Diseño digital en Verilog: Comparador de palabras de N bits

Universidad de Costa Rica
Facultad de Ingeniería Eléctrica

IE0323 Circuitos Digitales I GR003

Fecha de entrega: 29/11/2023

Número de grupo de proyecto: 3

Profesor:
Prof. Rafael Esteban
Badilla Alvarado

Miembros:
Lidia Magaly García González, B83169

Roger Daniel Piovet García, C15990

Fabricio Arguijo Cantillo, B70645

Índice

1. Enunciado

En este proyecto, se busca desarrollar un circuito digital en Verilog capaz de comparar dos palabras de N bits, A y B, para cualquier N mayor o igual a 3. El objetivo principal es detectar si la palabra A es mayor que la palabra B y noticarlo a través de una señal llamada Z, la cual es activa en bajo. Para lograr esto, se utilizará la metodología de diseño mediante redes iterativas. El proyecto se dividirá en dos partes, abordando en cada parte, un sentido diferente en el recorrido de las palabras de entrada: de izquierda a derecha y viceversa

2. Diseño

Los diseños de los circuitos digitales capaces de comparar dos palabras de N bits, A y B para $N \geq 3$ que se desarrollarán en Verilog se realizaron a partir de la metodología de diseño mediante redes iterativas. Esta metodología es utilizada para circuitos digitales combinacionales que deben procesar una gra cantidad de bits, o una cantidad desconocida de bits [2].

Se define una *red iterativa* como:

> Un conjunto de celdas de lógica combinacional idénticas en las cuáles la información es pasada de una celda a la siguiente de una manera lineal. Una excepción a la simetría de la estructura lo representan la primera y la última celda las cuales son, en general, diferentes.

Los diseños realizados para cada recorrido de las palabras de entrada (izquierda a derecha, y, derecha a izquierda) son parte de excepción a la simetría. Esto debido a que las celdas finales e iniciales correspondiente a cada diseño son estructuralmente distintas entre ellas en términos de compuertas y de puertos I/O, y distintas a la celda típica. Los pasos a seguir para el diseño de las redes iterativas basado en transición de estados son los siguientes:

1. Definir los estados que resuelven el problema.
2. Construir la Tabla de Transición de Estados.
3. Hacer la asignación de estados, la declaración de variables de estado y variables de salida.
4. Hacer la Tabla de Transición de Estados codificada.
5. Dibujar el diagrama de bloque de la celda típica.
6. Diseñar la celda típica.
7. Diseñar la celda inicial.
8. Diseñar la celda final.

Los pasos 2, 4, 5, 6, 7, y 8 son triviales, ya que tratan conceptos mecánicos que se han visto a profundidad a lo largo del curso. Esto incluye la construcción de tablas de transición de estados, realizar diagramas de bloques, o resolver mapas de Karnaugh de hasta 5 variables. El paso 1 no es trivial, ya que se deben definir los estados mínimos para la comparación eficiente de las palabras

A y B. Debe ser justificado al detalle porque esta definición de estados es la mínima, al igual que las transiciones necesarias. Esto es logrado inicialmente por medio de palabras de prueba, y posteriormente, verificado a partir de testbenchs en la implementación de Verilog del diseño.

Se solicita realizar una implementación conductual en Verilog para un recorrido de las palabras de entrada, y una implementación estructural en Verilog para el otro recorrido. Se necesita un criterio técnico para escoger el tipo de implementación que se escoge para cada recorrido. Dicho criterio puede ser obtenido de la parte de diseño. El criterio que se escogió parte de la cantidad de compuertas total de la red iterativa. La escogencia de implementación se realiza de la siguiente manera:

- El diseño con **más** compuertas se le aplicará una implementación conductual en Verilog.
- El diseño con **menos** compuertas se le aplicará una implementación estructural en Verilog

La justificación de haber escogido este criterio para determinar que tipo de implementación se le aplica a los recorridos de palabras se basa en que, mientras más compuertas tenga un diseño, más complicado será implementarlo estructuralmente en Verilog, al igual que depurarlo. Sin embargo, el hecho de que el diseño tenga una gran cantidad de compuertas no implica que su funcionalidad sea compleja. Debido a esto, podría ser posible sustituir una implementación estructural por una implementación conductual, la cual utilice estructuras de control de Verilog para emular el funcionamiento de las celdas de la red iterativa.

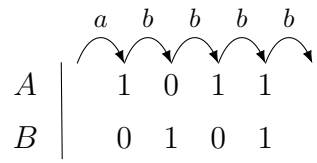
2.1. Izquierda a derecha

2.1.1. Definición de estados

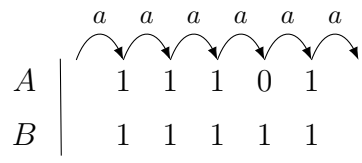
Para la definición de estados, se realizaron algunas palabras de prueba correspondientes a la primera palabra A y la segunda palabra B. La única restricción que se consideró para escoger estas palabras de prueba es que fueran ambas de tamaño N, con $N \geq 3$. Considere los estados a y b, definidos tal que:

- a: Hasta el momento, la primera palabra es menor o igual a la segunda palabra porqué los pares de bits en la misma posición en ambas palabras recibidos han sido los mismos, o se ha recibido un bit igual a cero en la primera palabra y un bit igual a uno en la segunda palabra, o el estado se encuentra presente en la celda inicial **(estado inicial)**.
- b: La primera palabra es mayor a la segunda palabra porqué se ha recibido un bit igual a uno en la primera palabra y un bit igual a cero en la segunda palabra en la misma posición en ambas palabras.

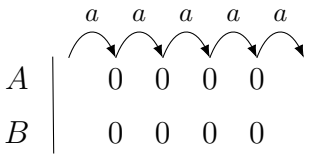
El estado a es el único estado que califica como estado inicial, debido a que no se trata de una afirmación, sino de un estado que puede ser transitorio. Se puso a prueba la definición de estados anterior por medio de 4 palabras de prueba:



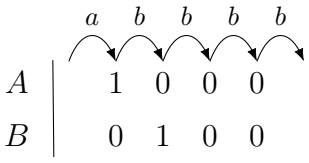
(a): $A = 11_{10}$ y $B = 5_{10}$



(b): $A = 29_{10}$ y $B = 31_{10}$



(c): $A = 0_{10}$ y $B = 0_{10}$



(d): $A = 8_{10}$ y $B = 4_{10}$

Figura 1: Palabras de prueba para la definición de estados de a y b

Note que, con esta definición de estados, en los cuatro casos, la red iterativa finaliza exitosamente. Esto debido a que para cada par de palabras de prueba

- En la sub-figura (a), $11_{10} > 5_{10}$ y el estado de salida es b.
- En la sub-figura (b), $29_{10} < 31_{10}$ y el estado de salida es a.
- En la sub-figura (c), $0_{10} \leq 0_{10}$ y el estado de salida es a.
- En la sub-figura (d), $8_{10} > 4_{10}$ y el estado de salida es b.

A simple vista se podría decir que los estados a y b son los estados mínimos requeridos para un comparador de palabras de N bits diseñado por medio de la metodología de diseño mediante redes iterativas, con un recorrido de palabras de entrada de izquierda a derecha. Sin embargo, en estas pruebas hubo un caso en particular con respecto a los valores de los bits individuales de A y B.

Considere las posiciones i-ésima y j-ésima de A y B, tal que $0 \leq j < i \leq N - 1$. En la posición i-ésima se recibe $A_i = 0$ y $B_i = 1$ y el estado presente es a. Por tanto, el próximo estado será también a. Después, en la posición j-ésima de A y B se recibe $A_j = 1$ y $B_j = 0$ con estado presente a. Por tanto, el próximo estado va a ser b. Como se estaba analizando las palabras de izquierda a derecha, el primer par de bits diferentes entre las palabras que se reciben permite determinar cuál palabra es mayor. Sin embargo, con la definición de estados que se propuso, la red iterativa finalizará en un caso de fallo. Esto es debido a que el estado b es una afirmación, y en el caso propuesto, se está afirmando algo que no es cierto. Lo anterior mencionado se ejemplifica por medio de la siguiente palabra de prueba:

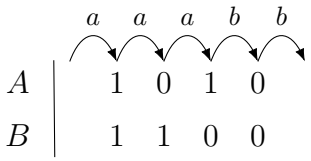
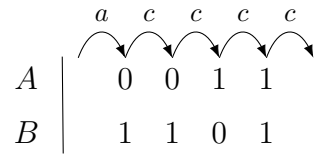


Figura 2: Contraejemplo de la definición de estados propuesta inicialmente.
$A = 10_{10}$ y $B = 12_{10}$

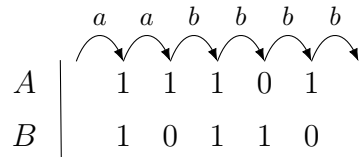
En el caso anterior, note que $10_{10} \leq 12_{10}$, pero el estado de salida de la red es b. Se propone una nueva definición de estados que intenta resolver este problema:

- a: Hasta el momento, la primera palabra es igual a la segunda palabra porqué los pares de bits en la misma posición en ambas palabras recibidos han sido los mismos, o el estado se encuentra presente en la celda inicial **(estado inicial)**.
- b: La primera palabra es mayor a la segunda palabra porqué se ha recibido un bit igual a uno en la primera palabra y un bit igual a cero en la segunda palabra en la misma posición en ambas palabras.
- c: La segunda palabra es mayor a la primera palabra porque se ha recibido un bit igual a cero en la primera palabra y un uno en la segunda palabra en la misma posición en ambas palabras.

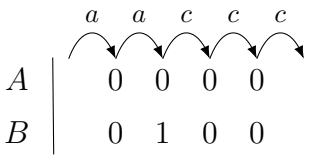
Note que el estado a sigue siendo el único estado que puede aplicar como estado inicial. En comparación a la definición de estados que se realizó inicialmente, el estado a solo corresponde al caso en donde, hasta el momento, $A = B$. Además de esto, se agregó un estado c el cuál es correspondiente al caso en donde $A < B$. Nuevamente, se realizan cuatro palabras de prueba con esta definición de estados



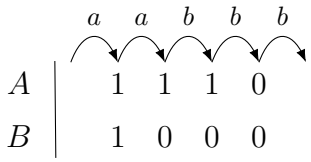
(a): $A = 3_{10}$ y $B = 13_{10}$



(b): $A = 29_{10}$ y $B = 22_{10}$



(c): $A = 0_{10}$ y $B = 4_{10}$



(d): $A = 14_{10}$ y $B = 8_{10}$

Figura 3: Palabras de prueba para la nueva definición de estados de a, b, y c

Note que, con esta definición de estados, en los cuatro casos, la red iterativa finaliza exitosamente. Esto debido a que para cada par de palabras de prueba:

- En la sub-figura (a), $3_{10} < 13_{10}$ y el estado de salida es c.

- En la sub-figura (b), $29_{10} > 22_{10}$ y el estado de salida es b.
- En la sub-figura (c), $0_{10} < 4_{10}$ y el estado de salida es c.
- En la sub-figura (d), $14_{10} > 8_{10}$ y el estado de salida es b.

Se avanzará el diseño con esta definición de estados. Su desempeño ante palabras de N bits será comprobado en la parte de pruebas, después de que el diseño sea implementado en Verilog.

2.1.2. Tabla de transición de estados

En base a las palabras de prueba y la definición de estados a la que se llegó en la sección anterior, se construye la siguiente tabla de transición de estados:

Estado Presente	Próximo Estado			
	$A_iB_i = 00$	$A_iB_i = 01$	$A_iB_i = 10$	$A_iB_i = 11$
a	a	c	b	a
b	b	b	b	b
c	c	c	c	c

Cuadro 1: Tabla de transición de estados para el diseño recorriendo las palabras de izquierda a derecha

En la tabla anterior, note que para los estados presentes b y c, sus próximos estados son ellos mismos, independientemente de el valor de los bits recibidos A_i y B_i. Esto debido a que las definiciones de los estados b y c son afirmaciones. Entonces, cuando una celda cae al estado b o c, proveniendo del estado a, los próximos estados de las siguientes celdas van a ser los mismos.

2.1.3. Asignación de estados y declaraciones de variables de estado y de salida

Para el recorrido de las palabras de entrada de izquierda a derecha, se abordará el diseño con la siguiente codificación de estados

- a: 10
- b: 01
- c: 11

Las variables de estado serán p y q para las variables de estado presente, P y Q para las variables de próximo estado, y Z para la salida final de la red iterativa.

2.1.4. Tabla de transición de estados codificada

En base a la asignación de estados anterior y el cuadro 1, se obtiene la siguiente tabla de transición de estados codificada

Estado Presente pq	Próximo Estado PQ			
	$A_iB_i = 00$	$A_iB_i = 01$	$A_iB_i = 10$	$A_iB_i = 11$
01	01	11	10	01
10	10	10	10	10
11	11	11	11	11

Cuadro 2: Tabla de transición de estados codificada recorriendo las palabras de izquierda a derecha

2.1.5. Diagrama de bloque de celda típica

El diagrama de bloque de la celda típica correspondiente al presente diseño posee 4 entradas: las variables de estado presente p y q, los bits de A y B en la posición i-ésima A_i y B_i. En este contexto, $1 \leq i \leq N-2$. Esto es debido a que la red posee celdas típicas únicamente para los bits de las palabras A y B en las posiciones acotadas por 1 y $N-2$, inclusive, ya que la celda correspondiente a los bits en la posición $N-2$ de ambas palabras es la celda inicial, mientras que los bits en la posición 0 corresponden a la celda final. Esto debido a que se están recorriendo las palabras de izquierda a derecha. El diagrama de bloque posee dos salidas: las variables próximo estado P y Q.

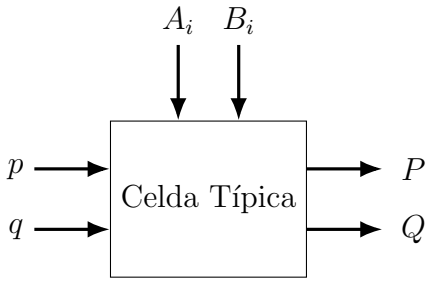


Figura 4: Diagrama de bloque de celda típica para el recorrido de las palabras de entrada de izquierda a derecha

2.1.6. Diseño de celda típica

En base a el cuadro 2, se realizan dos tablas de verdad con entradas $p\,q\,A_i\,B_i$, en este mismo orden de procedencia. Cada tabla de verdad corresponderá a una de las variables de estado P y Q. Para las entradas, se anotará en la tabla todas las posibles combinaciones entre estas, abarcando desde el valor 0000_2 hasta el 1111_2. Esto se realiza con el propósito de obtener la expansión en mintérminos de cada variable de estado.

Índice	Entradas				Salidas	
	p	q	A_i	B_i	P	Q
0	0	0	0	0	X	X
1	0	0	0	1	X	X
2	0	0	1	0	X	X
3	0	0	1	1	X	X
4	0	1	0	0	0	1
5	0	1	0	1	1	1
6	0	1	1	0	1	0
7	0	1	1	1	0	1
8	1	0	0	0	1	0
9	1	0	0	1	1	0
10	1	0	1	0	1	0
11	1	0	1	1	1	0
12	1	1	0	0	1	1
13	1	1	0	1	1	1
14	1	1	1	0	1	1
15	1	1	1	1	1	1

Cuadro 3: Tabla de verdad basada en el cuadro 2 con entradas $p\,q\,A_i\,B_i$ y salidas P y Q

En base a la tabla de verdad anterior, se obtienen las expansiones en mintérminos de P y Q

$$P(p,\, q,\, A_i,\, B_i) = \sum m(5,\, 6,\, 8,\, 9,\, 10,\, 11,\, 12,\, 13\,14,\, 15) + d(0,\, 1,\, 2,\, 3) \tag{1}$$

$$Q(p,\, q,\, A_i,\, B_i) = \sum m(4,\, 5,\, 7,\, 12,\, 13,\, 14,\, 15) + d(0,\, 1,\, 2,\, 3) \tag{2}$$

A partir de estas expansiones en mintérminos, se mapea cada función en mapas de Karnaugh de 4 variables y se soluciona buscando todos los implicantes primos para obtener la función mínima de P y Q.

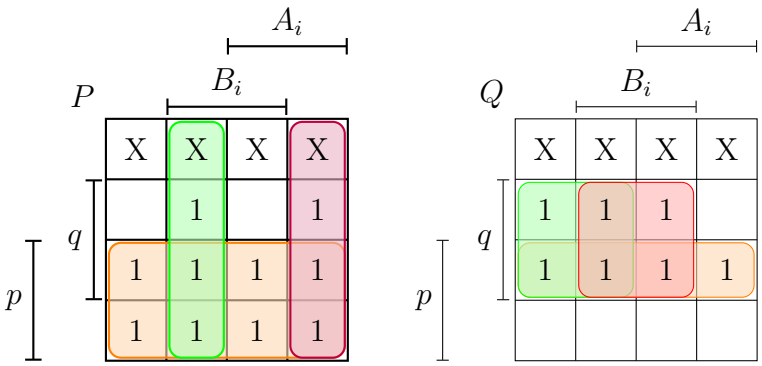


Figura 5: Solución de los mapas de Karnaugh correspondientes a las variables de estado P y Q

Al incluir los implicantes esenciales y aquellos implicantes primos que cubren los unos no cubiertos por implicantes esenciales para ambas variables de estado, se obtienen las siguientes funciones mínimas

$$\begin{aligned} P &= p + \overline{A_i}B_i + A_i\overline{B_i} \\ &= p + A_i \oplus B_i \end{aligned} \tag{3}$$

$$\begin{aligned} Q &= pq + q\overline{A_i} + qB_i \\ &= q(p + \overline{A_i} + B_i) \end{aligned} \tag{4}$$

Donde el color de cada término es el color del implicante al que corresponde en el mapa de Karnaugh de cada variable de estado. En base a estas ecuaciones lógicas, el esquemático de la celda típica es el siguiente

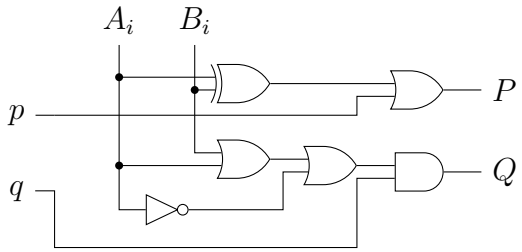


Figura 6: Esquemático de celda típica recorriendo las palabras de izquierda a derecha

Note que se necesitan 5 compuertas de dos entradas para implementar la celda típica con la asignación de estados correspondiente al recorrido de las palabras de entrada de izquierda a derecha.

2.1.7. Diseño de la celda inicial

El estado que se escogió como inicial fue el estado a. Debido a que la codificación del estado a es 01, se evalúan las ecuaciones de P y Q en $p = 0$ y $q = 1$

$$\begin{aligned} P(0,\, 1,\, A_i,\, B_i) &= 0 + A_i \oplus B_i \\ &= A_i \oplus B_i \qquad (5) \\ Q(0,\, 1,\, A_i,\, B_i) &= 1 \cdot (0 + \overline{A_i} + B_i) \\ &= \overline{A_i} + B_i \qquad (6) \end{aligned}$$

El esquemático de la celda inicial es el siguiente

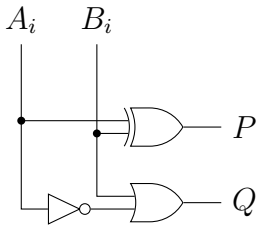


Figura 7: Esquemático de celda inicial recorriendo las palabras de izquierda a derecha

Note que se necesitan dos compuertas de dos entradas para implementar la celda inicial con la asignación de estados correspondiente al recorrido de palabras de entrada de izquierda a derecha.

2.1.8. Diseño de celda final

Para la celda final, note que si la red termina en el estado b, $Z = 0$ debido a que esta salida final es activa en bajo. Por lo contario, $Z = 1$. Esto sería cuando la red termina en el estado a y b. En base a esto, y en el cuadro 2, se construye la siguiente tabla de verdad con entradas $p\, q\, A_i\, B_i$ y salida Z

Índice	Entradas				Salidas
	p	q	A_i	B_i	Z
0	0	0	0	0	X
1	0	0	0	1	X
2	0	0	1	0	X
3	0	0	1	1	X
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	1

Cuadro 4: Tabla de verdad basada en el cuadro 2 con entradas $p\,q\,A_i\,B_i$ y salidas Z

A partir de la tabla de verdad anterior, se obtiene la expansión en mintérminos de Z

$$Z(p,\, q,\, A_i,\, B_i) = \sum m(4,\, 5,\, 7,\, 12,\, 13,\, 14,\, 15) + d(0,\, 1,\, 2,\, 3) \tag{7}$$

A partir de esta expansiones en mintérminos, se mapea la función en un mapa de Karnaugh de 4 variables y se soluciona buscando todos los implicantes primos para obtener la función mínima de Z.

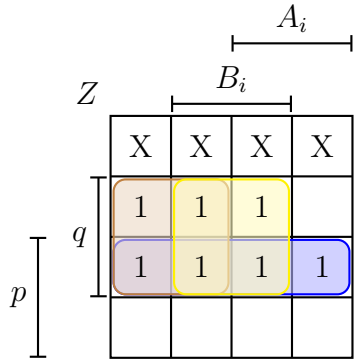


Figura 8: Solución del mapa de Karnaugh correspondientes a la salida final Z

La función mínima vendrá dada por

$$\begin{aligned} Z &= pq + q\overline{A_i} + qB_i \\ &= q(p + \overline{A_i} + B_i) \end{aligned} \tag{8}$$

Por tanto, el esquemático de la celda final es el siguiente

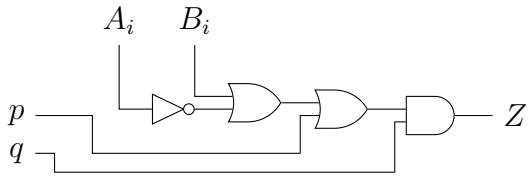


Figura 9: Esquemático de celda final para el recorrido de palabras de entrada de izquierda a derecha

Note que se necesitan 3 compuertas de dos entradas para implementar la celda final con la asignación de estados correspondiente al recorrido de las palabras de entrada de izquierda a derecha.

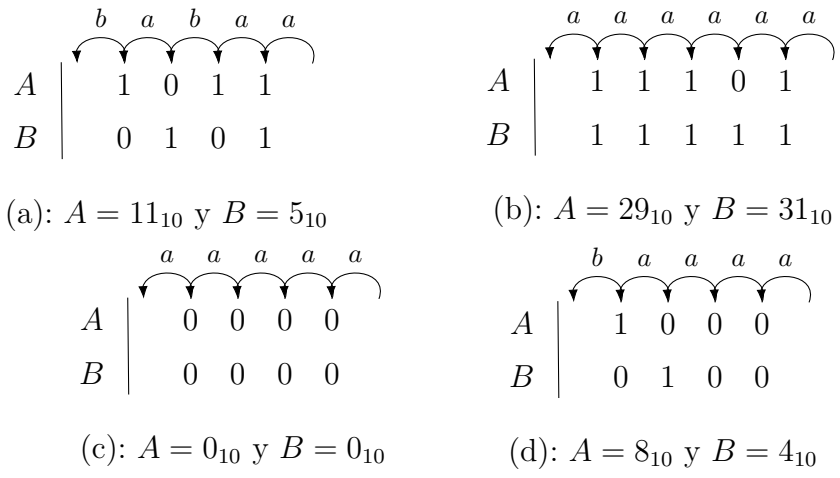
2.2. Derecha a izquierda

2.2.1. Definición de Estados

De forma similar a la definición de estados realizada para el recorrido de palabras de izquierda a derecha, se realizaron algunas palabras de prueba correspondientes a la primera palabra A y la segunda palabra B. Considere los estados a y b, definidos tal que:

- a: Hasta el momento, la primera palabra es menor o igual a la segunda palabra porqué los pares de bits en la misma posición en ambas palabras recibidos han sido los mismos, o se ha recibido un bit igual a cero en la primera palabra y un bit igual a uno en la segunda palabra, o el estado se encuentra presente en la celda inicial **(estado inicial)**.
- b: Hasta el momento, la primera palabra es mayor a la segunda palabra porqué se ha recibido un bit igual a uno en la primera palabra y un bit igual a cero en la segunda palabra en la misma posición en ambas palabras.

Nuevamente, estado a es el único estado que califica como estado inicial Se puso a prueba la definición de estados anterior por medio de las mismas 4 palabras de prueba utilizadas para la definición de estados preliminar del recorrido de palabras de entrada de izquierda a derecha



(a): $A = 11_{10}$ y $B = 5_{10}$

(b): $A = 29_{10}$ y $B = 31_{10}$

(c): $A = 0_{10}$ y $B = 0_{10}$

(d): $A = 8_{10}$ y $B = 4_{10}$

Figura 10: Palabras de prueba para la definición de estados de a y b

Note que, con esta definición de estados, en los cuatro casos, la red iterativa finaliza exitosamente. Esto debido a que para cada par de palabras de prueba

- En la sub-figura (a), $11_{10} > 5_{10}$ y el estado de salida es b.
- En la sub-figura (b), $29_{10} < 31_{10}$ y el estado de salida es a.
- En la sub-figura (c), $0_{10} \leq 0_{10}$ y el estado de salida es a.
- En la sub-figura (d), $8_{10} > 4_{10}$ y el estado de salida es b.

Se avanzará el diseño con esta definición de estados.

2.2.2. Tabla de transición de estados

En base a las palabras de prueba y la definición de estados a la que se llegó en la sección anterior, se construye la siguiente tabla de transición de estados

Estado Presente	Próximo Estado			
	$A_iB_i = 00$	$A_iB_i = 01$	$A_iB_i = 10$	$A_iB_i = 11$
a	a	a	b	a
b	b	a	b	b

Cuadro 5: Tabla de transición de estados para el diseño recorriendo las palabras de derecha a izquierda

2.2.3. Asignación de estados y declaraciones de variables de estado y de salida

Para el recorrido de las palabras de entrada de derecha a izquierda, se abordará el diseño con la siguiente codificación de estados

- a: 1
- b: 0

Se asignará p como la variable de estado presente, P la variable de próximo estado, y Z la salida final de la red.

2.2.4. Tabla de transición de estados codificada

En base a la asignación de estados anterior y el cuadro 5, se obtiene la siguiente tabla de transición de estados codificada

Estado Presente	Próximo Estado			
	$A_iB_i = 00$	$A_iB_i = 01$	$A_iB_i = 10$	$A_iB_i = 11$
1	1	1	0	1
0	0	1	0	0

Cuadro 6: Tabla de transición de estados para el diseño recorriendo las palabras de derecha a izquierda

2.2.5. Diagrama de bloque de celda típica

El diagrama de bloque de la celda típica correspondiente al presente diseño posee 3 entradas: la variable de estado presente p, y los bits de A y B en la posición i-ésima A_i y B_i. El diagrama de bloque posee una salida: las variable de próximo estado P.

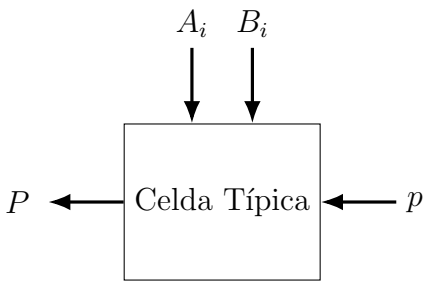


Figura 11: Diagrama de bloque de celda típica para el recorrido de las palabras de entrada de derecha a izquierda

Note la diferencia de puertos entre este bloque de celda típica y la celda típica del diseño anterior. Esto es debido a las distinta cantidad de bits en las asignaciones de estado, y la orientación de las celdas debida al recorrido de las palabras.

2.2.6. Diseño de celda típica

En base a el cuadro 6, se realiza una tabla de verdad con entradas $p\,A_i\,B_i$ correspondiente a la variable de estado P.

Índice	Entradas			Salida
	p	A_i	B_i	P
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Cuadro 7: Tabla de verdad basada en el cuadro 6 con entradas $p\,A_i\,B_i$ y salida P

En base a la tabla de verdad anterior, se obtiene la expansion en mintérminos de P

$$P(p,\ A_i,\ B_i) = \sum m(1,\ 4,\ 5,\ 7) \tag{9}$$

A partir de esta expansión en mintérminos, se mapea cada función en mapas de Karnaugh de 4 variables y se soluciona buscando todos los implicantes primos para obtener la función mínima de P

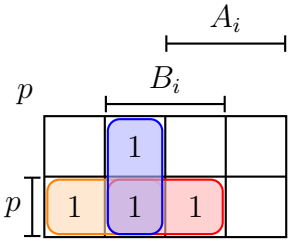


Figura 12: Solución del mapa de Karnaugh correspondiente a la variable de estado P

Al incluir los implicantes esenciales y aquellos implicantes primos que cubren los unos no cubiertos por implicantes esenciales para ambas variables de estado, se obtienen las siguientes funciones mínimas

$$\begin{aligned} P &= p\overline{A_i} + \overline{A_i}B_i + pB_i \\ &= p(\overline{A_i} + B_i) + \overline{A_i}B_i \end{aligned} \tag{10}$$

En base a estas ecuaciones lógicas, el esquemático de la celda típica es el siguiente

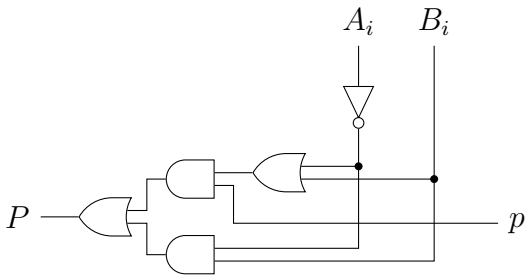


Figura 13: Esquemático de celda típica para el recorrido de palabras de entrada de derecha a izquierda

Note que se necesitan 4 compuertas de dos entradas para implementar la celda típica con la asignación de estados correspondiente al recorrido de las palabras de entrada de derecha a izquierda.

2.2.7. Diseño de la celda inicial

El estado que se escogió como inicial fue el estado a. Debido a que la codificación del estado a es 1, se evalúa la ecuación de P en $p = 1$

$$\begin{aligned} P(1,\, A_i,\, B_i) &= 1 \cdot (\overline{A} + B) + \overline{A}B \\ &= \overline{A}(1 + B) + B \\ &= \overline{A} + B \end{aligned} \tag{11}$$

El esquemático de la celda inicial es el siguiente

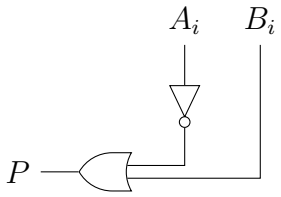


Figura 14: Esquemático de celda inicial para el recorrido de palabras de entrada de derecha a izquierda

Note que se necesita solo una compuerta de dos entradas para implementar la celda inicial con la asignación de estados correspondiente al recorrido de palabras de entrada de derecha a izquierda.

2.2.8. Diseño de celda final

Para la celda final, note que si la red termina en el estado b, $Z = 0$ debido a que esta salida final es activa en bajo. Por lo contario, $Z = 1$. En base a esto, y en el cuadro 6, se construye la siguiente tabla de verdad con entradas $p\, A_i\, B_i$ y salida Z

Índice	Entradas			Salida
	p	A_i	B_i	Z
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Cuadro 8: Tabla de verdad basada en el cuadro 6 con entradas $p\,A_i\,B_i$ y salida Z

Note que esta tabla de verdad es la tabla de verdad para el diseño de la celda típica. Debido a esto, la función mínima de Z será equivalente a la ecuación 11

$$Z(p,\ A_i,\ B_i) = p(\overline{A_i} + B_i) + \overline{A_i}B_i \tag{12}$$

Por tanto, el esquemático de la celda final es el siguiente

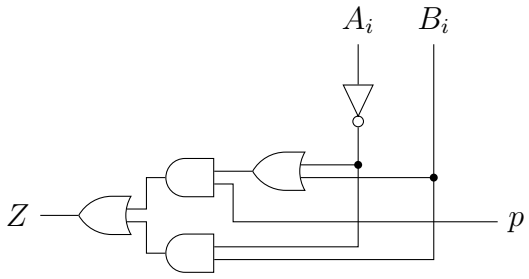


Figura 15: Esquemático de celda final para el recorrido de palabras de entrada de derecha a izquierda

Note que se necesitan 4 compuertas de dos entradas para implementar la celda típica con la asignación de estados correspondiente al recorrido de las palabras de entrada de derecha a izquierda.

Habiendo realizado los diseños para ambos recorridos de las palabras de entrada, se llega a la conclusión que la red iterativa más básica que se puede formar con estos diseños $N = 3$ se puede implementar con 9 compuertas con el recorrido de derecha a izquierdas, mientras que, con el recorrido de palabras de izquierda derecha, el diseño puede ser implementado con 10 compuertas. Recordando el criterio que se indicó en la parte inicial del diseño, la implementación se realizó de la siguiente manera

- El recorrido de palabras de entrada de izquierda a derecha se implementó conductualmente en Verilog debido a que posee el mayor número de compuertas en su diseño
- El recorrido de palabras de entrada de derecha a izquierda se implementó estructuralmente en Verilog debido a que posee el menor número de compuertas en su diseño

3. Implementación

3.1. Conductual

La descripción conductual se enfoca en describir la funcionalidad del código, a diferencia de la descripción estructural que se enfoca en describir la estructura interna y sus componentes y como estos están conectados entre sí [1].

Al realizar todas las asignaciones posibles se eligió el caso que requería menos compuertas, ese caso fue el de derecha a izquierda, el cual se implementó de forma estructural, ya que es detallista con las compuertas y más descriptivo para una función lógica no muy extensa. Para el caso con más compuertas se eligió código conductual que compacta más el posible código, para no generar un código más extenso como sería el estructural con una función lógica extensa.

Se creó un archivo para cada módulo, se creó un módulo para celda inicial, celda típica y final, y red iterativa. También para sus respectivos testbench. Se empieza creando el módulo de celda típica, debido a que de este se deriva la celda inicial y la celda final.

3.1.1. Celda típica

Para el módulo se tiene como entradas, `Ai` y `Bi` que son las variables de estado, y `p` y `q` que son las variables de estado presente, y como salida `P` y `Q` que representan las variables de próximo estado. Se crean los estados a y b y c, justo como en la parte de diseño y se les asigna un valor de 01,10 y 11 respectivamente. Al ser un código conductual, que se enfoca en las entradas y salidas, se utiliza el bloque `always`, que sirve para definir como las variables cambian, en este caso, la lista de sensibilidad se conforma de `Ai`, `Bi`,`p` y `q`, o sea, las entradas de la celda típica. Cada vez que una o algunas o todas de ellas cambian se ejecuta el código de `if else`, el cual describe que valor de próximo estado P debe producir dependiendo de A_i, B_i y del estado actual p, como en la tabla de transición previamente diseñada.

```
/* Definición de módulo para la celda típica de la
   red iterativa analizando las palabras de izquierda
   a derecha */

module celdaTipicaIzqDer (
    input p, q, Ai, Bi,
    output reg P, Q
);

    //asignacion de estados a:01, b:10, c:11
    reg [1:0] a = 2'b01;
    reg [1:0] b = 2'b10;
    reg [1:0] c = 2'b11;
```

```
14
15      always @(p or q or Ai or Bi) begin
16          // always cada vez que p, q, Ai o Bi cambien se ejecuta lo siguiente
            ↪  segun corresponda
17          // Y asi cambiar el proximo estado P, segun el estado actual p, q, y
            ↪  A y B.
18
19          // estado presente a
20          if (p == a[1] && q == a[0]) begin
21              if (Ai == Bi)
22              begin
23                  // si los bits son iguales, próximo estado a
24                  P = a[1];
25                  Q = a[0];
26              end
27              else if (Ai > Bi)
28              begin
29                  // si AB=10, próximo estado b
30                  P = b[1];
31                  Q = b[0];
32              end
33              else if (Ai < Bi)
34              begin
35                  // si AB=01, próximo estado c
36                  P = c[1];
37                  Q = c[0];
38              end
39          end
40          // estado presente b
41          else if (p == b[1] && q == b[0]) begin
42              // el próximo estado siempre es b con estado presente b
43              P = b[1];
44              Q = b[0];
45          end
46
47          // estado presente c
48          else if (p == c[1] && q == c[0]) begin
49              // el próximo estado siempre es c con estado presente c
50              P = c[1];
51              Q = c[0];
52          end
```

```
    end
endmodule
```

3.1.2. Celda inicial

De la materia estudiada en el curso, se sabe que la celda inicial, se deriva de la celda típica, evaluada en su estado inicial. Por esta razón en los pasos de creación de una red iterativa, se diseñaba primero la celda típica y despúes la inicial y final. Se realizó una instanciación por descripción nombrada del módulo de la celda típica con el nombre de `celdaInit`, se evalúa la celda típica en `pinit` y `qinit` con los valores de 0 y 1, que corresponden a los valores de estado inicial que se eligió en el diseño. Para este módulo se decidió nombrar a las entradas con `An_1` y `Bn_1` ya que esta posición de bits serían los primeros en entrar a la red iterativa por la celda inicial.

```
/* Definición de módulo para la celda inicial de la
   red iterativa analizando las palabras de izquierda
   a derecha */

module celdaInicialIzqDer (
    input An_1, Bn_1,
    output Pinit, Qinit
);

    reg pinit = 1'b0; // variable de estado p = 0 del estado inicial
    reg qinit = 1'b1; // variable de estado q = 1 del estado inicial

    /* Evaluar celda típica en el estado inicial a: 01. Se realiza
       una instanciación nombrada */
    celdaTipicaIzqDer celdaInit (.p(pinit), .q(qinit), .Ai(An_1), .Bi(Bn_1),
    ↪  .P(Pinit), .Q(Qinit));
endmodule
```

3.1.3. Celda final

El módulo de celda final no tiene próximos estados P y Q como los anteriores módulos, más bien tiene que tener una sola salida `Zout`, que esté siempre en alto, excepto cuando A_0 es mayor que B_0, o si el estado presente en la celda final es c. La construcción de este módulo fue análoga a la construcción del módulo `celdaTipicaIzqDer`, donde también se definieron los estados a,b, y c, con arrays de dos bits, para ser utilizados adentro de la sentencia `always`. Este tiene como lista de sensibilidad a las entradas, que se conforman por las variables de estado presente `pout` y `qout` y los bits `A0` y `B0`. Los casos condicionales se ejecutan en función de el cuadro 4 que, dependiendo

del estado presente y del valor de los bits de entrada, la salida final Z puede activarse o no.

```
/* Definición de módulo para la celda final de la
   red iterativa analizando las palabras de izquierda
   a derecha */

module celdaFinalIzqDer(
    input pout, qout, A0, B0,
    output reg Zout
);

    //asignacion de estados a:01, b:10, c:11
    reg [1:0] a = 2'b01;
    reg [1:0] b = 2'b10;
    reg [1:0] c = 2'b11;

    always @(pout or qout or A0 or B0) begin
        // always cada vez que pout, qout, A0 o B0 cambien se ejecuta lo
        ↪  siguiente segun corresponda
        // Y asi cambiar la salida Z

        // estado presente a
        if (pout == a[1] && qout == a[0]) begin
            if (A0 == B0)
            begin
                // si los bits son iguales, Zout = 1
                Zout = 1;
            end
            else if (A0 > B0)
            begin
                // si AB=10, Zout = 0
                Zout = 0;
            end
            else if (A0 < B0)
            begin
                // si AB=01, Zout = 1
                Zout = 1;
            end
        end
        // estado presente b
        else if (pout == b[1] && qout == b[0]) begin
```

```
            // el próximo estado siempre es b con estado presente b,
            ↪ entonces Zout = 0
            Zout = 0;
        end

        // estado presente c
        else if (pout == c[1] && qout == c[0]) begin
            // el próximo estado siempre es c con estado presente c,
            ↪ entonces Zout = 1
            Zout = 1;
        end
    end

endmodule
```

3.1.4. Red iterativa

Este módulo `redIterativaIzqDer` tiene como parametro N, el cúal indica cuantos bits tienen las palabras A y B, en el input se definió el tamaño de A y B con `[N - 1:0]` y como salida se tiene `Zout`. Se definió `P` y `Q` como `wire`. Para crear N celdas, en este módulo se utilizó `generate`, y un `for`. Se utilizó `generate` debido a que, se identificó que el próximo estado de las celdas típicas siempre es el estado presente de la siguiente celda. Las únicas excepciones a esta regla son la celda inicial y final, las cuáles no tienen una celda previa o una celda posterior, respectivamente. Se crearon 3 casos para el `for`

- cuando `i` vale 0: en esta posición se tiene la celda final, entonces se instancia posicionalmente `celdaFinalIzqDer`, con los valores de `A`, `B` y `P` en la posición 0.
- cuando `i` es `N-1` : esta sería la posición de la celda inicial. En este caso, se instancia `celdaInicialIzqDer`.
- caso `default`: para este caso se tienen $N-2$ celdas típicas, ya que anteriormente se restó el caso inicial y el caso final. Entonces para estos casos restantes se instancia `CeldaTipicaIzqDer` almacenando A, B en la posición i, con estado anterior `(P[i-1])` y con estado próximo `P(P[i])`.

```
/* Definición de módulo para la red iterativa analizando
   las palabras de izquierda a derecha */

module redIterativaIzqDer
#( parameter N = 3 ) (
    input [N - 1:0] A, B,
```

```
7       output Zout
8   );
9       /* arrays con las variables de próximo estado P y Q para las celdas de
        ↪  la red iterativa */
10      wire [N - 1:0] P;
11      wire [N - 1:0] Q;
12      genvar i; // variable para la elaboración del bloque generate
        ↪  correspondiente a la celda en la posicón i
13
14      /* bloque generate para generar las celdas que conforman la red
        ↪  iterativa por medio de un ciclo for.
15         genera una celda inicial, una celda final, y N - 2 celdas típicas */
16      generate
17          for (i = N - 1; i >= 0 ; i = i - 1)
18          begin
19              case (i)
20                  // generar una instancia de celdaInicial en la posición N -
                    ↪  1 de la red.
21                  N - 1 :
22                  begin
23                      celdaInicialIzqDer celdaInit (.An_1(A[N - 1]), .Bn_1(B[N -
                        ↪  1]), .Pinit(P[N - 1]), .Qinit(Q[N - 1]));
24                  end
25                  // generar una instancia de celdaFinal en la posición 0 de
                    ↪  la red.
26                  0 :
27                  begin
28                      celdaFinalIzqDer celdaFin (.pout(P[1]), .qout(Q[1]),
                        ↪  .A0(A[0]), .B0(B[0]), .Zout(Zout));
29                      /* P[1] y Q[1] son los próximos estados de la última
                        ↪  celda típica,
30                      y por tanto son el estado presente de la celda final*/
31                  end
32                  // generar N - 2 instancias de celdaTipica, entre la celda
                    ↪  inicial y la celda final
33                  default :
34                  begin
35                      celdaTipicaIzqDer celdaTyp (.p(P[i + 1]), .q(Q[i + 1]),
                        ↪  .Ai(A[i]), .Bi(B[i]), .P(P[i]), .Q(Q[i]));
36                      /* de forma similar a la celda final, el estado presente
                        ↪  de una celda tipica es el próximo estado
```

```
                        de la celda anterior. corresponde al estado presente
                        ↪  P[i + 1] y Q[i + 1] */
                end
            endcase
        end
    endgenerate
endmodule
```

3.2. Estructural

3.2.1. Celda típica

Ya que la implementación estructural se realizó al recorrido de palabras de entrada de derecha a izquierda, el módulo de celda típica `celdaTipicaDerIzq` fue implementado a partir de la ecuación lógica 10 y la sentencia `assign`, ya que la lógica combinacional de la celda requiere un continuous assignment. Se utilizó la sintaxis de Verilog reservada para operadores lógicas para obtener esta función.

```
/* Definición de módulo para la celda típica de la
   red iterativa analizando las palabras de derecha a izquierda
*/

module celdaTipicaDerIzq (
    input p, Ai, Bi,
    output P
);

    /* wires para la construcción de las funciones lógicas
    de P */
    wire s0, s1, s2;

    /* función lógica para la variable de próximo estado P */
    assign s0 = ~Ai | Bi;
    assign s1 = ~Ai & Bi;
    assign s2 = p & s0;

    assign P = s2 | s1;
endmodule
```

3.2.2. Celda inicial

De forma similar a la implementación anterior, el módulo de celda inicial `celdaInicialDerIzq` consistió de una instanciación del módulo de celda típica, evaluando su estado presente en el estado inicial a:1

```
module celdaInicialDerIzq (
    input A0, B0,
    output Pinit
);
    reg a = 1'b1; //estado inicial

    celdaTipicaDerIzq celdaInit (.p(a), .Ai(A0), .Bi(B0), .P(Pinit)); /*
    ↪  Evaluar celda típica en el estado inicial a: 1 */

endmodule
```

3.2.3. Celda final

El módulo de celda final `celdaFinalDerIzq` se implementó a partir de la ecuación 12.

```
/* Definición de módulo para la celda final de la
   red iterativa analizando las palabras de derecha a izquierda
*/

module celdaFinalDerIzq (
    input p, An_1, Bn_1,
    output Z
);
    /* wires para la construcción de las funciones lógicas
    de P */
    wire s0, s1, s2;

    /* función lógica para la variable de próximo estado P */
    assign s0 = ~An_1 | Bn_1;
    assign s1 = ~An_1 & Bn_1;
    assign s2 = p & s0;

    assign Z = s2 | s1;

endmodule
```

3.2.4. Red iterativa

El módulo de red iterativa `redIterativaDerIzq` fue construido a partir del módulo `redIterativaIzqDer`, con la diferencia que, en la posición 0 de la red se coloca una instancia de celda inicial, y en la posición $N-1$ una instancia de celda final.

```
/* Definición de módulo para la red iterativa analizando
   las palabras de derecha a izquierda */

module redIterativaDerIzq
#( parameter N = 3 ) (
    input [N - 1:0] A, B,
    output wire Zout
);
    wire [N - 2:0] P;
    genvar i;

    generate
        for (i = 0; i <= N-1; i = i + 1) // instancias de celdaInicial y
        ↪  celdaFinal dentro del for/generate????
        begin
            case (i)
                0 : begin
                    celdaInicialDerIzq celdaInit (.A0(A[0]), .B0(B[0]),
                    ↪  .Pinit(P[0]));
                end
                N-1 : begin
                    celdaFinalDerIzq celdaFin (.An_1(A[N-1]), .Bn_1(B[N-1]),
                    ↪  .p(P[N-2]), .Z(Zout));
                    /* P[1] y Q[1] son los próximos estados de la última
                    ↪  celda típica,
                    y por tanto son el estado presente de la celda final*/
                end
                default : begin
                    celdaTipicaDerIzq celdaTyp (.p(P[i - 1]), .Ai(A[i]),
                    ↪  .Bi(B[i]), .P(P[i]) );
                end
            endcase
        end
    endgenerate
endmodule
```

4. Pruebas

Las pruebas se realizaron por medio de testbenchs en verilog, en donde se pusieron a prueba los módulos descritos en la sección anterior. Esto fue realizado para ambas implementaciones de la red iterativa.

4.1. Izquierda a derecha

Para el recorrido de las palabras de entrada de izquierda a derecha, se realizaron cinco testbenchs

- `celdaInicialIzqDer_tb`
- `celdaTipicaIzqDer_tb`
- `celdaFinalIzqDer_tb`
- `redIterativaIzqDer_tb`
- `redIterativaIzqDer_general_tb`

Cada testbench pone a prueba el módulo correspondiente a su nombre. Se realizó un testbench para cada módulo que se definió en el proyecto con el propósito de verificar el correcto funcionamiento de cada uno de forma individual. Note que se realizaron dos módulos para la red iterativa. Esto fue debido a que uno de estos módulos, `redIterativaIzqDer_tb`, realiza pruebas sobre el módulo `redIterativaIzqDer` con palabras de prueba para A y B predefinidas, mientras que el módulo `redIterativaIzqDer_general_tb` realiza pruebas con casos de esquina, al igual que abarca todas las posibles combinaciones entre A y B con una cantidad de bits N predefinida.

4.1.1. Celda inicial

El testbench `celdaInicialIzqDer_tb` pone a prueba el módulo `celdaInicialIzqDer`. Ya que este siempre tiene el mismo estado presente a ya que es el estado inicial, las pruebas que este testbench realiza corresponde a las posibles combinaciones que pueden haber entre las variables A y B.

```
/* Testbench para la celda inicial correspondiente al
   diseño de la red iterativa analizando las palabras
   de bits A y B de izquierda a derecha */

`timescale 1 ns/10 ps // Definición del timescale

module celdaInicialIzqDer_tb;
    /* Rango de tiempo period correspondiente
       a cada combinación binaria de las palabras
       A y B */
```

```
    localparam period = 20;

    /* Declaración de bits de prueba correspondientes
       a las palabras A y B */
    reg An_1, Bn_1;

    /* variables de próximo estado P y Q para la
       celda Inicial */
    wire [1:0] prox_estado;      // P = prox_estado[1], Q = prox_estado[0]

    /* instanciación de celdaInicial como una descripción
       nombrada  para someterla a pruebas */
    celdaInicialIzqDer DUT (.Pinit(prox_estado[1]), .Qinit(prox_estado[0]),
    ↪  .An_1(An_1), .Bn_1(Bn_1));

    initial
        begin
            /* Archivo para la visualización de los
               resultados de las pruebas en gtkwave */
            $dumpfile("celdaInicial_tb.vcd");

            /* descargar en el archivo del dumpfile
               las variables en el módulo celdaInicial_tb */
            $dumpvars(1, celdaInicial_tb);

            /* Pruebas en base a la tabla de
               transición de estados */
            An_1 = 0; Bn_1 = 0;     //AB = 00
            #period;
            An_1 = 0; Bn_1 = 1;     //AB = 01
            #period;
            An_1 = 1; Bn_1 = 0;     //AB = 10
            #period;
            An_1 = 1; Bn_1 = 1;     //AB = 11
            #period;
            $finish;
        end
endmodule
```

Primero define el `timescale` con el intervalo de tiempo y precisión que se desea visualizar las formas de onda en gtkwave. En el módulo, se define una variable `period` como `localparam`, corres-

pondiente al delay entre cada prueba que se realice dentro del bloque initial. `reg An_1` y `reg Bn_1` corresponderán a los bits de prueba de las palabras A y B, mientras que `wire [1:0] prox_estado` corresponde a un bus de datos que transporta el próximo estado de la celda inicial. Se inicializa el módulo `celdaInicialIzqDer` como una descripción nombrada, alambrando a cada puerto la variable correspondiente en el testbench.

En el bloque initial, se utilizan los system tasks `$dumpfile` y `$dumpvars` para asignar las variables que se exportarán al archivo `.vcd` que posteriormente se abrirán en gtkwave, después de haber compilado el testbenchs junto a los módulos que correspondan. Por último, se realizan todas las combinaciones binarias entre `An_1` y `Bn_1` para poner el módulo a prueba y se finaliza la simulación por medio del system task `$finish`.

4.1.2. Celda tipica

El testbench `celdaTipicaIzqDer_tb` pone a prueba el módulo `celdaTipicaIzqDer`. A diferencia del módulo anterior, el módulo correspondiente a la celda típica puede tener distintos estados presentes.

```
/* Testbench para la celda típica correspondiente al
   diseño de la red iterativa analizando las palabras
   de bits A y B de izquierda a derecha */

`timescale 1 ns/10 ps // Definición del timescale

module celdaTipicaIzqDer_tb;
    /* Rango de tiempo period correspondiente
       a cada combinación binaria de las palabras
       A y B */
    localparam period = 20;

    /* variable utilizada para for de pruebas */
    integer counter;

    /* Declaración de bits de prueba correspondientes
       a las palabras A y B */
    reg Ai, Bi;

    /* variables de próximo estado P y Q
       y de estado presente p y q para la
       celda típica */
    reg [1:0] estado;           // p = estado[1], q = estado[0]
    wire [1:0] prox_estado;     // P = prox_estado[1], Q = prox_estado[0]

```

```
    /* instanciación de celdaTípicaIzqDer como descripción nombrada
       para someterla a pruebas */
    celdaTipicaIzqDer DUT (.p(estado[1]), .q(estado[0]), .P(prox_estado[1]),
    ↪  .Q(prox_estado[0]), .Ai(Ai), .Bi(Bi));

    initial
        begin
            /* Archivo para la visualización de los
               resultados de las pruebas en gtkwave */
            $dumpfile("celdaTipica_tb.vcd");

            /* descargar en el archivo del dumpfile
               las variables en el módulo celdaTipica_tb */
            $dumpvars(1, celdaTipicaIzqDer_tb);

            /* Pruebas en base a la tabla de
               transición de estados */
            for (counter = 0 ; counter < 3 ; counter = counter + 1)
            begin
                case (counter)
                    0 : estado = 2'b01; // *** Estado presente: a:01 ***
                    1 : estado = 2'b10; // *** Estado presente: b:10 ***
                    2 : estado = 2'b11; // *** Estado presente: c:11 ***
                endcase

                Ai = 0; Bi = 0;     //AB = 00
                #period;
                Ai = 0; Bi = 1;     //AB = 01
                #period;
                Ai = 1; Bi = 0;     //AB = 10
                #period
                Ai = 1; Bi = 1;     //AB = 11
                #period;
            end
            $finish;
        end
endmodule
```

La estructura de este testbench es la misma que `celdaInicialIzqDer_tb`, a diferencia que se utiliza una estructura de control `for` en el bloque initial. Para esto, en el módulo primero se define una variable `integer counter` encargada de manejar las iteraciones del `for`. En cada iteración

del `for`, se cambia el estado presente y se abarcan todas las posibles combinaciones de los bits de las palabras de entrada `Ai` y `Bi`.

4.1.3. Celda final

El testbench `celdaFinalIzqDer_tb` pone a prueba el módulo `celdaFinalIzqDer`. Nuevamente, este testbench tiene la misma estructura de los testbench anteriores, con la diferencia de que los bits de las palabras de entrada corresponden a `reg A0, B0`, y se define la salida de la red `wire Zout`.

```
/* Testbench para la celda final correspondiente al
   diseño de la red iterativa analizando las palabras
   de bits A y B de izquierda a derecha */

`timescale 1 ns/10 ps // Definición del timescale

module celdaFinalIzqDer_tb;
    /* Rango de tiempo period correspondiente
       a cada combinación binaria de las palabras
       A y B */
    localparam period = 20;

    /* variable utilizada para el índice del for de pruebas */
    integer counter;

    /* Declaración de bits de prueba correspondientes
       a las palabras A y B */
    reg A0, B0;

    /* variables de estado presente p y q, y salida
       Z para la celda final */
    reg [1:0] estado;      // p = estado[1], q = estado[0]

    /* Salida de la red iterativa Z */
    wire Zout;

    /* instanciación de celdaFinal como una descripción
       nombrada para someterla a pruebas */
    celdaFinalIzqDer DUT (.pout(estado[1]), .qout(estado[0]), .Zout(Zout),
    ↪  .A0(A0), .B0(B0));

    initial
```

```
        begin
            /* Archivo para la visualización de los
               resultados de las pruebas en gtkwave */
            $dumpfile("celdaFinal_tb.vcd");

            /* descargar en el archivo del dumpfile
               las variables en el módulo celdaFinal_tb */
            $dumpvars(1, celdaFinalIzqDer_tb);

            /* Pruebas en base a la tabla de
               transición de estados */
            for (counter = 0 ; counter < 3 ; counter = counter + 1)
            begin
                case (counter)
                    0 : estado = 2'b01; // *** Estado presente: a:01 ***
                    1 : estado = 2'b10; // *** Estado presente: b:10 ***
                    2 : estado = 2'b11; // *** Estado presente: c:11 ***
                endcase
                A0 = 0; B0 = 0;     //AB = 00
                #period;
                A0 = 0; B0 = 1;     //AB = 01
                #period;
                A0 = 1; B0 = 0;     //AB = 10
                #period
                A0 = 1; B0 = 1;     //AB = 11
                #period;
            end
            $finish;
        end
endmodule
```

4.1.4. Red iterativa con casos explícitos

El testbench `redIterativaIzqDer_tb` pone a prueba el módulo `redIterativaIzqDer` con palabras de prueba limitadas. De forma similar a los testbenchs anteriores, se define `localparam 20` para el manejo de los delay en las pruebas. Además de esto, se define `localparam N` para definir la cantidad de bits de las palabras de A y B para las cuales que se deseen realizar las pruebas. En este caso, se definieron casos explícitos de 4 bits. Además de esto, se definió `reg [N - 1:0] A` y `reg [N - 1:0] B` para cada palabra de prueba. Las combinaciones de palabras de prueba que se escogieron fueron las siguientes

- $A = 1111_2$ y $B = 0100_2$

- $A = 0011_2$ y $B = 0100_2$
- $A = 1000_2$ y $B = 0000_2$
- $A = 0000_2$ y $B = 0000_2$

```
/* Testbench para el diseño de la red iterativa
   analizando las palabras de bits A y B de
   izquierda a derecha, con casos explícitos */

`timescale 1 ns/10 ps // Definición de timescale

module redIterativaIzqDer_tb;
    /* tamaño de bits N de las palabras A y B,
       y tiempo period para cada combinación
       binaria entre A y B */
    localparam N = 4, period = 20;

    // Declaración de las palabras A y B de N bits
    reg [N - 1:0] A = 0;
    reg [N - 1:0] B = 0;

    // Salida Z de la red iterativa
    wire Zout;

    /* instanciación del módulo redIterativaIzqDer como
       una descripción nombrada */
    redIterativaIzqDer  #(.N(N)) DUT (.A(A), .B(B), .Zout(Zout));

    initial
        begin
            /* Archivo para la visualización de los
               resultados de las pruebas en gtkwave */
            $dumpfile("red_tb.vcd");

            /* descargar en el archivo del dumpfile
               las variables en el módulo redIterativaIzqDer_tb */
            $dumpvars(1, redIterativaIzqDer_tb);

            /* Pruebas en base a algunas combinaciones binarias
               entre A y B.
               *** Si A >  B => Zout = 0 ***
               *** Si A <= B => Zout = 1 ***
```

```
            */
            A = 4'b1111;    // A = 10
            B = 4'b0100;    // B = 4
            #period;

            A = 4'b0011;    // A = 3
            B = 4'b0100;    // B = 4
            #period;

            A = 4'b1000;    // A = 8
            B = 4'b0000;    // B = 0
            #period;

            A = 4'b0000;    // A = 0
            B = 4'b0000;    // B = 0
            #period;
            $finish;
        end
endmodule
```

4.1.5. Red iterativa con casos generales

El testbench `redIterativaIzqDer_general_tb` pone a prueba el módulo `redIterativaIzqDer` con casos de prueba exhaustivos. Primero, se usó un `for` para realizar todos los posibles casos de esquina, y condicionales `if else` para verificar si se realizó la comparación con éxito por medio del módulo que se está poniendo a prueba. Se utilizó una estructura de control basada en dos `for` anidados para recorrer todas las posibles posibilidades de comparación entre las palabras A y B. Estos utilizan `A_counter` y `B_counter` para contar desde 0 hasta $2^N - 1$. Esto abarca todos los posibles valores de A y B con N bits, de tal manera que el valor de estas palabras se actualiza en cada iteración de su ciclo `for` correspondiente.

```
/* Testbench para el diseño de la red iterativa
   analizando las palabras de bits A y B de
   izquierda a derecha, con casos generales */

`timescale 1 ns /10 ps // Definición de timescale

module redIterativaIzqDer_general_tb;
    /* tamaño de bits N de las palabras A y B,
       y tiempo period para cada combinación
```

```
10          binaria entre A y B */
11      localparam N = 3, period = 20;
12
13      // Declaración de las palabras A y B de N bits
14      reg [N - 1:0] A;
15      reg [N - 1:0] B;
16
17      // índices utilizados para los for de las pruebas
18      integer A_counter;
19      integer B_counter;
20      integer counter;
21
22      // Salida Z de la red iterativa
23      wire Zout;
24
25      /* instanciación del módulo redIterativaIzqDer como
26         una descripción nombrada */
27      redIterativaIzqDer  #(.N(N)) DUT (.A(A), .B(B), .Zout(Zout));
28
29      initial
30          begin
31              /* Archivo para la visualización de los
32                 resultados de las pruebas en gtkwave */
33              $dumpfile("red_general_tb.vcd");
34
35              /* descargar en el archivo del dumpfile
36                 las variables en el módulo redIterativaIzqDer_tb */
37              $dumpvars(1, redIterativaIzqDer_general_tb);
38
39              // Casos de esquina entre A y B
40              for (counter = 0 ; counter < 4 ; counter = counter + 1)
41              begin
42                  case (counter)
43                      0 : begin A = {N{1'b1}}; B ={N{1'b1}};end
44                      1 : begin A = {N{1'b1}}; B ={N{1'b0}};end
45                      2 : begin A = {N{1'b0}}; B ={N{1'b1}};end
46                      3 : begin A = {N{1'b0}}; B ={N{1'b0}};end
47                  endcase
48                  #period;
49                  if (A <= B  && Zout==1)
50                      $display("\n A = %b \n B = %b \n A =< B y Zout = %b.
                        ↪  Prueba exitosa\n", A, B, Zout);
```

```
51              else if (A > B && Zout==0)
52                  $display("\n A = %b \n B = %b \n A > B y Zout = %b. Prueba
                    ↪  exitosa\n", A, B, Zout);
53              else
54                  $display("\n A = %b \n B = %b \n A <= B y Zout = %b.
                    ↪  Prueba fallida\n", A, B, Zout);
55          end
56
57          /* Pruebas en base a las posibles combinaciones binarias
58             entre A y B.
59             *** Si A >  B => Zout = 0 ***
60             *** Si A <= B => Zout = 1 ***
61          */
62          for (A_counter = 0; A_counter < 2**N ; A_counter = A_counter + 1)
63          begin
64              for (B_counter = 0; B_counter < 2**N; B_counter = B_counter +
                ↪  1)
65              begin
66                  // A > B y Zout = 0
67                  if (A > B && Zout == 0)
68                      $display("\n A = %b \n B = %b \n A > B y Zout = %b.
                        ↪  Prueba exitosa\n", A, B, Zout);
69                  // A <= B y Zout = 1
70                  else if (A <= B && Zout == 1)
71                      $display("\n A = %b \n B = %b \n A <= B y Zout = %b.
                        ↪  Prueba exitosa\n", A, B, Zout);
72                  // Prueba fallida
73                  else
74                      $display("\n A = %b \n B = %b \n A <= B y Zout = %b.
                        ↪  Prueba fallida\n", A, B, Zout);
75                  B = B + 1;
76                  /* Al llegar a la última iteración del loop, no realizar
                    ↪  #period
77                     para evitar pruebas innecesarias al visualizar las
                       ↪  formas de onda
78                     en gtkwave
79                  */
80                  if (B_counter != 2**N - 1) #period;
81              end
82              A = A + 1;
83              if (A_counter != 2**N - 1) #period;
```

```
            end
            $finish;
        end
endmodule
```

Para los casos de esquina mostrados en el código anterior, se debe tomar en cuenta que la sintáxis `A = {N{1'b1}}` indica concatenación N veces del valor binario que se coloque entre los corchetes curvos. En este caso, la expresión mostrada indica que se le asigna a la variable `A` un valor binario correspondiente a la concatenación de N unos de un bit.

4.2. Derecha a izquierda

Para el recorrido de las palabras de entrada de derecha a izquierda, se realizaron cinco testbenchs

- `celdaInicialDerIzq_tb`
- `celdaTipicaDerIzq_tb`
- `celdaFinalDerIzq_tb`
- `redIterativaDerIzq_tb`
- `redIterativaDerIzq_general_tb`

La funcionalidad y construcción de estos testbenchs son virtualmente iguales a los testbenchs realizados para el recorrido de las palabras de entrada de izquierda a derecha. Las únicas consideraciones que se deben tomar al comparar los testbenchs análogos entre ambos casos es la nombración de variables (por ejemplo: en el caso de izquierda a derecha, para la celda inicial se utiliza `An_1` y `Bn_1`, mientras que en el otro caso se utiliza `A0` y `B0` debido al orden de recorrido de las palabras), los puertos en las celdas que conforman cada celda, y la asignación de estado. Se debe notar que los dos recorridos poseen cantidades de estado diferente, y por tanto, asignaciones diferentes. Esto provoca que la instanciación de módulos y la codificación de estados no sea consistente entre ambos conjuntos de testbenchs.

4.2.1. Celda inicial

El testbench `celdaInicialDerIzq_tb` pone a prueba el módulo `celdaInicialDerIzq`.

```
/* Testbench para la celda inicial correspondiente al
   diseño de la red iterativa analizando las palabras
   de bits A y B de derecha a izquierda */

`timescale 1 ns/10 ps // Definición del timescale

```

```
module celdaInicialDerIzq_tb;
    /* Rango de tiempo period correspondiente
       a cada combinación binaria de las palabras
       A y B */
    localparam period = 20;

    /* Declaración de bits de prueba correspondientes
       a las palabras A y B */
    reg A0, B0;

    /* variable de próximo estado P para la
       celda Inicial */
    wire P;      // P = prox_estado[1]

    /* instanciación de celdaInicial como una descripción
       nombrada  para someterla a pruebas */
    celdaInicialDerIzq DUT (.Pinit(P),.A0(A0), .B0(B0));

    initial
        begin
            /* Archivo para la visualización de los
               resultados de las pruebas en gtkwave */
            $dumpfile("celdaInicial_tb.vcd");

            /* descargar en el archivo del dumpfile
               las variables en el módulo celdaInicial_tb */
            $dumpvars(1, celdaInicialDerIzq_tb);

            /* Pruebas en base a la tabla de
               transición de estados */
            A0 = 0; B0 = 0;     //AB = 00
            #period;
            A0 = 0; B0 = 1;     //AB = 01
            #period;
            A0 = 1; B0 = 0;     //AB = 10
            #period;
            A0 = 1; B0 = 1;     //AB = 11
            #period;
            $finish;
        end
endmodule
```

4.2.2. Celda tipica

El testbench `celdaTipicaDerIzq_tb` pone a prueba el módulo `celdaTipicaDerIzq`.

```
/* Testbench para la celda típica correspondiente al
   diseño de la red iterativa analizando las palabras
   de bits A y B de derecha a izquierda */

`timescale 1 ns/10 ps // Definición del timescale

module celdaTipicaDerIzq_tb;
    /* Rango de tiempo period correspondiente
       a cada combinación binaria de las palabras
       A y B */
    localparam period = 20;

    /* variable utilizada para for de pruebas */
    integer counter;

    /* Declaración de bits de prueba correspondientes
       a las palabras A y B */
    reg Ai, Bi;

    /* variables de próximo estado P
       y de estado presente p, para la
       celda típica */
    reg p;            // p = estado
    wire P;           // P = prox_estado
    /* instanciación de celdaTípica para someterla
       a pruebas */
    celdaTipicaDerIzq DUT (.p(p), .P(P), .Ai(Ai), .Bi(Bi));

    initial
        begin
            /* Archivo para la visualización de los
               resultados de las pruebas en gtkwave */
            $dumpfile("celdaTipica_tb.vcd");

            /* descargar en el archivo del dumpfile
               las variables en el módulo celdaTipica_tb */
            $dumpvars(1, celdaTipicaDerIzq_tb);

```

```
            /* Pruebas en base a la tabla de
               transición de estados */
            for (counter = 0 ; counter < 2 ; counter = counter + 1)
            begin
                case (counter)
                    0 : p = 1'b1; // *** Estado presente: a:1 ***
                    1 : p = 1'b0; // *** Estado presente: b:0 ***
                endcase
                Ai = 0; Bi = 0;     //AB = 00
                #period;
                Ai = 0; Bi = 1;     //AB = 01
                #period;
                Ai = 1; Bi = 0;     //AB = 10
                #period
                Ai = 1; Bi = 1;     //AB = 11
                #period;
            end
            $finish;
        end
endmodule
```

4.2.3. Celda final

El testbench `celdaFinalDerIzq_tb` pone a prueba el módulo `celdaFinalDerIzq`.

```
/* Testbench para la celda final correspondiente al
   diseño de la red iterativa analizando las palabras
   de bits A y B de derecha a izquierda */

`timescale 1 ns/10 ps // Definición del timescale

module celdaFinalDerIzq_tb;
    /* Rango de tiempo period correspondiente
       a cada combinación binaria de las palabras
       A y B */
    localparam period = 20;

    /* variable utilizada para el índice del for de pruebas */
    integer counter;

```

```
16
17      /* Declaración de bits de prueba correspondientes
18         a las palabras A y B */
19      reg An_1, Bn_1;
20
21      /* variable de estado presente p, y salida
22      Z para la celda final */
23      reg p;      // p = estado presente
24
25      /* Salida de la red iterativa Z */
26      wire Zout;
27
28      /* instanciación de celdaInicial como una descripción
29         nombrada  para someterla a pruebas */
30      celdaFinalDerIzq DUT (.p(p), .Z(Zout), .An_1(An_1), .Bn_1(Bn_1));
31      initial
32          begin
33              /* Archivo para la visualización de los
34                 resultados de las pruebas en gtkwave */
35              $dumpfile("celdaFinal_tb.vcd");
36
37              /* descargar en el archivo del dumpfile
38                 las variables en el módulo celdaInicial_tb */
39              $dumpvars(1, celdaFinalDerIzq_tb);
40
41              /* Pruebas en base a la tabla de
42                 transición de estados */
43              for (counter = 0 ; counter < 2 ; counter = counter + 1)
44              begin
45                  case (counter)
46                      0 : p = 1'b1; // *** Estado presente: a:1 ***
47                      1 : p = 1'b0; // *** Estado presente: b:0 ***
48                  endcase
49                  An_1 = 0; Bn_1 = 0;     //AB = 00
50                  #period;
51                  An_1 = 0; Bn_1 = 1;     //AB = 01
52                  #period;
53                  An_1 = 1; Bn_1 = 0;     //AB = 10
54                  #period
55                  An_1 = 1; Bn_1 = 1;     //AB = 11
56                  #period;
```

```
            end
            $finish;
        end
endmodule
```

4.2.4. Red iterativa con casos explícitos

El testbench `redIterativaDerIzq_tb` pone a prueba el módulo `redIterativaDerIzq` con casos limitados.

```
/* Testbench para el diseño de la red iterativa
   analizando las palabras de bits A y B de
   derecha a izquierda, con casos explícitos */

`timescale 1 ns/10 ps // Definición de timescale

module redIterativaDerIzq_tb;
    /* tamaño de bits N de las palabras A y B,
       y tiempo period para cada combinación
       binaria entre A y B */
    localparam N = 4, period = 20;

    // Declaración de las palabras A y B de N bits
    reg [N - 1:0] A;
    reg [N - 1:0] B;

    // Salida Z de la red iterativa
    wire Zout;

    /* instanciación del módulo redIterativaIzqDer como
       una descripción nombrada */
    redIterativaDerIzq  #(.N(N)) DUT (.A(A), .B(B), .Zout(Zout));

    initial
        begin
            /* Archivo para la visualización de los
               resultados de las pruebas en gtkwave */
            $dumpfile("red_tb.vcd");

            /* descargar en el archivo del dumpfile
```

```
            las variables en el módulo redIterativaIzqDer_tb */
        $dumpvars(1, redIterativaDerIzq_tb);

        /* Pruebas en base a algunas combinaciones binarias
            entre A y B.
            *** Si A >  B => Zout = 0 ***
            *** Si A <= B => Zout = 1 ***
        */
        A = 4'b1010;    // A = 10
        B = 4'b0100;    // B = 4
        #period;

        A = 4'b0011;    // A = 3
        B = 4'b0100;    // B = 4
        #period;

        A = 4'b1000;    // A = 8
        B = 4'b0000;    // B = 0
        #period;

        A = 4'b0000;    // A = 0
        B = 4'b0000;    // B = 0
        #period;
        $finish;
    end
endmodule
```

4.2.5. Red iterativa con casos generales

El testbench `redIterativaDerIzq_tb` pone a prueba el módulo `redIterativaDerIzq` con casos exhaustivos.

```
/* Testbench para el diseño de la red iterativa
   analizando las palabras de bits A y B de
   izquierda a derecha, con casos generales */

`timescale 1 ns /10 ps // Definición de timescale

module redIterativaDerIzq_general_tb;
    /* tamaño de bits N de las palabras A y B,
```

```
9         y tiempo period para cada combinación
10        binaria entre A y B */
11    localparam N = 3, period = 20;
12
13    // Declaración de las palabras A y B de N bits
14    reg [N - 1:0] A;
15    reg [N - 1:0] B;
16
17    // índices utilizados para los for de las pruebas
18    integer A_counter;
19    integer B_counter;
20    integer counter;
21
22    // Salida Z de la red iterativa
23    wire Zout;
24
25    /* instanciación del módulo redIterativaIzqDer como
26       una descripción nombrada */
27    redIterativaDerIzq  #(.N(N)) DUT (.A(A), .B(B), .Zout(Zout));
28
29    initial
30        begin
31            /* Archivo para la visualización de los
32               resultados de las pruebas en gtkwave */
33            $dumpfile("red_general_tb.vcd");
34
35            /* descargar en el archivo del dumpfile
36               las variables en el módulo redIterativaIzqDer_tb */
37            $dumpvars(1, redIterativaIzqDer_general_tb);
38
39            // Casos de esquina entre A y B
40
41            for (counter = 0 ; counter < 4 ; counter = counter + 1)
42            begin
43                case (counter)
44                    0 : begin A = {N{1'b1}}; B ={N{1'b1}};end
45                    1 : begin A = {N{1'b1}}; B ={N{1'b0}};end
46                    2 : begin A = {N{1'b0}}; B ={N{1'b1}};end
47                    3 : begin A = {N{1'b0}}; B ={N{1'b0}};end
48                endcase
49                #period;
```

```
50                  if (A <= B  && Zout==1)
51                      $display("\n A = %b \n B = %b \n A =< B y Zout = %b.
                        ↪  Prueba exitosa\n", A, B, Zout);
52                  else if (A > B && Zout==0)
53                      $display("\n A = %b \n B = %b \n A > B y Zout = %b. Prueba
                        ↪  exitosa\n", A, B, Zout);
54                  else
55                      $display("\n A = %b \n B = %b \n A <= B y Zout = %b.
                        ↪  Prueba fallida\n", A, B, Zout);
56              end
57
58              /* Pruebas en base a las posibles combinaciones binarias
59                 entre A y B.
60                 *** Si A >  B => Zout = 0 ***
61                 *** Si A <= B => Zout = 1 ***
62              */
63              for (A_counter = 0; A_counter < 2**N ; A_counter = A_counter + 1)
64              begin
65                  for (B_counter = 0; B_counter < 2**N; B_counter = B_counter +
                    ↪  1)
66                  begin
67                      // A > B y Zout = 0
68                      if (A > B && Zout == 0)
69                          $display("\n A = %b \n B = %b \n A > B y Zout = %b.
                            ↪  Prueba exitosa\n", A, B, Zout);
70                      // A <= B y Zout = 1
71                      else if (A <= B && Zout == 1)
72                          $display("\n A = %b \n B = %b \n A <= B y Zout = %b.
                            ↪  Prueba exitosa\n", A, B, Zout);
73                      // Prueba fallida
74                      else
75                          $display("\n A = %b \n B = %b \n A <= B y Zout = %b.
                            ↪  Prueba fallida\n", A, B, Zout);
76                      B = B + 1;
77                      /* Al llegar a la última iteración del loop, no realizar
                        ↪  #period
78                         para evitar pruebas innecesarias al visualizar las
                           ↪  formas de onda
79                         en gtkwave
80                      */
81                      if (B_counter != 2**N - 1) #period;
```

```
                end
                A = A + 1;
                if (A_counter != 2**N - 1) #period;
            end
            $finish;
        end
endmodule
```

5. Resultados y análisis

5.1. Izquierda a derecha

5.1.1. Celda típica

Primero, se cambió del directorio `./verilog` al directorio `./verilog/izquierda-derecha` para realizar la compilación de los testbenchs. En el directorio `verilog`, se ejecutó el siguiente comando en la terminal

```
$ cd izquierda-derecha
```

Se compiló, ejecutó el archivo `.vvp`, y se abrió la gráfica en gtkwave del testbench por medio del los siguientes comandos en la terminal

```
$ iverilog -o tipica_out.vvp .\celdaTipica.v .\celdaTipica_tb.v
$ vvp tipica_out.vvp
$ gtkwave celdaTipica_tb.vcd
```

Se obtuvo la siguiente forma de onda en gtkwave

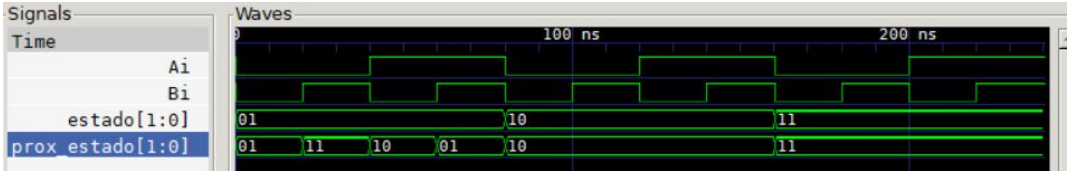


Figura 16: Forma de onda de pruebas realizadas sobre el módulo `celdaTipicaIzqDer`

Note que el módulo funciona correctamente, ya que:

- Con estado presente b : 10 o c : 11, independientemente de los valores de `Ai` y `Bi`, el próximo estado es b : 10 o c : 11, respectivamente.
- Con estado presente a : 01, si `Ai`=`Bi`, el próximo estado es a : 01

- Con estado presente a : 01, si `Ai`=1 y `Bi`=0, el próximo estado es b : 10. Si `Ai`=0 y `Bi`=1, el próximo estado es c : 11.

5.1.2. Celda inicial

Se compiló, ejecutó el archivo `.vvp`, y se abrió la gráfica en gtkwave del testbench correspondiente a la celda inicial por medio del los siguientes comandos en la terminal

```
$ iverilog -o inicial_out.vvp .\celdaTipica.v .\celdaInicial.v
↪  .\celdaInicial_tb.v
$ vvp inicial_out.vvp
$ gtkwave celdaInicial_tb.vcd
```

Se obtuvo la siguiente forma de onda en gtkwave

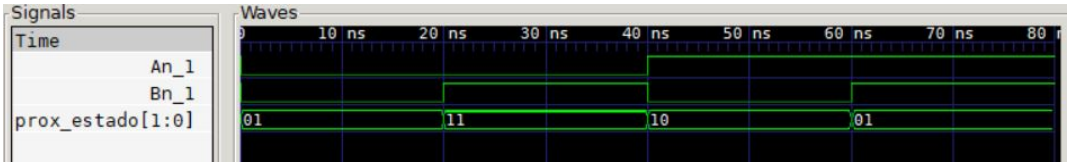


Figura 17: Forma de onda de pruebas realizadas sobre el módulo `celdaInicialIzqDer`

Note que el módulo funciona correctamente, ya que

- Si `Ai`=`Bi`, el próximo estado es a : 01
- Si `Ai`=1 y `Bi`=0, el próximo estado es b : 10. Si `Ai`=0 y `Bi`=1, el próximo estado es c : 11.

Esto es debido a que la celda inicial siempre tiene estado presente a : 01.

5.1.3. Celda final

Se compiló, ejecutó el archivo `.vvp`, y se abrió la gráfica en gtkwave del testbench correspondiente a la celda final por medio del los siguientes comandos en la terminal

```
$ iverilog -o final_out.vvp .\celdaFinal.v .\celdaFinal_tb.v
$ vvp final_out.vvp
$ gtkwave celdaFinal_tb.vcd
```

Se obtuvo la siguiente forma de onda en gtkwave

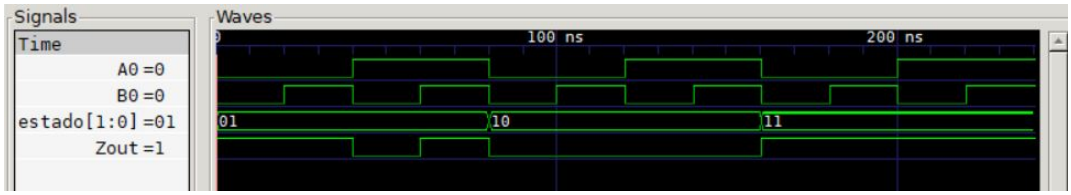


Figura 18: Forma de onda de pruebas realizadas sobre el módulo `celdaFinalIzqDer`

Note que el módulo funciona correctamente, ya que:

- Con estado presente b : 10, independientemente de los valores de `Ai` y `Bi`, la salida $Z = 0$
- Con estado presente c : 11, independientemente de los valores de `Ai` y `Bi`, la salida $Z = 1$
- Con estado presente a : 01, el único caso en donde $Z = 0$ es cuando `Ai`=1 y `Bi`=0

5.1.4. Red iterativa con casos explícitos

Se compiló, ejecutó el archivo `.vvp`, y se abrió la gráfica en gtkwave del testbench correspondiente a la red iterativa con casos explícitos por medio del los siguientes comandos en la terminal

```
$ iverilog -o red_out.vvp .\celdaInicial.v .\celdaTipica.v .\celdaFinal.v
↪  .\red.v .\red_tb_.v
$ vvp red_out.vvp
$ gtkwave red_tb.vcd
```

Se obtuvo la siguiente forma de onda en gtkwave Note que el módulo funciona correctamente, ya

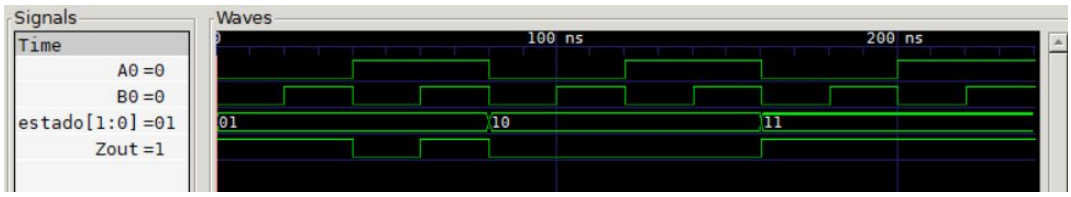


Figura 19: Forma de onda de pruebas realizadas sobre el módulo `redIterativaIzqDer` con casos explícitos

que para las comparaciones de las palabras de prueba A y B respectivamente

- $15_{10} > 4_{10}$, entonces $Z = 0$
- $3_{10} < 4_{10}$, entonces $Z = 1$
- $8_{10} > 0_{10}$, entonces $Z = 0$
- $0_{10} = 0_{10}$, entonces $Z = 1$

5.1.5. Red iterativa con casos generales

Para este último caso, se mostrará una gráfica para el caso con $N = 3$. Esto es debido a que, para valores más grandes de N, el tiempo de ejecución es muy grande y las formas de onda en gtkwave se vuelven muy saturadas de datos, por lo que es poco eficiente la visualización de datos através de esta herramienta. Se compiló, ejecutó el archivo `.vvp`, y se abrió la gráfica en gtkwave del testbench correspondiente a la red iterativa con casos generales por medio del los siguientes comandos en la terminal

```
$ iverilog -o red_out.vvp .\celdaInicial.v .\celdaTipica.v .\celdaFinal.v
↪  .\red.v .\red_general_tb.v
$ vvp red_out.vvp
$ gtkwave red_general_tb.vcd
```

Se obtuvo la siguiente forma de onda en gtkwave

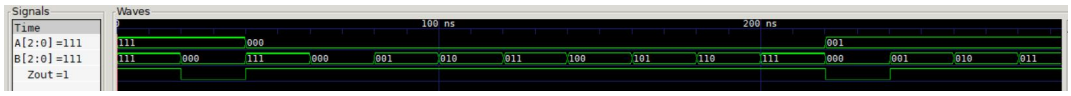


Figura 20: Forma de onda de pruebas realizadas sobre el módulo `redIterativaIzqDer` con casos generales

Note primero que se obtiene el valor de Z deseado para los casos de esquina, y para los casos generales que proceden a estos, Z únicamente se pone en bajo cuando A supera en magnitud a B.

5.2. Derecha a izquierda

5.2.1. Celda típica

Primero, se cambió del directorio `./verilog` al directorio `./verilog/derecha-izquierda` para realizar la compilación de los testbenchs. En el directorio `verilog`, se ejecutó el siguiente comando en la terminal

```
$ cd derecha-izquierda
```

Debido a que los archivos en donde se encuentra cada módulo poseen el mismo nombre que poseen los archivos en donde se encuentran los módulos para el recorrido de las palabras de entrada de izquierda a derecha pero en un directorio diferente, no se listarán de nuevo los comandos en terminal, ya que los comandos en la terminal ya mencionados pueden ser utilizados para compilar, ejecutar, y abrir las formas de onda en gtkwave para los módulos correspondientes al recorrido de las palabras de derecha a izquierda.

Para la celda típica, se obtuvo la siguiente forma de onda en gtkwave

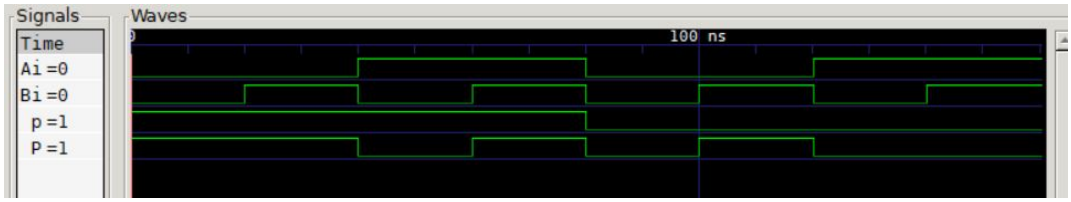


Figura 21: Forma de onda de pruebas realizadas sobre el módulo `celdaTipicaDerIzq`

Note que el módulo funciona correctamente, ya que:

- Con estado presente $a : 1$, el próximo estado es $b : 0$ únicamente si `Ai`=1 y `Bi`=0
- Con estado presente $b : 0$, el próximo estado es $a : 1$ únicamente si `Ai`=0 y `Bi`=1

5.2.2. Celda inicial

Se obtuvo la siguiente forma de onda en gtkwave

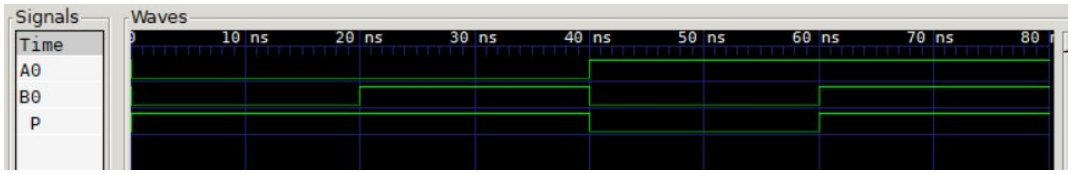


Figura 22: Forma de onda de pruebas realizadas sobre el módulo `celdaInicialDerIzq`

Note que el módulo funciona correctamente, ya que el próximo estado es $b : 0$ únicamente si `Ai`=1 y `Bi`=0. Esto es debido a que la celda inicial siempre tiene estado presente $a : 1$.

5.2.3. Celda final

Se obtuvo la siguiente forma de onda en gtkwave

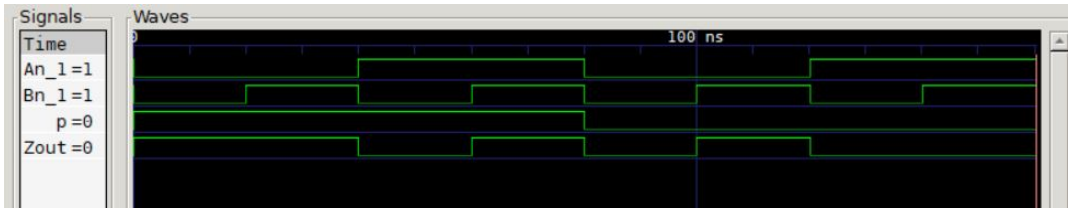


Figura 23: Forma de onda de pruebas realizadas sobre el módulo `celdaFinalDerIzq`

Note que el módulo funciona correctamente, ya que esta forma de onda es equivalente a la forma de onda para la celda típica mostrada anteriormente. Esto es debido a que Z posee la misma función lógica que la variable de próximo estado P

5.2.4. Red iterativa con casos explícitos

Se obtuvo la siguiente forma de onda en gtkwave

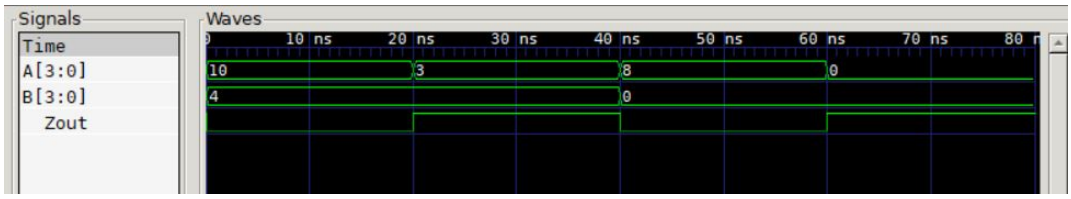


Figura 24: Forma de onda de pruebas realizadas sobre el módulo `redIterativaDerIzq` con casos explícitos

Note que el módulo funciona correctamente, ya que para las comparaciones de las palabras de prueba A y B respectivamente

- $10_{10} > 4_{10}$, entonces $Z = 0$
- $3_{10} < 4_{10}$, entonces $Z = 1$
- $8_{10} > 0_{10}$, entonces $Z = 0$
- $0_{10} = 0_{10}$, entonces $Z = 1$

5.2.5. Red iterativa con casos generales

Se obtuvo la siguiente forma de onda en gtkwave

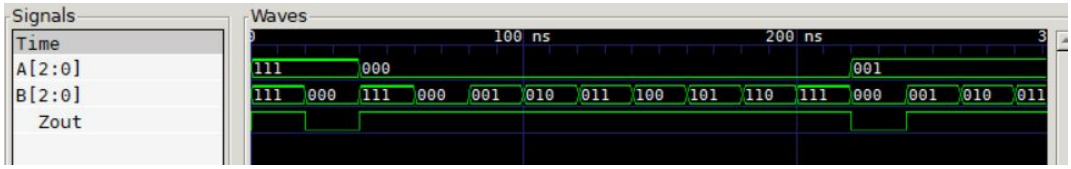


Figura 25: Forma de onda de pruebas realizadas sobre el módulo `redIterativaDerIzq` con casos generales

Note primero que se obtiene el valor de Z deseado para los casos de esquina, y para los casos generales que proceden a estos, Z únicamente se pone en bajo cuando A supera en magnitud a B.

Referencias

[1] Geovanny Delgado Cascante. *Lenguaje de descripción de Hardware: VERILOG*. Universidad de Costa Rica, Escuela de Ingeniería Eléctrica.

[2] Geovanny Delgado Cascante. *Redes Iterativas I Parte*. Universidad de Costa Rica, Escuela de Ingeniería Eléctrica.