

# PRÁCTICA 1

## Estudio previo

El código suministrado tiene dos partes. La fuente Mis tipos y el test bench. En el primero se definen los tipos que vamos a usar, señales y subprogramas asociados. Se ha creado un tipo vector formado por tipos Mi\_Logica. Se ha definido la función para resolver la lógica (function resuelve) y el tipo resuelto como subtipo del tipo base.

En el cuerpo del paquete es donde podemos ver lo que hace la función de resolución. Se ha definido el tipo tabla como una matriz del rango completo de Mi\_Logica tanto en filas como en columnas. Ahora definimos la constante tabla de donde se va a sacar la operación. Podría tener cualquier otra distribución, siempre y cuando se respete la propiedad conmutativa, distributiva y elemento neutro. Se inicializa la variable parcial de tipo Mi\_Logica al elemento neutro 'U'.

Ahora empezamos la recursión que nos resuelve la lógica, con un bucle for, usando la variable parcial y recorriendo el array de entrada. Una vez terminado el bucle, se nos devuelve el valor de parcial. El array de entrada tendrá tantas posiciones como drivers haya para la misma señal de tipo resuelto.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

Package Mis_Tipos is

    type Mi_Logica is ('U','0','1','X');
    type Mi_Logica_Vector is array( Natural range <> ) of Mi_Logica;

    function Resuelve( x : Mi_Logica_Vector ) return Mi_Logica;
    subtype Mi_Logica_Resuelta is Resuelve Mi_Logica;
end Mis_Tipos;

package body Mis_Tipos is
    function Resuelve( x : Mi_Logica_Vector ) return Mi_Logica is
        type Tabla is array(Mi_Logica range 'U' to 'X',Mi_Logica range 'U' to 'X')
        of Mi_Logica;

        constant Operacion : Tabla := (('U','0','1','X'),
                                         ('0','0','X','X'),
                                         ('1','X','1','X'),
                                         ('X','X','X','X'));

        variable Parcial : Mi_Logica := 'U';
    begin
        for i in x'Range loop
            Parcial := Operacion(Parcial,x(i));
        end loop;
        return Parcial;
    end Resuelve;
end Mis_Tipos;
```

Una vez explicado el paquete, podemos ver el test\_bench. En el test bench se tiene que incluir el paquete que hemos creado anteriormente para poder usar los tipos y las funciones que se definen. Se han creado dos señales, una de tipo Mi\_Logica y otra de tipo resuelta.

Se definen dos procesos, y ambos hacen transacciones en las dos señales, lo que implica que cada señal tiene dos drivers. Esto no supone un problema para la señal de tipo resuelto ya que tiene una función de resolución. Sin embargo, sí que supone un problema para la señal de tipo Mi\_Logica.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.Mis_Tipos.all;

entity TB is
-- Port ( );
end TB;

architecture Behavioral of TB is

    signal s_sr : Mi_Logica      := 'U';
    signal s_r  : Mi_Logica_Resuelta := 'U';
begin

    Bus_End_Point_1: process
    begin
        s_sr <= '0' after 5ns, '1' after 10ns, 'X' after 12 ns;
        s_r  <= '0' after 5ns, '1' after 10ns, 'X' after 20 ns;
        wait;
    end process Bus_End_Point_1;


    Bus_End_Point_2: process
    begin
        s_sr <= '0' after 5ns, '1' after 12ns, 'X' after 15 ns;
        s_r  <= '0' after 5ns, '1' after 12ns, 'X' after 15 ns;
        wait;
    end process Bus_End_Point_2;

end Behavioral;
```

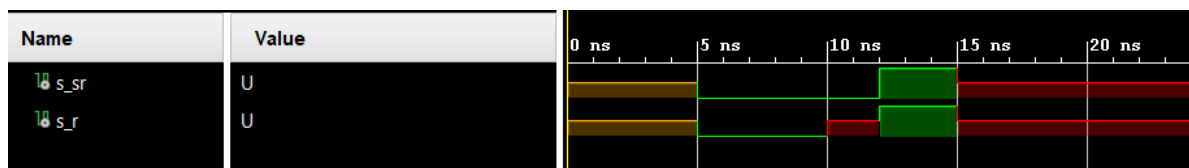
## Esta línea no compilará

Este mensaje significa que, efectivamente, al intentar runear la simulación, la línea generará un fallo. Esto se debe a que la señal no está resuelta y tiene dos drivers, por lo que no sabe a qué driver tiene que “hacer caso”. Esto no pasa con la otra señal, porque en el paquete se ha definido que el tipo `Mi_Logica_Resuelta` tiene una operación de resolución de variable.

Este es el error que saca vivado ante esta situación.

 [XSIM 43-3249] File C:/Users/usuario/Desktop/javi/eii/Electronica/practica1/practica1.srscs/sources\_1/new/TB.vhd, line 11. Unresolved signal "s\_sr" is multiply driven.

Sin embargo, si comentamos en uno de los procesos la instrucción que no nos dejaba compilar, ahora la simulación sí se llevará a cabo. Obtenemos el siguiente resultado.



La señal no resuelta (`s_sr`) es fácil de analizar. Como ahora solo hay un driver, en mi caso, el del segundo proceso, tenemos la inicialización a ‘U’, ‘0’ a los 5 ns, ‘1’ a los 12 ns y ‘X’ a los 15 ns, como se ve en la simulación.

El caso de la señal resuelta es más complicado ya que tenemos que ver los dos drivers que actúan y meterlos en la tabla de resolución. Al principio en ambos drivers tenemos ‘U’ por lo que la salida es ‘U’. Es lo mismo que ocurre entre 5 y 10 ns. Ambos son ‘0’, luego la señal vale ‘0’. Sin embargo en 10 ns, uno de los drivers pasa a ‘1’. En este caso la salida es ‘X’ según la operación de resolución. A los 12 ns el otro driver pasa a tener un ‘1’. Como ambos son ‘1’, según la tabla la señal vale ‘1’. Cuando uno de los drivers pasa a tener ‘X’ en 15 ns la señal pasa a valer ‘X’. Esto ya no cambia porque el último cambio que se hace es del otro driver que también pasa a ‘X’.

## Circuitos en los que es útil

La utilidad que yo le veo es para hacer operaciones utilizando una única señal. Con la lógica resuelta se pueden implementar operaciones tanto aritméticas como lógicas. Solo es necesaria una señal, dos procesos y una tabla de resolución que refleje la operación que se quiere implementar.