P3

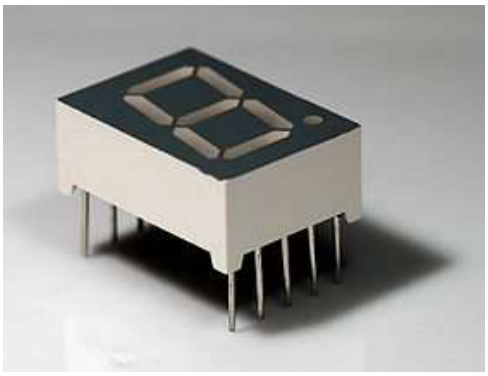
Diseño e implementación de un decodificador de 7 segmentos

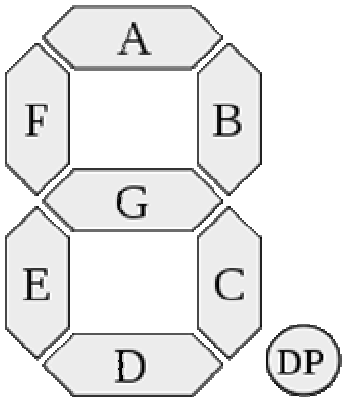
1.- Introducción y objetivos

En esta práctica se van a implementar un decodificador de 7 segmentos. Además de la descripción, la simulación y la síntesis (realizadas en la práctica P2 con otro diseño), se completarán todas las fases del diseño digital en FPGA y se observará el resultado en una placa de desarrollo.

2.- El decodificador de 7 segmentos

El decodificador de 7 segmentos es un circuito digital que convierte números binarios (en el caso que nos ocupa los correspondientes a los valores decimales del 0 al 9) para adaptarlos a un display de 7 segmentos como el de la figura.





Así, la salida del decodificador debe ser capaz de mostrar los siguientes dígitos decimales:

Por conveniencia, los segmentos A…G se nombrarán como led(0)… led(6).

Como tenemos 10 configuraciones distintas, vamos a necesitar 4 entradas, pero de ellas no necesitamos todos los valores (16). Los 6 valores restantes se emplean para simplificar las ecuaciones lógicas.

El decodificador de 7 segmentos responde a la tabla de la verdad que se muestra a continuación:

bcd(3)	bcd(2)	bcd(1)	bcd(0)	led(6) g	led(5) f	led(4) e	led(3) d	led(2) c	led(1) b	led(0) a
0	0	0	0	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	0	0	1
0	0	1	0	0	1	0	0	1	0	0
0	0	1	1	0	1	1	0	0	0	0
0	1	0	0	0	0	1	1	0	0	1
0	1	0	1	0	0	1	0	0	1	0
0	1	1	0	0	0	0	0	0	1	0
0	1	1	1	1	1	1	1	0	0	0
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X

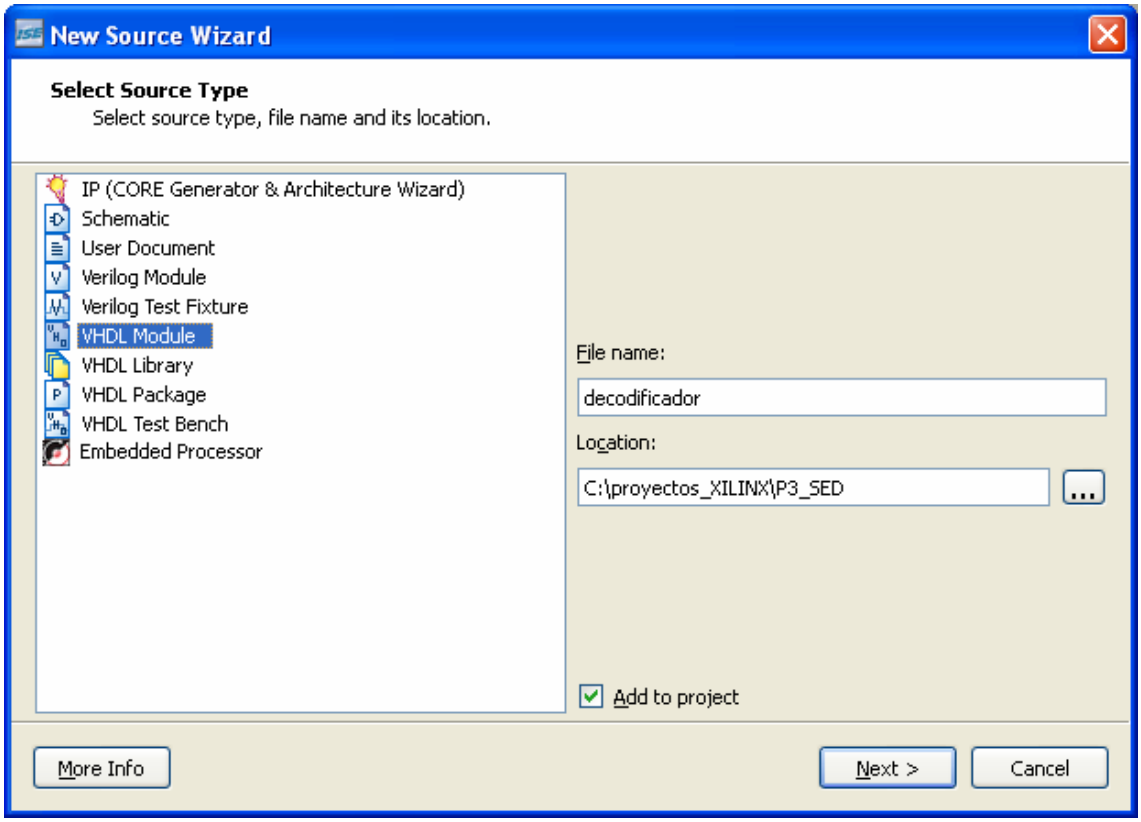
En nuestro caso, el display de 7 segmentos de la placa de desarrollo a usar tiene lógica negativa. Esto supone que cada led del display se enciende cuando el valor de su segmento es 0, y permanece apagado cuando vale 1. Esta consideración se tiene en cuenta en la tabla.

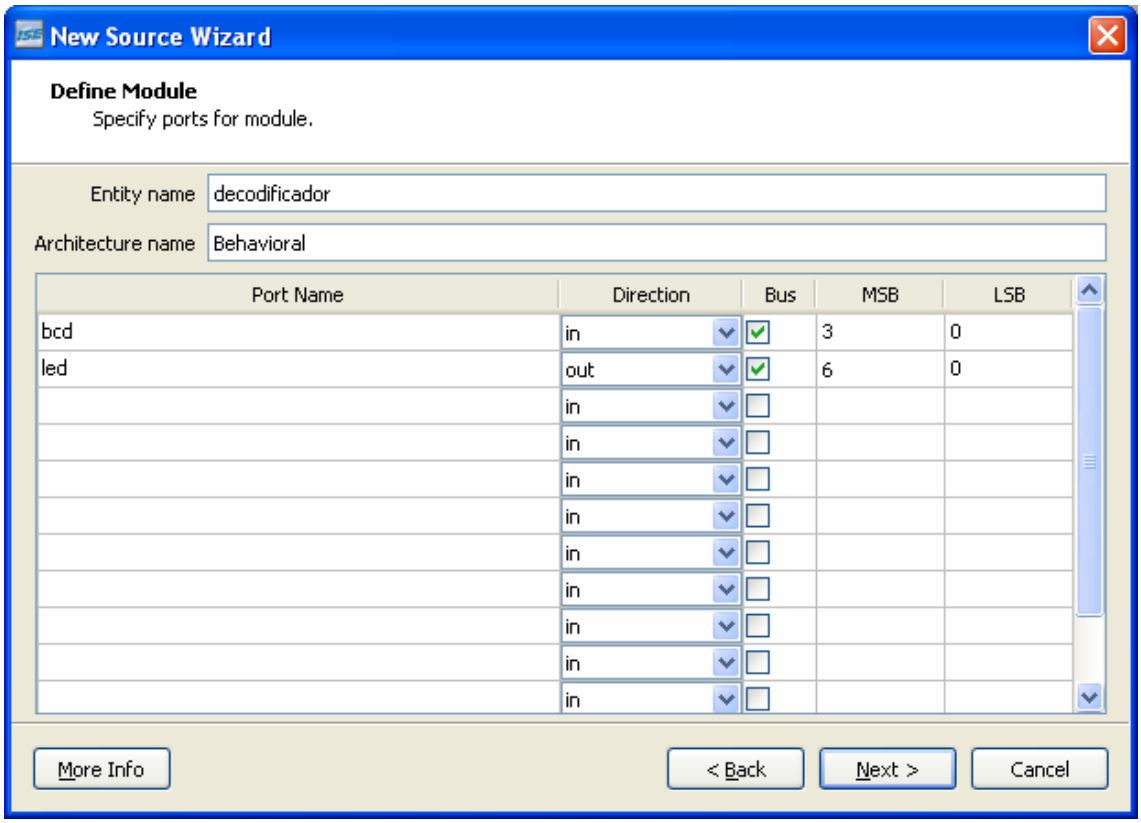
3.- Descripción VHDL

Una vez creado un nuevo proyecto como se realizó en la práctica 2, debemos crear un nuevo módulo VHDL llamado **decodificador** en la que estén implementadas las 7 funciones simplificadas usando mapas de Karnaugh.

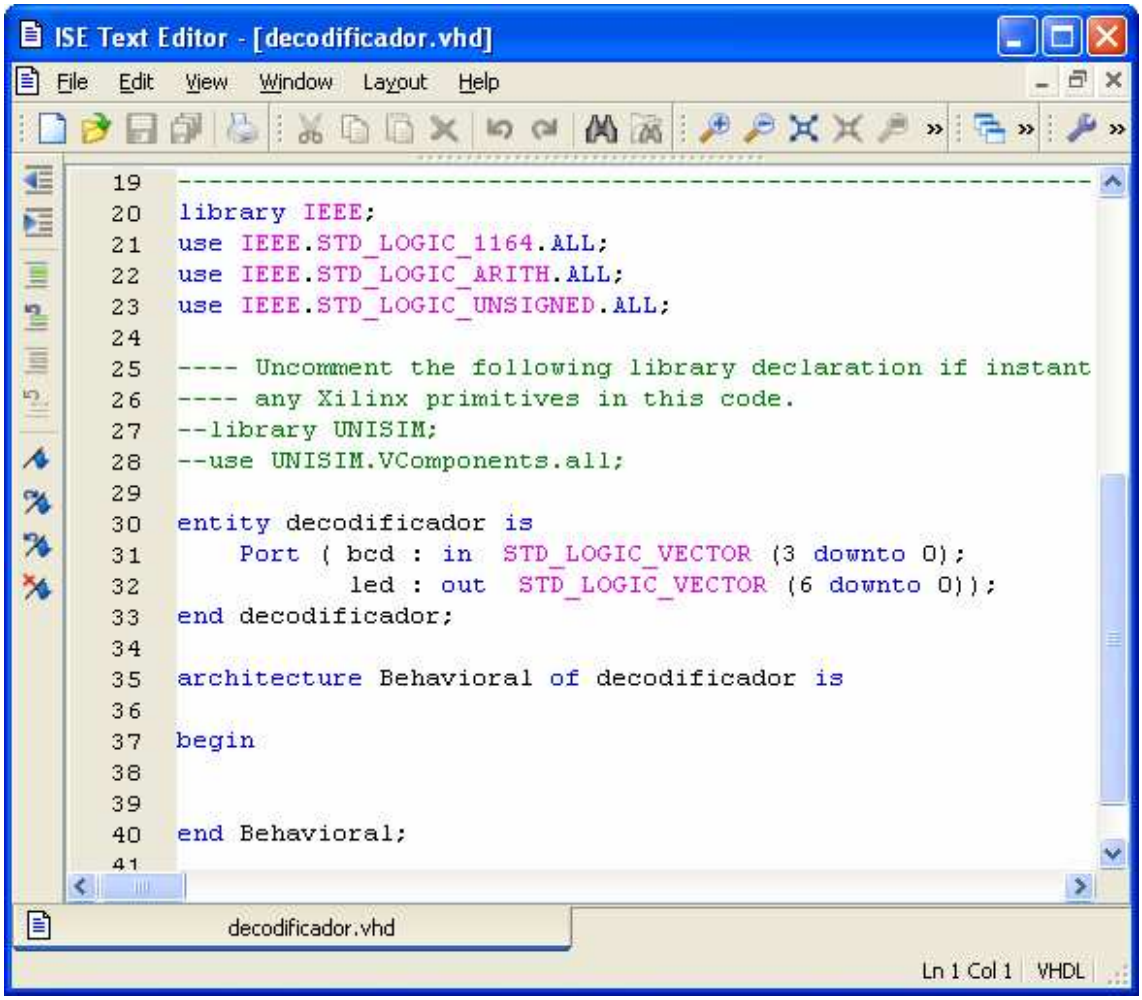
El diseño tiene 4 entradas y 7 salidas. No obstante, como tanto las entradas como las salidas están relacionadas, las vamos a agrupar en dos grupos de señales, denominadas *buses*. Así, nuestro diseño va a constar de un bus de entrada **bcd** de 4 bits de longitud (de 3 a 0), y un bus de salida **led** de 7 bits (de 6 a 0).

Para hacer esto, se marca en la declaración de los puertos la casilla de *bus*, y a continuación se introduce el tamaño del mismo. MSB es el acrónimo de bit más significativo en inglés y LSB, bit menos significativo.





Tras completar el asistente, se genera el esqueleto del código VHDL que hay que completar con la descripción del diseño (ver figura en la página siguiente). El profesor hará los comentarios pertinentes respecto al nuevo tipo de puertos declarados.



A continuación se simplifican las funciones empleando los mapas de Karnaugh. Si hacemos esto, las expresiones lógicas a implementar son:

$$led(6) = \overline{bcd(3)} \cdot \overline{bcd(2)} \cdot \overline{bcd(1)} + bcd(2) \cdot bcd(1) \cdot bcd(0)$$

$$led(5) = bcd(1) \cdot bcd(0) + \overline{bcd(3)} \cdot \overline{bcd(2)} \cdot bcd(0) + \overline{bcd(3)} \cdot \overline{bcd(2)} \cdot bcd(1)$$

$$led(4) = bcd(0) + \overline{bcd(3)} \cdot bcd(2) \cdot \overline{bcd(1)}$$

$$led(3) = \overline{bcd(3)} \cdot \overline{bcd(2)} \cdot \overline{bcd(1)} \cdot \overline{bcd(0)} + bcd(2) \cdot \overline{bcd(1)} \cdot \overline{bcd(0)} + bcd(2) \cdot bcd(1) \cdot bcd(0)$$

$$led(2) = \overline{bcd(3)} \cdot \overline{bcd(2)} \cdot bcd(1) \cdot \overline{bcd(0)}$$

$$led(1) = bcd(2) \cdot \overline{bcd(1)} \cdot bcd(0) + bcd(2) \cdot bcd(1) \cdot \overline{bcd(0)}$$

$$led(0) = \overline{bcd(3)} \cdot \overline{bcd(2)} \cdot \overline{bcd(1)} \cdot bcd(0) + bcd(2) \cdot \overline{bcd(1)} \cdot \overline{bcd(0)}$$

Para describir estas funciones, se proporciona el código que puede ser copiado en el script de VHDL. Nótese que se han introducido señales intermedias, definidas en la parte declarativa de la arquitectura. Estas señales, al ser de conexionado interno, pueden leerse y escribirse, tal y como se efectúa en el código. De esta manera, las ecuaciones lógicas que hay que escribir no ocupan líneas demasiado largas y también ganamos en claridad.

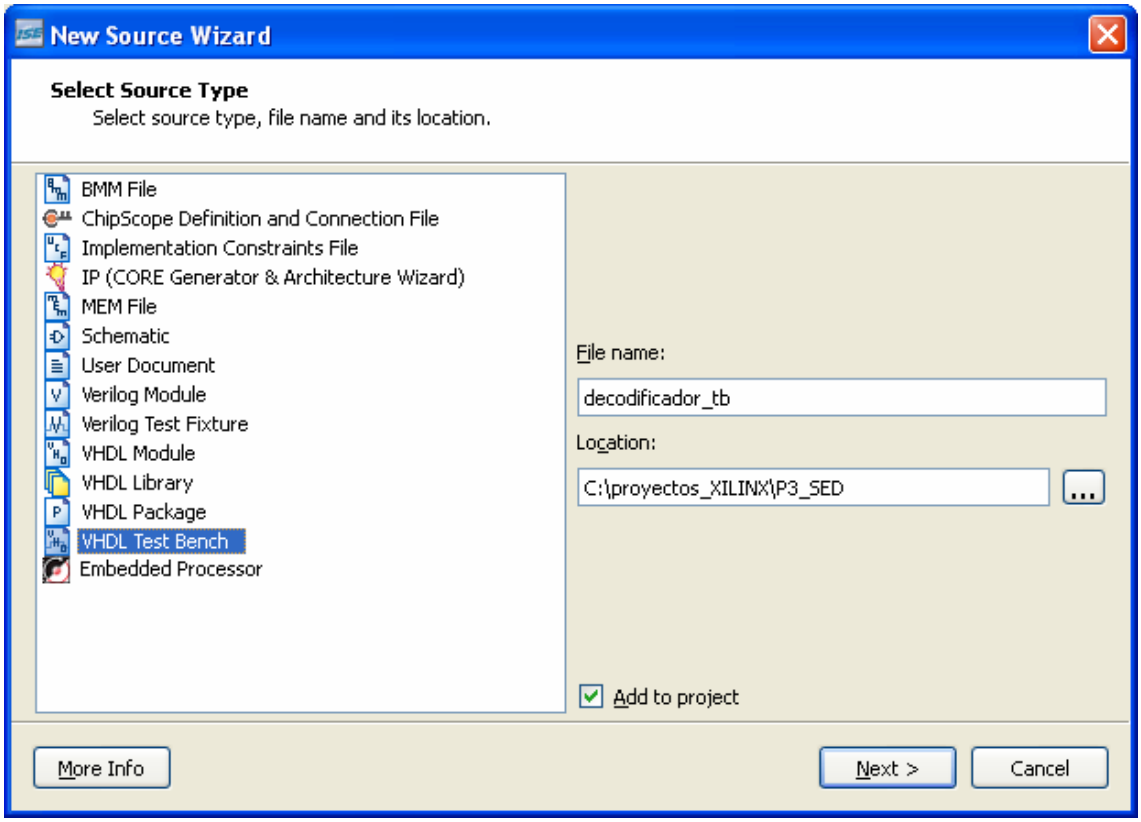
A continuación se muestra la parte del código que describe el decodificador de 7 segmentos. Si se ha comprendido el código, se recomienda usar las herramientas de copiar y pegar para añadirlo al script abierto en el entorno de desarrollo.

```
architecture Behavioral of decodificador is

    signal and1_led6, and2_led6                 : std_logic;
    signal and1_led5, and2_led5, and3_led5 : std_logic;
    signal and1_led4                            : std_logic;
    signal and1_led3, and2_led3, and3_led3 : std_logic;
    signal and1_led1, and2_led1                 : std_logic;
    signal and1_led0, and2_led0                 : std_logic;

begin

    and1_led6 <= ((not(bcd(3))) and (not(bcd(2)))) and (not(bcd(1)));
    and2_led6 <= (bcd(2) and bcd(1)) and bcd(0);
    and1_led5 <= bcd(1) and bcd(0);
    and2_led5 <= ((not(bcd(3))) and (not(bcd(2)))) and bcd(0);
    and3_led5 <= ((not(bcd(3))) and (not(bcd(2)))) and bcd(1);
    and1_led4 <= (bcd(2) and not(bcd(3))) and (not(bcd(1)));
    and1_led3 <= ((not(bcd(3))) and (not(bcd(2)))) and (bcd(0) and (not(bcd(1))));
    and2_led3 <= ((bcd(2)) and (not(bcd(1)))) and (not(bcd(0)));
    and3_led3 <= (bcd(2) and bcd(1)) and bcd(0);
    and1_led1 <= (bcd(2) and not(bcd(1))) and bcd(0);
    and2_led1 <= (bcd(2) and bcd(1)) and (not(bcd(0)));
    and1_led0 <= ((not(bcd(3))) and (not(bcd(2)))) and (bcd(0) and (not(bcd(1))));
    and2_led0 <= (bcd(2) and not(bcd(1))) and (not(bcd(0)));

    led(6) <= and1_led6 or and2_led6;
    led(5) <= (and1_led5 or and2_led5) or and3_led5;
    led(4) <= bcd(0) or and1_led4;
    led(3) <= (and1_led3 or and2_led3) or and3_led3;
    led(2) <= ((not(bcd(3))) and (not(bcd(2)))) and (bcd(1) and (not(bcd(0))));
    led(1) <= and1_led1 or and2_led1;
    led(0) <= and1_led0 or and2_led0;

end Behavioral;
```

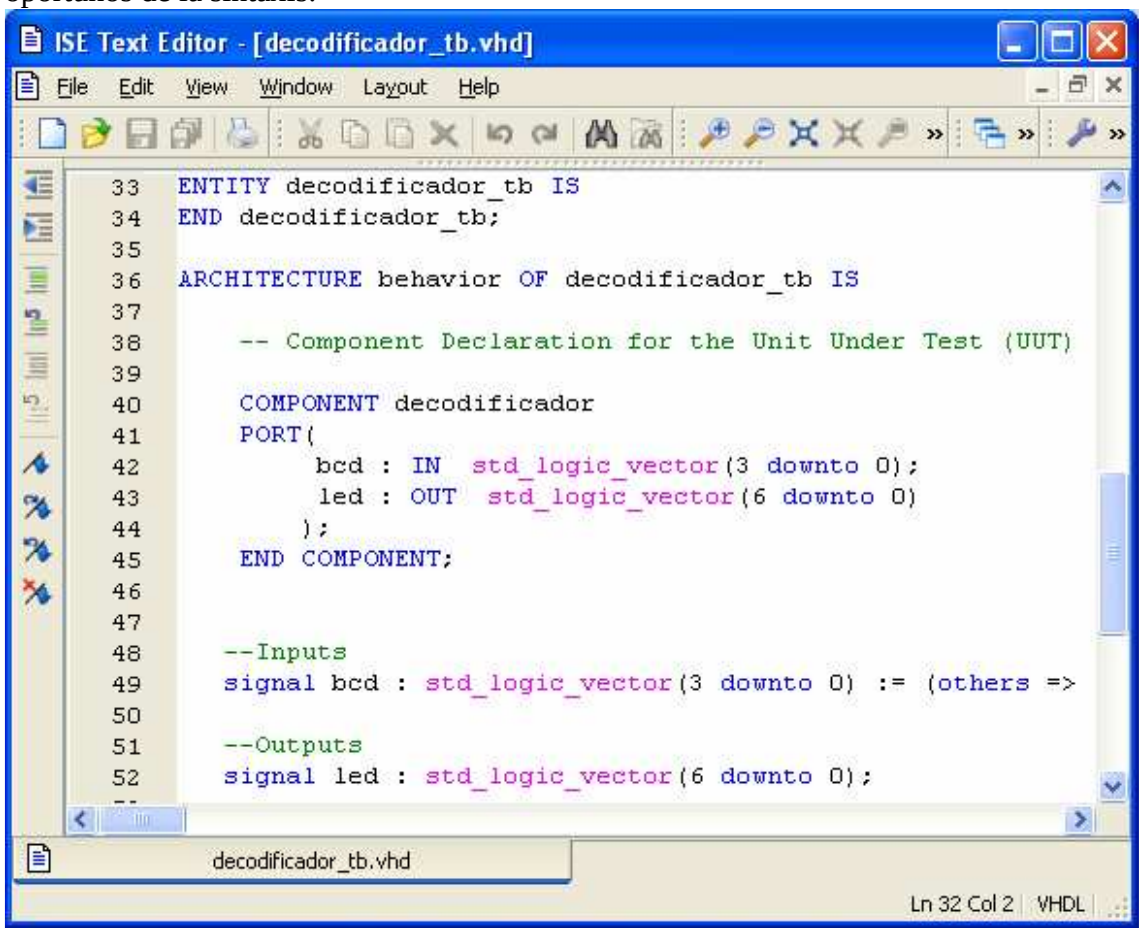
4.- Simulación

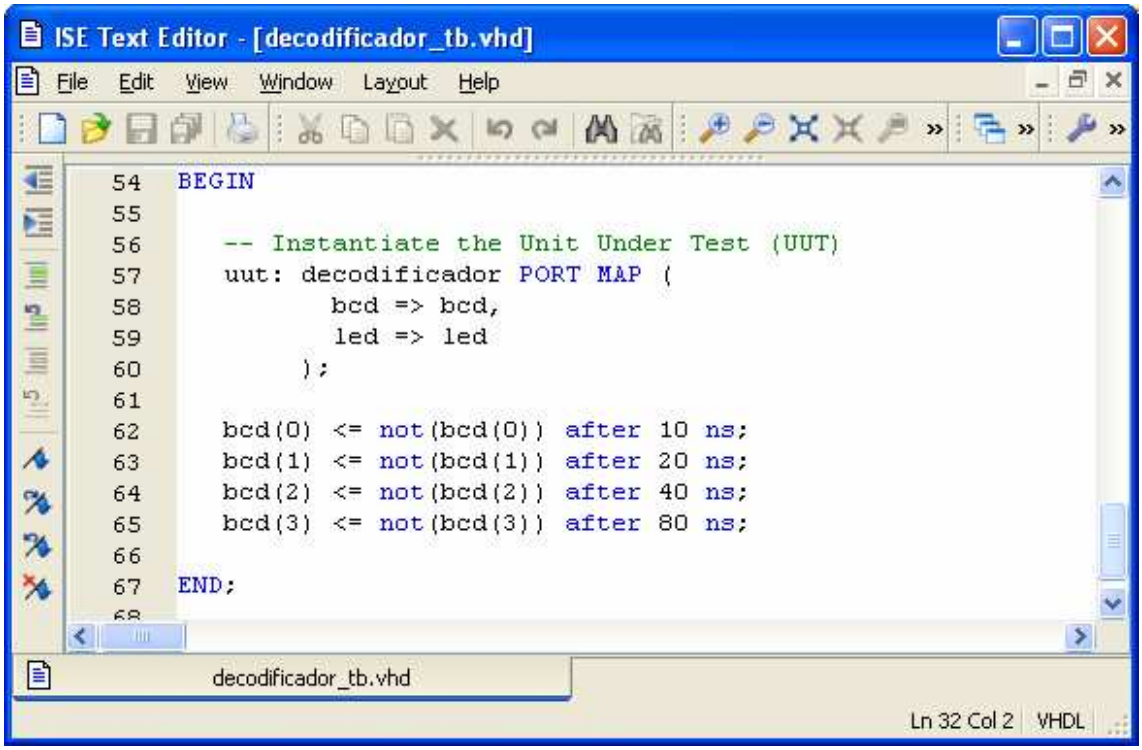
Una vez descrito el circuito, tenemos que comprobar que hace lo que deseamos. Debemos realizar una simulación que nos dé unos valores de salida acordes con la tabla de funcionamiento que decodificador que figura al principio de este documento.

Nuevamente creamos una nueva fuente, un test-bench de nombre **decodificador_tb**, que asociaremos al fichero que queremos testear, **decodificador.vhd**. En la página siguiente figura la primera ventana del asistente.

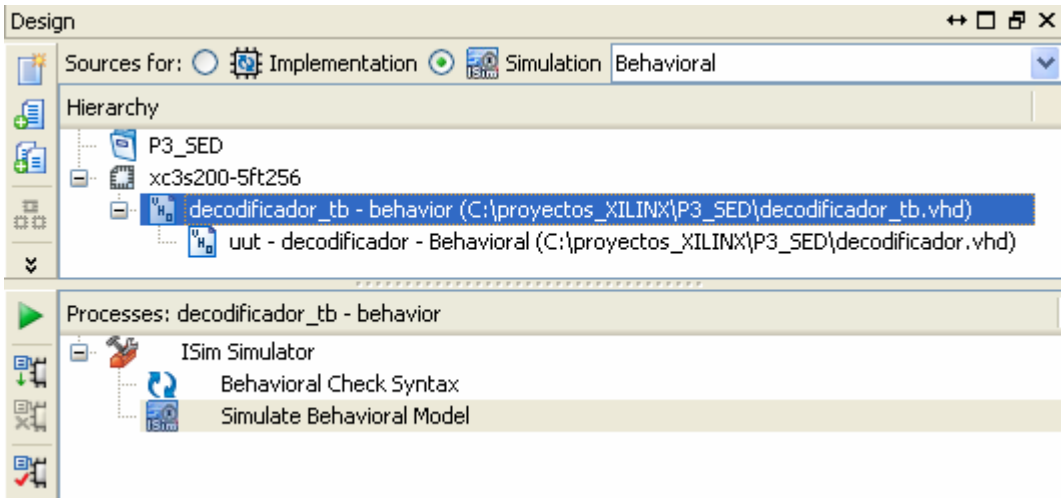


A continuación se muestra el código del test-bench. El profesor hará los comentarios oportunos de la sintaxis.





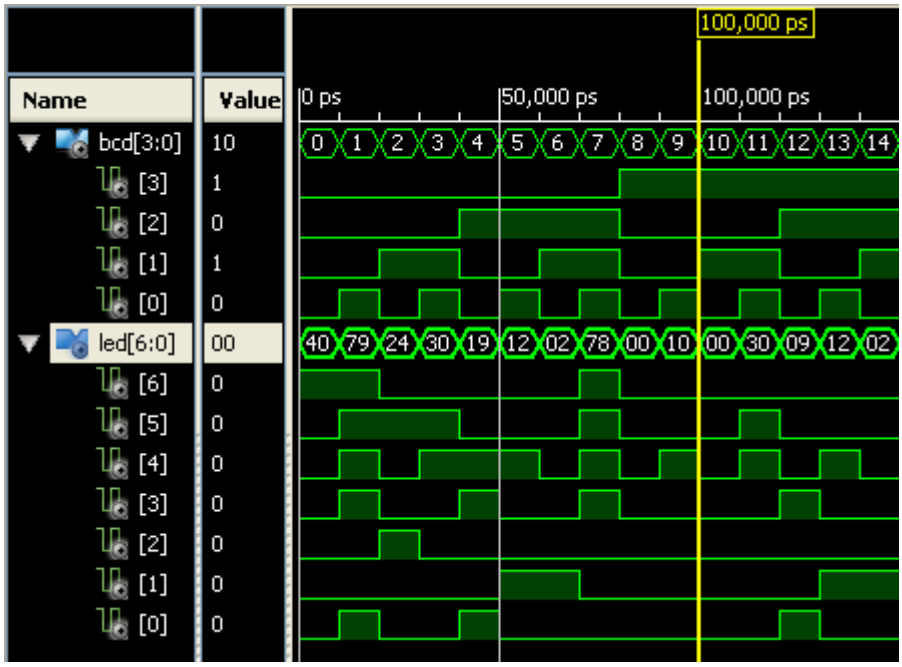
Podemos comprobar la jerarquía del proyecto, y ver que, efectivamente, el test-bench encapsula al código VHDL que vamos a simular. Todo esto en la vista *Sources for Simulation*.



Manteniendo seleccionado el test-bench (como en la figura). Ejecutamos el comando *Simulate Behavioral Model* para ver los valores de salida de la simulación y contrastarlos con la tabla de la verdad del circuito.

RECORDATORIO:
Una sóla simulación simultáneamente. Si deseamos lanzar una nueva simulación, deberemos cerrar la actual.

A continuación se muestran los resultados de la simulación y la tabla de la verdad en la que se ha incluido una nueva columna con los datos agrupados en notación hexadecimal. El profesor comentará la manera de visualizar la simulación como aparece en la figura.

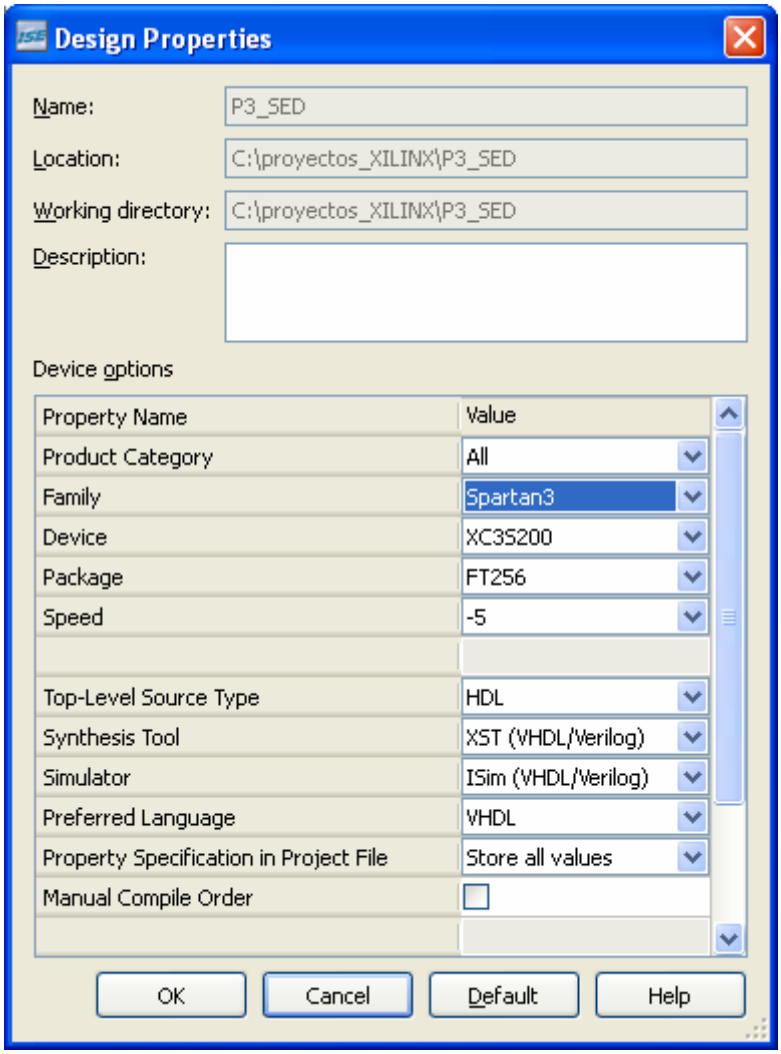


bcd(3)	bcd(2)	bcd(1)	bcd(0)	led(6) g	led(5) f	led(4) e	led(3) d	led(2) c	led(1) b	led(0) a	led (hexa)
0	0	0	0	1	0	0	0	0	0	0	40
0	0	0	1	1	1	1	1	0	0	1	79
0	0	1	0	0	1	0	0	1	0	0	24
0	0	1	1	0	1	1	0	0	0	0	30
0	1	0	0	0	0	1	1	0	0	1	19
0	1	0	1	0	0	1	0	0	1	0	12
0	1	1	0	0	0	0	0	0	1	0	02
0	1	1	1	1	1	1	1	0	0	0	78
1	0	0	0	0	0	0	0	0	0	0	00
1	0	0	1	0	0	1	0	0	0	0	10
1	0	1	0	X	X	X	X	X	X	X	XX
1	0	1	1	X	X	X	X	X	X	X	XX
1	1	0	0	X	X	X	X	X	X	X	XX
1	1	0	1	X	X	X	X	X	X	X	XX
1	1	1	0	X	X	X	X	X	X	X	XX
1	1	1	1	X	X	X	X	X	X	X	XX

5.- Selección de la placa de desarrollo

Para los siguientes pasos de la fase de diseño es necesario especificar el tipo de dispositivo FPGA sobre el que se a implementar las funciones lógicas.

El profesor explicará cómo se selecciona la FPGA y qué FPGA se va a emplear. La ventana donde se realiza esta elección aparece en la siguiente figura:



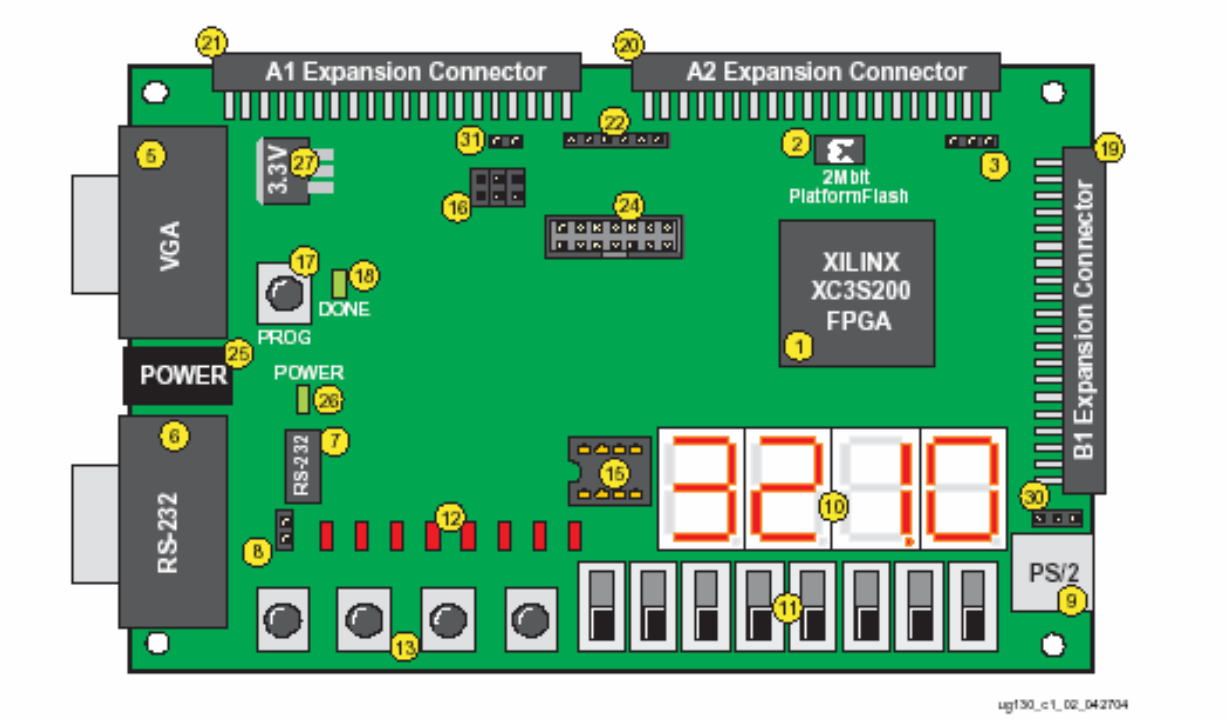
6.- Asignación de pines de la FPGA

En este punto el diseñador puede abstraerse de todos los procesos que tienen lugar a partir de ahora. No obstante, el entorno permite modificar y seleccionar parámetros y características en todos estos procesos, a conveniencia por parte del propio diseñador.

Una de las cosas que puede hacer es lo que se denomina **restricciones de usuario** (**user constrains** en inglés). Estas restricciones pueden ser temporales (especificar unos ciertos requerimientos de velocidad, por ejemplo) o de ubicación (en qué lugar físico de la FPGA implementaremos nuestro circuito lógico) o de asignación de pines (en qué pines físicos del dispositivo FPGA pondremos nuestros puertos de entrada y salida).

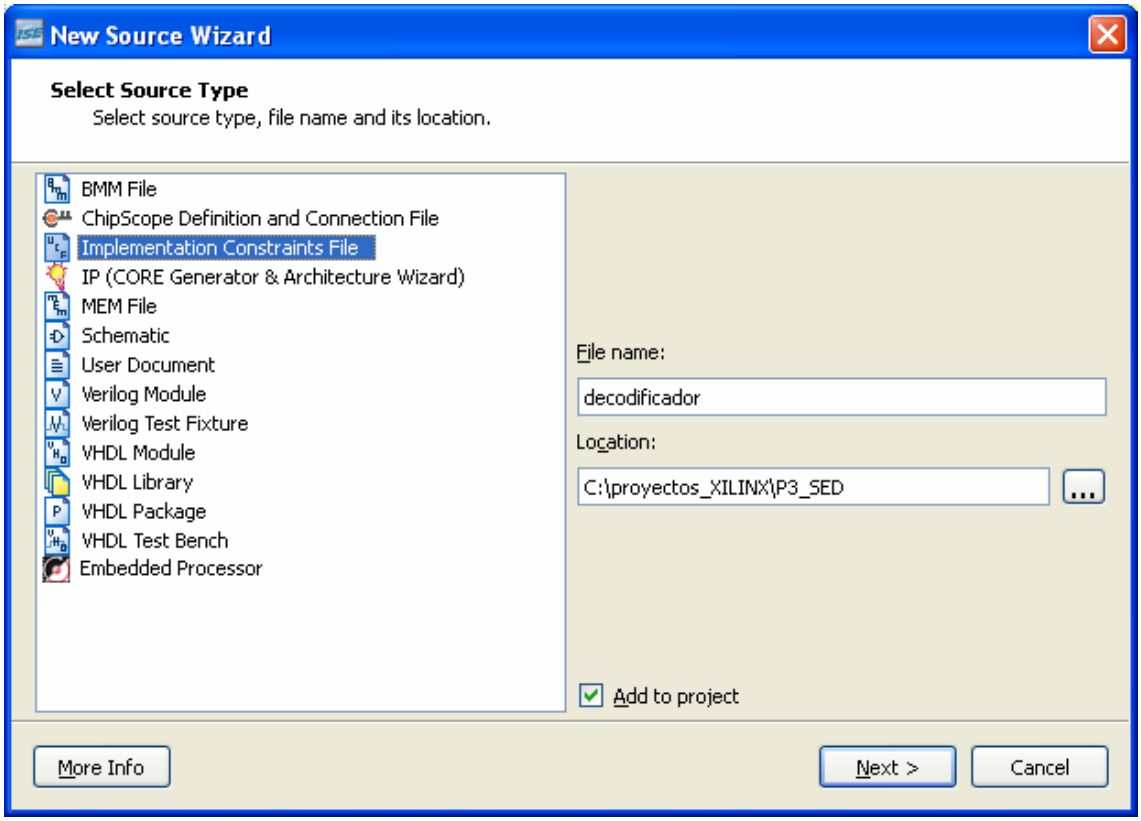
La placa de desarrollo suele constar de una FPGA y ciertos periféricos conectados a ella que son útiles para comprobar en el propio circuito nuestro diseño. Estos periféricos suelen ser interruptores, botones, leds, displays, conectores serie, VGA y otros muchos.

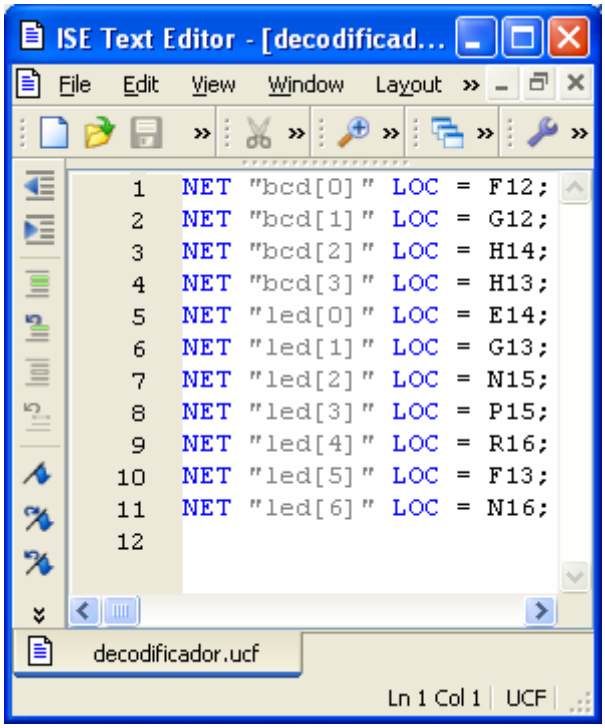
Un ejemplo se presenta en el siguiente esquema:



En nuestro diseño deseamos conectar la entrada **bcd** a 4 interruptores como los de la figura (11) y la salida **led** a un display como los de la figura (10). Normalmente los fabricantes proveen en las hojas de características de sus placas de una serie de tablas que asocian cada periférico con un pin de la FPGA.

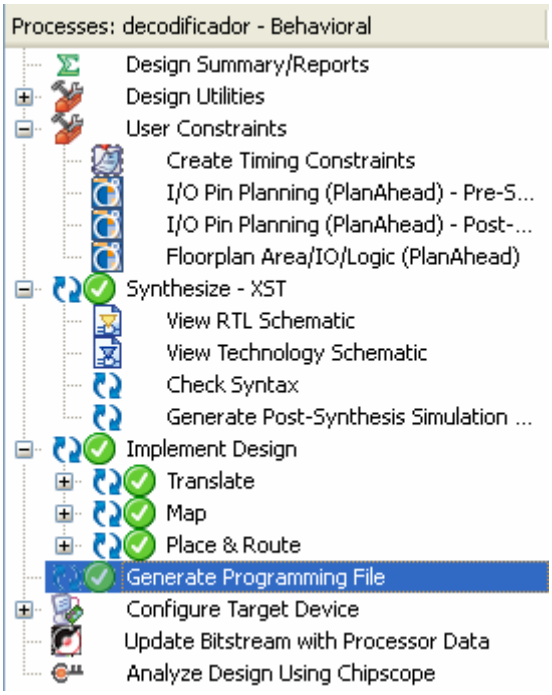
Para realizar la asignación debemos crear una nueva fuente (*New Source*).





En el asistente seleccionamos *Implementation Constraints File*. Esto generará un fichero de texto con extensión UCF (acrónimo de *User Constraints File*) en el que introduciremos la asignación de pines a cada puerto de entrada o salida de nuestro diseño.
En la figura aparece un ejemplo de asignación de pines para una FPGA Spartan-3 XS200. El profesor proporcionará la información de los pines de la FPGA de la que se disponga ese día en el laboratorio.

7.- Implementación y generación del fichero de configuración



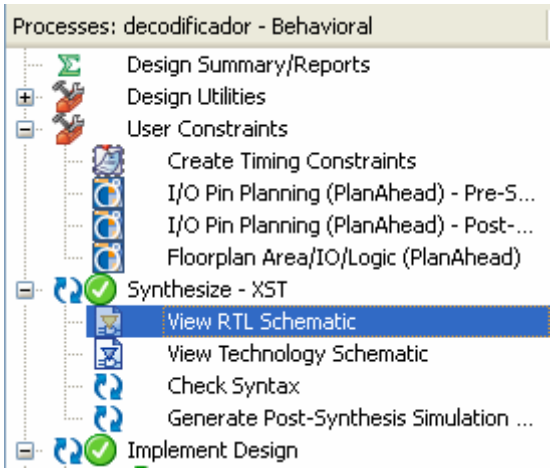
Los últimos pasos del diseño se realizan ejecutando el comando *Generate Programming File*. Haciendo esto, se ejecutarán todos los pasos necesarios para implementar el diseño (*Implement Design*: *Translate*, *Map*, y *Place & Route*). Si todo va bien, aparecerá en cada una de estas fases un icono confirmativo de color verde, como en la figura.

El profesor comentará en qué consiste cada una de estas etapas.

Si todo se ha ejecutado satisfactoriamente se generará el archivo **decodificador.bit**, con toda la información para configurar satisfactoriamente la FPGA. El profesor le indicará cómo.

8.- Ejercicios

1. Ejecutar el comando *View RTL Schematic*. Observar los resultados que se obtienen.
2. Ejecutar el comando *View Technology Schematic*. Observar los resultados que se obtienen. Las LUT que aparecen son generadores de funciones implementadas con memorias de 16 posiciones de 1 bit.



3. Para diseños lógicos complejos, la descripción a través de VHDL de flujo de datos (ecuaciones lógicas) no resulta ser la manera de descripción más eficaz. Sustituir la descripción del decodificador por la que se muestra en la figura, que responde a una descripción en un nivel de abstracción mayor usando VHDL funcional o comportamental.
El sintetizador se encarga de traducir esta descripción a hardware. Simular el diseño usando esta forma alternativa de descripción para verificar que los resultados que se obtienen son los mismos. Repetir ejercicios 1 y 2 nuevamente.