigo, f_in siendo exactamente la ecuación P mostrada, y siendo g_in exactamente la ecuación Y mostrada, finalizando de esta manera este modulo y creando la celda inicial.

Seguidamente de la celda inicial se planteó la celda tipica, la cual se encuentra conectada a la celda inicial de forma directa:

Figura X:

```
module celda_tipica(
    input z, y, g_in,
    input [2:0] A, B,
    output f_mid, g_mid
);

//todas las salidas de Z
assign f_mid = (((z | ~A) & (z | B)) & (y | z));

//todas las salidas de la funcion Y
assign g_mid = (y & g_in);

endmodule
```

Modelo conductual de celda típica de izquierda a derecha

En esta celda de plantea un modulo llamado “celda_tipica”, donde recibe 5 entradas, las primeras tres son entradas de un bit, los cuales son Z, Y y g_in, las variables Z y Y, son variables definidas al momento de definir los estados en la sección 3.1, por lo que dependiendo el estado en el que se encuentre, estas variables están constantemente cambiando, y g_in es la conexión directa que recibe la celda tipica por parte de la celda inicial, y esta entrada es el cable que une estas celdas, luego también se vuelven a recibir los vectores A y B, los cuales funcionan de la misma manera que lo vectores recibidos en la primera celda, sin embargo estos no necesariamente pasan por la celda inicial, y finalmente tenemos dos salidas principales, las cuales son f_mid y g_mid, las cuales son las que se definen en la logica de la celda en base a las siguientes ecuaciones:

$$Y(y, \rho, A, B) = y + \bar{A}B + A\bar{B}$$

$$Z(y, \rho, A, B) = \rho\bar{A} + \rho B + y\rho$$

Donde la ecuación Y corresponde a g_mid, donde se encuentra la casualidad que uno de sus terminos es exactamente igual a g_in, por lo que aquí es donde se

refleja la relación de estas celdas, donde simplemente se agrega una and y, luego por parte de la ecuación Z se relaciona con f_mid, donde se define la ecuación exactamente de la misma forma de manera conductual.
Finalmente, la ultima celda, tiene una peculiaridad, que se muestra de la siguiente manera:

Figura X:

```
3  module celda_final(
4      input f_mid,
5      output f
6  );
7
8  assign f = f_mid;
9
10 endmodule
```

Modelo conductual de celda final de izquierda a derecha

Esta celda tiene la peculiaridad de que a la hora de plantear las ecuaciones, resulta que todas sus compuestas son exactamente iguales a f_mid planteado en la celda tipica como se muestra en la siguiente ecuación:

$$f(y, p, A, B) = pA' + pB + yp = P(yp, A, B)$$

Lo cual provoca que solamente sea necesario pasar un cable directo a través de este modulo para que funcione correctamente la celda final, conectandolo con la salida f, la cual es la salida final, la cual varia entre 0 y 1, siendo 1 cuando A y B son iguales o B es mayor que A, y f = 0 cuando A es mayor que B

Uniendo todas las celdas mediante conexiones llegamos a todo el sistema, lo cual es un archivo de codigo en el que solo se dedica a hacer todas las conexiones y hacer que todo funcione en conjunto, conectando todo, tanto salidas como entradas:

Figura X:

```v
//MainModule
`include "celda_inicial.v"
`include "celda_tipica.v"
`include "celda_final.v"

module system(
    input [2:0] A, B,
    input z, y,
    output f
);

//conexiones generadas para el cableado de cada celda
wire f_in;
wire g_in;
wire f_mid;
wire g_mid;

//instanciacion de la celda inicial
celda_inicial C1 (
    .A(A),
    .B(B),
    .f_in(f_in),
    .g_in(g_in)
);

//instanciacion de la celda tipica
celda_tipica C2(
    .z(z),
    .y(y),
    .A(A),
    .B(B),
    .g_in(g_in),
    .f_mid(f_mid),
    .g_mid(g_mid)
);

//instanciacion de la celda final
celda_final C3(
    .f_mid(f_mid),
    .f(f)
);

endmodule
```

Modelo conductual de todo el sistema de izquierda a derecha

Modelo Estructural de derecha a izquierda:

Para la red iterativa de derecha a izquierda utilizamos la implementación estructural, esto se refiera a la manera en el diseño y como se organiza el sistema que queremos armar, esta implementación se centra en los detalles técnicos, la arquitectura de software, la codificación y la disposición física de los componentes. Conociendo acerca de implementación estructural pasamos al código. Partiendo de los esquemáticos de las celdas en sección 3.1 nos enfocaremos en sus componentes internos es decir en las compuertas lógicas, comenzaremos con la Celda Típica:

Figura X

Verilog > Verilogproyecto > CeldaTipicaBit.v

```
1   module CeldaTipicaBit(output P_mid,
2   input[2:0] A, B,
3   input P_in, P
4   );
5
6       wire not_B, and_or1, and_or2, and_or3 ;          //Cables intermedios
7
8       and gate_and1(and_or1, P_in);
9       and gate_and2(and_or2, not_B, P);
10      and gate_and3(and_or3, A, P);
11      not gate_not1(not_B, B);
12      or gate_or(P_mid, and_or1, and_or2, and_or3);
13
14  endmodule
15
```

Modelo estructural de celda típica de derecha a izquierda

En esta celda es la primera que se genera y de ella dependen las otras celdas, si observamos el esquemático (1) podemos observar que el código incluye las tres compuertas AND el OR que une todas las salidas de las AND además del NOT que niega la salida del B, al realizar esta celda se deben de tener ciertas consideraciones, es cierto que el dibujo solo presenta 3 entradas A y B palabras que se están comparando y P que es la asignación de estados, pero debemos de considerar la entrada de la celda Inicial que se ha definido como P_in, esta entrada es el cable que nos une la celda típica e inicial, pro condición de estado definimos que cuando tenemos el valor de P = 0 este nos define la celda inicial y si

observamos su esquemático esta es igual a la compuerta AND nombrada como gate_and1 es decir que ahí es donde se conecta.

Ahora hablaremos de la celda inicial como se menciono con anterioridad su salida lógica va conectada a la celda típica pero ella tiene su propio esquemático que debemos representar

Figura Y

Verilog > Verilogproyecto > CeldaInicialBit.v

```
1   module CeldaInicialBit(output P_in,
2   input[2:0] A, B
3
4   );
5
6       wire not_B;          //Cables intermedios
7
8       and gate_and(P_in, not_B, A);
9       not gate_not(not_B, B);
10
11  endmodule
12
```

Modelo estructural de celda inicial de derecha a izquierda

El observar las compuestas lógicas que posee el esquemático (1), corresponden a una AND y un NOT que niega la entrada B y su salida esta conectada a la celda típica

La ultima celda, tiene una peculiaridad que se puede ver en su respectivo modulo:

Figura Z

Verilog > Verilogproyecto > CeldaFinalBit.v

```
1   module CeldaFinalBit(output Z_out,
2   input P_mid
3   );
4
5       not not_gate(Z_out, P_mid);
6
7   endmodule
8
```

Modelo estructural de celda final de derecha a izquierda

Si observamos su correspondiente esquemático (3) podemos observar que la cantidad de compuertas lógicas y la distribución es la misma que el de la celda típica con la diferencia de que esta negando todas las entradas es decir es igual a la celda típica solo que negada, por ello nos tomamos la libertad de negar la salida de la celda típica en el modulo por ello es que la única compuerta lógica presente es la del NOT ya que agarra la salida P_mid y al invierte dando como resultado nuestro Z_out.

Para finalizar con los códigos tenemos uno el cual su unica funcion es la de unir las tres celdas en una, este se utilizara para realizar los TestBench de las tres celdas ya que nos permite hacer todas las conexiones y hacer que todo funcione en conjunto, conectando todo, tanto salidas como entradas en un solo código:

```
module CableadoBit( output Z_out,
    input [2:0] A,
    input [2:0] B,
    input P
    );

    wire P_in; // Cable intermedio entre CeldaInicial y CeldaTipica
    wire P_mid; // Cable intermedio entre CeldaTipica y CeldaFinal

    // Instancias de las celdas
    CeldaInicialBit inicial(
        .P_in(P_in),
        .A(A),
        .B(B)
    );

    CeldaTipicaBit tipica(
        .P_mid(P_mid),
        .P(P),
        .A(A),
        .B(B),
        .P_in(P_in)
    );

    CeldaFinalBit final(
        .Z_out(Z_out),
        .P_mid(P_mid)
    );

endmodule
```

3.3 - Pruebas

Pruebas de Modelo Conductual de izquierda a derecha:

Antes de pasar a la prueba final del sistema completo, se debe hacer pruebas basicas en las celdas unicamente para verificar su funcionamiento correcto, por lo que estas pruebas se realizarán unicamente mediante el uso de un unico bit, ya que con un solo bit bastará para el estudio de cada caso, y como se mostrará mas adelante, el algoritmo utilizado para el analisis de las palabras se hace mediante vectores, por lo que es posible analizar bit por bit y esto nos permite hacer estas pruebas basicas iniciales, y son más que suficiente para los propositos del sistema.

Prueba en celda inicial:

Figura X:

```
1  // module_1_tb.v
2  `timescale 1 ns/10 ps // time-unit = 1 ns, precision = 10 ps
3  module celda_inicial_tb();
4    reg A, B; //[2:0]
5    wire f_in, g_in;
6   // duration for each bit = 20 * timescale = 20 * 1 ns = 20ns
7   localparam period = 20;
8   celda_inicial UUT (.A(A), .B(B), .f_in(f_in), .g_in(g_in));
9
10  initial // initial block executes only once
11  begin
12  $dumpfile ("module_1_tb.vcd");
13  //$dumpvars(1, module_1_tb);
14  $dumpvars(1, celda_inicial_tb);
15
16  // values for A and B
17   A = 0;
18   B = 0;
19   #period; // wait for period
20   A = 1;
21   B = 0;
22   #period;
23   A = 0;
24   B = 1;
25   #period;
26   A = 1;
27   B = 1;
28   #period;
29
30   $finish; // Termina la simulación después de completar las pruebas
31  end
32 endmodule
```

Codigo utilizado en la prueba (testbench) de la celda inicial

Este codigo mostrado se ejecuta utilizando los siguientes comandos de manera consecutiva:

iverilog -o out.vvp .\celda_inicial.v .\module_1_tb.v

vvp .\out.vvp

gtkwave .\module_1_tb.vcd

Una vez ejecutados estos comandos se abrirá una ventana de gtkwave el cual mostrará el comportamiento de la celda, *comportamiento el cual será extendido en la sección 3.4.*

Ademas, como se muestra en el codigo, lo primero que se hace es definir todos los inputs de la celda como tipo reg, y todos los outputs como tipo wire, esto con el fin de conectar el testbench con la celda y generar esas conexiones en la linea 8 del codigo, una vez realizado todo este procedimiento se introducen todas las posibles combinaciones de A y B con un solo bit, y así mostrando todos los caso posibles de un bit.

Prueba en celda típica:

Figura X:

```
// module_2_tb.v
`timescale 1 ns/10 ps // time-unit = 1 ns, precision = 10 ps
module celda_tipica_tb();
  reg z, y, g_in;
  reg A, B; //[2:0]
  wire f_mid, g_mid;

  // duration for each bit = 20 * timescale = 20 * 1 ns = 20ns
  localparam period = 20;
  celda_tipica UUT (
    .z(z), .y(y), .g_in(g_in),
    .A(A), .B(B),
    .f_mid(f_mid), .g_mid(g_mid)
  );

  initial begin
    $dumpfile ("module_2_tb.vcd");
    $dumpvars(1, celda_tipica_tb);

    // caso de y = 0 y z = 1
    A = 0; B = 0; g_in = 0;
    y = 0; z = 1;
    #period;

    A = 0; B = 0; g_in = 1;
    y = 0; z = 1;
    #period;

    A = 0; B = 1; g_in = 0;
    y = 0; z = 1;
    #period;

    A = 0; B = 1; g_in = 1;
    y = 0; z = 1;
    #period;

    A = 1; B = 0; g_in = 0;
    y = 0; z = 1;
    #period;

    A = 1; B = 0; g_in = 1;
    y = 0; z = 1;
    #period;
```

Codigo utilizado en la prueba (testbench) de la celda típica

Este codigo mostrado se ejecuta utilizando los siguientes comandos de manera consecutiva:

iverilog -o out.vvp .\celda_tipica.v .\module_2_tb.v

vvp .\out.vvp

gtkwave .\module_2_tb.vcd

Una vez ejecutados estos comandos se abrirá una ventana de gtkwave el cual mostrará el comportamiento de la celda, *comportamiento el cual será extendido en la sección 3.4.*

Cabe recargar que el codigo de la imagen no esta completo ya que este llega casi hasta las 120 lineas de codigo, y se hizo con el fin de probar todos los casos posibles por los que podría pasar la celda, por lo que se debe su extension peculiar.

Ademas, como se muestra en el codigo, lo primero que se hace es definir todos los inputs de la celda como tipo reg, y todos los outputs como tipo wire, esto con el fin de conectar el testbench con la celda y generar esas conexiones en la linea 10 a la 14 del codigo, una vez realizado todo este procedimiento se introducen todas las posibles combinaciones de A,B, y, z con un solo bit, y así mostrando todos los caso posibles de un bit.

Prueba en celda final:

Figura X:

```
1  // module_3_tb.v
2  `timescale 1 ns/10 ps // time-unit = 1 ns, precision = 10 ps
3  module celda_final_tb();
4    reg f_mid;
5    wire f;
6
7    // duration for each bit = 20 * timescale = 20 * 1 ns = 20ns
8    localparam period = 20;
9    celda_final UUT (
10     .f_mid(f_mid),
11     .f(f)
12   );
13
14   initial begin
15     $dumpfile ("module_3_tb.vcd");
16     $dumpvars(1, celda_final_tb);
17
18     // values for f_mid
19     f_mid = 0;
20     #period;
21
22     f_mid = 1;
23     #period;
24
25     f_mid = 0;
26     #period;
27
28     f_mid = 1;
29     #period;
30
31     $finish; // Termina la simulación después de completar las pruebas
32   end
33 endmodule
34
```

Codigo utilizado en la prueba (testbench) de la celda final

Este codigo mostrado se ejecuta utilizando los siguientes comandos de manera consecutiva:

iverilog -o out.vvp .\celda_final.v .\module_3_tb.v

vvp .\out.vvp

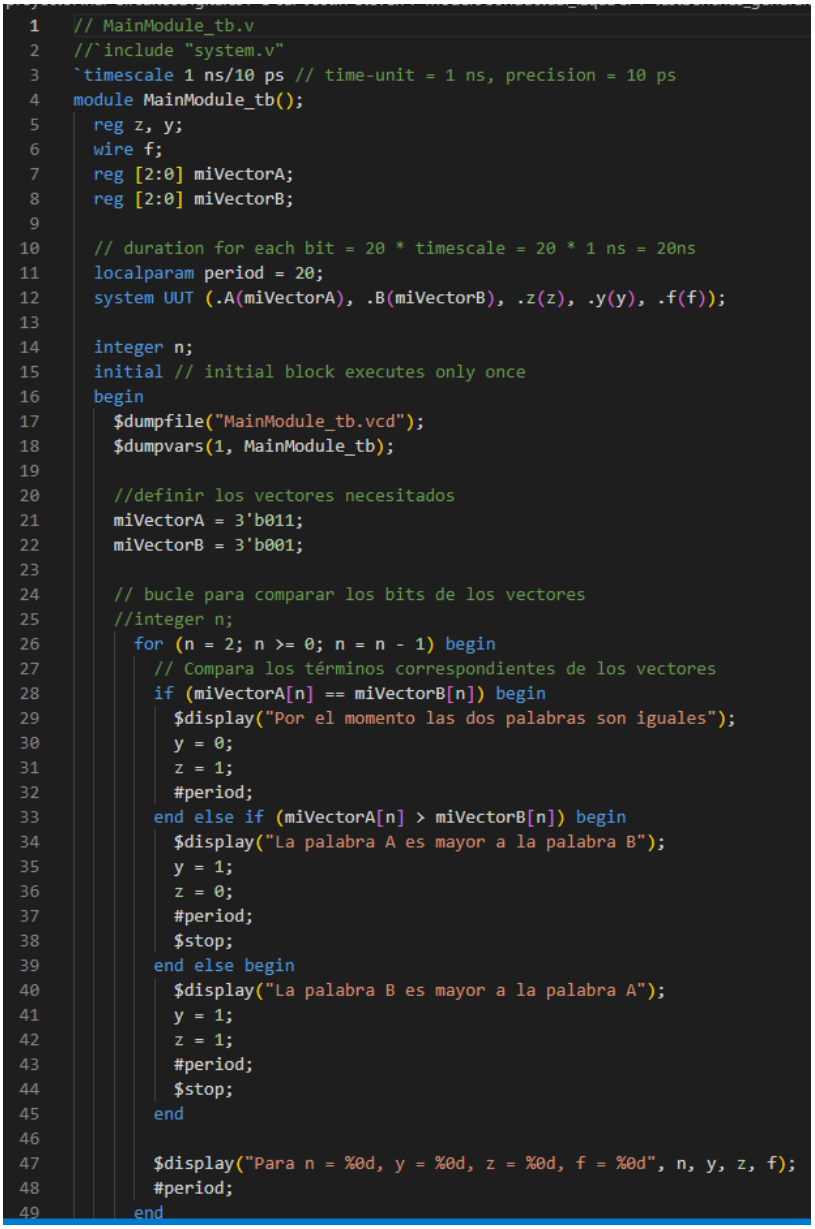
gtkwave .\module_3_tb.vcd

Una vez ejecutados estos comandos se abrirá una ventana de gtkwave el cual mostrará el comportamiento de la celda, *comportamiento el cual será extendido en la sección 3.4.*

Esta celda notoriamente es la que tiene el test más simple, ya que esta solo tiene un cable que pasa directo hacia la salida, así que el valor de entrada va a ser igual al de la salida, y el test está estructurado de forma similar a los anteriormente expuestos.

PRUEBA DE TODO EL SISTEMA:

Figura X:



```v
// MainModule_tb.v
//`include "system.v"
`timescale 1 ns/10 ps // time-unit = 1 ns, precision = 10 ps
module MainModule_tb();
  reg z, y;
  wire f;
  reg [2:0] miVectorA;
  reg [2:0] miVectorB;

  // duration for each bit = 20 * timescale = 20 * 1 ns = 20ns
  localparam period = 20;
  system UUT (.A(miVectorA), .B(miVectorB), .z(z), .y(y), .f(f));

  integer n;
  initial // initial block executes only once
  begin
    $dumpfile("MainModule_tb.vcd");
    $dumpvars(1, MainModule_tb);

    //definir los vectores necesitados
    miVectorA = 3'b011;
    miVectorB = 3'b001;

    // bucle para comparar los bits de los vectores
    //integer n;
      for (n = 2; n >= 0; n = n - 1) begin
        // Compara los términos correspondientes de los vectores
        if (miVectorA[n] == miVectorB[n]) begin
          $display("Por el momento las dos palabras son iguales");
          y = 0;
          z = 1;
          #period;
        end else if (miVectorA[n] > miVectorB[n]) begin
          $display("La palabra A es mayor a la palabra B");
          y = 1;
          z = 0;
          #period;
          $stop;
        end else begin
          $display("La palabra B es mayor a la palabra A");
          y = 1;
          z = 1;
          #period;
          $stop;
        end

        $display("Para n = %0d, y = %0d, z = %0d, f = %0d", n, y, z, f);
        #period;
      end
```

Codigo utilizado en la prueba (testbench) de la celda final

Este testbench cuando se planteó, se hizo para palabras de 3 bits, sin embargo, haciendo ligeras modificaciones es sencillo extenderlo o disminuirlo a la cantidad de bit que se desee, ya que su funcionamiento es simple, ya que toma un vector de X cantidad de bit y los pasa por un ciclo for, este for tiene 3 condicionales,donde cada condicional representa un estado del sistema, y esto hace que cambien los valores de y y Z de forma iterativa, lo que permite un rapido diagnostico del sistema, mostrando cual palabra es mayor, y haciendo la ejecución del grafico de gtkwave se puede confirmar el valor de la salida final f de forma sencilla.

Este codigo mostrado se ejecuta utilizando los siguientes comandos de manera consecutiva:

iverilog -o out.vvp .\system.v .\MainModule_tb.v

vvp .\out.vvp

gtkwave .\MainModule_tb.vcd

Pruebas de Modelo Estructural de derecha a izquierda :

Se debe de comenzar con una prueba trivial en cada uno de los 3 módulos para probar si el comportamiento esperado se refleja en las pruebas, estas pruebas al ser independientes no tienen un orden en específico, pero comenzaremos con la celda inicial. Debemos de crear un nuevo código para realizar la prueba que se ve de la siguiente forma:

Figura X

```
`timescale 1 ns/100 ps

module CeldaInicial_tb;
    reg A, B;
    wire P_in;
    localparam period = 10;

    // Instancia del módulo CeldaInicial
    CeldaInicial uut (
        .A(A),
        .B(B),
        .P_in(P_in)
    );

    // Inicialización de las entradas A y B
    initial begin
        $dumpfile ("modulo_1_tb_.vcd");
        $dumpvars(1, CeldaInicial_tb);

        // Valores para A y B
        A = 0;
        B = 0;
        #period; // Periodod de espera
        A = 0;
        B = 1;
        #period;
        A = 1;
        B = 0;
        #period;
        A = 1;
        B = 1;
        #period;

    end

endmodule
```

Modulo de TestBench para celda Inicial de derecha a izquierda

La celda inicial solo tiene dos entradas A y B, que al ser de un bit posee 4 combinaciones posibles esta prueba se encarga de entregarnos la salida para cada una de las combinaciones, el resultado de la salida nos ayudará para darnos cuenta si la lógica interna de la celda inicial plasmada en el módulo CeldaInicial es correcto

Para la celda tipica dada su cantidad de entradas su codigo queda algo grande por ello se obto por mostrar las partes en las que se diferencia con respecto a la celda inicial

Verilog > verilogbueno > CeldaTipica_tb.v

```
1   `timescale 1ns/100ps
2
3   module CeldaTipica_tb;
4
5       reg A, B, P_in, P;
6       wire P_mid;
7       localparam period = 10;
8
9       // Instancia del módulo my_or
10      CeldaTipica uut (
11          .P_mid(P_mid),
12          .A(A),
13          .B(B),
14          .P_in(P_in),
15          .P(P)
16      );
17
```

Un cambio lógico con respecto a la inicial es los registros, valores que varían y pertenecen a las entradas en la celda definidos en el modulo de CeldaTipica de igual forma se tiene instanciar estas entradas en el TestBech

```
18    // Inicialización de las entradas A, B, y P_in
19      initial begin
20         $dumpfile("modulo_2_tb_.vcd");
21         $dumpvars(1, CeldaTipica_tb);
22
23         // Valores para A, B Y P
24         A = 0;
25         B = 0;
26         P = 0;
27         P_in = 0;
28
29         #period; // Periodod de espera
30         A = 0;
31         B = 1;
32         P = 0;
33         P_in = 0;
34         #period;
35         A = 1;
36         B = 0;
37         P = 1;
38         P_in = 1;
39         #period;
40         A = 1;
41         B = 1;
42         P = 0;
43         P_in = 0;
44         #period;
45
46      end
47  endmodule
```

Para este cambio se deben de considerar varias cosas, lo primero es que de igual forma que en el anterior se siguen revisando las 4 combinaciones posibles generadas por A y B, el valor de P está definido por la asignación de estados en donde tenemos P = 1 solo cuando A es mayor a B en todos los demas casos P = 0 y por último el valor de P_in esta definido por la logica de el la celda de entrada en donde el unico caso donde este vale 1 es cuando A = 1 y B = 0.

Para finalizar tenemos el TestBech de la celda final, que se muestra a continuación

```verilog
`timescale 1ns/100ps

module CeldaFinal_tb;

    reg P_mid;
    wire Z_out;
    localparam period = 10;

    // Instancia del módulo my_or
    CeldaFinal uut (
        .Z_out(Z_out),
        .P_mid(P_mid)
    );

   // Inicialización de las entradas A, B, y P_in
    initial begin
        $dumpfile("modulo_3_tb_.vcd");
        $dumpvars(1, CeldaFinal_tb);

        // Valores para A y B
        P_mid = 1;

        #period; // Periodod de espera
        P_mid = 0;
        #period;

    end
endmodule
```

Dado que P_mid es un único registro y este posee un bit, solo tiene dos posibles casos cuanto este vale 1 y cuando vale 0

Para visualizar el comportamiento del código en GTKwave es necesario un proceso de pasos que vamos a denominar “El proceso de compilación” vamos a tomar como ejemplo la CeldaInicial este proceso comienza de la siguiente forma:

1. Entramos al Símbolo del sistema (CMD) en la computadora y nos vamos a posicionar en la carpeta que contiene nuestros módulos, para ello utilizamos el comando **cd** seguido de la ubicación del archivo, para este caso **cd desktop\verilog\verilogproyecto**

2. Compilar los archivos verilog, el siguiente código compilara los archivos verilog y generara un archivo ejecutable llamado salidaInicial

```
iverilog -o salidaInicial CeldaInicial.v CeldaInicial_tb.v
```

3. Ejecutar simulación el código, este codigo esto ejecutara la simulación utilizando el ejecutable generado

```
vvp salidaInicial
```

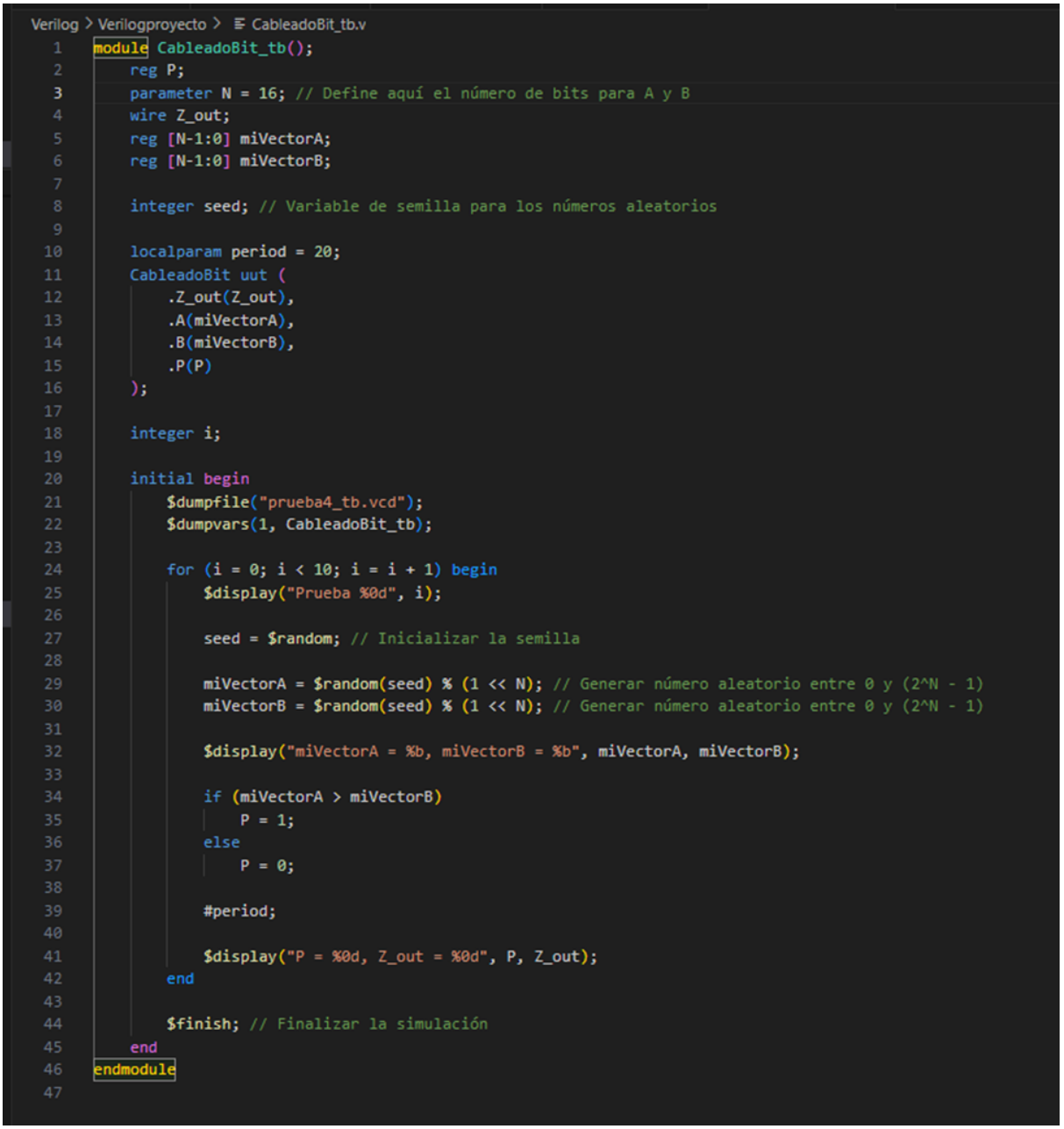
4. Por último el generador de VCD, ejecuta la simulación y genera un archivo con el nombre que se le escriba en el $dumpfile

```
vvp -vcd salidaInicial
```

Siguiendo estos pasos podemos generamos un archivo el cual es capaz de leer el GTKwave, se debe de realizar el mismo proceso paso a paso para cada una de las celdas a la que se le realice un TestBench, debemos tener cuidado de escribir de forma correcta los nombres de lo módulos ademas de que estos tiene que estar guardados en el miedo lugar

PRUEBA DE TODO EL SISTEMA DE DERECHA A IZQUIERDA

El código utilizado para realizar la prueba de las celdas juntas es el siguiente:



```
Verilog > Verilogproyecto > CableadoBit_tb.v
module CableadoBit_tb();
    reg P;
    parameter N = 16; // Define aquí el número de bits para A y B
    wire Z_out;
    reg [N-1:0] miVectorA;
    reg [N-1:0] miVectorB;

    integer seed; // Variable de semilla para los números aleatorios

    localparam period = 20;
    CableadoBit uut (
        .Z_out(Z_out),
        .A(miVectorA),
        .B(miVectorB),
        .P(P)
    );

    integer i;

    initial begin
        $dumpfile("prueba4_tb.vcd");
        $dumpvars(1, CableadoBit_tb);

        for (i = 0; i < 10; i = i + 1) begin
            $display("Prueba %0d", i);

            seed = $random; // Inicializar la semilla

            miVectorA = $random(seed) % (1 << N); // Generar número aleatorio entre 0 y (2^N - 1)
            miVectorB = $random(seed) % (1 << N); // Generar número aleatorio entre 0 y (2^N - 1)

            $display("miVectorA = %b, miVectorB = %b", miVectorA, miVectorB);

            if (miVectorA > miVectorB)
                P = 1;
            else
                P = 0;

            #period;

            $display("P = %0d, Z_out = %0d", P, Z_out);
        end

        $finish; // Finalizar la simulación
    end
endmodule

```

Este código está creado para trabajar con cualquier cantidad de bit ya que todo lo que depende de los bits está en terminos de un N-1 para poder cambiar ese valor en un único parámetro, podemos decir que la ciencia del este radica en, los valores de A y B que se definen como vectores para poder almacenar palabras aleatorias de N bits y el for el cual permite generar 10 combinaciones de palabras aleatorias de N bits para A y para B

Para poder compilar está codigo es exactamente el mismo procedimiento que para todos con una pequeña diferencia, el paso 2 donde se escribe el código que compila los archivos verilog ahora será el siguiente

```
iverilog -o TBcableado CableadoBit_tb.v CableadoBit.v CeldaInicialBit.v CeldaTipicaBit.v  CeldaFinaBitl.
```

Únicamente dejar en claro que cuando se aumenta la cantidad de bits en el TestBench de todo el sistema los input de las celdas deben aumentar sus bits de la siguiente forma

```
module CeldaTipicaBit(output P_mid,
input [2:0] A, B,
input P_in, P
);
```

En este caso específico para que puede correr 3 bits, dicho cambio se realiza en todos los módulos

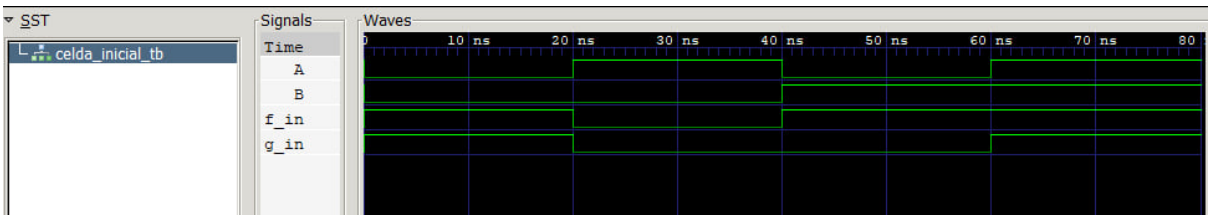
3.4 - Análisis de resultados

Análisis de resultados de Modelo Conductual de izquierda a derecha:

En esta sección se mostrará los resultados de las graficas de comportamiento de cada celda y de todo el sistema, con el fin de explicar su funcionamiento y entenderlo, ademas de poder verificar el correcto funcionamiento de cada uno.

Analisis de resultados en celda inicial:

Al ejecutarse la prueba se obtiene el siguiente grafico:



La cual es una grafica que muestra los valores de f_in y g_in conforme el comportamiento de las entradas A y B, la principal relevancia que nos muestra esta grafica es que se presenta el correcto funcionamiento de la celda inicial, ya que respeta el funcionamiento delas ecuaciones de la celda inicial anteriormente mencionadas:

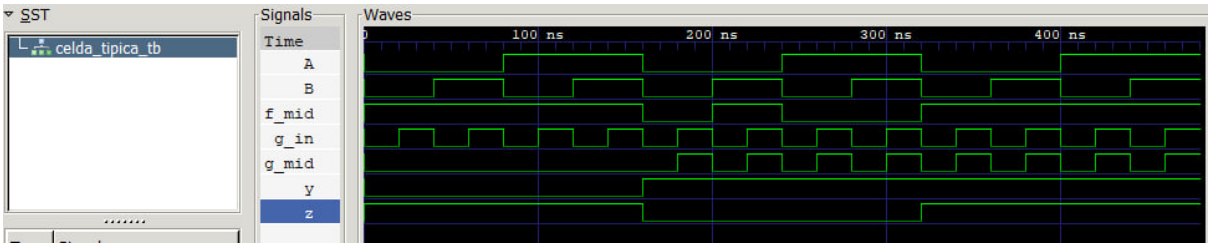
$$Y(0,1,A,B) = \acute{A}B + AB'$$

$$p(0,1,A,B) = \acute{A} + B$$

funcionando igual que ellas y respondiendo correctamente.

Análisis de resultados en celda típica:

Al ejecutarse la prueba se obtiene el siguiente grafico:



La cual es una grafica que muestra los valores de f_mid y g_mid conforme el comportamiento de las entradas A, B, z y Y, la principal relevancia que nos muestra esta grafica es que se presenta el correcto funcionamiento de la celda típica, ya que respeta el funcionamiento delas ecuaciones de la celda inicial anteriormente mencionadas, haciendo enfasis en que solamente se colocaron 3 casos de Y y de Z, ya que estas variables solo toman 3 posibles combinanciones

$$Y(y,p,A,B) = y + \acute{A}B + A\acute{B}$$

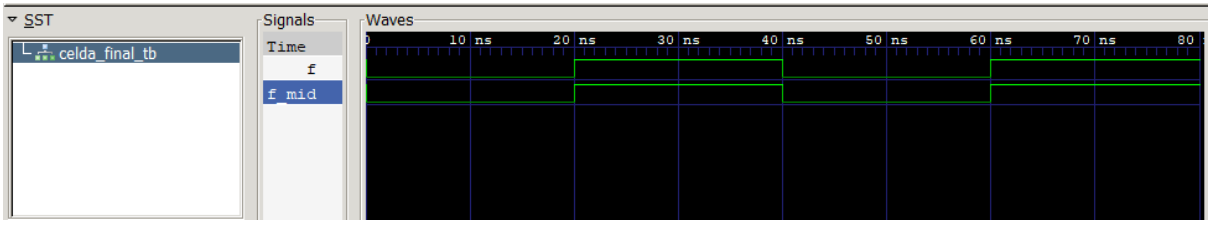
$$Z(y,p,A,B) = p\acute{A} + pB + yp$$

:

funcionando igual que ellas y respondiendo correctamente.

Análisis de resultados en celda final:

Al ejecutarse la prueba se obtiene el siguiente grafico:



Como se mencionó anteriormente, esta es la celda más facil de analizar, ya que al ser un cable que pasa directo desde f_mid hasta f, podemos decir que f = f_mid, así que como se muestra en la grafica, la ecuación se cumple y cada valor de f_mid se replica en f.

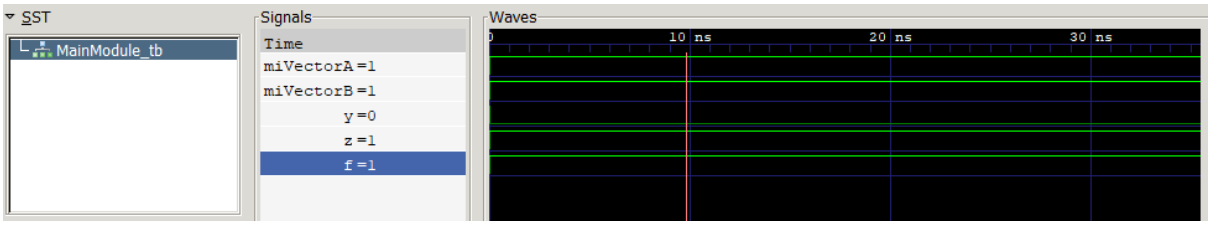
ANALISIS DE RESULTADOS DEL SISTEMA COMPLETO:

Esta sección es **LA MAS IMPORTANTE** de todas las pruebas, ya que en esta se verifica si es que todo el sistema funciona o no, por lo que se dividirá en 3 principales pruebas, una con 1 bit en los 3 posibles casos, una con 3 bits en los 3 posibles casos, y otra con 16 bits con los 3 posibles casos:

1 bit en 3 posibles casos:

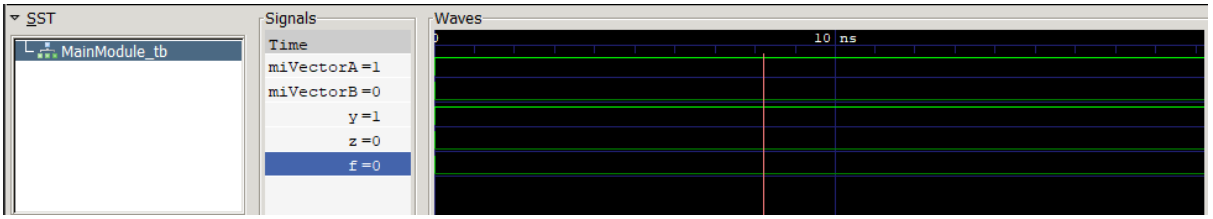
caso vectores iguales:

```
Por el momento las dos palabras son iguales
Para y = 0, z = 1, f = 1
```



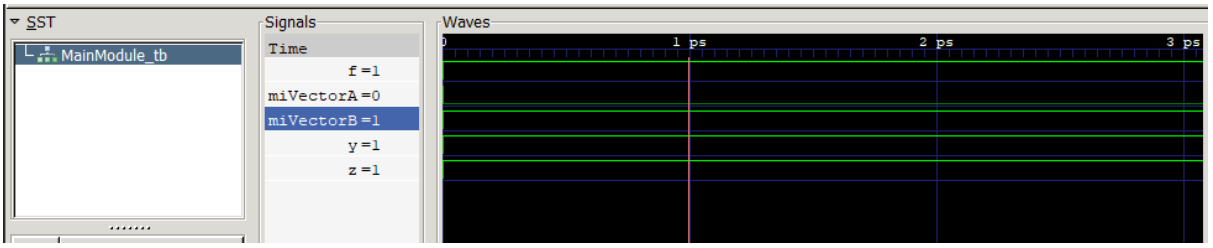
caso A > B:

```
VCD info: dumpfile MainModule_tb.vcd opened for output.
La palabra A es mayor a la palabra B
```



caso B > A:

```
VCD info: dumpfile MainModule_tb.vcd opened for output.
La palabra B es mayor a la palabra A
```

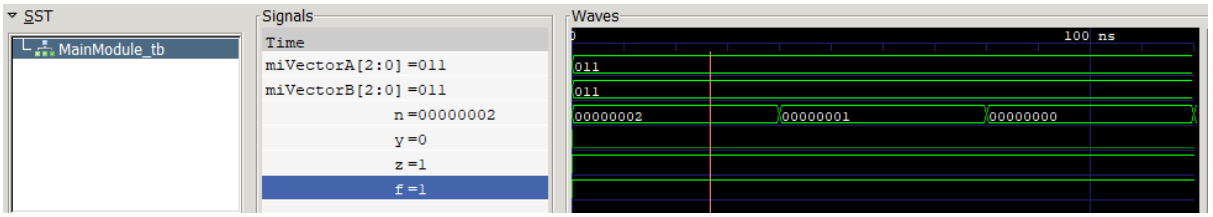


Como se muestra en los 3 casos usando un bit, el codigo funciona perfectamente y me arroja el resultado escrito, definiendo los valores correctos de Y y Z, ademas de un correcto valor de F, tomando en cuentra que *siempre que A y B sean iguales, oN es mayor a A, F siempre va a ser igual a uno, y en caso de que A sea mayor a B, F va a ser igual a 0*

3 bit en 3 posibles casos:

caso vectores iguales: (011 y 011)

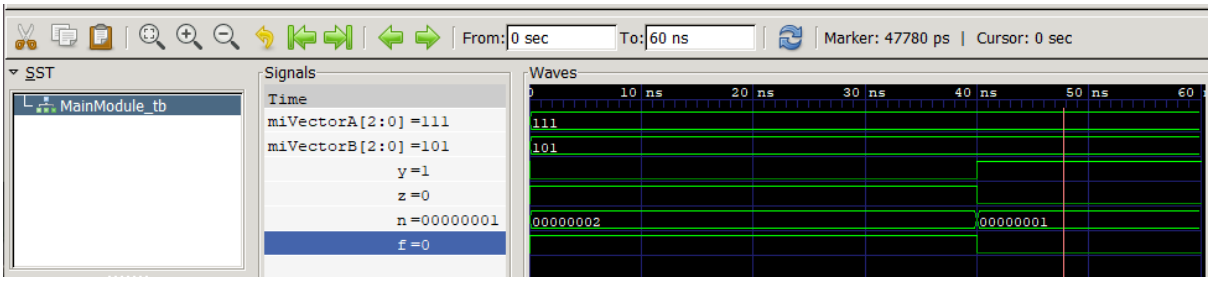
```
VCD info: dumpfile MainModule_tb.vcd opened for output.
Por el momento las dos palabras son iguales
Para n = 2,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 1,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 0,  y = 0, z = 1, f = 1
```



Como se muestra, en cada uno de los bit se muestra el mismo resultado de F = 1, y = 0, z = 1, justo como se habia planteado anteriormente

caso A > B: (A=111 y B=101)

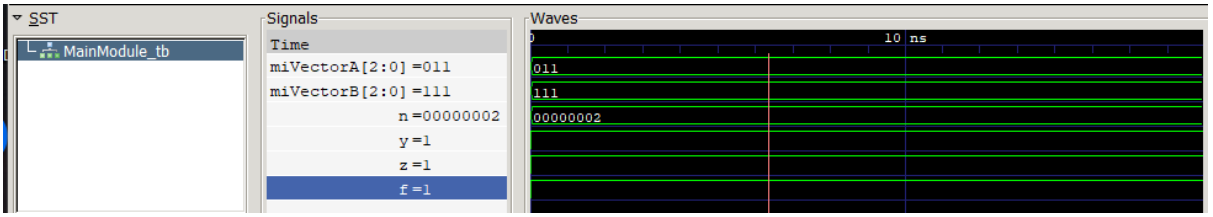
```
Para n = 2,  y = 0, z = 1, f = 1
La palabra A es mayor a la palabra B
```



Como se muestra, justo el segundo bit es donde se hace el cambio y se detecta que A es mayor que B, por lo tanto en la consola lo imprime y F resulta ser igual a 0.

caso B > A: (B=111 y A=011)

```
VCD info: dumpfile MainModule_tb.vcd opened for output.
La palabra B es mayor a la palabra A
```

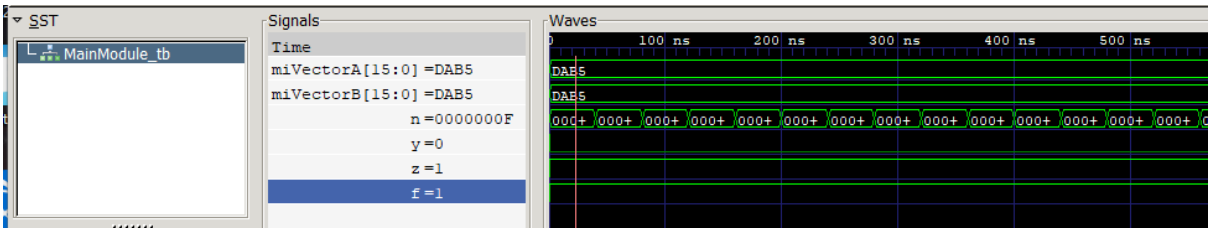


En este caso casualmente el primer bit era donde detectaba donde B era mayor que A, asi que la grafica termina ahi, solo detecta el primer termino, cumpliendo los valores de y y Z, y ademas le puso el valor de f = 1.

16 bit en 3 posibles casos:

caso vectores iguales: (`16'b1101101010110101`)

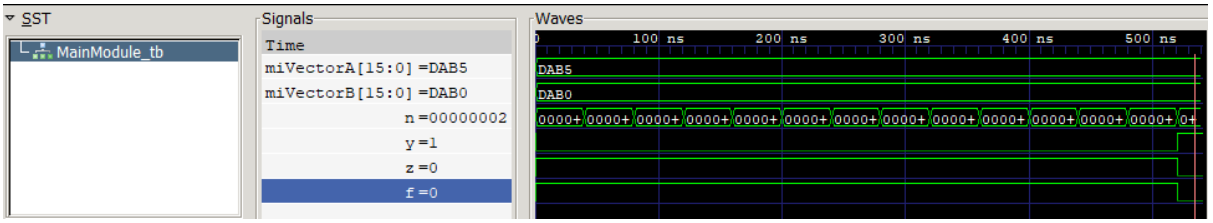
```
Por el momento las dos palabras son iguales
Para n = 15,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 14,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 13,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 12,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 11,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 10,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 9,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 8,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 7,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 6,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 5,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 4,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 3,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 2,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 1,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 0,  y = 0, z = 1, f = 1
```



Como se muestra, el algoritmo funciona igual solo que ahora en vez de hacer 3 iteraciones, hace 16.

caso A > B: (A=`16'b1101101010110101` y B=`16'b1101101010110000`)

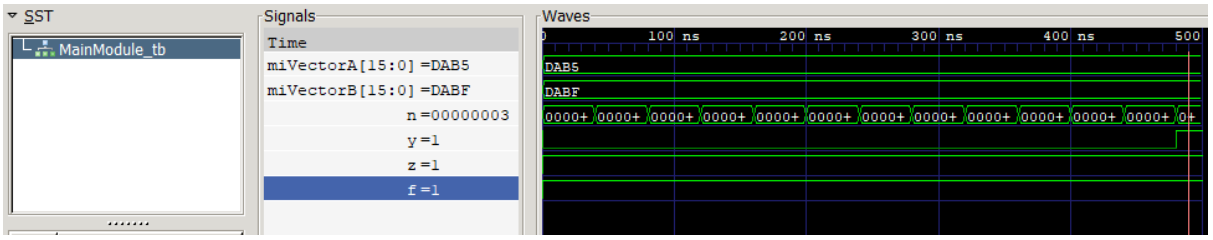
```
VCD info: dumpfile MainModule_tb.vcd opened for output.
Por el momento las dos palabras son iguales
Para n = 15,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 14,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 13,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 12,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 11,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 10,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 9,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 8,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 7,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 6,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 5,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 4,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 3,  y = 0, z = 1, f = 1
La palabra A es mayor a la palabra B
```



Como se muestra en la grafica, a partir del N = 2, encontramos una discrepancia la cual la indica al programa que A es mayor a B, mostrandolo mediante un cambio en la grafica, y dando un F = 0.

caso B > A: (B=`16'b1101101010111111` y A=`16'b1101101010110101`):

```
VCD info: dumpfile MainModule_tb.vcd opened for output.
Por el momento las dos palabras son iguales
Para n = 15,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 14,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 13,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 12,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 11,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 10,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 9,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 8,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 7,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 6,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 5,  y = 0, z = 1, f = 1
Por el momento las dos palabras son iguales
Para n = 4,  y = 0, z = 1, f = 1
La palabra B es mayor a la palabra A
```



Como se muestra en la grafica, justo en el N = 3, el sistema vuelve a notar una discrepancia que le indica que B es mayor que A, y ademas, la salida que muestra es F = 1.

Por lo que una vez analizados todos estos casos, podemos deducir y confirmar que el sistema funciona para multiples casos, y que su funcionamiento, da igual el dato que reciba, no cambia, ya que el funcionamiento es independiente de el valor de los bit que se coloquen, ya que este programa funciona recorriendo vectores, lo cual permite evitar errores logicos relacionados a los bits.

Análisis de resultados de Modelo Estructural de derecha a izquierda

En esta sección se mostraran los resultados obtenidos en las pruebas, estos resultados se obtien al realizar el proceso de compilación, esto con el fin de explicar y comprobar el correcto funcionamiento de cada modulo.

Analisis de resultados en celda inicial

Al ejecutar el proceso de compilación y ejecutar el archivo en GTKwave no da el siguiente diagrama de tiempo:

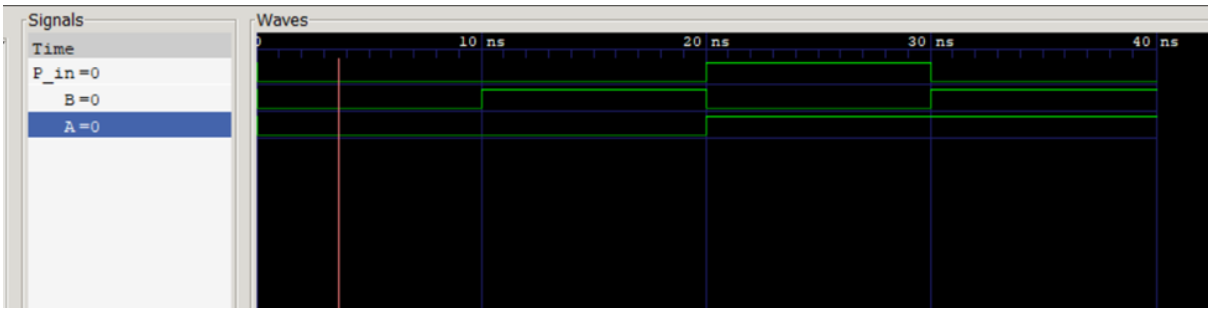


Figura. Diagrama de tiempo celda inicial de derecha a izquierda

En donde podemos observar que el único caso en que la salida P_in se activa en alto es cuando A es mayor a B, si observamos el esquemático (x) de la celda inicial al tener un AND en la sailda la única forma de que su valor sea 1 es cuando ambas entradas son 1 y esto solo ocurre con A=1 Y B=0 ya que este ultimo tiene un negado que lo hace 1, comprobando de esta forma que la prueba es un éxito.

Análisis de resultados en celda típica:

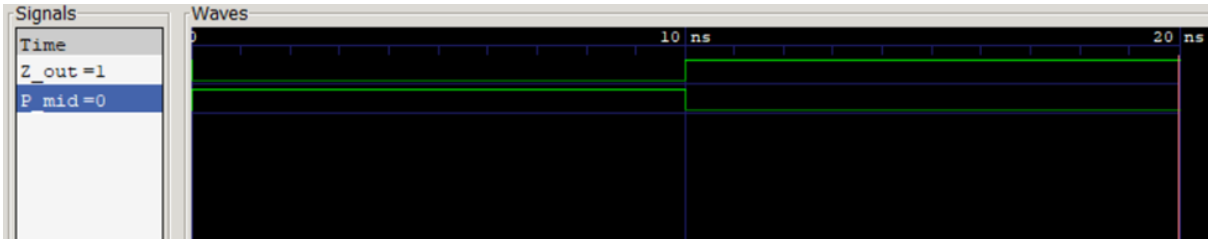
Al ejecutar el proceso de compilación y ejecutar el archivo en GTKwave no da el siguiente diagrama de tiempo

Figura. Diagrama de tiempo celda inicial de derecha a izquierda

En este caso tenemos que tener en mente una cosa y es el valor de P, aunque no aparezca en el resultado en el código del Testbench tiene un gran valor ya que este nos define cómo se va a comportar la celda ya que el valor de P puede ser 1 o 0 y el único caso en donde este es 1 es cuando A es mayor a B como se definió en la asignación de estados, con todo esto y fijándonos en el esquemático () en el momento que P =1 tenemos A = 1, B = 0 y debido a que B está negado cada AND recibe en cada entrada 1 y conociendo que salida de AND es 1 solo cuando sus dos salidas equivalen a 1 y siguiendo con este análisis a OR llegan tres 1 y esta compuerta es 1 solo con que una de sus entradas sea 1 entonces podemos afirmar que es solo en este caso cuando P_mid puede estar en alto, es decir, la prueba es un éxito.

Análisis de resultados en celda final:

Al ejecutar el proceso de compilación y ejecutar el archivo en GTKwave no da el siguiente diagrama de tiempo



En donde se puede apreciar claramente que la salida Z_out es contrario a la entrada P_mid es decir que la prueba es un éxito.

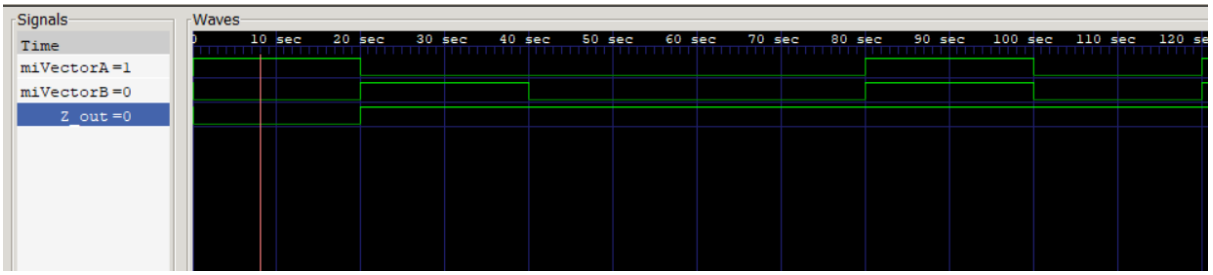
Comprobando que las tres celdas funcionan correctamente ahora si podemos pasar al el análisis del sistema completo

ANALISIS DE RESULTADOS DEL SISTEMA COMPLETO:

El primer caso que se debe de comprobar es cuando N vale 1 bit, lo que se busca es que cumpla con el enunciado del proyecto el cual dice que debe de activar la salida Z en bajo cuando detecte que A es mayora a B, ejecutamos el código y nos arroja el siguiente resultado

```
Prueba 0
miVectorA = 1, miVectorB = 0
P = 1, Z_out = 0
Prueba 1
miVectorA = 0, miVectorB = 1
P = 0, Z_out = 1
Prueba 2
miVectorA = 0, miVectorB = 0
P = 0, Z_out = 1
Prueba 3
miVectorA = 0, miVectorB = 0
P = 0, Z_out = 1
Prueba 4
miVectorA = 1, miVectorB = 1
P = 0, Z_out = 1
Prueba 5
miVectorA = 0, miVectorB = 0
P = 0, Z_out = 1
Prueba 6
miVectorA = 1, miVectorB = 1
P = 0, Z_out = 1
Prueba 7
miVectorA = 1, miVectorB = 1
P = 0, Z_out = 1
Prueba 8
miVectorA = 1, miVectorB = 0
P = 1, Z_out = 0
Prueba 9
```

Cabe mencionar que el código generado para el sistema completo me genera 10 combinaciones de palabras abarcan las 4 combinaciones posibles para A y B de un bit solo que el hacer 10 palabras de prueba algunos algunos se repiten, del cual se puede verificar su correcto funcionamiento ya que Z = 0 solo en el caso en que a es mayor, esto se puede ver también con el diagrama de tiempo



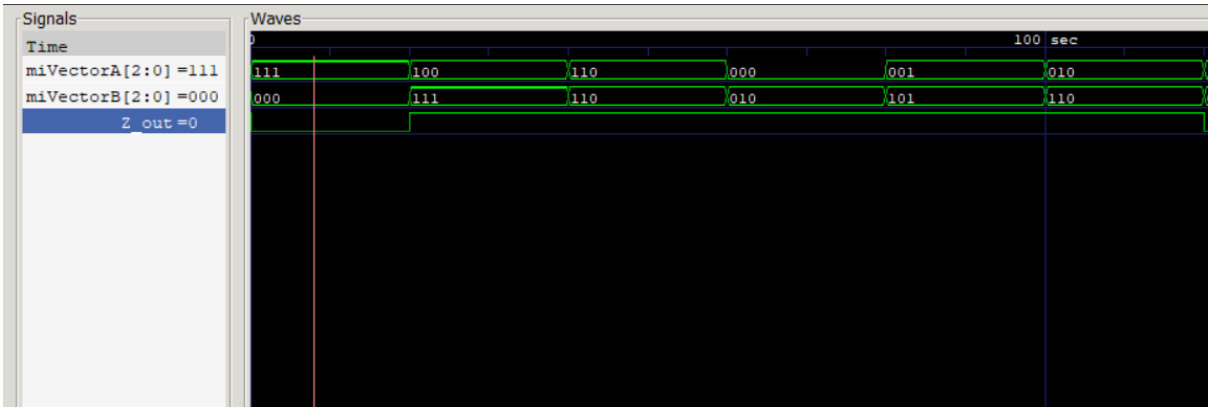
Y con esto solo reafirmamos el correcto funcionamiento del código

Caso para 3 bit

A continuación vamos a empezar a poner nuestro código a prueba verificando si verdaderamente funciona para cualquier N bit por ello vamos a ejecutar el codigo para un N = 3

```
Prueba 0
miVectorA = 111, miVectorB = 000
P = 1, Z_out = 0
Prueba 1
miVectorA = 100, miVectorB = 111
P = 0, Z_out = 1
Prueba 2
miVectorA = 110, miVectorB = 110
P = 0, Z_out = 1
Prueba 3
miVectorA = 000, miVectorB = 010
P = 0, Z_out = 1
Prueba 4
miVectorA = 001, miVectorB = 101
P = 0, Z_out = 1
Prueba 5
miVectorA = 010, miVectorB = 110
P = 0, Z_out = 1
Prueba 6
miVectorA = 101, miVectorB = 001
P = 1, Z_out = 0
Prueba 7
miVectorA = 101, miVectorB = 111
P = 0, Z_out = 1
Prueba 8
miVectorA = 001, miVectorB = 100
P = 0, Z_out = 0
Prueba 9
miVectorA = 010, miVectorB = 011
P = 0, Z_out = 1
```

Al ejecutar el código nos imprime este análisis en el cual podemos comprobar que efectivamente el código las palabras para generar un resultado dando una salida Z = 0 solo cuando el A es mayor a B, para terminar de corroborar vamos a ir al Gtkwave para obtener su respectivo diagrama de tiempo



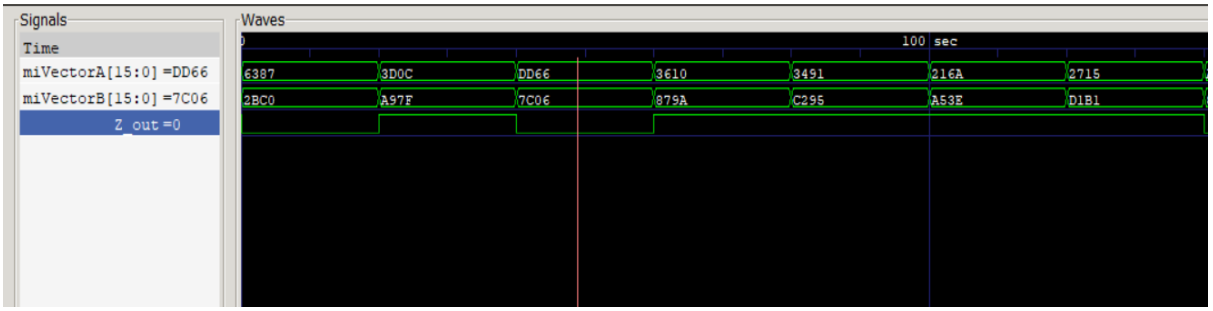
Observando el diagrama de tiempo podemos afirmar que el código funciona, es decir la prueba es un éxito

Caso para 16 bit

Como última prueba y dar por correcto el código vamos a implementar 16 bits y verificar si efectivamente funciona para N bits, compilamos el codigo y nos da lo siguiente

```
Prueba 0
miVectorA = 0110001110000111, miVectorB = 0010101111000000
P = 1, Z_out = 0
Prueba 1
miVectorA = 0011110100001100, miVectorB = 1010100101111111
P = 0, Z_out = 1
Prueba 2
miVectorA = 1101110101100110, miVectorB = 0111110000000110
P = 1, Z_out = 0
Prueba 3
miVectorA = 0011011000010000, miVectorB = 1000011110011010
P = 0, Z_out = 1
Prueba 4
miVectorA = 0011010010010001, miVectorB = 1100001010010101
P = 0, Z_out = 1
Prueba 5
miVectorA = 0010000101101010, miVectorB = 1010010100111110
P = 0, Z_out = 1
Prueba 6
miVectorA = 0010011100010101, miVectorB = 1101000110110001
P = 0, Z_out = 1
Prueba 7
miVectorA = 1010010000000101, miVectorB = 1000000101001111
P = 1, Z_out = 0
```

Se logra visualizar que también se cumple , ahora vamos a generar el archivo vcd para leerlo en GTKwave y comprobar que funciona correctamente



Si transformamos la primera combinación tenemos que A = 25479 y B = 11200 en el que claramente A es mayor a B por lo tanto la salida Z es activa en 0 otro ejemplo es la prueba 2 donde A = 15628 y B = 43391 donde claramente A es menor que B por lo tanto Z está en 1 y asi con todas la palabras esto se puede diferenciar más cuando compilamos el codigo

4 - Conclusiones

En este proyecto, nos embarcamos en el diseño de un circuito digital en Verilog con el propósito de comparar palabras de N bits, A y B, siendo N mayor o igual a 3. El objetivo primordial fue detectar si la palabra A es mayor que la palabra B y comunicarlo a través de la señal Z, activa en bajo. Para lograr esta comparación, se empleó la metodología de diseño basada en redes iterativas, dividiendo el proyecto en dos partes: un análisis de las palabras de entrada de izquierda a derecha y otra de derecha a izquierda.

La implementación de redes iterativas permitió abordar este desafío de manera eficiente y precisa. Se logró diseñar un circuito funcional que realizó la comparación bit a bit de las palabras de entrada con dos TestBench distintos para comprar su eficiencia y precisión.

Se adquirirá una comprensión más profunda sobre la importancia y aplicaciones de las redes iterativas en el diseño de circuitos digitales. Esta metodología demuestra su relevancia al permitirnos resolver el problema planteado de manera efectiva, independientemente de la longitud de las palabras de entrada.

5 - Referencias Bibliográficas

What is a Verilog testbench? (n.d.). MATLAB & Simulink. https://la.mathworks.com/discovery/verilog-testbench.html

Admin. (2023, June 20). *Verilog module*. ChipVerify. https://www.chipverify.com/verilog/verilog-modules

M.Sc., I. G. (s.f.). *redes iterativas.* Recuperado el 27 de Noviembre de 2023, de monografias.com: https://www.docsity.com/es/redes-iterativas-2d/3115362/

Saldaña, M. A. (s.f.). *Redes Iterativas.* Recuperado el 27 de Noviembre de 2023, de pdfslide.tips: https://pdfslide.tips/documents/redes-iterativas.html?page=1

Universidad Técnica Federico Santa María. (s.f.). *Microsoft Word - ap5.doc.* Recuperado el 27 de Noviembre de 2023, de ramos.elo.utfsm.cl: http://ramos.elo.utfsm.cl/~elo212/docs/lsilva-ap5.pdf