Memoria

Sistemas Electrónicos para la Automatización

Docente: *Jorge Romero Sánchez*

Práctica: Señales Resueltas

***Izan Amador Bustos**: izan.amador@uma.es*

Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Contenido

Lista de figuras

Lista de códigos

1. Especificaciones

- Explicar en detalle en un estudio previo, cómo funciona el código suministrado (para la memoria).
- Generar un proyecto en Vivado para ejecutar la simulación.
- ¿Qué significan las líneas: “Esta línea no compilará”?.
- ¿En qué tipo de circuitos se puede utilizar la lógica resuelta con múltiples drivers?
- Elaborar un informe que ilustre el procedimiento de diseño, simulación, respondiendo a las 2 cuestiones. Podéis utilizar capturas de pantalla de los cronogramas de simulación.

2. Estudio teórico

2.1. Sobre la asignación concurrente de señales en VHDL

Existen dos tipos de asignación de señales en VHDL:

- **Concurrente**: ocurre a nivel de arquitectura y no depende del orden de escritura.
- **Secuencial**: ocurre a nivel de proceso y depende del orden de escritura.

Si hay una asignación múltiple a nivel de arquitectura, **se producen un número de drivers equivalente al número de procesos fuentes** y la asignación se considera incorrecta si la señal no es del tipo resuelta.

Además, la decisión de cómo se actualiza el driver no depende del orden, si no de la función de resolución de la lógica resuelta en cuestión.

2.2. Lógica Resuelta y no resuelta

Un ejemplo sencillo para entender el concepto de función de resolución es establecer un paralelismo con un circuito electrónico:

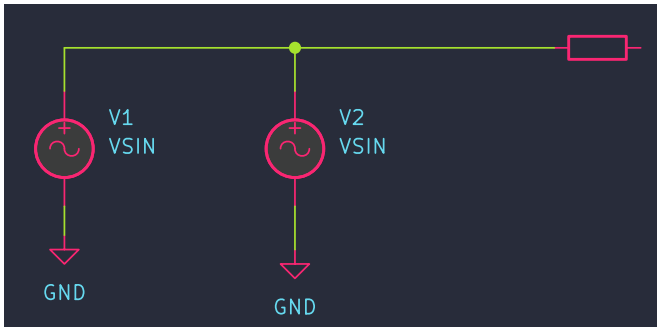

Figura 1: Circuito explicativo para el concepto de función de resolución

Cuando dos señales que provienen de nodos diferentes confluyen en el mismo nodo, existe una función de resolución que decide la asignación de la señal de salida, que se conoce como **Leyes de Kirchoff**.

De manera similar al ejemplo de la Figura 1, es necesario una lógica que permita la resolución de la asignación concurrente de señales en VHDL. Esta función de resolución se implementa mediante una tabla que expresa las diferentes salidas en función de las relaciones de las posibles confluencias de asignaciones en la entrada.

Para la librería std_logic esta es la tabla de la función de resolución:

	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'-'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Figura 2: Tabla de la función de resolución de la librería std_logic

En la que se codifica la salida dependiendo de las dos señales de entrada que haya que resolver.

Requisitos
Para que una función de resolución sea implementable cabe destacar que es fundamental que se cumplan tres propiedades:

- **Conmutatividad**: $f(a,b) = f(b,a)$
- **Asociatividad**: $f[(a,b),c] = f[a,f(b,c)]$
- **Elemento neutro** $f(e,a) = f(a,e) = a$

Por lo tanto, si no se cumplen las tres propiedades, la tabla se no se consideraría de resolución, dando lugar al concepto de **lógica no resuelta**.

Es imprescindible tener en cuenta las propiedades anteriores a la hora de diseñar un tipo de lógica resuelta.

3. Diseño

3.1. Paquete Mis_Tipos

```
1  Package Mis_Tipos is
2  -- En este paquete defino mis tipos de datos constantes
3  -- y señales globales, subprogramas, etc.
4
5    type Mi_Logica is ('U', '0', '1', 'X');  -- La lÃ³gica base no resuelta.
6    type Mi_Logica_Vector is array(Natural range <>) of Mi_Logica;
7
8    -- La función de resolución cumple la Conmutativa, Asociativa y Elemento Neutro.
9    function Resuelve(x : Mi_Logica_Vector)
10     return Mi_Logica;
11
12   subtype Mi_Logica_Resuelta is Resuelve Mi_Logica;  -- El tipo resuelto como
13
14 end Mis_Tipos;  -- subtipo del tipo base.
15
```

```
16  package body Mis_Tipos is   -- Aquí defino el cuerpo de los subprogramas.
17
18    function Resuelve(x : Mi_Logica_Vector) return Mi_Logica is
19      type Tabla is array(Mi_Logica range 'U' to 'X', Mi_Logica range 'U' to 'X')
20
21        of Mi_Logica;
22                                  -- 'U'  '0'  '1'  'X'
23                                     ----------------------
24      constant Operacion : Tabla := (('U', '0', '1', 'X'),   --|'U'
25                                     ('0', '0', 'X', 'X'),   --|'0'
26                                     ('1', 'X', '1', 'X'),   --|'1'
27                                     ('X', 'X', 'X', 'X'));  --|'X'
28      variable Parcial : Mi_Logica := 'U';  -- El valor inicial es importante:
29
30    begin  -- Siempre es el Elemento Neutro.
31
32      for i in x'Range loop
33        Parcial := Operacion(Parcial, x(i));  -- RecursiÃ³n con propiedades:
34  -- Asociativa:f(f(x,y),z)=f(x,f(y,z))
35      end loop;  -- Conmutativa: f(x,y)=f(y,x)
36
37      return Parcial;
38
39    end Resuelve;
40
41    end Mis_Tipos;
42
```

Código 1: Mis_tipos

En el paquete *Mis_Tipos*, se pretende definir una lógica base no resuelta, llamada *Mi_Lógica*, y un subtipo llamado *Mi_Logica_Resuelta* que sí se encuentre resuelto, mediante una función de resolucion llamada *Resuelve*.

Para comprobar si la función es efectivamente de resolución, se ha de comprobar que **la tabla cumple las propiedades de asociatividad, conmutatividad, y elemento neutro**, como es el caso.

Cabe destacar que existen dos valores *X* y *U*, en la definición del tipo *Mi_Logica*, que no permitirían la síntesis del sistema.

Al final de la función se incluye un bucle for que sirve para indexar el valor de la solución de cada combinación de la tabla *Operacion*, obteniendo por ejemplo:

- U:= Operacion(U,U)
- 0 := Operacion(U,0)
- X := Operacion(0,X)
- X := Operacion(X,X)

Es importante establecer un valor inicial ya que se trata de una indexación recursiva y debe de establecerse un valor predeterminado.

3.2. Test Bench

```
1  library IEEE;
2  use work.Mis_Tipos.all;  -- La biblioteca lógica WORK es implícita en cada diseño.
3
4  entity Mi_Logica_TB is   -- Por tanto todos mis packages pertenecen a ella.
5
6  end Mi_Logica_TB;
7
```

```
8  architecture Comportamiento of Mi_Logica_TB is
9
10   signal s_sr     : Mi_Logica          := 'U';  -- Señales sin resolver (sr).
11   signal s_r      : Mi_Logica_Resuelta := 'U';  -- Señales resueltas (r).
12
13 begin
14
15   Bus_End_Point_1 : process  -- Fuente 1 para la señal resuelta (genera un driver).
16   begin
17    -- ¡Esta línea no compilará!
18    s_sr <= '0' after 5 ns, '1' after 10 ns, 'X' after 12 ns;
19     s_r  <= '0' after 5 ns, '1' after 10 ns, 'X' after 20 ns;
20     wait;
21   end process Bus_End_Point_1;
22
23   Bus_End_Point_2 : process  --Fuente 2 para la señal resuelta (otro driver).
24   begin
25     -- ¡Esta línea no compilará¡!
26    s_sr <= '0' after 5 ns, '1' after 12 ns, 'X' after 15 ns;
27     s_r  <= '0' after 5 ns, '1' after 12 ns, 'X' after 15 ns;
28     wait;
29   end process Bus_End_Point_2;
30
31 end Comportamiento;
```

Código 2: Test bench original

El archivo del banco de pruebas pretende comparar ambas lógicas.

De esta manera, se define una señal sin resolver y una señal resuelta para comparar su comportamiento mediante la asignación desde distintas fuentes. Ambas fuentes generan drivers que deben de ser resueltos mediante su correspondiente función de resolución.

4. Simulación abortada

4.1. Test Bench

```
1  architecture Comportamiento of Mi_Logica_TB is
2
3    signal s_sr     : Mi_Logica          := 'U';  -- Señales sin resolver (sr).
4    signal s_r      : Mi_Logica_Resuelta := 'U';  -- Señales resueltas (r).
5
6  begin
7
8    Bus_End_Point_1 : process  -- Fuente 1 para la señal resuelta (genera un driver).
9    begin
10    -- ¡Esta línea no compilará!
11    s_sr <= '0' after 5 ns, '1' after 10 ns, 'X' after 12 ns;
12     s_r  <= '0' after 5 ns, '1' after 10 ns, 'X' after 20 ns;
13     wait;
14   end process Bus_End_Point_1;
15
16   Bus_End_Point_2 : process  --Fuente 2 para la señal resuelta (otro driver).
17   begin
18     -- ¡Esta línea no compilará¡!
19    s_sr <= '0' after 5 ns, '1' after 12 ns, 'X' after 15 ns;
20     s_r  <= '0' after 5 ns, '1' after 12 ns, 'X' after 15 ns;
21     wait;
22   end process Bus_End_Point_2;
23
24 end Comportamiento;
```

Código 3: Fragmento del test bench abortado

En el primer caso, la simulación es abortada ya que se produce una asignación de múltiples drivers mediante dos fuentes en un tipo de lógica no resuelta, que el compilador no sabe cómo solucionar.

4.2. Error

```
ERROR: [XSIM 43-3249] File C:/Users/izana/Documents/GitHub/SEA/practica1/practica1.srcs/sim_1/new/Mi_Logica_TB.vhd, line 9. Unresolved signal "s_sr" is multiply driven.
INFO: [USF-XSim-69] 'elaborate' step finished in '1' seconds
INFO: [USF-XSim-99] Step results log file:'C:/Users/izana/Documents/GitHub/SEA/practica1/practica1.sim/sim_1/behav/xsim/elaborate.log'
ERROR: [USF-XSim-62] 'elaborate' step failed with error(s). Please check the Tcl console output or 'C:/Users/izana/Documents/GitHub/SEA/practica1/practica1.sim/sim_1/behav/xsim/elaborate.log' file for mor
ERROR: [Vivado 12-4473] Detected error while running simulation. Please correct the issue and retry this operation.
ERROR: [Common 17-39] 'launch_simulation' failed due to earlier errors.
```

Figura 3: Error de simulación

Por lo tanto en la Figura 3, se aborta la simulación, identificando el error claramente por consola.

5. Simulación sin errores

5.1. Test Bench

```
1  architecture Comportamiento of Mi_Logica_TB is
2
3    signal s_sr     : Mi_Logica          := 'U';  -- Señales sin resolver (sr).
4    signal s_r      : Mi_Logica_Resuelta := 'U';  -- Señales resueltas (r).
5
6  begin
7
8    Bus_End_Point_1 : process  -- Fuente 1 para la señal resuelta (genera un driver).
9    begin
10    -- ¡Esta línea no compilará!
11    s_sr <= '0' after 5 ns, '1' after 10 ns, 'X' after 12 ns;
12     s_r  <= '0' after 5 ns, '1' after 10 ns, 'X' after 20 ns;
13     wait;
14   end process Bus_End_Point_1;
15
16   Bus_End_Point_2 : process  --Fuente 2 para la señal resuelta (otro driver).
17   begin
18     -- ¡Esta línea no compilará¡!
19   -- s_sr <= '0' after 5 ns, '1' after 12 ns, 'X' after 15 ns;
20     s_r  <= '0' after 5 ns, '1' after 12 ns, 'X' after 15 ns;
21     wait;
22   end process Bus_End_Point_2;
23
24 end Comportamiento;
```

Código 4: Fragmento del test bench simulado

Para resolver el error de compilación, **es necesario eliminar la asignación múltiple mediante distintas fuentes en la lógica no resuelta**. Para ello, el único cambio necesario es comentar una de las dos líneas de la asignación de la lógica no resuelta.

De esta manera, la señal con la lógica resuelta no presenta ningún problema con la asignación de múltiples drivers y se elimina la asignación múltiple en la no resuelta, resultando en una simulación exitosa.

5.2. Cronograma de simulación

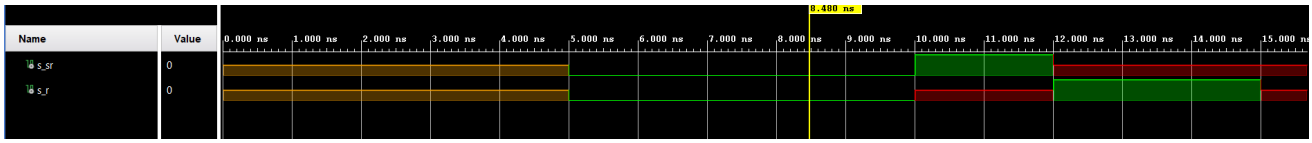

Figura 4: Cronograma de simulación

En el cronograma de la simulación de la Figura 4, se puede apreciar cómo se realizan las asignaciones pertinentes definidas en el archivo del banco de pruebas en los tiempos que corresponden.

En primer lugar ambas señales se encuentran al valor lógico indefinido *U*, para después alterar sus valores entre 0,1 y *X*, en función de los retardos asignados.

6. Primera cuestión

¿Qué significan las líneas: "Esta línea no compilará"?

Significa que el compilador abortará la simulación al encontrarse con una asignación concurrente de múltiples drivers desde distintas fuentes con un tipo de lógica no resuelta que no presenta una función de resolución que permita solucionar la concurrencia entre ambas asignaciones.

7. Segunda cuestión

¿En qué tipo de circuitos se puede utilizar la lógica resuelta con múltiples drivers?

Existen circuitos en los que la lógica resuelta con múltiples drivers puede ser necesaria, por ejemplo en el caso de las señales que van a ser manipuladas por diferentes procesos para lectura o escritura.

En concreto, en una microcomputadora, el bus de comunicaciones entre los distintos elementos de la misma, debe de tener múltiples drivers que deben de ser resueltos mediante una lógica, para realizar la correcta interconexión entre la memoria, CPU, periféricos etc.

Principalmente se pueden utilizar en comunicaciones.