UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES

ARAGON

INGENIERIA EN COMPUTACIÓN

“TEORÍA Y PRÁTICA DE LA PROGRAMACIÓN DE CPLD DE XILINX” EN LA MODALIDAD DE: DESARROLLO DE UN CASO PRÁCTICO

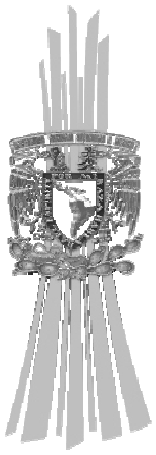
T R A B A J O E S C R I T O Q U E

P R E S E N T A

CARLOS ANTONIO GENARO OLIVAS

P A R A O B T E N E R E L T I T U L O D E :

I N G E N I E R O E N C O M P U T A C I Ó N



ASESOR:

DR. JUAN MANUEL LÓPEZ CARRETO

MEXICO, ARAGON ENERO DE 2008

Este trabajo está dedicado a mi Padre Dios,
a quien agradezco por todo lo recibido en la vida.
Lo dedico también a mis Papás que lo han dado todo por mi.
Y a mis padrinos, mis abuelitos,
que me inculcaron siempre
el mayor conocimiento del universo, el de Cristo.

Gracias también a Paco y Ale, por estar siempre cerca.
Gracias a José, Ricardo, Oscar y Medel
por la amistad durante todos estos años.
Gracias a la UNAM por la educación gratuita.
Gracias a mi Maestro, el Dr. Juan Manuel
por el soporte, el apoyo y las oportunidades.

“La inteligencia consiste no sólo en el conocimiento, sino también en la destreza de aplicar los conocimientos en la práctica”

ARISTÓTELES

Contenidos

Resumen

La historia del CPLD comienza en los años 70’s cuando solo se contaba con dispositivos lógicos sencillos y estaba en auge el desarrollo de tarjetas impresas, entonces fue creada una matriz programable en la que se podía realizar cualquier diseño combinacional a partir de compuertas AND y OR, el desarrollo de estás matrices, su simplificación con el uso de solamente un tipo de compuertas programables, manteniendo el otro tipo de compuertas como no programable, aunado a la evolución de la programación de estas compuertas a partir de circuitos CMOS que pueden ser regrabados, con ayuda de las macroceldas y la microminiaturización de la tecnología digital permitieron sentar las bases para crear el CPLD.

El CPLD por sí mismo es un circuito programable pero requiere de un lenguaje de programación específico para ser programado.

Este lenguaje es VHDL, un lenguaje que nos permite describir hardware, es decir, nos permite decidir cómo se comportarán las señales en un circuito cuya base es de matrices programables y macroceldas.

El diseño de sistemas en VHDL es sencillo, pues el lenguaje acepta tipos de descripciones que son fáciles de razonar y después se encarga de ajustarlos y convertirlos en una función que pueda ser llevada a la matriz AND/OR. Para usar el lenguaje, es preciso contar con el software que se encargue de traducirlo y grabarlo en el CPLD, en este trabajo ese software es Xilinx ISE, un poderoso programa en el cual se pueden hacer todo tipo de diseños en VHDL para todos los dispositivos que existen entre los productos que maneja Xilinx, ya sean CPLD ó FPGA.

En el primer capítulo de este trabajo se analiza la historia del CPLD y de la lógica programable, así como la arquitectura del CPLD, en el segundo capítulo se trabaja sobre el lenguaje VHDL, de manera teórica, puede resultar exhausto hacer toda la lectura del lenguaje, pero es importante tener una referencia, posteriormente se describe el uso del software y sus posibilidades de diseño, sus herramientas, beneficios y características.

Ya entrando en la parte práctica, las primeras prácticas son de programas realmente muy sencillos; sin embargo, el gran valor de estas es que ayudan a conocer el lenguaje y

el software. En las primeras prácticas se aprende a crear nuevos diseños, a simular, y a implementar diseños basados en dispositivos lógicos sencillos como los multiplexores y los contadores, mostrando cómo un CPLD puede trabajar como cualquiera de estos y además, como muchos de estos dispositivos reunidos en uno solo.

En el capítulo 4, que corresponde a las Prácticas Intermedias se explota el hecho de poder diseñar sistemas que involucran varios tipos de dispositivos, pero se lleva al siguiente nivel, pues se hacen diseños que comúnmente se hacían con el uso de dispositivos más sencillos sin la necesidad de usar los mismos. Estos diseños son los basados en máquinas de estados, son descritos con las herramientas con que VHDL dispone para tratar diseños más avanzados, como son vectores, sentencias de control, procesos, tipos de datos personalizados, etc… Una amplia gama de herramientas y ventajas que hacen más simple y más poderoso el uso de estas tecnologías. Todas estas herramientas se trabajan en este capítulo.

Finalmente en el capítulo 5 se trabaja sobre máquinas de estados, explotando todas las posibilidades de Xilinx ISE y VHDL, este capítulo consuma la unión de estas dos herramientas en conjunto con la capacidad del CPLD de albergar diseños relativamente grandes y complejos. Por último se realiza el diseño e implementación de una memoria RAM que puede ser utilizada dentro de un diseño como puede ser un microprocesador, por poner un ejemplo.

En este trabajo encontrará más que los fundamentos necesarios para empezar a trabajar con CPLDs, encontrará los fundamentos necesarios para empezar a diseñar sistemas digitales en CPLD.

1. El CPLD

En este primer capítulo se trata la evolución de los circuitos programables iniciando desde los PAL (*Programmable Array Logic*), PLD (*Programmable logic Device*), GAL (*Generic Array Logic*), CPLD (*Complex Programmable Logic Device*) y FPGA (*Field Programmable Gate Array*); incluyendo sus principales características, diferencias, virtudes, etc. Posteriormente se analiza la arquitectura interna del CPLD de manera detallada, describiendo sus modos de operación, de manera tal que el lector tenga un panorama concreto de lo que es un CPLD y lo que posteriormente se le podrá exigir.

1.1 Historia de la lógica programable

A finales de los años 70's los dispositivos lógicos estándar y las tarjetas impresas estaban en auge. Entonces alguien se preguntó: "¿Qué tal si damos a los diseñadores la capacidad de implementar distintas interconexiones en un dispositivo más grande?". Esto permitiría a los diseñadores integrar muchos dispositivos lógicos básicos en uno sólo.

Para ofrecer lo último en flexibilidad de diseño, Ron Cline[1] de Signetics, tuvo la idea de hacer dos planos programables, que fueran capaces de implementar cualquier combinación de compuertas AND y OR. Pasando por las compuertas OR los productos de las compuertas AND utilizando matrices programables.

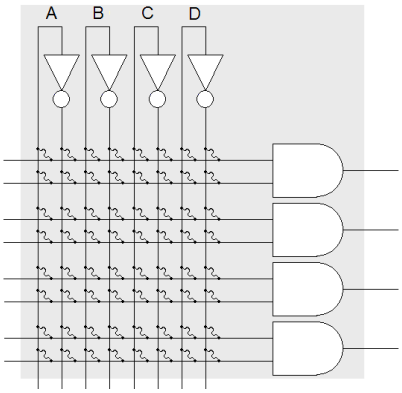


Fig. 1.1 Matriz AND no programada

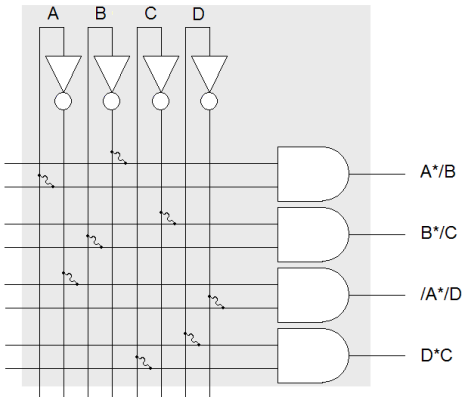


Fig. 1.2 Matriz AND programa

Una matriz programable es una red de conductores distribuidos en filas y columnas con un fusible para cada punto de intersección, la programación de la matriz se realiza mediante la fundición de los fusibles de los puntos de intersección.

[1] Ronald Cline era Diseñador para Signetics a nivel Silicio, tiempo después, Philips compró Signetics y Ron pasó a ocupar el puesto de Director de Ingeniería.

El dispositivo con matrices OR y AND programables es conocido como PLA *"Programmable Logic Array"* y su diagrama de flujo es el siguiente:

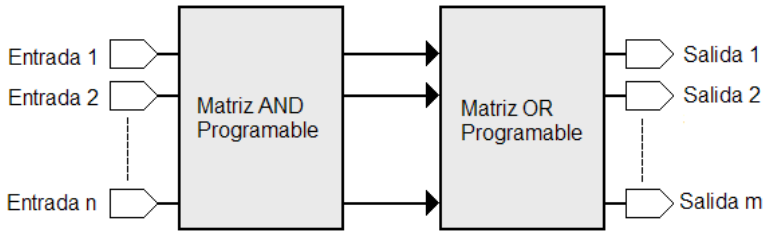


Fig. 1.3 Diagrama de flujo del PLA

Esta arquitectura resultó ampliamente flexible, pero debido a la geometría y los tiempos de reloj con que se manejaba (10μs), el tiempo de propagación (comúnmente llamado *"delay"*) de la entrada con respecto a la salida se volvió muy grande, lo que hizo que estos nuevos dispositivos se volvieran lentos.

Las principales características del PLA son:

- Dos planos programables.
- Cualquier combinación de ANDs y ORs.
- Compartición de términos AND a través de multiples ORs.
- La más alta densidad de lógica programable para el usuario.
- Alto contenido de fusibles, es más lento que los PAL.
- Arreglo de lógica programable.

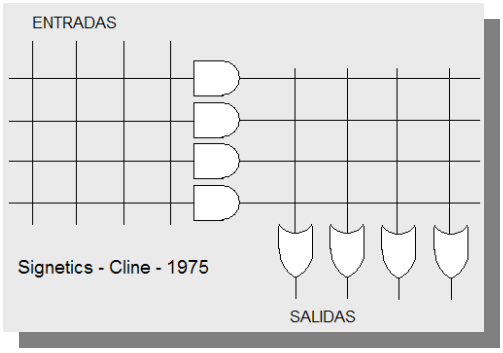


Fig. 1.4 Diagrama del PLA

Años más tarde, MMI[2], que era considerada como la segunda fuente de arreglos PLA, después de fabricar el PLA, lo modificó para crear la arquitectura PAL (Programable Array Logic) dejando como fija una de las dos matrices programables. Esta arquitectura,

[2] MMI fue comprada por AMD, consolidándose como Vantis, bajo esta razón social, fue adquirida en 1999 por Lattice Semiconductor, conocido proveedor de PALs y GALs. John Birkner fue cofundador de MMI.

a diferencia de la PLA tenía el arreglo de compuertas OR fijo, y el arreglo AND era el programable, esta nueva arquitectura resultó mucho más rápida que la PLA y requirió de software menos complejo, aunque no tenía toda la flexibilidad de la estructura PLA.[3]

Otras arquitecturas siguieron a ésta, como el PLD *"Programmable Logic Device"*, y esta categoría de dispositivos que se conoce como SPLD *"Simple PLD"*.

Las características del PAL son:

- Un plano programable OR Fijo / AND.
- Combinación finita de ANDs y ORs.
- Media densidad lógica disponible para el usuario.
- Menor cantidad de fusibles.
- Mayor velocidad que los PLA's.
- Lógica de arreglo programable.

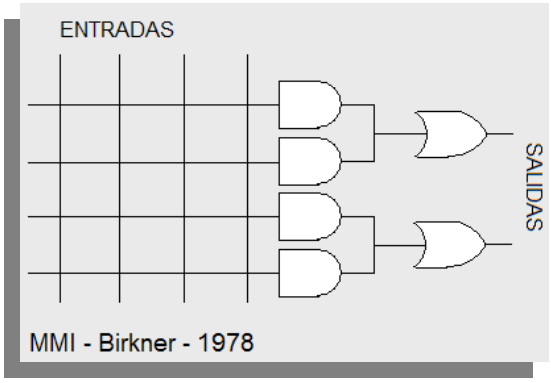


Fig. 1.5 Diagrama del PAL

Esta arquitectura tiene muchos caminos de interconexión vertical y horizontal, en cada juntura existe un fusible, con la ayuda de herramientas de software, los diseñadores pudieron seleccionar que junturas no serían conectadas "quemando" sus fusibles; esto era hecho mediante un dispositivo programador.

Los pines de entrada eran conectados a los conductores verticales, los horizontales a las compuertas AND-OR, llamadas Términos Producto, y estos a su vez a flip-flops, cuyas salidas eran vinculadas a los pines de salida. Los PLDs ofrecieron hasta 50 veces más compuertas en un mismo encapsulado que los dispositivos lógicos discretos, esto fue un gran avance, sin mencionar que el número de dispositivos en existencia era menor.

Un desarrollo un poco más reciente de los PLD es la GAL *"Generic Array Logic"*, es básicamente igual que la PAL, con la diferencia de que no tiene fusibles ni tecnología

bipolar, sino tecnología E^{2}CMOS "*Electronically Erasable CMOS*", esto la convierte en tecnología reprogramable. La manera de funcionar de la GAL es exactamente la misma que la PAL.

La figura 1.6 muestra un ejemplo de una GAL no programada:

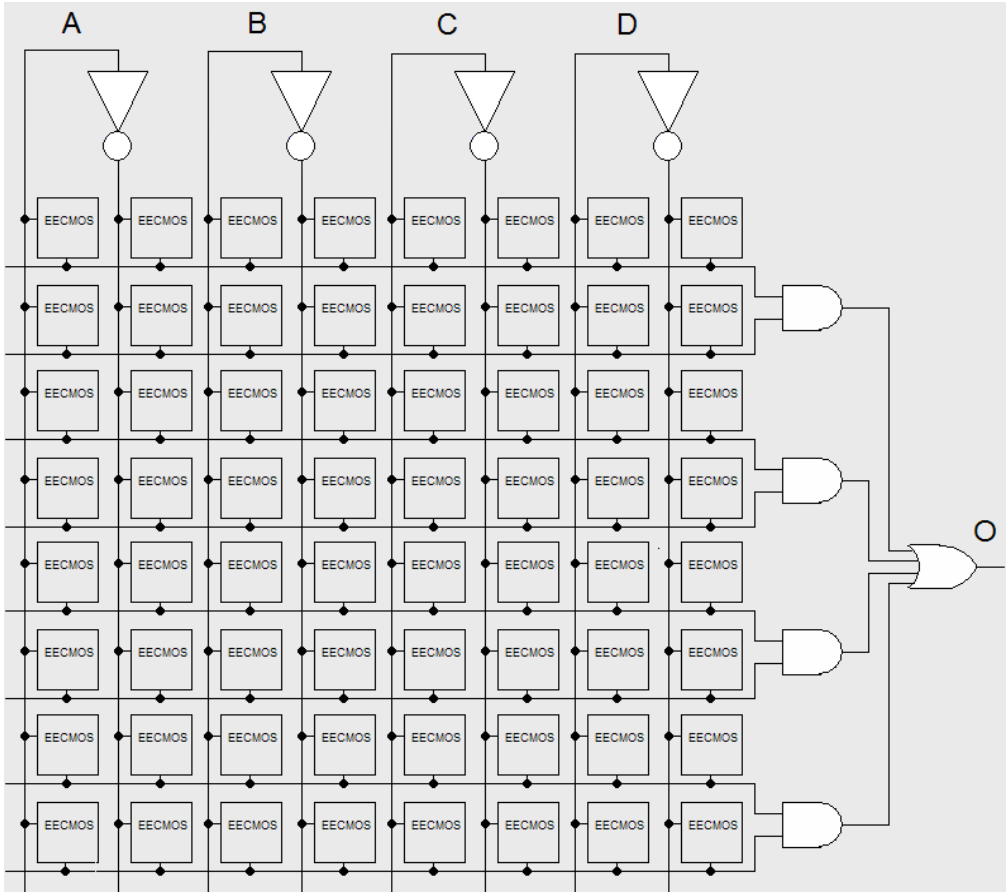


Fig. 1.6 Diagrama lógico de una GAL no programada

Las celdas E^{2}CMOS activadas conectan las variables deseadas o sus complementos con las apropiadas entradas de las puertas AND. Las celdas E^{2}CMOS están desactivadas cuando una variable o su complemento no se utilizan en un determinado producto. La salida final de la puerta OR es una suma de productos.

En la figura 1.7 se ejemplificará la implementación de la función
O = /A*C + B*/D + /B*/C + A*D en un arreglo tipo GAL.

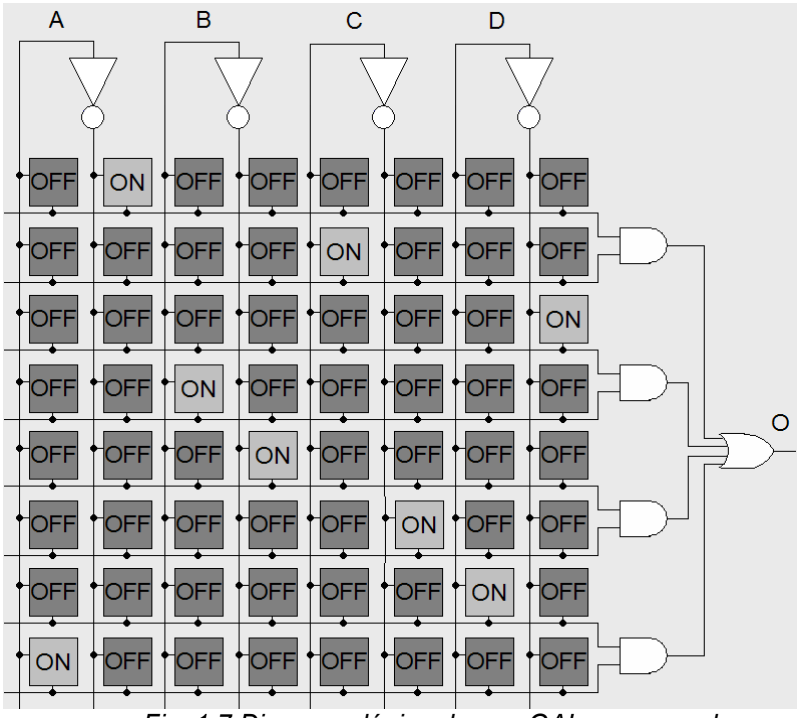


Fig. 1.7 Diagrama lógico de una GAL programada

La tecnología PLD se ha movido en los últimos tiempos hacia compañías como Xilinx, que produce dispositivos CMOS de ultra bajo consumo de potencia *"Ultra-Low Power"* basados en la tecnología de memorias Flash. Los PLDs Flash tienen la ventaja de que pueden ser borrados y reescritos una y otra vez programando y borrando electrónicamente el dispositivo.

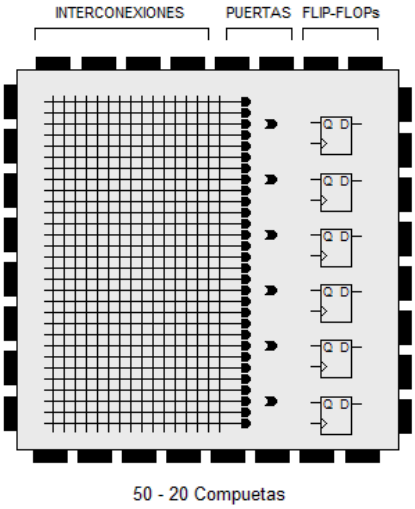


Fig. 1.8 Diagrama lógico general del CPLD

Actualmente hay disponibles PLDs mucho más sofisticados, los CPLD "*Complex PLD*" y los FPGA *"Field Programmable Gate Array"*.

Esencialmente, un CPLD es una versión más grande de un PLD, cuentan con una matriz de interconexión interna centralizada que se utiliza para conectar las macroceldas del dispositivo. Al igual que en los SPLD, la matriz se programa para conectar de forma selectiva señales de entrada a un conjunto de compuertas AND que a su vez están conectadas con compuertas OR.

Las salidas de las compuertas OR se conectan a su vez con macroceldas configurables, que permiten al usuario configurar la polaridad de la salida, seleccionar operaciones combinacionales o de registro, proporcionar la funcionalidad tri-estado y, opcionalmente realimentar la señal a la matriz de interconexión.

El camino de una señal de entrada a través del CPLD es el mismo siempre, independientemente de la función lógica que sea implementada. La señal entra a través de un Buffer o Inversor hasta la matriz de interconexión, de allí pasa a las compuertas AND/OR y de posteriormente es conducida a la macrocelda de salida.

Además, las señales de entrada, salida y reloj del CPLD se encuentran en pines específicos siempre, por lo que son idénticos entre sí. Esta arquitectura limita la flexibilidad del diseño en cierto grado, pero también simplifica el análisis del diseño ya que la temporización es fácil de calcular, así que antes de comenzar el diseño se pueden calcular las velocidades de entrada/salida.

Los CPLDs pueden manejar velocidades de hasta 5 nanosegundos, lo que equivale a 200MHz. En 1985 Xilinx introdujo una idea completamente nueva: Combinar el control del usuario y el tiempo de los PLDs con las densidades y costo-beneficios de los arreglos de compuertas.

A los usuarios les agradó el concepto y nació el FPGA. Acualmente, Xilinx[3] es aún el productor número uno de FPGA's en todo el mundo. Un FPGA es una estructura regular de celdas lógicas (o módulos) interconectadas que está bajo el completo control del usuario. Esto significa que se puede diseñar, programar y hacer cambios al circuito en cualquier momento que se desee. Los FPGA no tienen una estructura de implementación predefinida como los CPLD, en cambio, tienen canales de rutado que se conectan entre las celdas lógicas de múltiples maneras distintas, para implementar las funciones lógicas deseadas.

[3] Xilinx es líder en el mercado de Dispositivos Lógicos Programables. Fuente: Artmam.net http://artmam.net/Digital_Signal_Processor.htm

1.2 Arquitectura simplificada de un CPLD

La arquitectura de un CPLD está dada fundamentalmente por la Macrocelda, una macrocelda está compuesta por dos unidades: la matriz programable de compuertas AND/OR y la OLMC *"Output Logic Macrocell"*.

A continuación se procede a explicar con detalle el funcionamiento de estas dos unidades:

1.2.1 Matrices AND/OR

Todo diseño lógico simplificado puede ser descrito con la ayuda de solamente dos tipos de compuertas, AND y OR, pues éstas pueden realizar la función de casi cualquier otro tipo de compuerta básica. Esto será ejemplificado mediante la solución a dos problemas comunes de diseñó lógico.

Ejemplo 1. Lógica del sumador completo

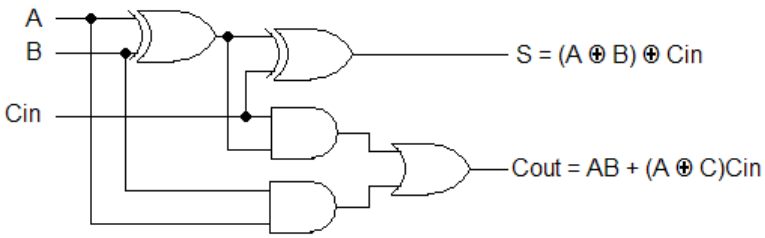


Fig. 1.9 Sumador Completo

Este sumador completo utiliza 3 tipos distintos de compuertas en su mínima expresión, ahora, convirtiendo sus ecuaciones tenemos:

S = /A*/B*Cin + /A*B*Cin + A*/B*/Cin + A*B*Cin

Cout = /A*B*Cin + A*/B*Cin + A*B*/Cin + A*B*Cin

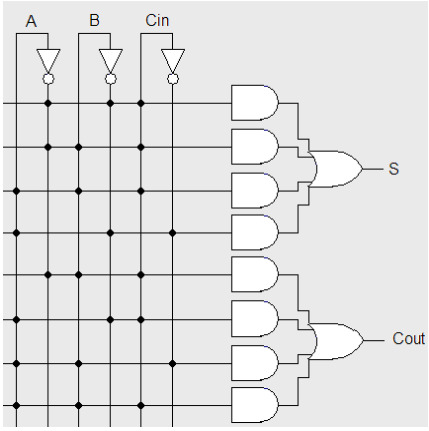


Fig. 1.10 Matriz programable AND-OR del sumador completo

Estas ecuaciones las transportamos a una matriz programable AND-OR

Difícilmente el diseño ocupará menos compuertas que las que usa el diseño a transportar, sin embargo, debido a que esta programación es dada por una herramienta de software independiente del diseño, no es menester del programador preocuparse de ello.

Véase ahora otro ejemplo de la ventaja de la matriz programable usando un ejemplo diferente, en ésta ocasión, un multiplexor 4:1 Con la siguiente tabla de verdad:

A	B	C	D	S0	S1	O
A	X	X	X	0	0	A
X	B	X	X	0	1	B
X	X	C	X	1	1	C
X	X	X	D	1	0	D

O = /S0*/S1*A + /S0*S1*B + S0*/S1*C+ S0*S1*D

La implementación queda finalmente así:

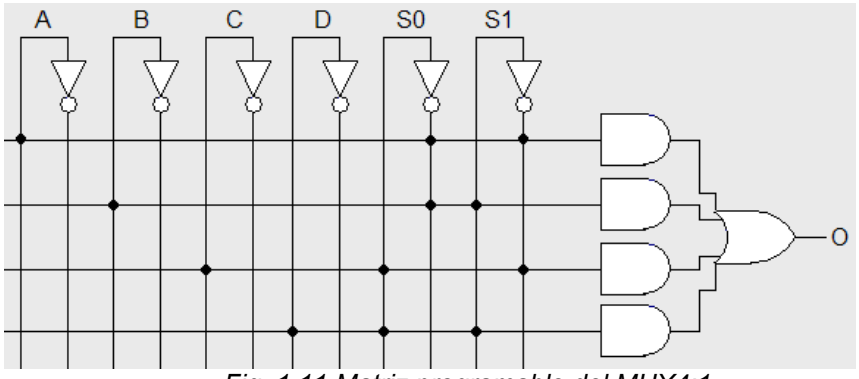


Fig. 1.11 Matriz programable del MUX4:1

Estos dos diseños pueden ser implementados con la misma matriz, suprimiendo los canales que no se usan, además. Pueden ser implementados juntos en la misma matriz si fuera necesario, es decir, lo que antes eran varios Dispositivos Simples ahora se pueden unir en uno sólo.

Eso es todo con respecto al funcionamiento y ventaja de diseño de las matrices programables; sin embargo, hasta este momento lo único que podemos hacer con estas matrices es simple diseño combinacional, pero una característica de un CPLD es que es capaz de implementar diseño combinacional y secuencial, lo que lo hace tan versátil. Lo que provee al CPLD de esta versatilidad es la OLMC.

1.2.2 OLMC – Output Logic Macrocell

Una OLMC se puede configurar como entrada o salida combinacional o como salida secuencial. En el modo secuencial, los pines que se definen como salidas secuenciales proceden de la salida Q de un flip-flop. Las OLMC se configuran internamente de forma automática, mediante la programación de un conjunto de celdas que están separadas de las celdas de la matriz lógica.

A continuación se muestra una OLMC, en este caso, de una GAL22V10, las entradas a la puerta OR de la OLMC procedentes de la matriz AND varían de diez a dieciséis. El resto de la lógica está dada en un flip-flop y dos multiplexores.

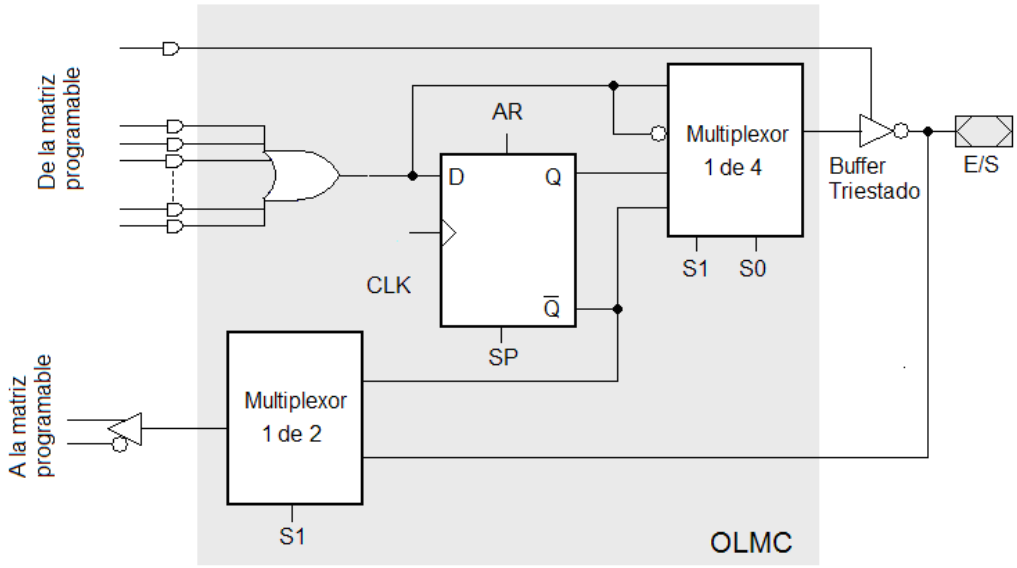


Fig. 1.12 Diagrama de una OLMC

El multiplexor 1 de 4 conecta su salida al buffer tri-estado en función del estado de las dos líneas de selección S0 y S1. Las entradas del multiplexor son la salida de la compuerta OR, su complemento es la salida Q del flip-flop y el complemento de ésta. Esto permite que la salida de la OLMC sea activa a nivel ALTO o a nivel BAJO en cada modo.

El multiplexor 1 de 2 conecta la salida del buffer tri-estado o la salida /Q del flip-flop a través de un buffer a la matriz AND en función del estado de S1. Los bits de selección, S0 y S1 para cada OLMC, se programan en un grupo dedicado de celdas de la matriz mediante el software de compilación, por lo que el usuario no puede manipular directamente esos bits.

El flip-flop que se utiliza es tipo D activado por flanco positivo. La entrada de reset asíncrona (AR) pone el flip-flop en el estado 0 lógico independientemente de la señal de reloj. La entrada de inicialización síncrona (SP) activa el flip-flop poniéndolo en estado 1, en el flanco de subida del impulso de reloj.

Las cuatro configuraciones de la OLMC son:

- Modo combinacional con salida a nivel BAJO.
- Modo combinacional con salida a nivel ALTO.
- Modo secuencial con salida a nivel BAJO.
- Modo secuencial con salida a nivel ALTO.

1.2.2.1. Selección del modo de la OLMC

En la OLMC de ejemplo, los modos se seleccionan con los bits S0 y S1, que se controlan mediante programación. En el modo combinacional S1S0=10 ó S1S0=11. Las figuras muestran los caminos lógicos a través de la OLMC para obtener una salida combinacional activa a nivel BAJO y a nivel ALTO, junto con el esquema lógico efectivo.

La inversión del multiplexor 1 de 4 y la cancelación del buffer de salida tri-estado dan lugar a la salida activa a nivel ALTO.

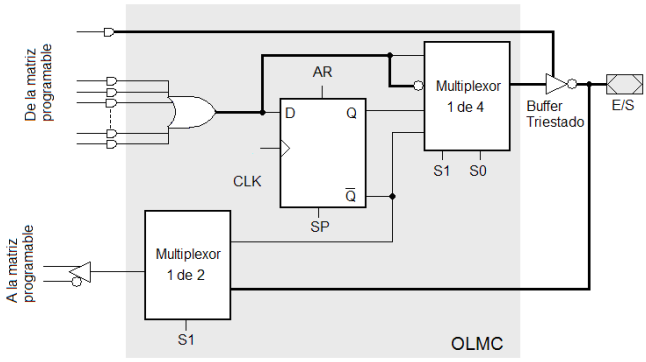


Fig. 1.13 OLMC en modo combinacional a nivel ALTO

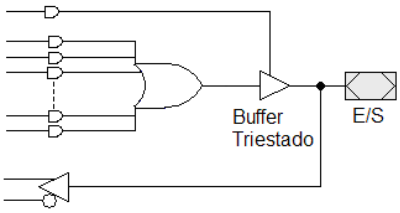


Fig. 1.14 Diagrama lógico efectivo

La inversión del buffer de salida tri-estado produce una salida activa a nivel BAJO.

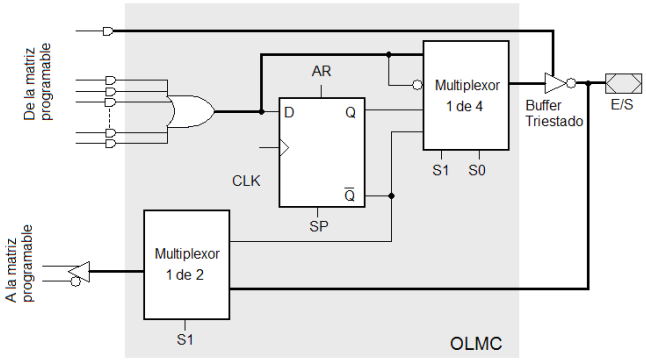


Fig. 1.15 OLMC en modo combinacional a nivel BAJO

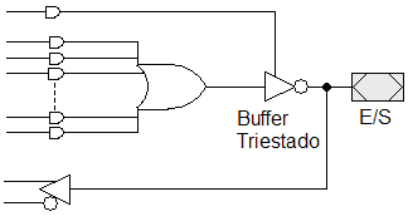


Fig. 1.16 Diagrama lógico efectivo

En el modo secuencial S1S0=00 ó S1S0 = 01. Las figuras siguientes muestran los caminos lógicos a través de la OLMC que proporcionan salidas secuenciales activas a nivel BAJO o a nivel ALTO, junto con la lógica efectiva de la OLMC para cada caso.

La inversión del flip-flop (Salida /Q) y la cancelación del buffer tri-estado dan a lugar una salida a nivel alto. Obsérvese que la realimentación a la matriz programable AND a través de un buffer se realiza a partir de la salida /Q del flip-flop y no del pin de salida como en el caso del modo combinacional, por lo que *una salida secuencial no puede utilizarse como entrada.*

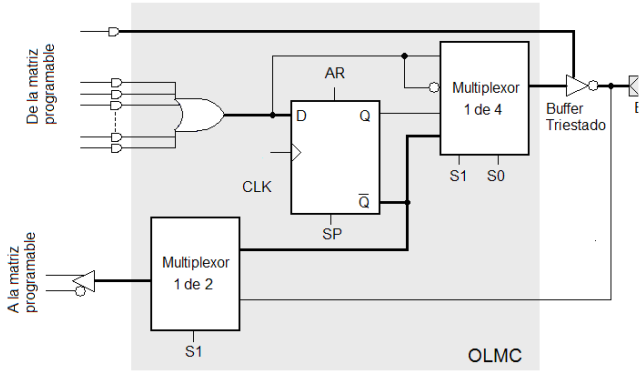


Fig. 1.17 OLMC en modo secuencial activo a nivel ALTO

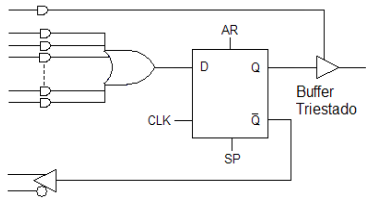


Fig. 1.18 Diagrama lógico efectivo

La inversión del buffer tri-estado produce una salida activa a nivel BAJO

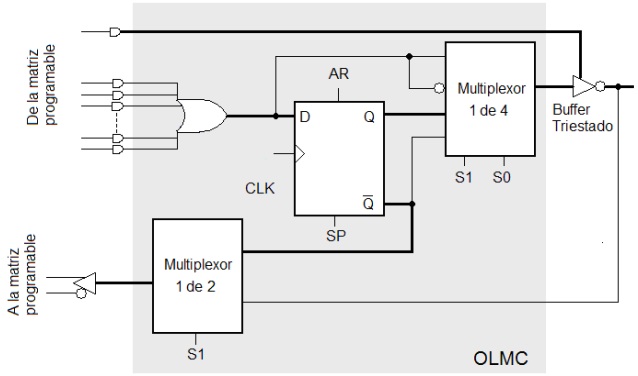


Fig. 1.19 OLMC en modo secuencial activo a nivel BAJO

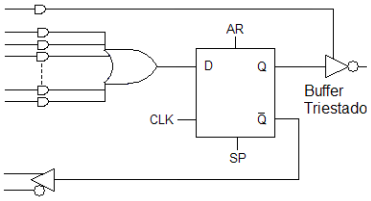


Fig. 1.20 Diagrama lógico efectivo

Finalmente, una macrocelda se obtiene cuando son conectadas las OLMC con las matrices programables. En el caso de los CPLDs de Xilinx, específicamente el caso del COOLRUNNER XPLA3 y COOLRUNNER-II, que son de bajo consumo y que cuentan con memoria EEPROM interna, se tienen las siguientes observaciones:

- Tienen hasta 384 macroceldas.
- Consumo en *StandBy* menor a 100μA.
- Consumo en funcionamiento inferior de más del 50% al de las tecnologías tradicionales.
- Tiempo de propagación de la señal de entrada a la salida predecible.
- Configuración JTAG[4].
- Excelentes posibilidades de Pin Locking.
- Alimentación 3.3V compatibles con entradas y salidas 5V.

[4] Una interfaz JTAG es una interfaz especial de cuatro o cinco pines agregadas a un chip, diseñada de tal manera que varios chips en una tarjeta puedan tener sus líneas JTAG conectadas y solo necesite conectarse a un "puerto JTAG" para acceder a todos los chips en un circuito impreso.

Conclusión

Conocer la arquitectura de los CPLD nos pone en contexto para aprender cómo se programan, es preciso conocer la arquitectura que uno va a programar, estar consciente de sus ventajas y de sus limitaciones.

Por medio de la historia e los CPLD hemos visto en qué principios y componentes está basado su funcionamiento, el desarrollo de la OLMC es fundamental para que los CPLD puedan realizar sus tareas, así como poder cambiar entre modos combinacionales y secuenciales.

2. VHDL y las herramientas de trabajo

El presente capítulo trata de las formas existentes para programar un CPLD o un FPGA, así como de las herramientas de trabajo de la marca Xilinx.

La mayoría de las empresas como XILINX, ALTERA, ACTEL, etc; dentro de sus ambientes de desarrollo además de un procesador de texto, incluyen un simulador gráfico.

Los objetivos de esta sección son los siguientes:

- Conocer las múltiples posibilidades ofrecidas por el lenguaje VHDL.
- Conocer la sintaxis y el juego de instrucciones utilizado en síntesis lógica VHDL.
- Conocer las principales ventajas y limitaciones de los diferentes estilos de escritura.
- Verificar las informaciones teóricas para la práctica.
- Aprovechar una base de ejemplos concretos.

2.1 VHDL

VHDL es la abreviatura de VHSIC *Hardware Description Language*, y VHSIC de sus siglas (*Very High Speed Integrated Circuit*), es un lenguaje de descripción de hardware, incorrectamente tratado como lenguaje de programación, pues su objetivo consiste en describir el funcionamiento y estructura de sistemas electrónicos, ASIC's, FPGA's y sistemas lógicos convencionales.

VHDL es un lenguaje de alto nivel conocido por permitir:

- El ensamble de modelos lógicos complejos.
- La simulación de los modelos de los componentes.
- La síntesis lógica.

2.1.1 Breve historia de VHDL

La necesidad de contar con un lenguaje como este se dio en 1981 cuando el Departamento de Defensa de los Estados Unidos debido a la crisis del ciclo de vida del hardware. El costo de reproducir hardware electrónico cuando las tecnologías se volvían obsoletas alcanzaba el punto de crisis, porque el funcionamiento de las partes no estaba debidamente documentado, y los varios dispositivos con que se contaba eran verificados individualmente usando una amplia gama de herramientas, lenguajes y simuladores incompatibles.

Lo que se requería era un lenguaje con un amplio rango de capacidad descriptiva, que pudiera trabajar con el mismo simulador o con cualquier otro para cualquier tecnología y tipo de metodología.

El proceso de estandarización fue buscado desde etapas tempranas por la industria, y dos años antes de tener el lenguaje estandarizado se lanzó un lenguaje previo (Baseline V7.2), todos los derechos del lenguaje fueron dados por el Departamento de Defensa (DoD) a IEEE para animar la aceptación de la industria. Fue el primer estándar de VHDL publicado, en 1987.

Una vez como estándar IEEE, VHDL pasó a revisión durante 5 años antes de ser entregado a la industria, la primera versión se completó en 1993, las herramientas de VHDL'93 ahora están disponibles.

2.1.2 Ensamble de modelos lógicos complejos

VHDL es un lenguaje de alto nivel que facilita la descripción del comportamiento de los modelos, que ofrece un fuerte nivel de abstracción, puesto que la posibilidad de implementar físicamente un diseño no es necesariamente tomada en cuenta a este nivel; además, permite la partición del diseño en varios elementos que permite dividir un diseño complejo en varios que resulten más simples de desarrollar de manera separada.

2.1.3 Simulación

Los modelos de comportamiento creados con VHDL pueden ser simulados con el fin de verificar su coherencia. El conjunto de modelos de comportamiento y los archivos de simulación constituyen a la vez una especificación y un medio de verificación. El comportamiento de un modelo y de su realización física deberán ser idénticos.

2.1.4 Síntesis lógica

Permite implementar físicamente un diseño, gracias a la utilización de herramientas de síntesis.

Usa solamente una parte reducida del juego de instrucciones de VHDL.

2.2 VHDL – Revisión General

- En VHDL el conjunto lógico descriptivo se conoce como Entidad *"Entity"*.
- VHDL permite definir los puertos *"Ports"* de entrada y salida de una Entidad mediante la definición de señales simples o buses.
- Definir un modelo de comportamiento sintetizable, llamado Arquitectura *"Architecture"* usando el juego de instrucciones soportado por la herramienta de síntesis.
- Unir diferentes módulos descritos separadamente (VHDL estructural)
- Estilo de escritura RTL *"Register Transfer Logic"*

2.3. Información sobre la sintaxis:

Las mayúsculas y minúsculas pueden se utilizadas indistintamente en VHDL, un objeto o palabra clave bien podría llamarse: MODULE, Module o module

Los comentarios se hacen mediante dos guiones medios seguidos, siendo comentario todo lo que se encuentre después de los guiones hasta el final de la línea del comentario.

Ejemplo:

```
--Esto es un comentario
architecture ARQUI of EJEMPLO is --Esto es otro comentario
begin
```

Los archivos de código VHDL usan la extensión VHD

Ejemplo: Ejemplo.vhd

2.3.1. La pareja Entidad/Arquitectura

Dos de las unidades básicas de diseño más importantes en VHDL son la pareja "Entidad/Arquitectura"

Entidad. Una entidad es una porción del código en la que se definen las entradas y salidas, su sintaxis es la siguiente:

```
entity NOMBRE is
port(            --lista de puertos de entrada y salida:
nombre de señal: modo y tipo;
);
end [NOMBRE];  --Se usan corchetes para indicar la
--opcionalidad de repetir el nombre de la entidad
```

El nombre dado a una entidad puede ser cualquiera, exceptuando las palabras reservadas del lenguaje. Se recomienda dar el mismo nombre a la entidad y al archivo VHDL.

La lista de puertos deberá estar comprendida entre dos paréntesis seguidos de un punto y coma.

Ejemplo: Plantear la declaración de la entidad del siguiente circuito:

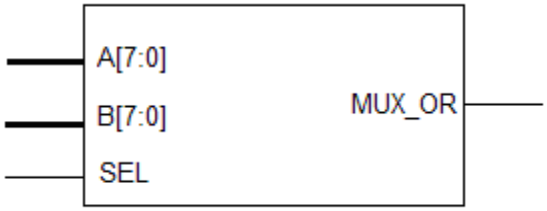


Fig. 2.1 Símbolo de ejemplo

```
entity EJEMPLO is port(
A, B: in bit_vector(7 downto 0);
S: in bit;
MUX_OUT: out bit_vector(7 downto 0);
MUX_OR: out bit);
end EJEMPLO;
```

Arquitectura. La arquitectura es la porción del código en la cual se hace la descripción del dispositivo a sintetizar.

Un requisito indispensable para una arquitectura, es que debe estar asociada a una entidad dentro del mismo archivo, por esto que se manejan como pareja.

La arquitectura está compuesta de dos partes: Declaratoria y Operatoria. Siguiendo con el ejemplo anterior, se declarará la arquitectura que describa el funcionamiento del mismo multiplexor:

```
architecture ARQUI of EJEMPLO is
--Parte declaratoria
signal MUX_OUT: bit_vector(7 downto 0); --Señal interna
-------------------------------------------------------
--Parte operatoria
begin
MUX_OUT<= A when SEL='0' else B;
MUX_OR <= '1' when MUX_OUT /= "00000000" else '0';
end ARQUI;
```

La Arquitectura puede contener los siguientes componentes complementarios en su parte Declaratoria:

- Declaración de Señales internas.
- Declaraciones de componentes.
- Declaraciones de constantes.
- Declaraciones de tipos de objetos.
- Declaraciones de subprogramas.

El código Operatorio puede contener:

Código RTL

Asignaciones concurrentes de señales. Los resultados de síntesis y simulación son independientes del orden de escritura de las instrucciones, como en el ejemplo precedente.

Asignaciones secuenciales: En simulación, ejecución de instrucciones en el orden de escritura (secuencialmente). En síntesis lógica, el mismo comportamiento es reproducido.

Código Estructural

Instanciación de cajas negras (Componentes): El módulo en curso de desarrollo puede llamar a otros sub-módulos (VHDL Estructural).

Resultados de síntesis:

```
architecture ARQUI of EJEMPLO is
--Parte declaratoria
signal MUX_OUT: bit_vector(7 downto 0); --Señal interna
-------------------------------------------------------
--Parte operatoria
begin
MUX_OUT<= A when SEL='0' else B;
-- Asignaciones condicionales de señales con "When -Else"
MUX_OR <= '1' when MUX_OUT /= "00000000" else '0';
end ARQUI;
```

Este código se comporta como lo haría el siguiente circuito:

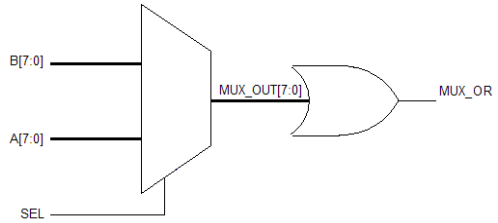


Fig. 2.2 Circuito lógico de ejemplo

2.3.2. Conjunción Entidad/Arquitectura

A continuación se muestran dos ejemplos de diseños descritos en VHDL mediante la implementación de la pareja Entidad/Arquitectura, nótese que el nombre del archivo referido es el mismo que el nombre de la entidad, y que la segunda arquitectura no contiene parte declaratoria:

EJEMPLO.vhd

```
entity EJEMPLO is port(
A, B: in bit_vector(7 downto 0);
S: in bit;
MUX_OUT: out bit_vector(7 downto 0);
MUX_OR: out bit);
end EJEMPLO;
architecture ARQUI of EJEMPLO is
signal MUX_OUT: bit_vector(7 downto 0);
begin
MUX_OUT<= A when SEL='0' else B;
MUX_OR <= '1' when MUX_OUT /= "00000000" else '0';
end ARQUI;
```

MUX2_1.vhd

```
entity MUX2_1 is port(
A_IN, B_IN, SEL: in bit;
SALIDA: out bit);
end;
architecture ARCHI of MUX2_1 is
begin
SALIDA <= (A_IN and not(SEL)) or (B_IN and SEL);
end;
--LOS OPERADORES AND, NOT y OR
--SON PROPIOS DEL LENGUAJE VHDL
```

Objetos que se pueden manipular en VHDL

En el lenguaje VHDL se pueden manipular 3 clases principales de objetos:

- Señales: Son similares a las señales encontradas en los esquemas, los PORTS declarados dentro de una entidad son señales. Las señales pueden también ser declaradas como BUS.
- Constantes: Permiten definir valores permanentes.
- Variables: Son utilizadas en los PROCESS ("Instrucciones Secuenciales").

La declaración de cualquiera de los objetos antes mencionados comprende:

- Clase: Señal, constante o variable.
- Nombre: A elección del creador, con la condición de que no se utilicen palabras reservadas.
- Modo: Esto aplica sólo a las señales, y se refiere a IN, OUT, INOUT, BUFFER.
- Tipo: Bit, Bit_Vector, Bolean, Integer…

2.3. Tipos de datos

Dentro de los tipos de objetos que se pueden manipular en VHDL, existen algunos tipos predefinidos, dentro de los cuales, los principales son:

- **BIT**: Puede tomar el valor de '0' ó '1'.
- **BIT_ VECTOR**: Es un grupo de bits (Bus).

Ejemplo:

```
Signal A: bit_vector(7 downto 0);
A<="01011010"; --Equivalente a a<=X"5A";
--X"valor" indica que se trata de un valor hexadecimal
```

- **BOOLEAN:** Puede tomar los valores TRUE o FALSE.
- **INTEGER:** Valor entero codificado sobre 32 bits. (de -2'147'483'648 a +2'147'483'647)

Un entero puede estar limitado en su declaración, a fin de evitar su codificación sobre 32 bits

Ejemplo:

```
signal VALOR: integer range 0 to 255;
begin
VALOR <=143 when init = '1' else 33;
```

- **TIPOS "User Defined": STD_LOGIC y STD_ULOGIC: Extensiones del tipo BIT, pueden tomar 9 valores diferentes:**

```
Type STD_ULOGIC is(
'U', --Uninitialized
'X', --Forcing Unknown
'0', --Forcing 0
'1', --Forcing 1
'Z', --High Impedance
'W', --Weak Unknown   _
'L', --Weak 0          |
'H', --Weak 1          |> No soportados en simulador
'-', --Don't care     _|
);   --Extraído del código fuente del paquete
    --"STD_LOGIC_1164".
```

STD_LOGIC: Este tipo de valor da una mayor potencia operacional que el tipo BIT, tanto para la simulación como para la síntesis. (Particularmente los valores 'Z' y '-' para la síntesis)

STD_LOGIC_VECTOR: Constituye un grupo de objetos similar al BIT_VECTOR, pero con los 9 estados posibles del STD_LOGIC para cada uno de los bits. Para utilizar estos tipos de datos (STD_LOGIC y STD_LOGIC_VECTOR), debemos declarar la utilización de la biblioteca IEEE que contiene el paquete particular (STD_LOGIC_1164), en el encabezado del archivo .VHD.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
Ejemplo de declaración de la biblioteca IEEE y del paquete
STD_LOGIC_1164:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity EJEMPLO is port(
A, B: in STD_LOGIC_VECTOR(7 downto 0);
SEL: in STD_LOGIC;
MUX_OR: out STD_LOGIC);
end;
```

Tipos enumerados: Principalmente usados para definir los estados de las máquinas de estado.

```
architecture ARCHI of MACHINE is
--Parte declaratoria de la arquitectura
type ESTADOS is (REPOSO, LECTURA, ESCRITURA);
signal ACTUAL, SIGUIENTE: ESTADOS;
--Las señales ACTUAL y SIGUIENTE podrán tomar los
--valores "REPOSO, "LECTURA" o escritura.
begin
--asignaciones
```

2.3.1. Reglas de asignación de los vectores de datos

- El orden en el cual se utiliza el vector (bus) debe ser el mismo que en la declaración del vector. (Valores crecientes o decrecientes en los índices).
- No es necesario utilizar el vector entero. (Utilización de una parte de la señales del bus).
- El ancho del bus (tamaño del vector) debe corresponder para la mayoría de las operaciones. (Excepto para comparación).

Ejemplo de la asignación de vectores:

```
Architecture ARCHI of VECTOR is
--Parte declaratoria de la arquitectura
signal D_IN, MASCARA, D_OUT: std_logic_vector(7 downto 0);
signal Q_OUT: std_logic_vector(7 downto 0);
constant FIJA: std_logic_vector(2 downto 0):= "010";
--La asignación de un valor a una constante o variable
--Se realiza mediante el símbolo ":=" en lugar del "<="
--acostumbrado para la asignación de señales
begin
  D_OUT<=D_IN and not(MASCARA); --Operaciones con buses,
     --todas son sobre 8 bits
  Q_OUT<=(D_IN(6 downto 2)) and not(MASCARA(7 downto 3)))
          & FIJA;
--El signo "&" es un operador llamado de concatencación;
end;
```

Ejemplo de un código equivalente al anterior:

```
D_OUT(7) <= D_IN(7) and not(MASCARA(7));
D_OUT(6) <= D_IN(6) and not(MASCARA(6));
D_OUT(0) <= D_IN(0) and not(MASCARA(0));
Q_OUT(7) <= D_IN(6) and not(MASCARA(7));
Q_OUT(6) <= D_IN(5) and not(MASCARA(6));
Q_OUT(3) <= D_IN(2) and not(MASCARA(3));
Q_OUT(2) <= FIJA(2); -- ó Q_OUT(2) <= '0';
Q_OUT(1) <= FIJA(1); -- ó Q_OUT(1) <= '1';
Q_OUT(0) <= FIJA(0); -- ó Q_OUT(0) <= '0';
```

Ejemplo común de una asignación incorrecta usando vectores:

```
Signal D_IN, MASCARA, D_OUT : std_logic_vector(7 downto 0);
begin
D_OUT<= D_IN(0 to 7) and not(MASCARA);
```

2.4. Operadores Lógicos de VHDL

Los operadores lógicos predefinidos de VHDL son: ***AND, OR, NAND, NOR, XOR, NOT,*** operan sobre los objetos de todas las clases (señales, clases y variables) y de tipo:

- bit
- bit_vector
- std_logic
- std_ulogic
- std_logic_vector
- std_ulogic_vector
- boolean

Los operandos siempre deben ser del mismo tipo y deben contener el mismo número de bits.

Ejemplo de utilización con bits simples:

```
entity OPE is port(
A, B, C: in bit;
s: out bit);
end;
architecture ARCHI of OPE is
```

```
begin
S<= (A and B) and not(C);
end;
```

Ejemplo de utilización con vectores:

```
library ieee;
use ieee.std_logic_1164.all;
entity OPE is port(A, B, C: in std_logic_vector(0 to 3);
s: out std_logic_vector(3 downto 0));
end;
architecture ARCHI of OPE is begin
S<= (A and B) and not(C);
end;
```

Cabe mencionar de manera más detallada la forma en que los operadores lógicos actúan sobre los vectores con un ejemplo equivalente:

Ejemplo de utilización con vectores:

```
library ieee;
use ieee.std_logic_1164.all;
entity OPE is port(A, B, C: in std_logic_vector(0 to 3);
s: out std_logic_vector(3 downto 0));
end;
architecture ARCHI of OPE is begin
S(0)<=(A(3) and B(3)) and not(C(3));
S(1)<=(A(2) and B(2)) and not(C(2));
S(2)<=(A(1) and B(1)) and not(C(1));
S(3)<=(A(0) and B(0)) and not(C(0));
end;
```

Los operadores lógicos pueden ser utilizados para describir arquitecturas cuyo funcionamiento se representa mediante las ecuaciones booleanas.

Ejemplos:

Circuito lógico simple con compuertas NOR

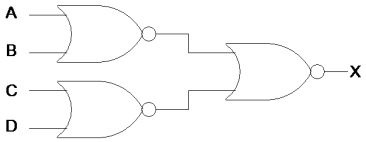


Fig. 2.3 Circuito lógico con compuertas NOR

```
library ieee;
use ieee.std_logic_1164.all;
entity EjeNOR is port(A, B, C, D: in std_logic;
X: out std_logic);
end;
architecture ARCHI of EjeNOR is begin
X<=(A NOR B) NOR (C NOR D);
end;
```

Circuito lógico simple con compuertas OR Y NAND

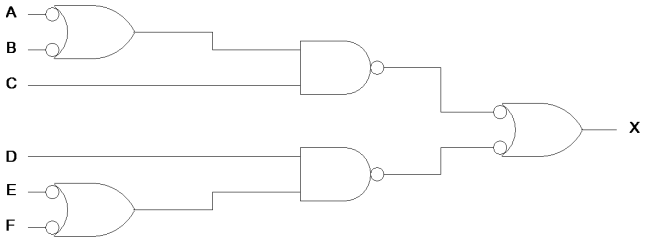


Fig. 2.4 Circuito lógico

```
library ieee;
use ieee.std_logic_1164.all;
entity Eje_OR_NAND is port(A, B, C, D, E, F: in std_logic;
X: out std_logic);   end;
architecture ARCHI of OPE is begin
X<= NOT( (NOT A OR NOT B) NAND C) OR NOT(D NAND (NOT E OR NOT
F));
end;
```

Tabla de verdad del sumador completo:

Entrada			Salida	
Cn	A	B	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

```
library ieee;
use ieee.std_logic_1164.all;
entity Tabla is port(Cn, A, B: in std_logic;
S, C: out std_logic);
end;
architecture ARCHI of Tabla is begin
S<= (NOT Cn AND (A XOR B) )OR (Cn AND (A XNOR B));
C<= (A AND B) OR (CN AND (A XOR B));
end;
```

2.5. Operadores relacionales

Los operadores relacionales de VHDL son:

- = (igual a).
- < (inferior a).
- > (superior a).
- /= (diferente de).
- <= (inferior o igual a).
- >= (superior o igual a).

Estos operan sobre objetos de tipo:

- bit, bit vector.
- std_logic, std_logic_vector.
- std_ulogic, std_ulogic_vector.
- Integer.
- Boolean.

Los operandos deben ser del mismo tipo, pero el número de bits comparados puede ser diferente.

La comparación de vectores se realiza a partir del bit más significativo (MSB), es importante tener esto en cuenta, pues los resultados pueden ser sorprendentes al momento de comparar vectores de diferente tamaño:

```
signal REG: std_logic_vector(4 downto 0);
signal CNT: std_logic_vector(3 downto 0);

 REG       CNT
"01111" > "0100"  --Esto resulta muy lógico, pero
"01111" < "1000"  --Esto resulta muy sorprendente.
```

Ésta es la razón por la que se recomienda hacer comparaciones con vectores que contengan el mismo número de bits.

La utilización de paquetes estandarizados hace posible que se efectúen operaciones relacionales entre objetos de tipo INTEGER y otro de tipo STD_LOGIC_VECTOR.

Información complementaria sobre los operadores relacionales

El resultado de una comparación es de tipo boolean, eso significa que puede tomar sólo los valores TRUE o FALSE.

Ejemplo

```
signal IGUAL, VENTANA: boolean;
signal COUNT: integer range 0 to 31;
begin
IGUAL <= (COUNT = 27);
VENTANA <= (COUNT >= 13) AND (COUNT <= 25);
--Operación Lógica entre dos Booleans
```

Nótese que los operadores lógicos relacionales distintos de "=" y "/=" son frecuentemente implementados en FPGA en forma de funciones aritméticas cableadas.

La comparación (distinta de "=" y "/=") de vectores puede resultar en la utilización de funciones aritméticas cableadas y por esto no serán optimizadas después de la

síntesis. Se recomienda ser prudente en la utilización de <, <=, >, >=. Dos porciones de código pueden tener un comportamiento equivalente, pero los resultados de síntesis pueden ser diferentes.

Ejemplo:

```
signal CNT: std_logic_vector(7 downto 0);
begin
VAL <= '1' when CNT<"10000000" else '0';
--En lugar de esto podemos escribir
--VAL <= '1' when CTN(7)='0' else '0';
--o más simple aún:
--VAL <= not CNT(7);
```

2.6. Operadores Aritméticos

Los operadores aritméticos de VHDL son:

- + (suma)
- * (multiplicación)
- ** (potencia)
- - (resta)
- / (división)
- mod (xxxxx)
- rem (xxxxx)
- abs (xxxxxxx)

Estos operadores son utilizados con objetos de tipo INTEGER, pero pueden igualmente operar sobre STD_LOGIC_VECTOR utilizando los paquetes STD_LOGIC_UNSIGNED y STD_LOGIC_ARITH de la IEEE.

Restricciones: la mayoría de las herramientas de síntesis solo autorizan las operaciones de multiplicación y división entre *constantes* o una *constante potencia de 2* y una *señal*.

Algunas arquitecturas de componentes programables tienen recursos de lógica específicos para una implementación eficaz de las funciones aritméticas y similares (en términos de velocidad y superficie de silicio usada)

Las herramientas de síntesis pueden hacer un uso de estos recursos más o menos oportuno según su conocimiento de la arquitectura del circuito usado. Disponen en general de opciones de síntesis, para elegir usar o no dichos recursos lógicos.

La comparación entre STD_LOGIC_VECTOR e INTEGER, es posible usando ciertos paquetes. Los nombres de la biblioteca y del paquete pueden ser diferentes según las herramientas.

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

entity ejemplo is port(TITI, TOTO: out std_logic);
end;

architecture Arqui of ejemplo is
signal CNT: std_logic_vector(7 downto 0);
begin
TOTO <= '1' when (CNT = 143) else '0';
TITI <= '1' when (CNT >= 75) and (CNT<=123) else '0';
end;
```

2.7. Operador de concatenación “&”

Permite la creación de vectores a partir de bits simples o de vectores.

Ejemplo

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity ejemplo is port(A, B, C, D: in std_logic;
STATUS: out std_logic_vector(7 downto 0));
end;

architecture Arqui of ejemplo is begin
STATUS <= "0000" & A & B & C & D;
--Equivalente a STATUS(7 downto 4) <= "0000";
--                STATUS(3) <= A; STATUS(2) <= B;
--                STATUS(1) <= C; STATUS(0) <= D;
end;
```

Otro ejemplo

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity ejemplo is port(A, B, C, D: in std_logic;
STATUS: out std_logic_vector(7 downto 0));
end;
architecture Arqui of ejemplo is
constant ZERO: std_logic_vector(3 downto 0):=”0000”;
begin
STATUS <= ZERO & A & B & C & D;
--Equivalente a STATUS(7 downto 4) <= "0000";
--                STATUS(3) <= A; STATUS(2) <= B;
--                STATUS(1) <= C; STATUS(0) <= D;
end;
```

2.8. Instrucciones Concurrentes y Secuenciales

Todas las operaciones se efectúan al mismo tiempo (en paralelo).

El orden de escritura de las instrucciones no afecta el resultado de síntesis o de simulación.

El estilo de escritura que se usa es comparable a los lenguajes de programación de los PALs (Ecuaciones Lógicas)

Ejemplo

```
entity CONCURRENTE i sport(A, B, C: in bit;
S, T: out bit );
end;

architecture ARQUI of CONCURRENTE is begin
S <= (A and B) and not(C);
T <= B xor C;
End Arqui;
```

Nótese que el orden de asignación de S y de T bien pudo haberse invertido y esto no habría modificado el resultado.

Ambas operaciones se realizan en modo combinacional y se realizan en paralelo.

2.9. Instrucciones Secuenciales

Las operaciones descritas como instrucciones secuenciales se realizan siguiendo una secuencia. En este caso, el orden si afecta los resultados de simulación y de síntesis. El estilo de escritura es parecido a los lenguajes informáticos de alto nivel.

Las instrucciones secuenciales se usan en partes específicas del código: PROCESS o subprogramas.

Ejemplo

```
arquitecture ARQUI of SECUENCIA is begin
process begin
 wait until CK'event and CK='1';
      if ENA='1' then COUNT <= COUNT + 1;
      end if;
 end process;
end;
```

CK'event está dado por la capacidad del CPLD y el FPGA de Xilinx para reconocer cuando se da un cambio en una señal, como es el paso de 0 a 1 y viceversa.

ENA = '1' se refiere a la utilización del Clock_Enable dedicado de los Flip Flops Xilinx.

2.10. Instrucciones Concurrentes

Asignación de señales por condición WHEN – ELSE

Sintaxis

signal_x <= valor_x when signal_y = valor_y else valor_z;

La sintaxis anterior debe entenderse de la siguiente manera:

Asignar a signal_x el valor valor_x cuando signal_y es igual a valor_y o de lo contrario, asignar el valor valor_z;

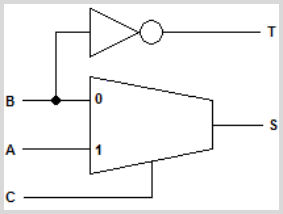
Nótese que el valor a escribir en caso "else" se escribe directamente sin el símbolo de asignación "<=".

Ejemplo:

```
entity CONCURRENTE i sport(A, B, C: in bit;
S, T: out bit);
end;
architecture ARQUI of CONCURRENT is begin
S <= A when C='1' else B;
T <= B xor C;
end ARQUI;
```

Ejemplo y figura:

```
entity CONCURRENTE i sport(A, B, C: in bit;
S, T: out bit);
end;
architecture ARQUI of CONCURRENT is begin
S <= A when C='1' else B;
T <= not B;
end ARQUI;
```

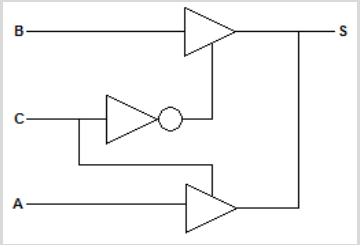


Estas instrucciones generan un circuito que se comporta de la siguiente forma:

Se observa porqué el orden de escritura no importa y porqué se dice que las operaciones se efectúan en paralelo.

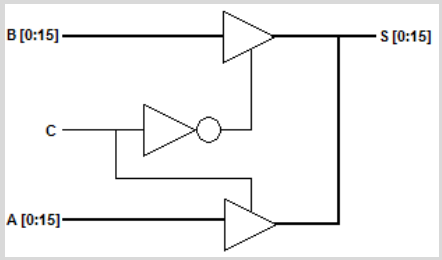
Ejemplo y figura

```
entity TRIESTADO i sport(
A, B, C: in std_logic;
S: out std_logic);
end;
architecture ARQUI of TRIESTADO is
begin
S <= A when C='0' else 'Z';
S <= B when C='1' else 'Z';
end;
```



Ejemplo y figura:

```
entity TRIESTADO i sport(
A, B, C: in std_logic_vector(0 to 15);
S: out std_logic_vector(0 to 15));
end;
architecture ARQUI of TRIESTADO is
begin
S <= A when C='0'
else "ZZZZZZZZZZZZZZZ";
S <= B when C='1'
else "ZZZZZZZZZZZZZZZ";
end;
```



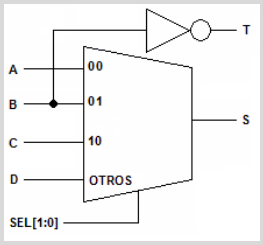
Asignación de señales por condición WITH – SELECT

Sintaxis: `when others` indica que señal_y tomará el valor valor_y para todo valor de señal_x que no haya sido indicado explícitamente.

Ejemplo y figura:

arquitecture ARQUI of CONCURRENTE is

```
 begin
 with SEL select
   s <= A when "00",
        B when "01",
        C when "10",
        D when others;
   T <= not B;
end;
```



Instrucción de bucle FOR i IN … GENERATE:

Sintaxis:

```
LABEL: for I in entero_a to entero_b generate
 --instrucciones concurrentes…
end generate;
```

GENERATE produce una iteración de las instrucciones concurrentes contenidas en el bucle. En número de iteraciones está determinado por los valores enteros “entero_a” y “entero_b”

La utilización de una etiqueta (LABEL) es obligatoria.

La instrucción GENERATE puede ajustarse al ancho de un bus automáticamente si se usa en lugar de los enteros el atributo RANGE:

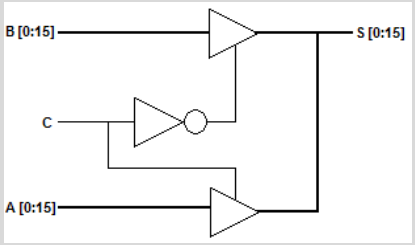
```
ETIQUETA: for I in DATA’range generate …
```

Range se refiere a la manera en que fue definido el Bus, o sea, un rango del tipo: (7 downto 0) ó (15 downto 0).

Ejemplos:

```
CICLO: for I in 0 to 15 generate
S(I) <= A(I) when C=’0’ else ‘Z’;
S(I) <= B(I) when C=’1’ else ‘Z’;
end generate;

CICLO: for I in S’range generate
S(I) <= A(I) when C=’0’ else ‘Z’;
S(I) <= B(I) when C=’1’ else ‘Z’;
end generate;
```



Range es un atributo predefinido de VHDL. Indica la gama de valores de los índices de un vector de datos. Es un código genérico, esto quiere decir que es totalmente independiente del ancho del bus.

Permite gran portabilidad, re-uso y facilita evoluciones, puesto que si se cambia el ancho del bus para reutilizar el diseño, no se necesita cambiar ésta línea de código para ajustarla al nuevo bus.

2.3 Herramientas de trabajo, el entorno de desarrollo ISE

Una vez que hemos visto las bases del funcionamiento del CPLD, tenemos que avocarnos a la tarea de desarrollar aplicaciones para este CPLD, debido a su gran versatilidad, podemos crear sistemas completos en un sólo CPLD valiéndonos de paradigmas de diseño Combinacional y/o Secuencial.

La herramienta de software que nos permitirá programar el CPLD mediante el lenguaje VHDL es el ISE 8 de Xilinx; en el cual escribiremos el código, lo compilaremos para posteriormente pasar a la fase de simulación en la cual se verificará el correcto funcionamiento del diseño antes de grabarlo en el CPLD.

Una vez que se haya simulado, el CPLD puede ser configurado para seleccionar los pines que servirán como entradas y salidas, esto también se hará mediante el software ISE, así como la misma grabación del CPLD físicamente, ISE es una herramienta muy completa.

El proceso general para implementar un diseño en un CPLD es el siguiente:

i. Escribir el código.

ii. Compilar el código.

iii. Simular el funcionamiento del diseño.

iv. Grabar el CPLD.

2.2.1 Escribiendo el código

Lo primero es iniciar el software ISE, mediante la siguiente ruta: *Menú Inicio/Todos los programas/Xilinx ISE 8/Project Navigator*.

Aparecerá la siguiente ventana después de mostrar *"The tip of the day"*:

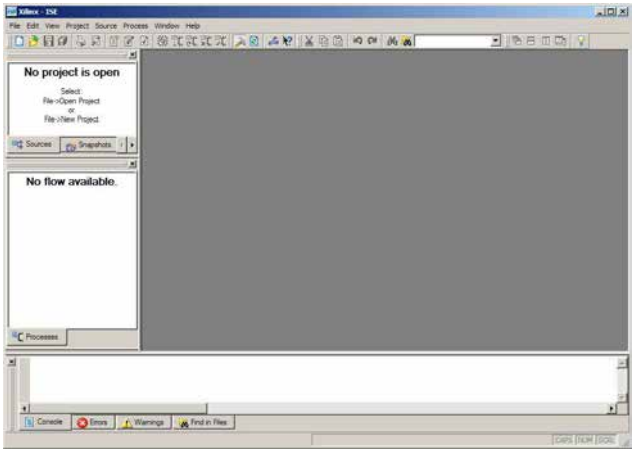


Fig. 2.5 Pantalla inicial de Xilinx ISE

Se pueden notar fácilmente las barras de menú y los controles principales, de entrada el software ISE nos sugiere que iniciemos un nuevo proyecto, para lo cual se hace click en el botón que tiene una hoja blanca en la parte superior izquierda, o mediante el menú *File/New Project*. Después aparecerá una ventana como la siguiente:

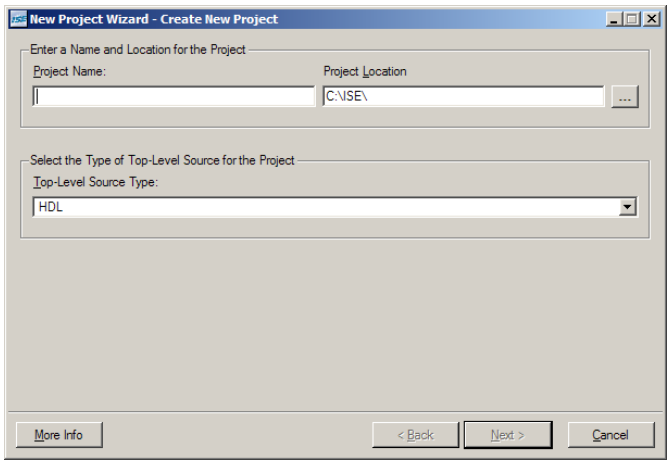


Fig. 2.6 Asistente de nuevo proyecto

Colocaremos el nombre del proyecto, que en realidad, puede ser cualquiera, también se puede seleccionar la carpeta en la que se guardará el proyecto.

ISE nos pedirá a continuación que le indiquemos el tipo de dispositivo que vamos a utilizar, es decir si se trata de un CPLD o un FPGA, en todo este curso nos basaremos en el CPLD XC2C256 de la familia *CoolRunner II* que es relativamente pequeño.

Por lo tanto tenemos que indicar a ISE la familia del producto, su número, el tipo de paquete y la velocidad a la que trabajará, así como la herramienta de síntesis, entre otras cosas que analizaremos más adelante.

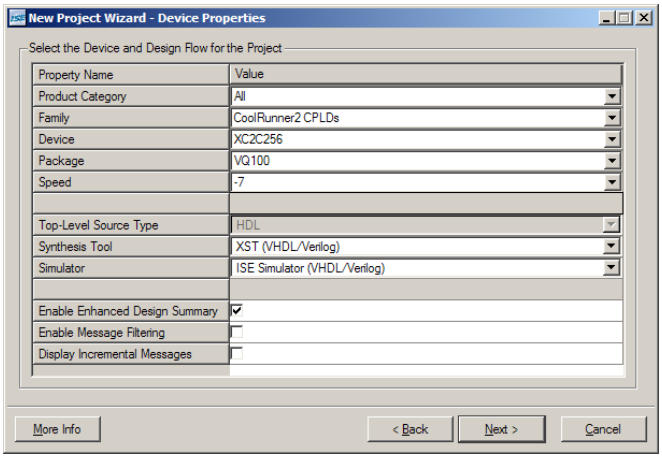


Fig. 2.7 Asistente de nuevo proyecto, propiedades del dispositivo

Después de esto ISE nos deja añadir los archivos de nuestro proyecto y es entonces cuando añadiremos nuestro *VHDL Module* con su respectivo nombre.

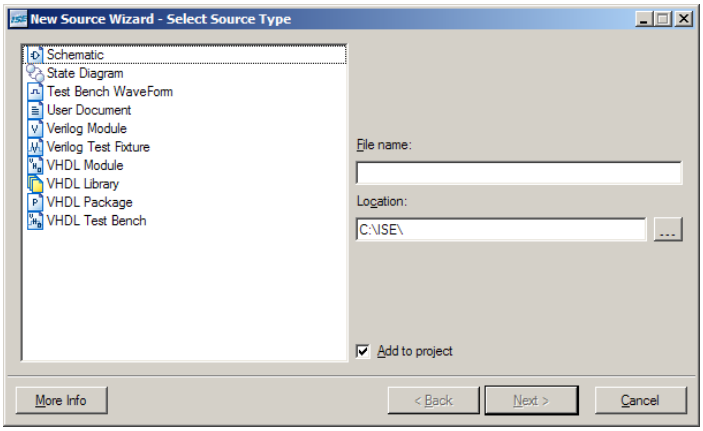


Fig. 2.8 Asistente de nuevo proyecto

Cuando terminemos con todas las actividades por las que nos va llevando este asistente de nuevo proyecto, por fin estaremos listos para empezar a capturar el código VHDL de nuestro diseño.

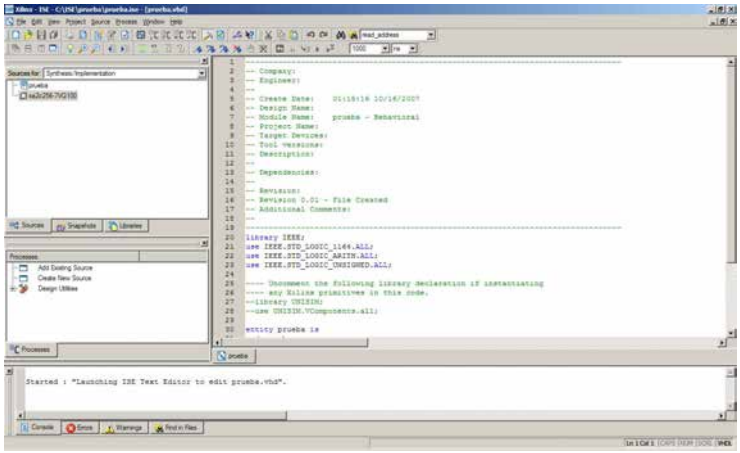


Fig. 2.9 Código VHDL en Xilinx ISE

La pantalla de ISE mostrará los componentes de nuestro diseño, así como las herramientas y los procesos de los que dispondremos para trabajar. ISE siempre nos mantendrá informados de todo lo que vaya resultando de cada proceso que hagamos, y también si hubo algún error o alguna advertencia que tengamos que considerar.

Por ahora no ahondaremos más en cada una de las opciones de ISE, pues lo haremos sobre la marcha en la primera práctica.

2.2.2 Compilando el código

Una vez que se ha escrito el código se procede a compilarlo, se puede hacer de manera general mediante la opción *Implement Design* que aparece en la ventana de *Processes*, misma que puede ser observada a continuación:

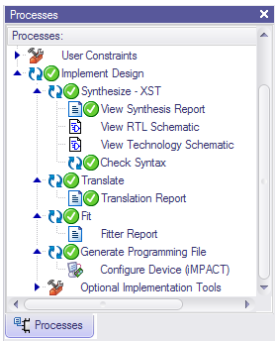


Fig. 2.10 Ventana "Process"

Lo que hará esta opción es llevar a cabo toda la implementación del diseño desde comprobar la sintaxis del código hasta generar el archivo que ha de ser cargado al CPLD, cada paso siempre después de concluir con el anterior; sin embargo, no siempre es del todo conveniente hacer todo el proceso de implementación.

Por ejemplo, si uno no está seguro de haber escrito un código 100% correcto, no tiene caso hacer todo el proceso para después proceder a corregirlo, ya que esto lleva mucho tiempo; en cambio, es suficiente con comprobar o "checar" la sintaxis (*Check Syntax*) para comprobar si el código es correcto al menos en su escritura.

2.2.3 Simulando

Xilinx ISE 8 nos permite implementar simulaciones de manera muy sencilla, lo primero será crear y añadir un archivo del tipo *Test Bench Waveform*, en el cual veremos todos nuestros puertos de entrada y salida, después de asignar valores de entrada a los puertos de entrada, podremos iniciar la simulación, para hacer esto tendremos 3 opciones, las cuales estarán siempre disponibles:

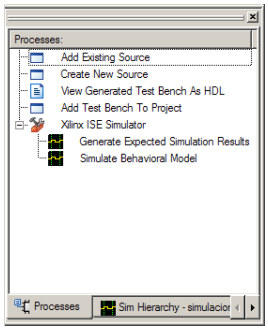


Fig. 2.11 Simulaciones disponibles en Behavioral Simulation

- *Generate Expected Simulation Results* Generará una simulación sobre la hoja en la que estaremos editando los puertos, y podremos hacer cambios inmediatamente.

- *Simulate Behavioral Model.* Hace la simulación en una nueva ventana, nos entrega el resultado en un área gris en la que ya no es visualizable la pauta de temporización que nos da la ventana de edición de simulación. Esto lo veremos a detalle en el desarrollo de la primera práctica.

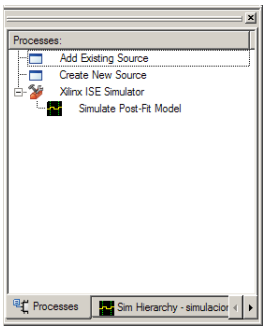


Fig. 2.12 Simulación disponible en Post Fit

- *Simulate Post-Fit Model* . Dará una simulación que se ajustará a la realidad, por lo que podremos ver el tipo de errores que tendría nuestro programa en un ambiente real, es decir, si generaría pulsos no deseados o pulsos que sólo se generan cuando cambian las señales procesadas comúnmente denominadas *"glitches"*.

Cabe mencionar que el diseño debe ser implementado antes de tratar de hacer las simulaciones.

Un error muy común al hacer la simulación es el siguiente:

ERROR:Simulator:222 - Generated C++ compilation was unsuccessful

La simulación es imposible de realizar y no hay forma de hacer que el sistema haga las simulaciones. Si el caso es tal siga estos pasos:

El error se debe a que Xilinx ISE 8.1i está instalado en una carpeta que tiene espacios en su nombre, como puede ser "C:\Archivos de programa\ISE" ó "C:\Program Files\ISE". Lo que se debe hacer para corregir el error es desinstalar Xilinx ISE e instalarlo en una carpeta que no tenga espacios, la que se sugiere es "C:\Xilinx\ISE".

La siguiente vez que se trate de simular, ya no habrá problema.

2.2.4 Grabando CPLD

Xilinx ISE 8 tiene un programa integrado llamado iMPACT, en el cual se lleva a cabo la grabación física del CPLD, también aparece en la ventana de *Processes*, bajo el procesos de *Generate Programming File.* La conexión del cable a la tarjeta del CPLD se

hará mediante el cable llamado JTAG, a iMPACT es lo único que le indicaremos para poder proceder a la grabación del dispositivo:

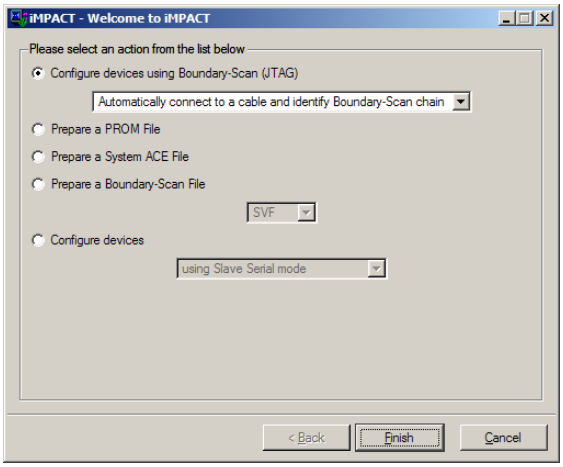


Fig. 2.13 Asistente de iMPACT

Conclusiones

A lo largo de todo este capítulo hemos visto las herramientas en las que se enfoca este trabajo, una no puede ser sin la otra. El CPLD, el software y el lenguaje de programación, en una analogía con los lenguajes de programación como C, BASIC, Java, etc… es como tener el compilador, el lenguaje y la plataforma reunidos todos en un solo trabajo.

Primero trabajamos sobre la plataforma, es decir, el dispositivo, el CPLD. Ahora tocó el turno a las otras dos herramientas por una parte, el lenguaje VHDL, hemos visto toda la parte teórica necesaria, con muchos ejemplos, sin embargo, hay que ponerlos en práctica, el software ISE es de trabajo profesional, pero no es un software complejo y difícil de aprender, ya hemos visto cómo es su funcionamiento básico, el siguiente paso será aprender a usar el software de manera conjunta con el lenguaje.

3. Prácticas básicas

Una vez que hemos echado un vistazo a las herramientas que vamos a usar, y que ya conocemos el funcionamiento del CPLD, estamos listos para comenzar a programar.

La manera en que se trabajará durante este curso teórico-práctico es partiendo de programas muy simples y modificándolos poco a poco para transformarlos en algo más complejo. En los primeros programas utilizaremos las estructuras más básicas de programación y usaremos el paradigma de diseño Combinacional, una vez que estas estructuras sean dominadas, estaremos listos para programar sistemas Secuenciales y máquinas de estado.

En base a la descripción sobre la programación en VHDL hecha en el capítulo dos, comenzamos con la primera práctica.

3.1 Práctica 1 – Circuitos SSI y Ecuaciones Booleanas en VHDL

Objetivo:

Conocer el uso de las ecuaciones booleanas en VHDL para describir circuitos lógicos combinacionales básicos de compuertas simples.

Antecedentes:

VHDL permite implementar circuitos lógicos que normalmente estamos acostumbrados a diseñar con compuertas básicas en un sólo dispositivo, la ventaja de esto es que el diseño puede cambiar totalmente y no se tiene que modificar ninguno de los dispositivos, sino que basta con reprogramar el dispositivo, en este caso usaremos el CPLD, el cual será programado mediante lo que en VHDL se conoce como ecuaciones booleanas, estas son ecuaciones con valores lógicos de ‘1’ ó ‘0’ y usan operadores lógicos como son AND, NAND, OR, NOR, XOR, XNOR, NOT.

Desarrollo:

Se implementará el circuito lógico mostrado en la figura 3.1 en el CPLD con la ayuda de ecuaciones booleanas y señales internas.

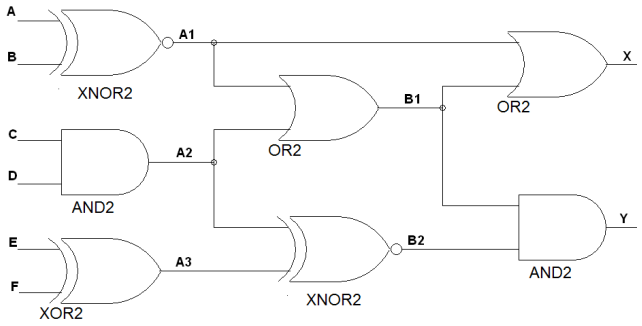


Fig. 3.1 Circuito lógico

Las señales internas serán A1, A2, A3, B1 y B2 y se usarán para almacenar temporalmente los resultados parciales de las ecuaciones.

A continuación, vamos a describir el proceso lógico que se ha de seguir para llevar a cabo con éxito la programación que de solución al circuito que se desea resolver.

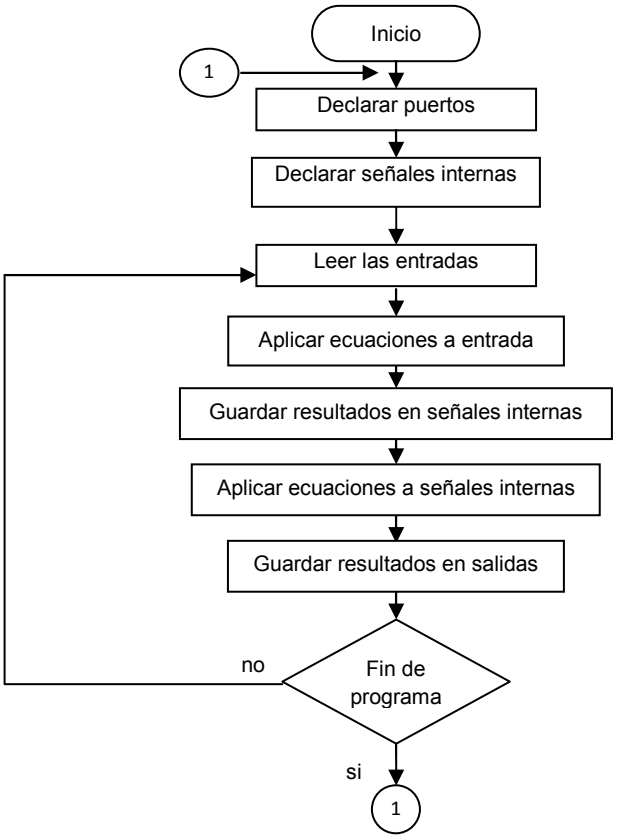


Fig. 3.2 Diagrama de flujo del programa

Dejaremos de lado el programa por un tiempo, para dar paso al detalle de la programación del código VHDL.

El CPLD Xilinx será programado mediante el software ISE, la manera general de trabajar en ISE es la siguiente (En las prácticas siguientes estos pasos ya no se listarán):

- Abrir ISE (Menú Inicio/Todos los programas/Xilinx ISE 8/Project Navigator)
- Crear un nuevo proyecto en el menú Archivo (File/New project)
- Dar un nombre al proyecto

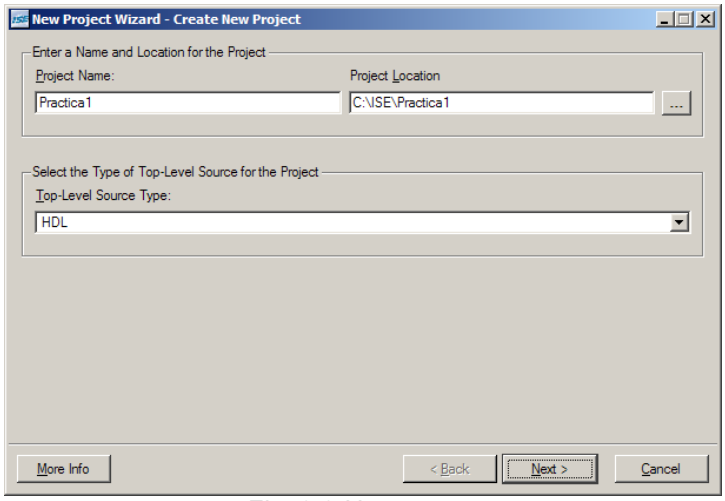


Fig. 3.3 Nuevo proyecto

- Seleccionar el CPLD y características del mismo en el que se implementará el proyecto.

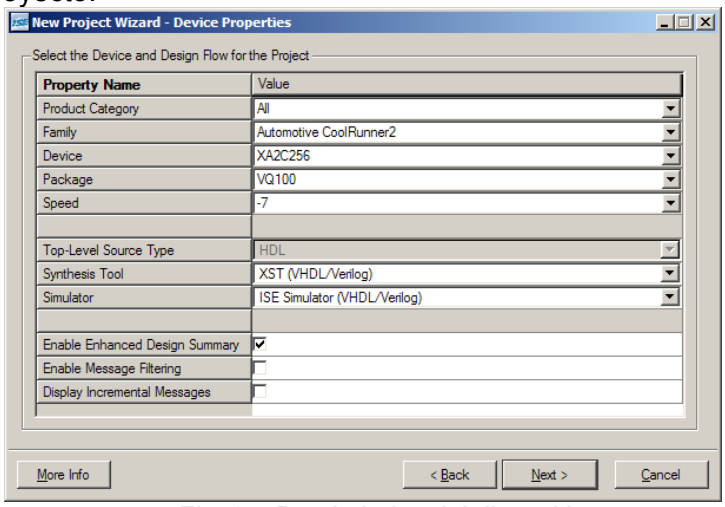


Fig. 3.4 Propiedades del dispositivo

- Añadir al proyecto un nuevo archivo del tipo *VHDL Module*

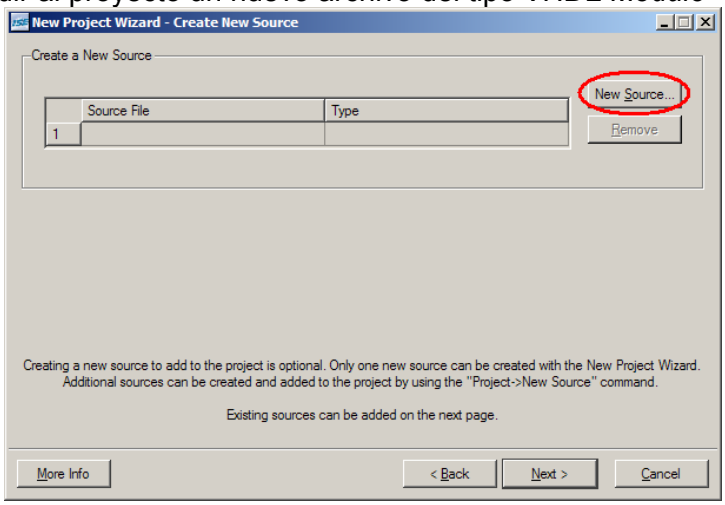


Fig. 3.5 Añadir módulo VHDL

- Darle un nombre al nuevo Archivo

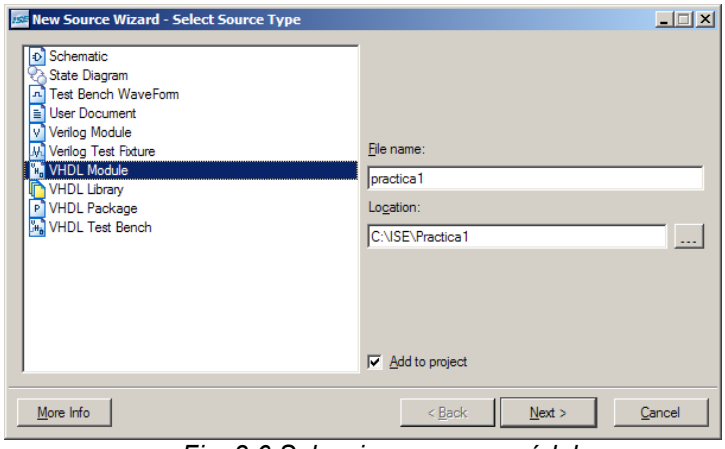


Fig. 3.6 Seleccionar nuevo módulo

- Se definen los puertos de entrada y se salida, aunque esto se puede hacer una vez que se ha iniciado a escribir el código, el nombre de la arquitectura es opcional y por *default* se llama *Behavioral*, como es la parte que describe el funcionamiento, la llamamos Función.

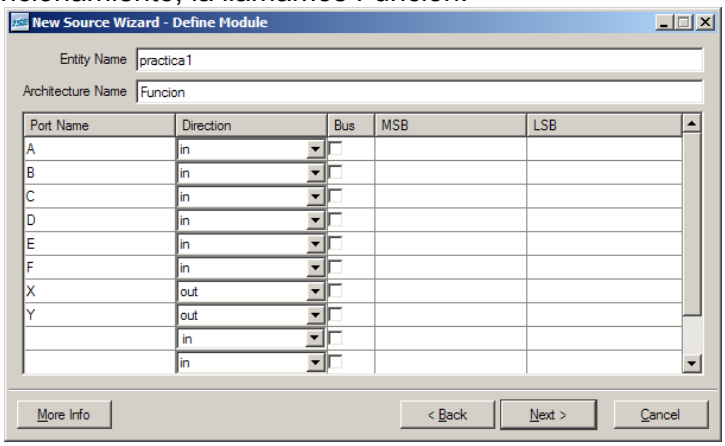


Fig. 3.7 Definir los puertos

- Finalmente ISE nos hace un resumen con todas las opciones que hemos seleccionado para el nuevo archivo.

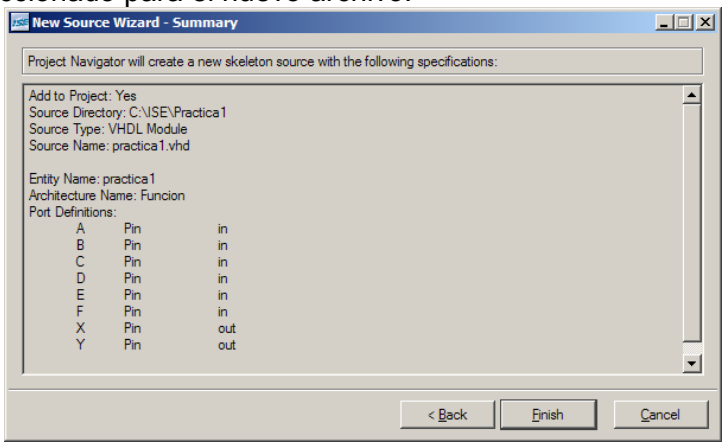


Fig. 3.8 Resumen

- Podemos continuar con el Asistente de nuevo proyecto.

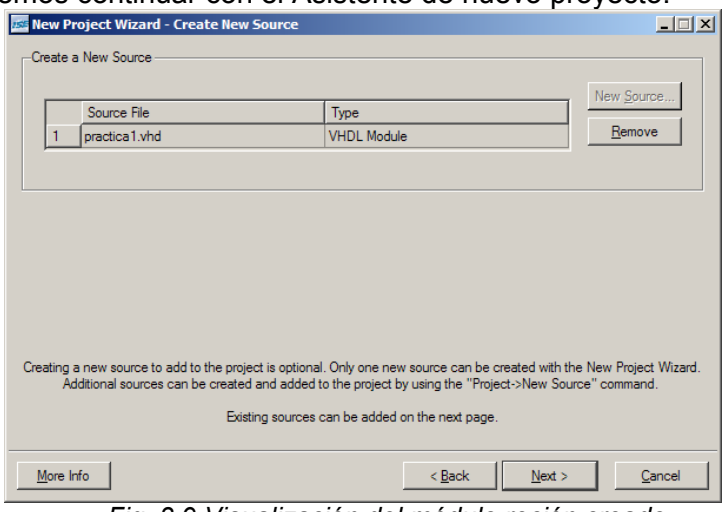


Fig. 3.9 Visualización del módulo recién creado

- El Asistente nos da la opción ahora de añadir un archivo previamente creado al nuevo proyecto, en este caso, no lo haremos.

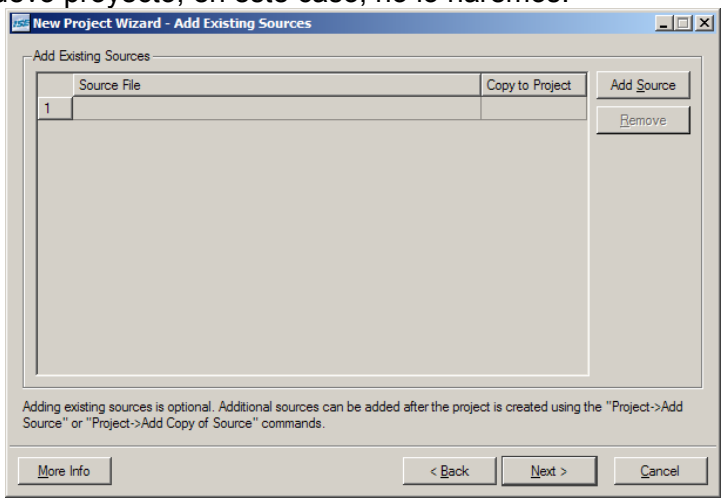


Fig. 3.10 Añadir módulos existentes

- ISE nos muestra un nuevo resumen, en este caso, se trata de un resumen de todo lo que serán los atributos y componentes del proyecto

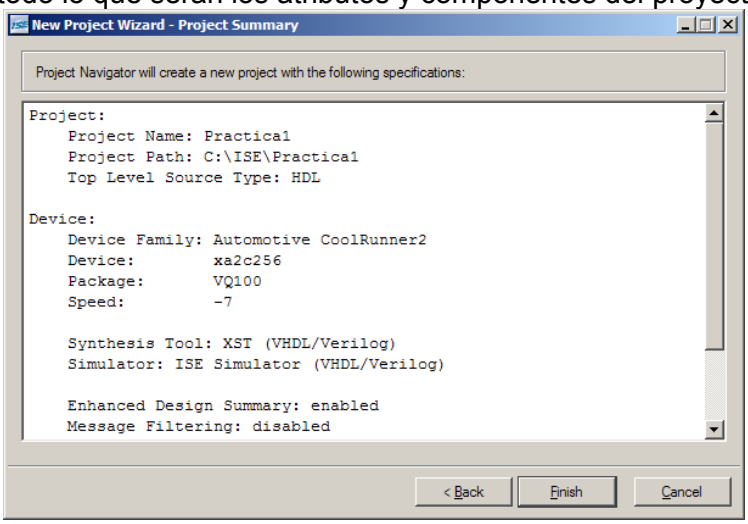


Fig. 3.11 Resumen del proyecto

- La pantalla de trabajo de ISE cambia totalmente, y ahora nos muestra el archivo que creamos con el código de VHDL.
- Además, ISE nos muestra en el listado de la izquierda el nuevo archivo que añadimos, además de que activa las opciones de Procesos en la ventana *Processes* que también está a la izquierda.
- El recuadro de mensajes nos informa que el proyecto ha sido creado con éxito y que hemos iniciado la edición del archivo 'practica1.vhd'

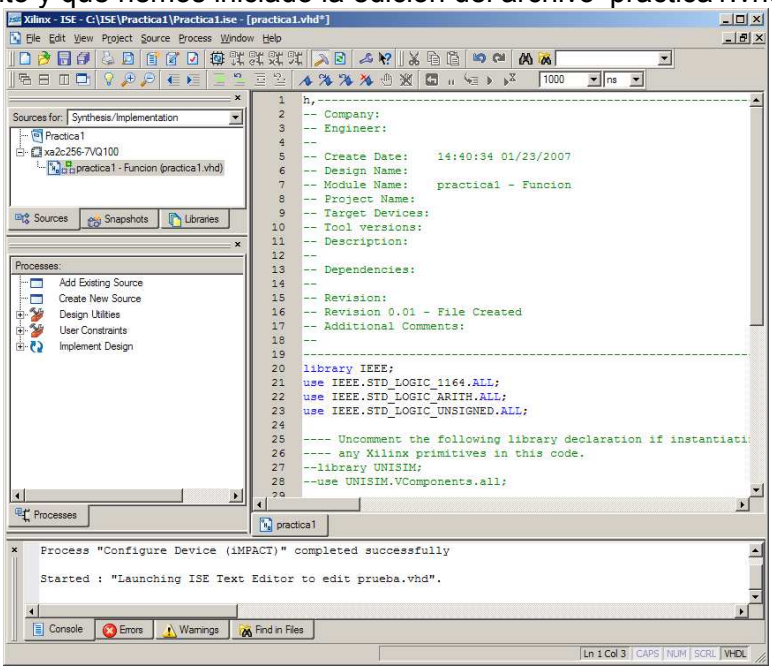


Fig. 3.12 Proyecto creado y módulo añadido

- Capturar el código
- El archivo que nos da ISE yo tiene una buena parte del código, que es la siguiente:

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

---- Uncomment the following library
---- declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity practica1 is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : in  STD_LOGIC;
           D : in  STD_LOGIC;
           E : in  STD_LOGIC;
           F : in  STD_LOGIC;
           X : out  STD_LOGIC;
           Y : out  STD_LOGIC);
end practica1;

architecture Funcion of practica1 is

begin


end Funcion;
```

- Como puede observarse, ISE nos listó los puertos tal y como se los indicamos, uno por uno, quizás ocupando unas cuantas líneas de más.
  Nos ha dejado listas las líneas de las librerías y listas las líneas en las que se ha de escribir el funcionamiento del programa.

Programación en VHDL:

En este circuito se hará especial énfasis en la sintaxis de la declaración de entidad y arquitectura, así como los puertos de entrada y salida y las señales internas que servirán de apoyo para la descripción del circuito aunque el diseño puede hacerse sin señales internas. Se busca también conocer con este código algunos errores enviados por VHDL.

El código que da solución al circuito lógico es el siguiente:

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

entity practica1 is port(
A,B,C,D,E,F: in std_logic;
X,Y: out std_logic);
end practica1;

architecture funcion of practica1 is
signal A1,A2,A3,B1,B2: std_logic;
begin
A1<=A XNOR B;
A2<=C AND D;
A3<=E XOR F;
B1<=A1 OR A2;
B2<=A2 XNOR A3;
X<=A1 OR B1;
Y<=B1 AND B2;
end funcion;
```

Nótese que se cambió la forma en que se declararon los puertos de entrada y de salida conforme los había escrito ISE, esto es por ahorrar espacio y líneas de código, aunque no cambia en lo absoluto el funcionamiento del programa.

La línea `signal A1,A2,A3,B1,B2: std_logic;` significa que se están declarando señales internas, luego se listan sus nombres y el tipo de dato que representan.

La línea `A1<=A XNOR B;` significa que el resultado de la compuerta XNOR aplicada a A y a B debe ser almacenado en A1, las demás líneas deben leerse de manera similar, si comparamos el orden de las líneas y las asignaciones con la forma en que está construido el circuito lógico a resolver, podremos observar que es muy importante llevar el mismo orden que niveles de compuertas para que las señales internas cumplan con su cometido.

- Implementar el diseño “Implement Design”
- Como vimos en la descripción de las Herramientas de Software, la implementación del programa la haremos mediante el botón *Implement Design*, ISE nos indicará palomeando el botón que el programa está correctamente escrito. En caso contrario, es preciso revisar que todos los pasos hasta este momento se hayan realizado correctamente.
- ISE nos mostrará también un reporte en el cual nos indica, entre otras cosas, algunas características que tendrá el CPLD, como qué pines han sido usados, cuántos son, cuántas macroceldas se usaron y en general todos los pormenores de la implementación del diseño en el CPLD de nuestra elección.

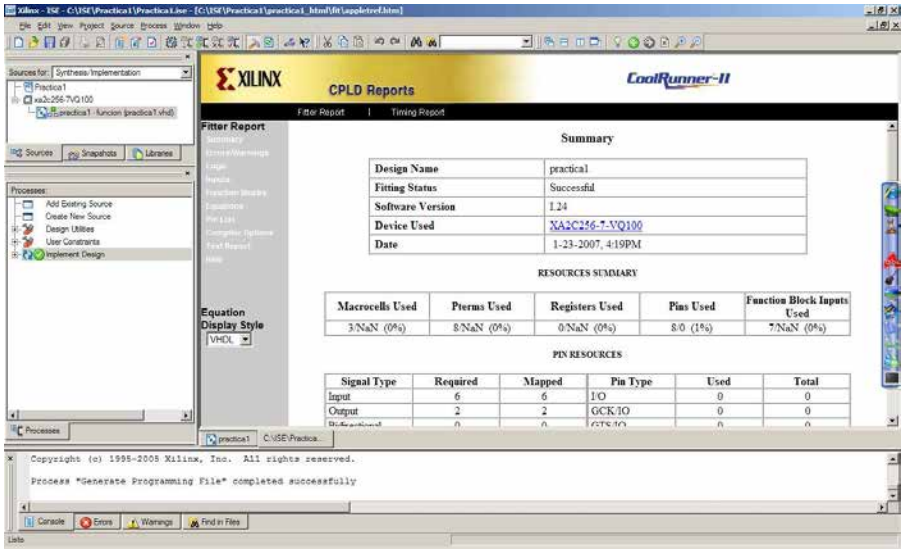


Fig. 3.13 Reporte del proyecto

Simulación:

- Añadir al proyecto un nuevo archivo del tipo *Test Bench WaveForm*

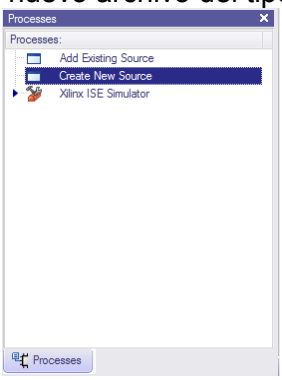


Fig. 3.14 Añadir una simulación al proyecto

- Para eso en la ventana de *Processes* haremos doble click *sobre Create new source*, lo que nos abrirá el asistente para crear un nuevo archivo para el proyecto.

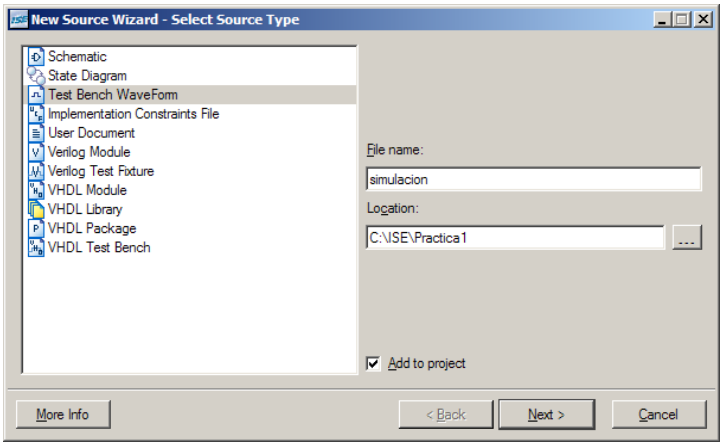


Fig. 3.15 Añadir el archivo de simulación

- En esta ventana podemos darle un nombre al archivo después de haber seleccionado *Test Bench WaveForm*, teniendo cuidado de no confundirlo con *VHDL Test Bench*. Esto nos creará un archivo de extensión .tbw el cual contendrá toda la información concerniente a la simulación.

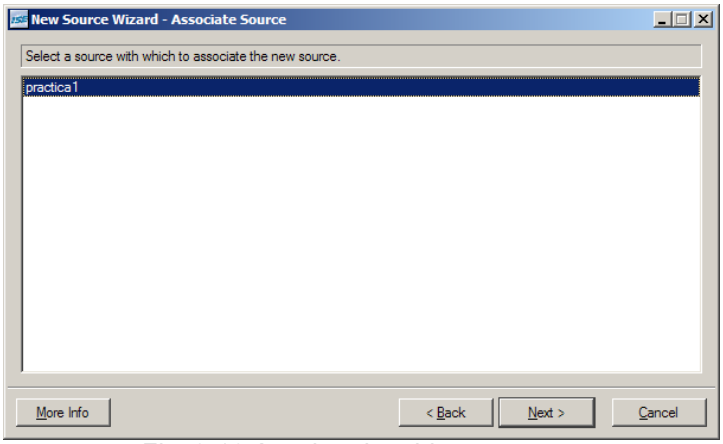


Fig. 3.16 Asociar el archivo

- La siguiente ventana en ser mostrada simplemente nos confirma que estamos añadiendo un archivo al proyecto “practica1” (o como le hallamos llamado)

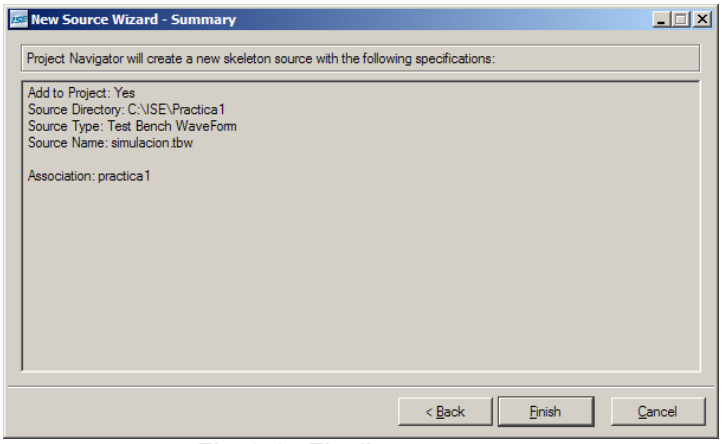


Fig. 3.17 Finalizar

- Nuevamente, ISE nos resume la acción que estamos a punto de confirmar, en este caso no lleva tantos pasos añadir el archivo.

- Dar los parámetros para la simulación

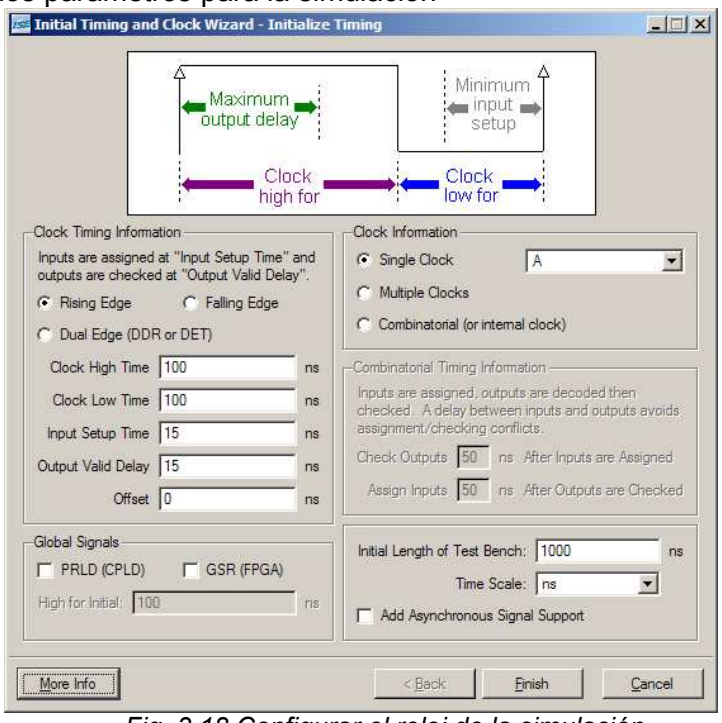


Fig. 3.18 Configurar el reloj de la simulación

- Una vez que añadimos el archivo aparece la siguiente ventana en la cual más que nada haremos la configuración del Clock.

- Como este primer programa es totalmente combinacional la configuración será diferente y especial con respecto a lo que hemos hecho anteriormente.

- En primer lugar, no tendremos un Clock como tal, por lo que seleccionaremos la opción *Combiational (or internal clock)*, la cual cancelará casi todas las demás opciones.

- En segundo lugar vamos a ponerle un límite de tiempo a la simulación, como estamos trabajando con señales combinacionales, lo único que haremos es comprobar que las compuertas estén funcionando como esperamos.

- Lo que haremos es simular toda la tabla de verdad que se esperaría de este diseño sin tener toda la tabla a la mano.

- Como tenemos 5 señales de entrada, tenemos que comparar 2^6 posibles combinaciones con 2^6 salidas; por lo tanto estamos hablando de 64 combinaciones, si damos un tiempo de 20ns a cada combinación necesitaremos un total de 1280ns con un tiempo de 10ns para cada tiempo alto y 10ns para cada bajo.

- Las opciones deben quedar como se ve en la siguiente imagen:

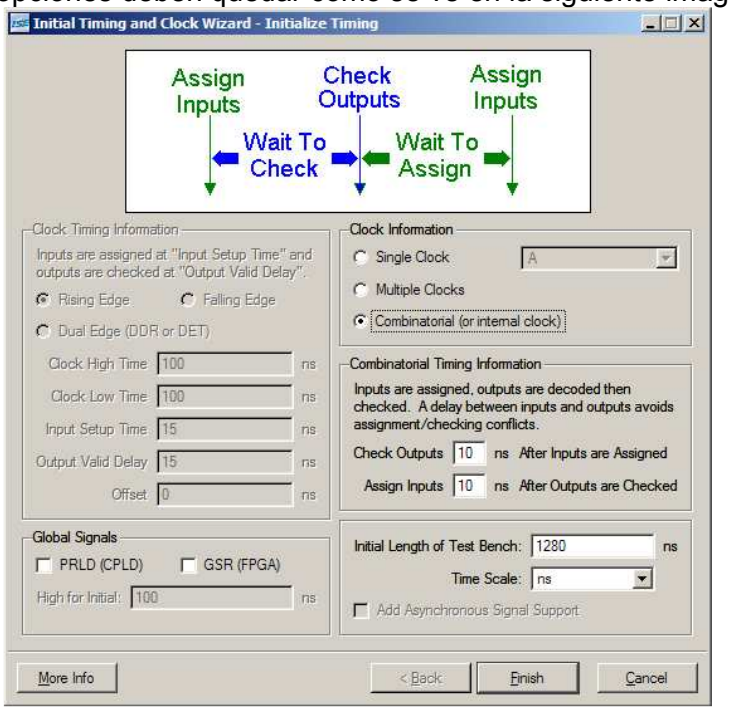


Fig. 3.19 Configurando el Reloj

- Una vez que han sigo configurados estos parámetros, ya se puede hacer la simulación.

- Hacer la simulación “Simulate Behavioral Model”

Para hacer la simulación será necesario hacer todas las combinaciones posibles, para eso lo que haremos es hacer que una señal cambie su estado de 0 a 1 a cierto ritmo constante, la siguiente señal va a cambiar de estado a la mitad de la velocidad y así sucesivamente.

Lo que se busca es emular el conteo binario del o al 63 con los pulsos altos de las señales.

En nuestro caso, se emula una tabla de verdad como la siguiente:

00000
00001
00010
00011
00100
00101
00110
00111
01000
….

Como puede observarse, una columna lleva un ritmo 0101, la otra un ritmo 0011 y la otra 00001111, eso es lo que haremos en la simulación.

Comenzaremos con la señal F, daremos click sobre el renglón que ocupa esta señal sobre uno de los cuadros verdes. Y del menú contextual que aparece seleccionamos *Set Value*.

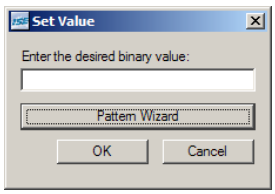


Fig. 3.20 Asignar valores a la señal

En el pequeño recuadro que aparece seleccionamos *Pattern Wizard*, en el cual configuraremos el pulso constante.

En esta primera señal (F), lo único que hay que configurar es que el pulso se repita 32 veces:

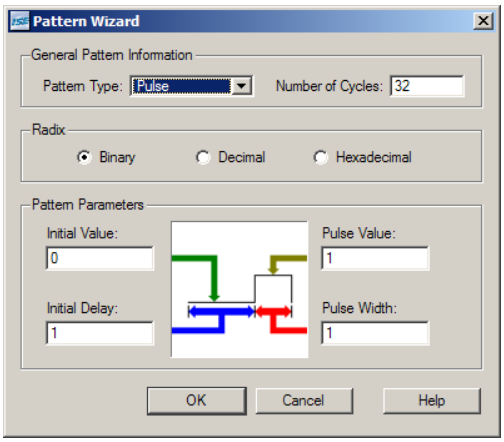


Fig. 3.21 Asistente para patrones de señales

En la siguiente señal (E) la diferencia es que el número de ciclos será la mitad, o sea, 16 y además, los valores de tiempo cambiarán al doble del anterior, es decir:

Initial Delay = 2; Pulse Width = 2

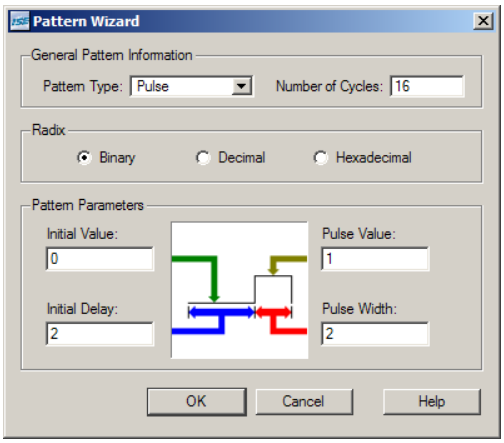


Fig. 3.22 Configurando el patrón de la señal

La señal D tendrá 8 ciclos y los tiempos de *Initial Delay* y *Pulse Width* de extensión 4, y así sucesivamente, cabe mencionar que no es necesario ser preciso en el valor del número de ciclos, si deben ser 8 y ponemos 16 no hay ningún problema. Claro que si es 16 y ponemos 8, no obtendremos los resultados esperados.

Una vez que hayamos terminado, obtendremos una ventana como la que se ve a continuación:

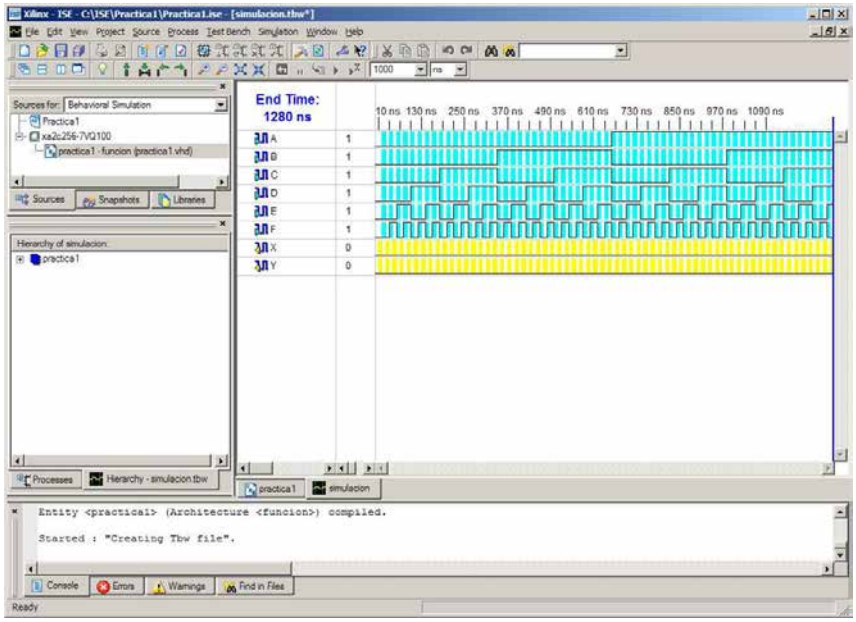


Fig. 3.23 Pantalla de simulación

Ahora procedemos a iniciar la simulación:

- En la ventana Sources for seleccionamos Behavioral Simulation.
- En la ventana Processes seleccionamos Generate Expected Simulation Results.
- ISE 8 nos pregunta si queremos guardar los cambios hechos a simulación.tbw, aceptamos, y finalmente tendremos el programa simulado, el resultado final debe ser el siguiente:

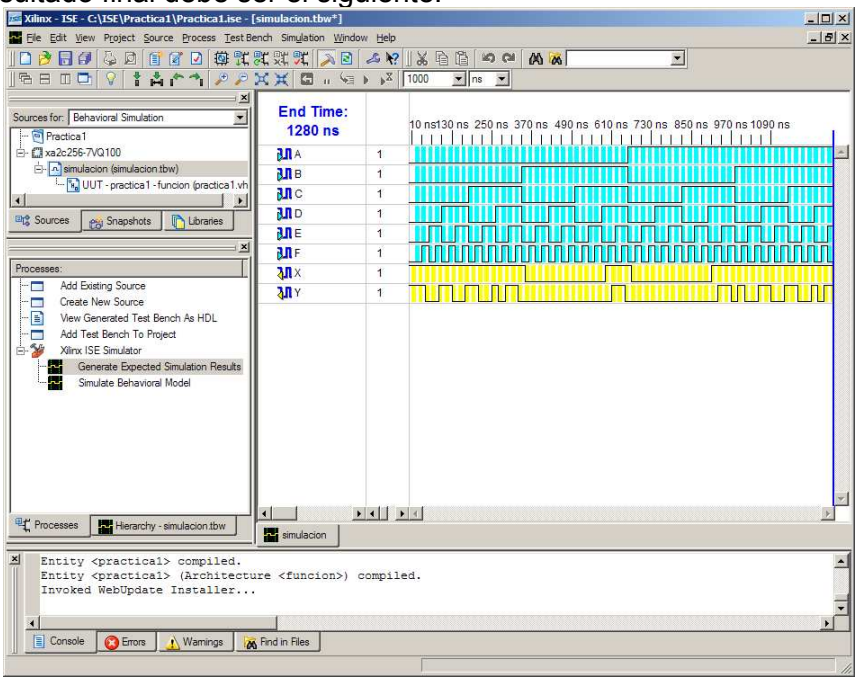


Fig. 3.24 Simulación realizada

3.2 Práctica 2 – Ecuaciones Booleanas, Asignaciones Condicionales y Vectores en VHDL

Objetivo:

Utilizar las ecuaciones booleanas para hacer operaciones lógicas sobre buses de datos en VHDL

Antecedentes:

Ahora que sabemos utilizar ecuaciones booleanas sobre circuitos lógicos simples, estamos listos para trabajar sobre vectores, el trabajo sobre vectores es una de las grandes bondades de VHDL ya que permite simplificar en gran parte el trabajo evitando que sea bit por bit.

El trabajo sobre vectores es lo mismo que trabajar con buses de datos completos de n bits. Con una simple operación podemos invertir el valor completo de un bus, o aplicar una operación a dos buses y obtener un bus que almacene el total de bits resultante. Cabe mencionar que es necesario que en las operaciones AND, OR; XOR, etc… es decir, de dos operandos, los vectores sean del mismo número de bits.

Cuando aplicamos una compuerta a un par de buses, por ejemplo, una AND a:

BUS1 = 00101011

BUS2 = 11001001

El resultado sería: BUSr = 00001001

Como puede verse, la compuerta AND trabaja bit por bit, y no tenemos la necesidad de escribir todo el código que se necesitaría, esto es de gran ayuda cuando se trabaja con buses que son cada vez más grandes, como 16 o 24 bits, por ejemplo.

Los bits en los vectores son diferenciados uno de otro mediante un número entero, siempre van ordenados, se pueden comparar a elementos de una matriz. También pueden ser listados al ser referenciados para tomar una parte del vector nada más, o sea, podemos tomar un vector de n bits y operarlo con la mitad de un vector de 2^n bits sin ningún problema. Lo único que hay que hacer es decirle a VHDL qué es lo que tiene que hacer, en qué orden debe numerar y trabajar los bits de los vectores y qué bits usará para trabajar, por default se sobre entiende que se trabajará con el vector completo.

Desarrollo:

Como desarrollo práctico, se programarán distintos vectores y se hará la simulación. La parte de la simulación es muy importante, pues Xilinx ISE nos permite trabajar con un

vector como si fuera un bus con un valor entero o hexadecimal en un instante de tiempo, es decir, no tenemos tampoco que dar en la simulación el valor exacto de todos los bits, Xilinx ISE también hace ese trabajo por nosotros.

Los vectores con los que se trabajará son los siguientes:

3 vectores de entrada:	1 vector de salida:
A de 3 bits	X de 8 bits
B de 8 bits	
C de 16 bits	

La manera en que se declara un vector es la siguiente:

```
A: in std_logic_vector(0 to 3);
```

Nótese que cada bit del vector está numerado ascendentemente, en caso de que se requiera que la numeración de bits sea descendente, se debe escribir de la siguiente forma: `A: in std_logic_vector(3 downto 0);`

Incluso, si así lo deseamos, podemos dar un rango como numeración para los bits, por ejemplo, si queremos que el vector A, que es de 4 bits, tenga los mismos numerados del 10 al 13, el vector puede ser declarado de la siguiente forma:

```
A: in std_logic_vector(10 to 13);
```

Como ejercicio, se realizarán algunas asignaciones sobre vectores mediante el uso de vectores. Sean los elementos de:

- Bus A: A2,A1,A0
- Bus B: B7,B6,…,B1,B0
- Bus C: C0,C1,…,C14,C15
- Bus X: X7, X6,…,X1,X0

La lógica del programa será la siguiente:

- Si bus A = “000”, es decir A2=0, A1=0 y A0=0, entonces
- Asignar el bus B al bus X
- Si bus A = “001”, es decir A2=0, A1=0 y A0=1, entonces
- Asignar bits C0-C7 al bus X
- Si bus A = “010” entonces
- Asignar bits C8-C15 a bus X
- Si bus A = “011” entonces
- Asignar bits (C0-C7 XOR C8-C15) a bus X
- Si bus A = “100” entonces
- Asignar bits (C0-C7 AND bus B) a bus X
- Si bus A = “101” entonces
- Asignar bits ((NOT C0-C7) OR bus B) a bus X
- Si bus A = “110” entonces

- Asignar bits ((NOT C4-C11) NAND bus B) a bus X
- Si bus A = “111” entonces
- Asignar bits B7,B6,B5,C0,C1,C2,C15,C14 a bus X, respetando el orden.

La forma de realizar la codificación en VHDL de las anteriores líneas, será mediante la asignación condicional WHEN…ELSE, cuya sintaxis fue vista en el capítulo 2.

Como puede notarse, toda asignación depende directamente del BUS A, por lo que la asignación condicional siempre estará referida a los valores de este bus.

Programación en VHDL:

El código solución para este problema se da a continuación; se sugiere, sin embargo, iniciar un nuevo proyecto para efectuar la programación y simulación de esta práctica:

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

entity practica2 is port(
A: in std_logic_vector(2 downto 0);
B: in std_logic_vector(7 downto 0);
C: in std_logic_vector(0 to 15);
X: out std_logic_vector(7 downto 0));
end practica2;

architecture Func of practica2 is
begin
 X <= B when A="000" else
      C(0 to 7) when A="001" else
      C(8 to 15) when A="010" else
      C(0 to 7) xor C(8 to 15) when A="011" else
      C(0 to 7) AND B when A="100" else
      NOT(C(0 to 7)) OR B when A="101" else
      NOT(C(4 to 11)) NAND B when A="110" else
      B(7 downto 5) & C(0 to 2) & C(15) & C(14);
end Func;
```

Como puede observarse, la codificación con la estructura WHEN…ELSE, nos permite listar todos los casos que se van a observar para una o varias señales de entrada, pues la condición puede ser cualquiera, en esta práctica en particular, todas las condiciones son sobre el bus A, pero puede trabajarse con otro.

Otro aspecto que debe ser considerado es que en el último o en los últimos casos ya no es necesario especificarlo, pues el operador lógico de la sentencia, ELSE abarca todos los casos que no fueron listados o considerados en la estructura.

En la penúltima línea de código se utilizó la concatenación de bits y de vectores, es decir, se tomaron grupos de bits y bits simples para unirlos en un sólo vector de 8 bits.

Simulación:

Para efectuar la simulación, primero debemos comprender la forma en que Xilinx ISE nos permite visualizar los buses y sus valores en un instante de tiempo.

En este caso seguimos el mismo procedimiento para iniciar la simulación que llevamos en la práctica pasada, ISE nos mostrará un conjunto de líneas grises, que no se parecen en nada a las series de rectángulos blancos y verdes o blancos y amarillos. Sin embargo, podemos notar que el bus es expandible desde su parte izquierda, Xilinx ISE nos permite editar el bus, ya sea, trabajando bit por bit o de una vez sobre todo el bus. Las maneras que tenemos para visualizar valores son mirando los bits como una cadena binaria, como un número hexadecimal, como un número decimal, etc…

Vamos a crear una secuencia que nos permita contar del 0 al 7 para pasar por todos los casos que corresponden a la programación del bus A. Una forma sencilla es dando click con el botón derecho del mouse sobre el valor del bus y seleccionar *Set Value*, a continuación, daremos *click* en *Pattern Wizard* y en el siguiente cuadro seleccionaremos las opciones siguientes:

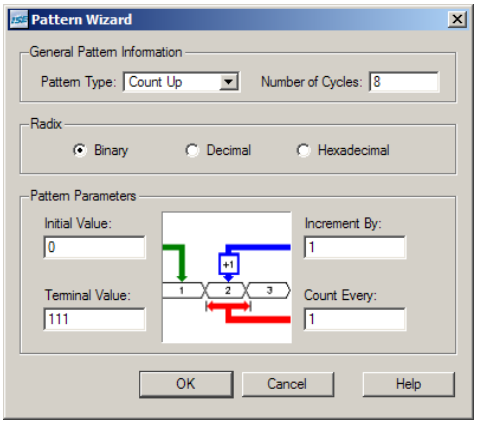


Fig. 3.25 Configurar el patrón de la señal

El asistente nos entregará un bus que debe lucir de la siguiente manera una vez que ha sido expandido:

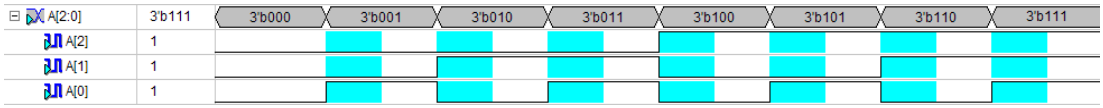


Fig. 3.26 Simulación del proyecto

Ya sea que veamos bit por bit o el valor del bus, siempre sabemos qué valor exactamente es el que tiene, en este caso, el bus muestra un "3'b" antes de cada valor binario porque nos sirve de guía de que se trata de un bus de 3 bits y lo estamos representando en forma binaria.

Lo mismo se hace para las siguientes señales. Para los motivos de simulación, es suficiente con generar 7 señales distintas, incluso, Xilinx ISE puede generar buses con valores aleatorios para que probemos nuestro programa, lo único que tenemos que hacer en el *Pattern Wizard* es seleccionar la opción *Random Bus* en lugar de *Count Up*. Con la diferencia de que es más conveniente visualizar los valores de los buses como hexadecimales en lugar de decimales o binarios, porque la comprensión es más sencilla.

Una vez que hemos terminado de dar valores a los buses, es tiempo de ejecutar la simulación. Para los valores aleatorios que se usaron en este documento, la simulación presentará los siguientes casos[5]:

Generate Expected Simulation Results

La simulación que obtenemos a partir de la opción *Generate Expected Simulation Results* es la siguiente:

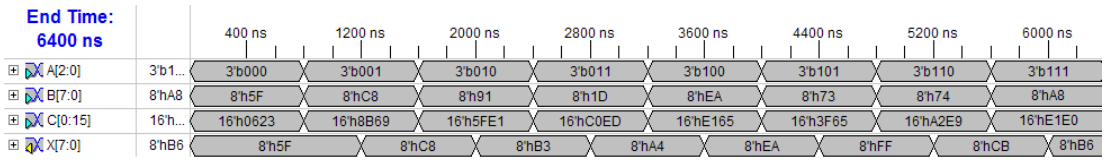


Fig. 3.27 Simulación estimada del proyecto

Para comprender la simulación es necesario comprender los valores hexadecimales, en primer lugar, el bus X tiene un valor 5F (Dejando de lado 8'h que significa 8 bits, formato hexadecimal), este 5F es el valor que tiene B en el primer instante,

[5] Se usó un tiempo total de 6400ns (Set end of test bench) y 400ns de Timing (Check Inputs, Asign Outputs en Rescale Timing)

cuando al mismo tiempo A vale 000, lo que cumple con la programación que hicimos; sin embargo, el valor de este bus X permanece siendo el mismo en el instante de tiempo que A y los demás buses cambian, lo que hace que el siguiente valor en el tiempo del bus X sea inexacto, por lo tanto, se nos presenta una situación en la que la simulación no es precisa.

Simulate Behavioral Model

Xilinx ISE está preparado para trabajar con este tipo de problemas, y es por eso que utilizaremos la simulación *Simulate Behavioral Model*, que nos permitirá visualizar los resultados de la simulación de la siguiente manera:

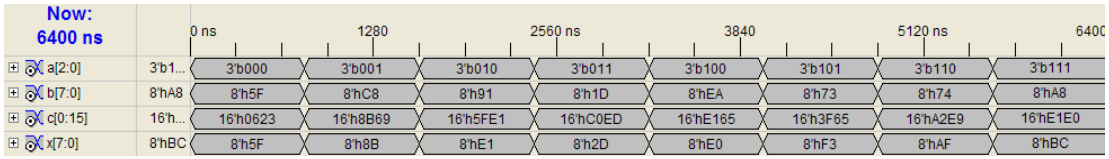


Fig. 3.28 Simulación del comportamiento ideal del proyecto

Esta simulación es mucho mejor, ya que nos da los valores exactos que tendrán las señales, ya que por tratarse de un diseño de tipo combinacional, el retraso que sufren las señales al ser asignadas es mínimo y lo que hace esta simulación es considerar los retrasos provocados por esas asignaciones como despreciables, o sea, como retrasos de 0s.

Continuando con la evaluación de la simulación obtenida, podemos observar que el valor del bus X cuando A vale 001 es exactamente el valor de las primeras dos cifras hexadecimales del bus C, y bien sabemos que dos cifras en hexadecimal equivalen a 8 bits en binario, por lo tanto, se nota que el programa está operando conforme a lo esperado.

Para el caso en que A vale 101, tenemos como resultado F3

Esto viene de la línea de código

```
NOT(C(0 to 7)) OR B when A="101" else
```

Por lo tanto, lo que hace es invertir h3F, que corresponde a los primeros 8 bits del bus C, el resultado es NOT 00111111 = 11000000 = C0.

Después, se hace un OR con el bus B que vale h73, el resultado es el siguiente

```
        11000000
OR      01110011
        11110011 = F3
```

Una vez que uno se ha acostumbrado a leer los valores en hexadecimal, es más sencillo saber si un programa funciona correctamente y si la simulación es correcta.

Simulate Post-Fit Model

Para terminar de ejemplificar los tres tipos de simulaciones que Xilinx ISE nos permite hacer, vamos a aclarar la razón de realizar esta tercera simulación.

La primera que vimos es una simulación muy sencilla y muy útil, sin embargo, como vimos también, puede ser muy inexacta.

La segunda simulación es muy buena y nos permite hacer muy buenas aproximaciones a los valores reales con los que trabajaría el CPLD una vez programado con el diseño en que estamos trabajando.

El resultado arrojado por la tercera simulación es el siguiente:

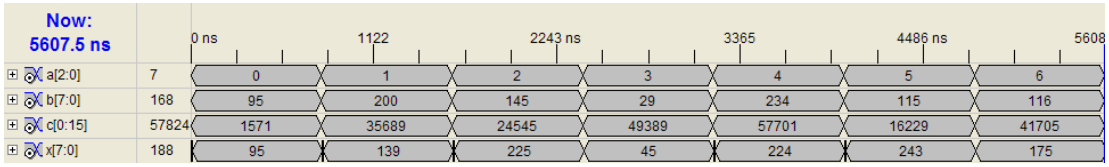


Fig. 3.29 Simulación real del proyecto

Algunas irregularidades pueden verse entre algunos de los intermedios en los valores del bus X, si observamos un poco más de cerca el bus en esa zona, veremos lo siguiente:

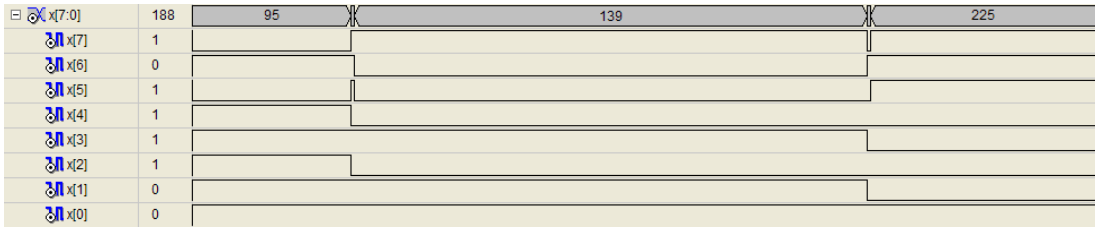


Fig. 3.30 Simulación real expandida para visualizar glitches

Lo que observamos anteriormente, ahora es más claro, son un conjunto de valores que toma el bus en pequeños intervalos de tiempo porque los valores de los bits que lo componen tardan en llegar a un estado en el cual permanezcan de manera estable. Mientras estos valores no se estabilicen, la señal de salida no será estable tampoco.

Esto podemos notarlo en esta simulación y no en las otras porque esta simulación es la más real, nos dice exactamente lo que sucedería si trabajáramos con señales como esas y a tales velocidades con el CPLD. De cualquier manera, no hay porqué asustarse, el tiempo que estamos manejando es de 800ns. Lo que equivale a manejar periodos de 0.0000004s, que es el tiempo en el que trabajaría un reloj de 1.25GHz, por lo tanto el error o glitch, es realmente insignificante cuando se trabaja con este CPLD.

Como ejercicios posteriores a esta práctica, se recomienda hacer asignaciones manualmente a los buses, así como cambiar las velocidades en las que trabaja el CPLD.

3.3 Práctica 3 – Circuitos MSI – Multiplexores y Decodificadores

Antecedentes:

Un Multiplexor es un dispositivo que nos permite seleccionar una de 2^n señales mediante un código de n bits. La señal seleccionada es enviada a una única salida, por lo que el multiplexor se concibe como una entidad con dos tipos de entrada, una que son todas las señales seleccionables y otra que son las señales de selección.

Comúnmente se entiende que los multiplexores funcionan con un bit, sin embargo, en el diseño lógico tradicional, un arreglo de multiplexores permite la selección de buses completos. Puede imaginarse entonces, que mediante VHDL se pueden diseñar multiplexores con gran facilidad y además, se requiere de exactamente las mismas líneas de programación salvo la declaración de puertos, para lograr que un programa funcione como multiplexor de un bit o de buses de n bits.

En el caso de los decodificadores, VHDL también es muy flexible, pues permite implementar entidades de este tipo en el que una señal de entrada puede ser enviada a una de 2^n señales de salida, al igual que en el multiplexor, la selección se hace mediante un bus de n bits de selección.

Desarrollo:

Trabajaremos el diseño de multiplexores y decodificadores mediante la técnica de la Asignación de señales por condición WITH-SELECT, en la cual, después de definir la señal que servirá como selección, se listarán los posibles casos, y se definirá el proceder que habrá de llevarse a cabo en los casos no listados.

Ésta forma de asignación es muy parecida a la WHEN-ELSE salvo que aquí las condiciones sólo aplican para una señal de selección y no se puede tener otro tipo de condiciones, sin embargo, como se podrá ver durante la programación, su uso es muy simple y versátil.

Programación en VHDL:

El código que corresponde a la implementación de un multiplexor es el siguiente:

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

entity practica3 is port(
A,B,C,D,E,F,G,H: in std_logic;          --8 Entradas
```

```
O: out std_logic;
SEL: in std_logic_vector(2 downto 0));  --3 bits de selección
end practica3;

architecture Funcionamiento of practica3 is
begin
 with SEL select
  O <=A when "000",
      B when "001",
      C when "010",
      D when "011",
      E when "100",
      F when "101",
      G when "110",
      H when others;
end Funcionamiento;
```

Nótese que para el caso de la asignación a la salida del valor de H, no se evalúa el caso como “111” sino que se le indica a ISE que H es el valor que asignará a todos los casos que no se hayan especificado, si en lugar de *others* ponemos el octavo caso, ISE nos manda un error en el cual nos dice que faltó un caso por contemplar, por lo que se entiende que este caso es *others*,

El equivalente de este programa es el circuito siguiente:

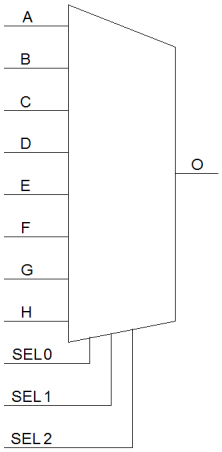


Fig. 3.31 Multiplexor 8:1

Si quisiéramos implementar un multiplexor de 4 bits, tendríamos que arreglarlos de la siguiente forma:

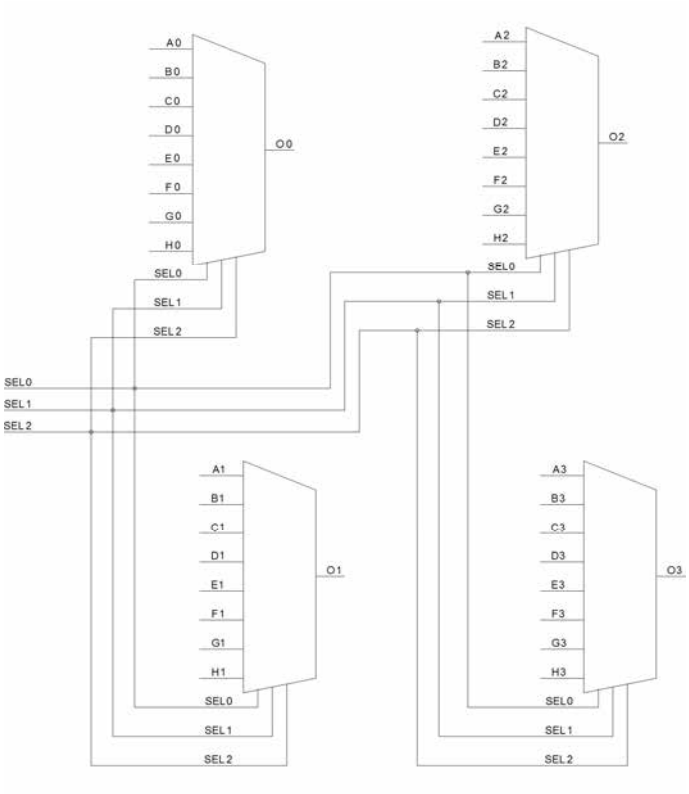


Fig. 3.32 Multiplexor 8:1 de 4 bits

Programando en VHDL, el único cambio que tenemos que hacer al código es este:

```
A,B,C,D,E,F,G,H: in std_logic_vector(3 downto 0);--8 Entradas
```

Es por eso que programar en VHDL reduce tanto el trabajo de diseño y aumenta la productividad al momento de crear sistemas completos. Aunque podríamos, no necesitamos hacer que el código sea igual que el diseño, mientras sepamos cuál es la tarea que tiene que desarrollar el dispositivo que estamos programando, no es necesario tener a la mano el diseño lógico tradicional.

Para el diseño de un decodificador, la forma de proceder es similar, dependiendo del valor que tenga la señal de selección, se habilitará poniendo en nivel bajo o cero lógico, una y sólo una de las salidas. El código solución se presenta a continuación, este código tiene la misma estructura WITH-SELECT con que se hizo el multiplexor, salvo una ligera diferencia:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity practica3 is port(
O: out std_logic_vector(7 downto 0);    --8 Salidas
SEL: in std_logic_vector(2 downto 0));  --3 bits de seleccion
end practica3;

architecture Funcionamiento of practica3 is
begin
 with SEL select
   when "000" => A <= "00000001";
   when "001" => A <= "00000010";
   when "010" => A <= "00000100";
   when "011" => A <= "00001000";
   when "100" => A <= "00010000";
   when "101" => A <= "00100000";
   when "110" => A <= "01000000";
   when others=> A <= "10000000";
end Funcionamiento;
```

Aquí se usa una estructura un poco distinta, la cual nos permitirá posteriormente asignar valores a distintas salidas empleando la estructura WITH-SELECT con la que se evalúan los posibles valores que puede tener una señal.

Conclusiones

La simulación de programas es un proceso largo de describir, porque involucra muchos aspectos como el tiempo, el patrón de la señal, la duración, el valor, etc… Pero una vez que es aprendida a realizar, la simulación es sencilla. Los primeros capítulos son en general largos, porque es preciso describir cada aspecto de las prácticas; sin embargo, se va dando por hecho que el lector conoce y aprende esos aspectos, por lo que no se repetirá su explicación, de allí que sea necesario realizar las prácticas en el orden sugerido, ya que los conocimientos de una permiten la realización de la siguiente.

4. Prácticas intermedias

En este capítulo se desarrollarán prácticas que involucran el uso de procesos y de ciclos de reloj. También se verá la forma en que se interpretan las simulaciones producto del desarrollo de diseños secuenciales. Se implementarán los operadores con que cuenta VHDL, y una herramienta tan imprescindible como es la sentencia IF THEN, se define también la forma de trabajar de los Procesos en VHDL y los Procesos regidos por un ciclo de reloj. Se realiza en este capítulo también un diseño esquemático en el que uno de los componentes es programado en su totalidad en VHDL por el usuario.

4.1 Práctica 4 – Operadores: Relacionales y Aritméticos

Objetivo:

Conocer el uso de los operadores relacionales y aritméticos en VHDL con el fin de realizar comparaciones de magnitudes con vectores y además, operaciones aritméticas básicas.

Antecedentes:

Con el previo conocimiento de los vectores, podemos ahora realizar operaciones más interesantes que las asignaciones y las concatenaciones.

Los primeros operadores de VHDL que conocimos fueron el de Asignación “<=” y el de Concatenación “&”, sin embargo, dentro de las muchas ventajas que ofrece VHDL se encuentran los operadores Relacionales y los Aritméticos, este tema fue cubierto en el capítulo 2 de este texto, por lo que sólo haremos una rápida mención del funcionamiento de estos operadores:

Los operadores relacionales de VHDL son:

- = (igual a).
- < (inferior a).
- > (superior a).
- /= (diferente de).
- <= (inferior o igual a).
- >= (superior o igual a).

Estos operan sobre objetos de tipo:

- bit, bit vector.
- std_logic, std_logic_vector.
- std_ulogic, std_ulogic_vector.
- integer.
- Boolean.

Como una introducción a los *Process* en VHDL, usaremos algunas líneas de código que nos permitirán conocer la manera en que se desarrolla un proceso.

```
process() begin
end process;
```

Un proceso es generalmente la parte secuencial de un diseño, quiere decir que podríamos encontrar una señal de reloj, en este caso, el *Process* lo ocuparemos para utilizar una instrucción secuencial de control, el IF-THEN.

Este IF es una sentencia lógica condicional en la cual a partir de una premisa se obtiene una conclusión con un valor lógico de verdadero o falso.

Un ejemplo sencillo de entender es el siguiente:

```
if seleccion > "0111" then
salida<='0'
else
salida<='1'
end if;
```

Cuando selección tiene un valor mayor al 7 binario de 4 bits la salida tiene una asignación de un valor bajo, en caso contrario, la salida tiene un valor alto. Ésta es la forma mínima en que se utiliza el IF-THEN.

Una de las formas más comunes de usar IF-THEN es de manera anidada, es decir, IF dentro de otros IF, VHDL simplifica la forma de escribir estas sentencias, por ejemplo, si queremos saber si un valor X es mayor, menor o igual que un valor Y tendríamos que hacer algo como lo que sigue:

```
if x>y then
  --es mayor
else
  if x<y then
    --es menor
  else
    es igual
  end if;
end if;
```

Los IF están anidados, pues hay un IF dentro de la parte ELSE del IF de mayor jerarquía, entiéndase, el primero en ser descrito. Si estuviéramos analizando una estructura con 10 condiciones siguiendo esta forma, nos veríamos en grandes aprietos anidando tantos IF.

Una forma de escribir el código anterior usando IF-THEN sin anidarlos es la siguiente:

```
if x>y then
  --es mayor
elsif x<y then
  --es menor
else
  --es igual
end if;
```

El número de líneas se reduce, pues el nuevo IF es incluido junto con el ELSE, para convertirse en ELSIF, esta nueva línea tiene su propio ELSE y además, todo forma parte de un mismo IF, por lo que solamente se hace el cierre una vez: END IF;

Volviendo al *Process*, entre paréntesis se va a listar las variables que componen lo que se llama Lista Sensitiva, esto lo explicaremos con más detalle en la siguiente práctica, por ahora no nos enfocaremos en *Process*.

El uso de los operadores aritméticos se hará de manera práctica, ya que su descripción fue hecha anteriormente.

Desarrollo:

Codificar una solución mediante el uso de IF-THEN para los siguientes casos:

Se tendrán dos buses de entrada, A y B ,de 4 bits y uno de salida de 8 bits, además de una línea de selección.

Cuando se seleccione “000”:

Si A>B entonces encender los 4 bits más significativos de la salida

Si A<B entonces encender los 4 bits menos significativos de la salida

Si A=B entonces encender los 2 bits más significativos y los dos menos significativos de la salida

Cuando se seleccione “001”:

Enviar el mayor a la salida

Cuando se seleccione “010”:

Enviar el menor a la salida

Cuando se seleccione “011”

Enviar a la salida la suma de ambas entradas

Cuando se seleccione “100”

Enviar a la salida la resta del mayor menos el menor

Cuando se seleccione “101”

Enviar a la salida la multiplicación de las entradas

Cuando se seleccione otro enviar A a la parte más significativa y B a la parte menos significativa

Con ese ejercicio queda cubierto el uso de todos los operadores relacionales.

Programación en VHDL:

El código que corresponde a la implementación de un multiplexor es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity practica4 is port(
A, B: in std_logic_vector(0 to 3);
sel: in std_logic_vector(0 to 2);
C: out std_logic_vector(7 downto 0));
end practica4;
architecture Behavioral of practica4 is
signal var: std_logic_vector(0 to 3);
begin
process(sel,A,B) begin
if sel="000" then
  if A/=B then
    if A>B then
    C<=x"F0";
  else
    C<=x"0F";
```

```
  end if;
  else
  C<=x"C3";
  end if;
elsif sel="001" then
  if A>B then
  var<=A;
  else
  var<=B;
  end if;
  C<=x"0" & var;
elsif sel="010" then
  if A<B then
  var<=A;
  else
  var<=B;
  end if;
  C<=x"0" & var;
elsif sel="011" then
  C<=A + B;
elsif sel="100" then
  if A>B then
  C<=x"0" & (A-B);
  else
  C<=x"0" & (B-A);
  end if;
elsif sel="101" then
  C<=B*A;
else
  C<=A & B;
end if;
end process;
end Behavioral;
```

4.2 Práctica 5 – Procesos

Objetivo:

Utilizar la arquitectura del CPLD para realizar procesos secuenciales regidos por un pulso de reloj.

Antecedentes:

Como sabemos, VHDL nos permite programar un dispositivo sabiendo si el valor que tiene una señal de nivel alto o bajo, pero hay una entrada especial en el CPLD que se diferencia de todas las demás porque tiene la capacidad no sólo de conocer el valor de la señal, sino el momento preciso en el que se da el cambio de señal, a este suceso en VHDL se le conoce como *Clock Event*, por lo tanto, si sabemos en qué momento cambia una señal y su valor, podemos manejar sistemas secuenciales que dependan del reloj, como son los contadores, los registros de corrimiento y cualquier máquina de estados que podamos codificar en un CPLD.

En la práctica anterior vimos cómo se maneja un proceso, ahora bien, parte fundamental de lo que compone al proceso es la lista sensitiva, en esta lista se encuentran las señales de las que depende directamente el proceso, las señales de entrada y las señales internas que serán leídas cuando se asigne de manera síncrona las salidas y demás señales internas.

Para hacer uso del *Clock Event*, podemos usar uno de los atributos de la señal que nos devolverá un valor lógico *True* o *False*, el cual pasaremos como argumento en un IF-THEN, considerando que nuestra variable reloj se llama clock:

```
if(clock'event) then
end if;
```

Eso nos permitiría realizar una acción cada vez que el reloj cambie de estado, sin embargo, se acostumbra que todas las asignaciones síncronas se realicen cuando el reloj hace el cambio a 0 ó a 1, en caso de que queramos realizar una acción cuando se tiene lo que se llama flanco de sub ida se hace lo siguiente:

```
if(clock'event and clock='1') then
end if;
```

El listado final para un proceso regido por un reloj es el siguiente:

```
process(clk) begin
```

```
    if(clk'event and clk='1') then
        --Acciones
   end if;
end process;
```

Desarrollo:

Es conveniente dar los primeros pasos en el desarrollo de códigos que involucran procesos con la creación de secuencias de encendido de bit por bit o por grupos de bits, pero algo muy importante que debe considerarse es el hecho de que la velocidad con la que el CPLD hará la secuencia será la misma del Oscilador que tenga conectado, por lo tanto, si tenemos conectado un Oscilador de 1.8432M, entonces el CPLD estará haciendo 1’843’200 instrucciones por segundo, por lo tanto, necesitaremos implementar dentro del propio diseño, una forma de control que nos permita saber cuando haya transcurrido cierta cantidad de tiempo.

Para esta práctica, haremos un programa que haga una secuencia de bits en un bus de 8bits, se encenderá el bit menos significativo, después se apagará para encender el segundo bit, después el tercero y así sucesivamente, cada cambio se hará en un segundo, por lo que se usará un contador, cuando el contador haya llegado a 1’843’200 entonces se realizará alguna acción, el contador será nuevamente puesto a cero y el proceso continuará indefinidamente.

Hay que recordar que para poder leer una señal, esta debe ser de tipo *In*, y las señales de tipo *Out* no pueden ser leídas, pues ISE nos dice que no son del tipo *In*; sin embargo, para hacer un secuenciador, enfocándonos por ahora a lo que es el bus de salida, debemos saber cuál es el último valor que fue asignado al bus, o incluso si éste tiene un valor incorrecto, por lo tanto, tenemos que leer la salida.

Lo que soluciona este problema son las señales de tipo *InOut*, estas señales pueden ser escritas como salidas, pero también pueden ser leídas, también son llamadas señales bidireccionales. Una vez que se tiene la señal de entrada/salida *InOut* se puede hacer la secuencia, como podrá verse en el código, la parte que corresponde al contador también necesita de una señal de entrada/salida, pues para aumentar el contador en uno, primero debe ser leído su valor.

El código que resuelve todos estos problemas es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity practica4 is port(
clk: in std_logic;
bus8: inout std_logic_vector(7 downto 0));
end practica4;
architecture Behavioral of practica4 is
signal contador: std_logic_vector(0 to 23);
begin
  process(clk,bus8) begin
    if(clk'event and clk='1') then
    if(contador<X"1C2000") then
        contador<=contador+1;
      else
        contador<=X"000001";
        case bus8 is
        when X"01"=> bus8 <= X"02";
        when X"02"=> bus8 <= X"04";
        when X"04"=> bus8 <= X"08";
        when X"08"=> bus8 <= X"10";
        when X"10"=> bus8 <= X"20";
        when X"20"=> bus8 <= X"40";
        when X"40"=> bus8 <= X"80";
        when X"80"=> bus8 <= X"01";
        when others=> bus8 <= X"01";
        end case;
      end if;
  end if;
  end process;
end Behavioral;
```

1C2000 es el equivalente en hexadecimal a 1.843M, cuando se trabaja con buses muy grandes, lo mejor es escribirlos con el formato de número hexadecimal, pues así uno se ahorra escribir muchos bits. De hecho, en este ejercicio, todos los valores, incluso los del bus de salida, fueron tratados con hexadecimales.

En el Case, se utiliza el código correspondiente a *others* para poner el Bus con el primer bit encendido, esto se hace así porque inicialmente, todos los bits estarán apagados, pero si hubiera cualquier otro valor, el secuenciador estaría listo para trabajar sin arrojar valores extraños.

Simulación:

Para el caso de la simulación, sería demasiado trabajo para ISE simular todos los ciclos que es necesario transcurran para que el valor de la salida cambie de un valor a otro; por lo tanto, cambiaremos el valor límite del contador para que solamente tenga que contar hasta 4, no se verá reflejado el tiempo total de ciclos que son necesarios para que transcurra un segundo, pero se podrá ver cómo se comporta el programa.

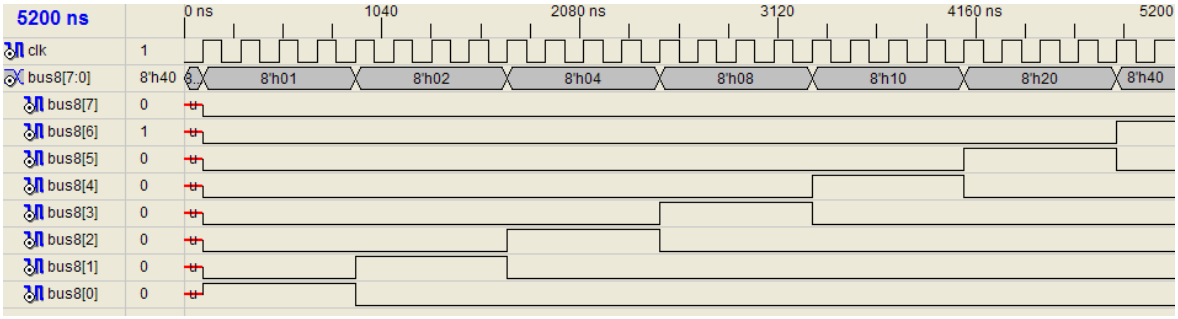


Fig. 4.1 Simulación del contador

El resultado es una gráfica como la que se muestra, en ella puede verse que el número de ciclos que transcurrieron para que la salida cambiara de valor es de 4, no se muestra el contador, ya que es una señal interna; pero se muestra su resultado, los procesos regidos por un reloj se llaman síncronos porque ocurren en el preciso instante en que el reloj cambia de estado.

La señal que se ve antes de que se asigne el primer valor a la salida, es un valor indeterminado, ya que hasta ese momento, como no ha transcurrido el primer ciclo de reloj, nada ha sido asignado a la salida, eso ocurre en una muy pequeña fracción de segundo, y lo más posible es que ese valor desconocido sea de ceros lógicos.

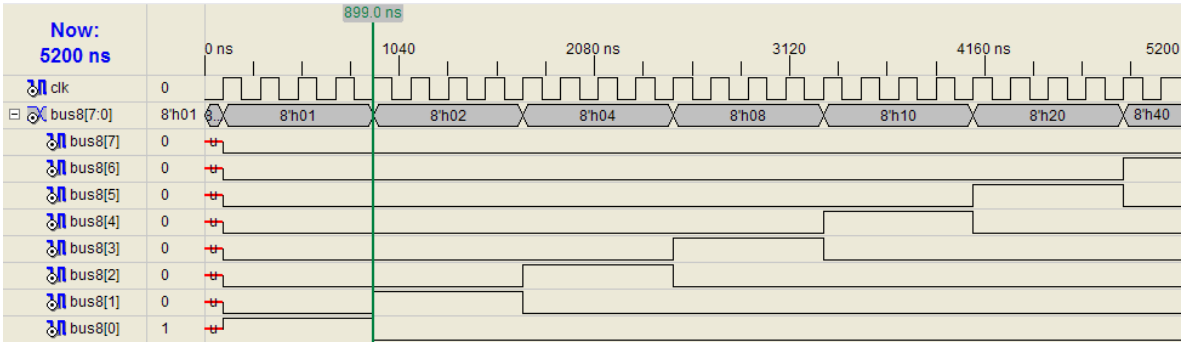


Fig. 4.2 Demostración de la sincronía en las asignaciones del contador

El indicador en la figura 4.2 muestra la sincronía que existe en el cambio de valor del bus de salida y el cambio en el reloj. Los procesos pueden manejar señales síncronas y asíncronas.

4.3 Práctica 6 –Contador basado en 74161 mediante esquemático.

Objetivo:

Programar un circuito basado en el diseño del contador 74161 mediante el desarrollo en VHDL del circuito del contador para ser posteriormente implementado y configurado en un diseño esquemático.

Antecedentes:

El Contador 74161 es un contador síncrono de 4 bits que tiene varias características:

- Reset Maestro, pone a 0 la cuenta
- Habilitador, el cual activa y desactiva el funcionamiento del circuito
- Cargador, nos permite cargar un número a la cuenta
-

Para hacer un contador de este tipo vamos a crear un símbolo que pueda integrarse en un esquemático, ese símbolo no será hecho con un esquemático, sino en lenguaje VHDL, esto con el fin de simplificar el trabajo.

La imagen siguiente muestra el funcionamiento del contador en 4 pasos:

Poner el contador a cero.
Cargar el 12 binario.
Contar 12, 13, 14, 15, 0, 1 , 2.
Inhibir.

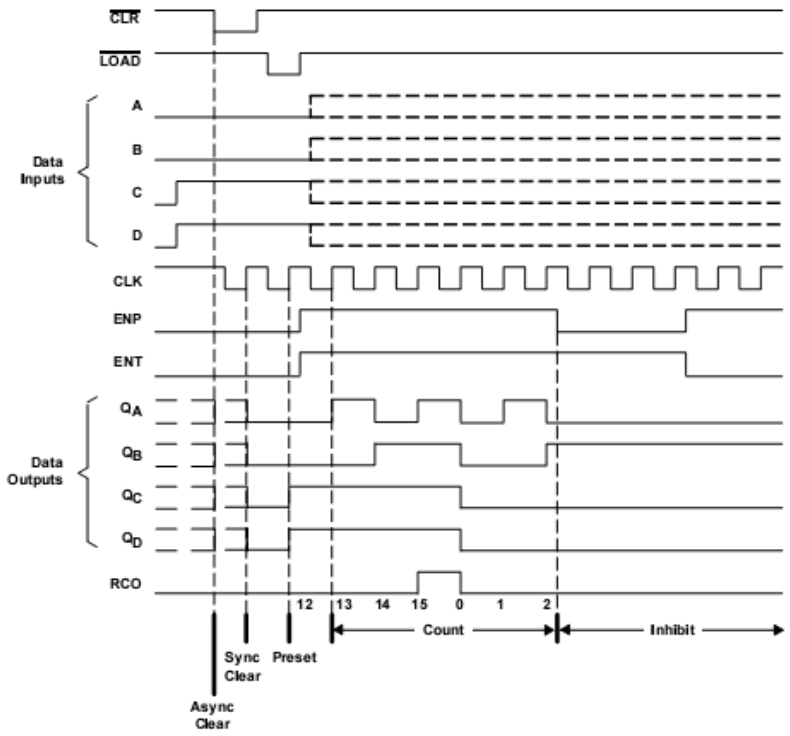


Fig. 4.3 Simulación del contador 74161 según el fabricante

La primer parte de esta práctica consistirá en programar el contador y hacer una simulación que resulte como la de la figura anterior.

La segunda parte consiste en crear un símbolo esquemático a partir de ese contador e incluirlo en un circuito esquemático en el cual se configurará para simplemente contar sin hacer cambios de señales en las entradas.

Desarrollo 1:

El primer paso es programar el contador, el código es el siguiente:

```
library ieee;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library UNISIM;
use UNISIM.Vcomponents.ALL;
entity cont8 is
   port ( clk  : in    std_logic;
          enp : in    std_logic;
          ent : in    std_logic;
          clr  : in    std_logic;
```

```
          d   : in    std_logic_vector (0 to 3);
          load  : in    std_logic;
          rco  : out    std_logic;
          q   : inout std_logic_vector (0 to 3));
end cont8;
--Contador basado en el 74161 de 8 bits
architecture func of cont8 is begin
process(clk, load, enp, ent, clr) begin
  if(clk'event and clk='1') then
    if(load='0') then
       q<=d;
    elsif(ent='1' and enp='1') then
      q<=q+1;
      if(q=x"E") then
       rco<='1';
      else
       rco<='0';
      end if;
    elsif(load='1' and (ent='0' or ent='0')) then
        q<=q;
    end if;
  end if;

  if(q="UUUU") then
   q<=X"0";
   rco<='0';
  end if;
  if(clr='0') then
   q<=X"0";
  end if;

end process;
end func;
```

Para entender mejor el código, lo analizaremos de abajo a arriba, nos encontramos con `if(clr='0')`, esa parte es la que hace la función del reset, no importando los otros valores de las entradas, cuando esta señal vale cero el contador, q, es puesto a cero.

Más arriba se encuentra `if(q="UUUU")`, si omitimos esta parte del código el circuito no funcionará del todo bien, lo que hace esta parte es ver si la señal de salida tiene valores válidos, si tratamos de sumar el contador sin antes haberlo inicializado tendremos salidas incorrectas, quizás altas impedancias, esta sección del código se asegura de que si el contador no tiene valores asignados entonces empiece en cero.

Más arriba tenemos varios IF anidados en `if(clk'event and clk='1')`, el primer IF nos habilita la carga síncrona, el segundo, la cuenta y a su vez tiene un IF que controla la habilitación de la señal de Carry (rco) cuando el contador ha llegado a su máximo valor y volverá a comenzar.

El siguiente IF, `elsif(load='1' and (ent='0' or ent='0'))`,es la sección en la que el contador se 'inhibe' y detiene su cuenta. Con estas simples instrucciones construimos un dispositivo capaz de actuar como un contador 74161.

La simulación se puede observar en la siguiente figura:

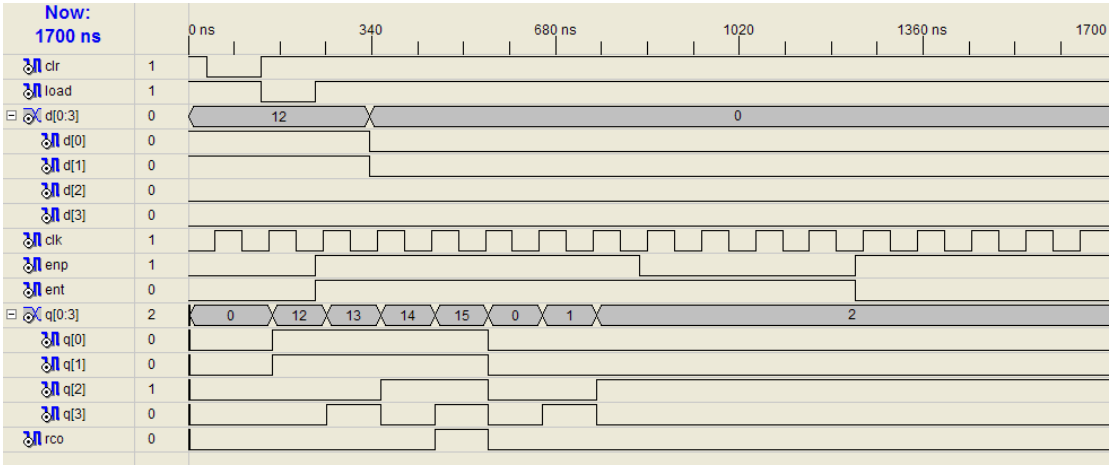


Fig. 4.4 Simulación del proyecto, idéntica a la del fabricante

El contador funciona a la perfección, con la diferencia de que aunque está listo para ser conectado en serie con otros contadores, solo hace falta una pequeña modificación del código para que el contador sea de 8 o de los bits que sea necesario.

Desarrollo 2:

Usaremos el mismo diseño de contador, pero el lector ha de hacer los cambios necesarios al código para que funcione como contador de 8 bits.

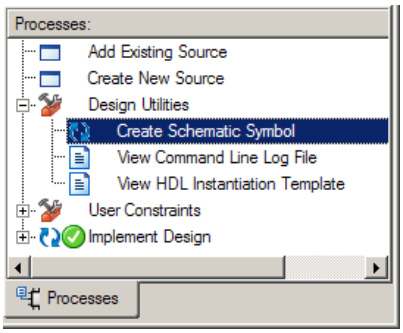


Fig. 4.5 Crear símbolo esquemático

El primer paso será generar el Símbolo Esquemático, para eso se selecciona el archivo .vhd del contador y en la ventana de procesos de ISE se selecciona la opción *Design Utilities* y luego *Create Schematic Symbol*.

Aparentemente no habrá cambio alguno.

A continuación, presionamos CTRL+N o *menú File->New*, esto creará un nuevo esquemático, lo salvamos.

Se añade el archivo al proyecto actual.

Se da *click* con el botón derecho sobre el archivo recién agregado y se selecciona la opción *Set as Top Module* y/o se borra el archivo del contador de la ventana de *Sources,* en la pestaña Symbols de esta ventana se busca el símbolo que acabamos de crear, y se añade al esquemático, debería verse así:

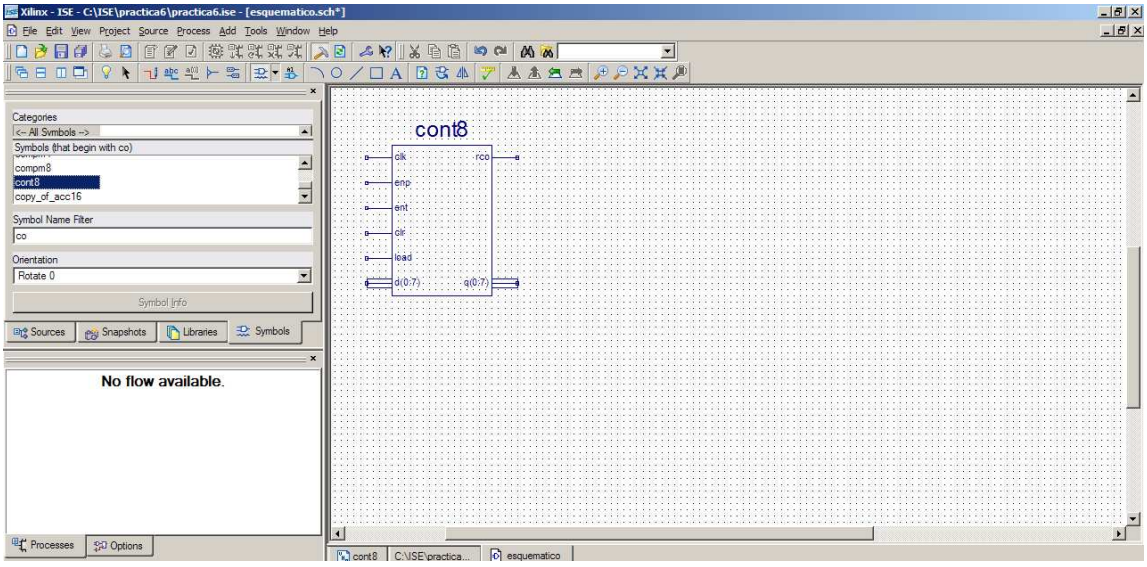


Fig. 4.6 Símbolo esquemático creado

Para configurar nuestro contador seguimos estos pasos:

Conectamos un bus a la entrada d(0:7), la herramienta de bus es la herramienta *Wire*, ésta dibuja Buses y Cables sencillos, se habilita con CTRL+W o el botón .

Notaremos que se pinta una línea gruesa, esa línea la extenderemos para que sea más larga aún.

Con la herramienta *BusTap* conectamos 8 *BusTap* a la línea de bus.

Es importante conectar cada *BusTap* a un Buffer por medio de un *Wire*, pues a la línea del Bus y a cada uno de los 8 *Wires* se les pondrá un nombre y tienen que ser seleccionables en el esquemático

Para introducir un Buffer al esquemático se busca el elemento *buf* en la lista. Todos los buffer que han sido conectados a los BusTap se conectan a tierra (elemento *gnd* en la lista).

Se selecciona la herramienta *Add Net Name* y en la ventana Processes se seleccionan las siguientes opciones:

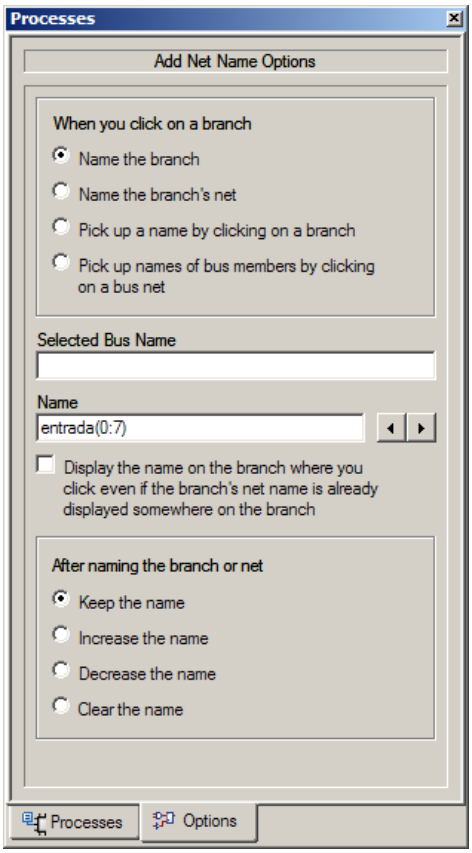


Fig. 4.7 *Configuración del nombre del pin.*

Con esas opciones se selecciona el Bus, de tal forma que aparecerá un nombre sobre él, ahora se llamará entrada(0:7) (ó el nombre que haya elegido).

Con los *Wires* que debe haber entre los *BusTap* y los Buffers se hará algo parecido, sólo que se llamará el primero entrada(0), y en la ventana de *Processes* se activa el radio que dice *Increase the name*, aunque esto se puede hacer manualmente.

Al terminar de hacer estas configuraciones se conectan las demás entradas a un elemento del tipo vcc, excepto la de reloj.

Con el botón *Add I/O Marker* se añaden los pines de entrada y salida, si al hacer click en el botón  la ventada *Processes* tiene activa la opción *Add an automatic marker*, basta con dar *click* sobre el pin CLK, que es una entrada para que se dibuje un puerto de entrada, un *click* en RCO para que se dibuje una salida y un click en q, que es una entrada/salida, automáticamente al hacer click sobre ésta, por ser una entrada/salida de 8 bits se dibuja el bus apropiado.

La figura 4.8 que se muestra a continuación es un ejemplo de como se debe ver finalmente el contador configurado.

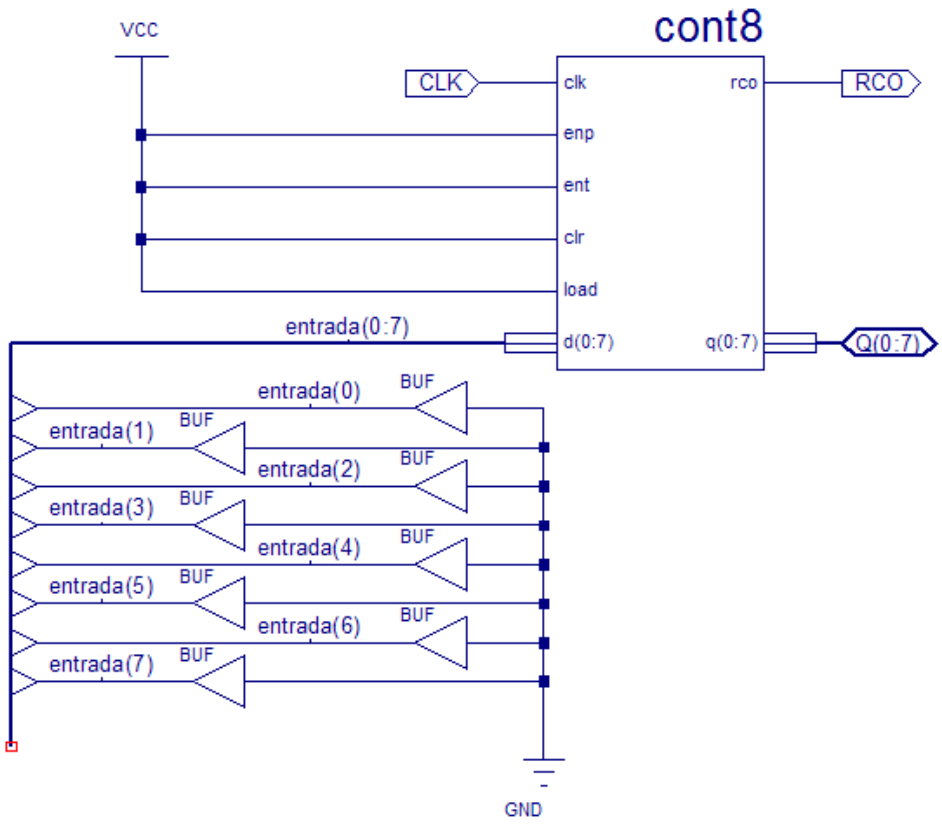


Fig. 4.8 Esquemático terminado

A continuación se hace la simulación del circuito.

Primero hacemos *click* en el botón *Check Schematic*  para comprobar que el circuito es correcto.

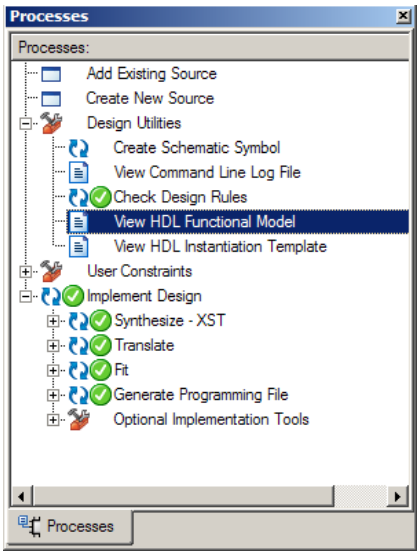


Fig. 4.9 Paso importante antes de simular, definir que se trate todo como VHDL

Después, implementar el diseño.

Si no existe problema con este diseño, se puede hacer la simulación, pero antes debe hacerse un cambio muy importante en la configuración, en la opción *Design Utilities* de la ventana *Processes*, se da *click* con el botón derecho en la opción *View HDL Functional Model.* En el cuadro de diálogo que aparece se selecciona VHDL en la opción *Functional Model Target Language*.

De no hacer esto la simulación no se llevará a cabo pues el simulador estará tomando el diseño como si fuera programado en Verilog[6].

Ese es el único cambio que hay que hacer para poder realizar la simulación.

Conclusiones

Los procesos son quizás la herramienta más poderosa del CPLD y VHDL, pues permiten definir el control del programa con ciclos de reloj eligiendo el momento preciso en el que se llevarán a cabo las acciones, esto permite tener procesos secuenciales, combinacionales y mixtos.

Ahora conocemos los operadores de VHDL, y la forma de usar los procesos, esto aunado a las sentencias de control que ya hemos revisado nos dan todas las herramientas necesarias para las prácticas avanzadas.

[6] Verilog es un Lenguaje de Descripción de Hardware (HDL) con el que se modelan sistemas electrónicos, fue creado para implementar circuitos analógicos, digitales y mixtos con una sintaxis parecida a la de C.

5. Prácticas Avanzadas

Una forma de trabajar para diseñar sistemas en VHDL es mediante el uso de máquinas de estados, en el diseño de máquinas de estados estará enfocado este capítulo, se diseñarán máquinas de estados usando diferentes técnicas clásicas para ejemplificar su uso dentro de Xilinx ISE y también la forma en que Xilinx ISE ayuda en el mejoramiento de estas técnicas como son el diseño mediante Multiplexores, Registros de corrimiento, contadores, memorias RAM, se puede trabajar incluso por encima de estos diseños clásicos mediante el uso de variables de estados que Xilinx ISE identifica y añade automáticamente al diseño facilitándolo y simplificándolo en gran medida.

Para terminar, se muestra el diseño de una memoria RAM para completar el conjunto de herramientas con que se puede potencializar el uso de los CPLD para llevar a cabo cualquier tipo de diseño digital.

5.1 Práctica 7 - Máquina de estados sobre un diseño basado en el Registro de corrimiento 74194

Objetivo:

En esta práctica se hará la primera máquina de estados de este conjunto de prácticas, el diseño estará basado en un registro de corrimiento, y mediante el empleo de dos procesos totalmente distintos; además, se hará el diseño desde cero primero para el registro y después se hará el mismo diseño pero codificado en VHDL. Al final de esta práctica el lector tendrá los conocimientos para desarrollar circuitos complejos, máquinas de estado y la base para hacer sistemas completos en un sólo diseño en VHDL valiéndose de uno o más procesos.

Antecedentes:

Es preciso para el desarrollo de esta práctica recordar el funcionamiento del registro de corrimiento (de cual de todos los registros de corrimiento estas hablando), su tabla de verdad es la siguiente:

S0	S1	Acción
0	0	Sostener (Hold)
0	1	Corrimiento a la izquierda
1	0	Corrimiento a la derecha
1	1	Cargar

Trataremos de manejar el menor número de cargas posibles realizando un mayor número de corrimientos, construiremos mapas de Karnaugh para cada entrada del Registro en que sea necesarios hacerlo, como una muestra extra se construirá también el esquemático; sin embargo, su fin es meramente ilustrativo.

Desarrollo:

A continuación se muestra el diagrama de estados de la máquina que se creará, dentro de cada estado se encuentra un número de 3 bits que lo identifica como único, la transición de un estado a otro está identificada por una flecha, el sentido indica de qué estado a qué estado se pasa, las flechas que tienen un texto son transiciones que dependen de una variable, las que no tienen variable son saltos directos. Los estados que tienen un par de flechas indican la salida que se activará. ||||||||||||||||||||||||||||||||||||||||

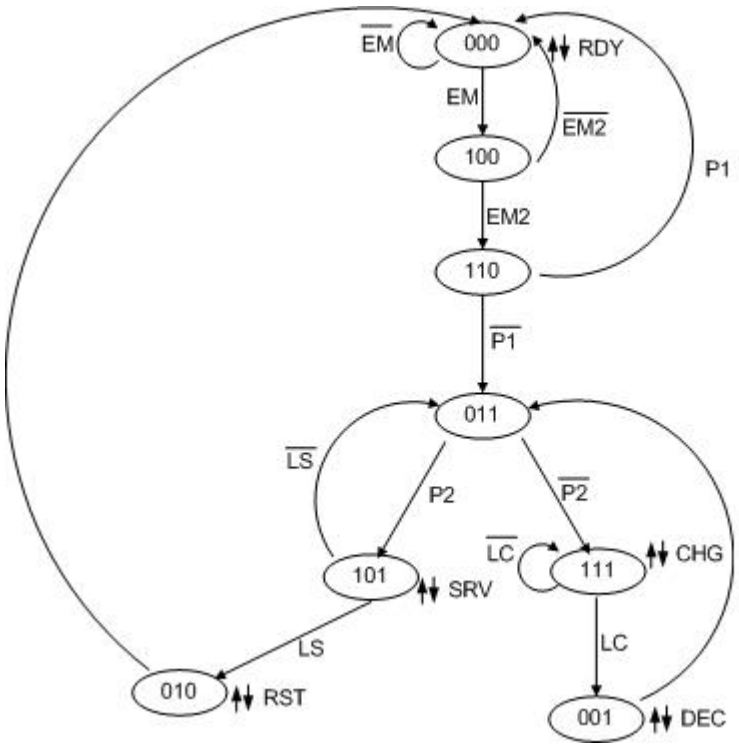


Fig. 5.1 Diagrama de estados

Estado 000

Obtendremos los mapas de Karnaugh, primero para el estado “000”, como puede verse ocurre un corrimiento a la derecha con una carga de uno “000”->”100” ó un Hold, es decir, se mantiene el estado (S=”10” ó S=”00”). Mirando la tabla de verdad observamos que S1 se conserva en cero, y que cuando S0 vale uno es cuando se hace el corrimiento, por lo tanto, la variable suscrita en el mapa de Karnaug será:

Q3 Q2	S0			
Q1	00	01	11	10
0	EM			
1				

Q3Q2	S1			
Q1	00	01	11	10
0	0			
1				

Como la carga es en el corrimiento a la derecha y vale uno, se agrega a R en su mapa, y la carga en L no importa por lo que se pone un *.

Q3 Q2	R			
Q1	00	01	11	10
0	1			
1				

Q3Q2	L			
Q1	00	01	11	10
0	*			
1				

No se hacen cargas, por lo que se colocan asteriscos también en P, que es la variable en la que se controlan los saltos de un estado a otro:

Q3 Q2	P3			
Q1	00	01	11	10
0	*			
1				

Q3Q2	P2			
Q1	00	01	11	10
0	*			
1				

Q3 Q2	P1			
Q1	00	01	11	10
0	*			
1				

Estado 100

En el diagrama de estados, el siguiente es “100”, éste puede regresar a “000” ó pasar a “110”, como puede notarse, se trata únicamente de corrimientos, a la derecha con carga de cero o a la izquierda con carga de uno. El salto a la derecha se hace con S0=1 y S1=0 (S=“10”), como depende de EM2, la tabla queda así para el estado “100”:

Q3 Q2	S0			
Q1	00	01	11	10
0	EM			EM2
1				

Q3Q2	S1			
Q1	00	01	11	10
0	0			~/EM2
1				

Agregamos las cargas, un uno para la izquierda, un cero para la derecha.

Q3 Q2	R			
Q1	00	01	11	10
0	1			0
1				

Q3Q2	L			
Q1	00	01	11	10
0	*			1
1				

Y nuevamente las cargas no llevan un valor, a partir de este punto el procedimiento será el mismo, no se mostrarán los mapas paso a paso, pero sí como se obtuvieron estado por estado.

"110" Se presenta un salto o un corrimiento a la derecha, S="11" ó S="10", S1=1 cuando P1, se agrega esta información a los mapas S0 y S1 y el salto al mapa R, se agregan ceros en los mapas de P, quedando los mapas de L y R con un asterisco en 110.

011 En este estado se presentan solamente corrimientos, a la derecha con carga de uno para caer en 101 ó a la izquierda con carga de 1 para caer en 111. La información de los mapas es S0=P2 y S1=~P2, pues los dos corrimientos en la tabla de verdad son el equivalente a una XOR, cuando P2=1 se hace corrimiento a la derecha (S=10), cuando P2=0 se hace corrimiento a la izquierda (S=01).

101 Muy parecido, se hacen dos corrimientos, pero ahora dependen de LS, la información de los mapas es similar salvo porque uno de los corrimientos es con una carga de 0.

010 Solamente hay un salto, S=11 se ponen ceros en los mapas de P, L y R llevan *

111 Realiza un hold ó un salto, S=00 ó S=11, el salto se da con LC=1, por lo tanto, en S0=LC y S1=LC, el salto es a 001, por lo que P1=1.

001 Tiene siempre un corrimiento a la izquierda con 1, S=01, L=1.

Q3 Q2	S0			
Q1	00	01	11	10
0	EM	1	0	EM2
1	0	P2	LC	LS

Q3Q2	S1			
Q1	00	01	11	10
0	0	1	P1	~EM2
1	1	~P2	LC	~LS

Q3 Q2	R			
Q1	00	01	11	10
0	1	*	*	1
1	*	1	*	0

Q3Q2	L			
Q1	00	01	11	10
0	*	*	*	0
1	1	1	*	1

Q3 Q2	P3			
Q1	00	01	11	10
0	*	0	0	*
1	*	*	0	*

Q3Q2	P2			
Q1	00	01	11	10
0	*	0	0	*
1	*	*	0	*

Q3 Q2	P1			
Q1	00	01	11	10
0	*	0	0	*
1	*	*	1	*

En base a los mapas de Karnaugh, se simplifican las ecuaciones de P, R y L, no es necesario simplificar S0 y S1 porque son muchas variables suscritas, en base a estas simplificaciones, las ecuaciones finales son:

R = Q2 xnor Q1

L = Q1

P3 = 0

P2 = 0

P1 = Q1

Y S0, S1 van tomando sus valores en cada caso mediante un multiplexor, es más fácil de verlo así.

Esta solución puede verse reflejada en el siguiente circuito, al cual las salidas son asignadas en cada estado mediante un decodificador.

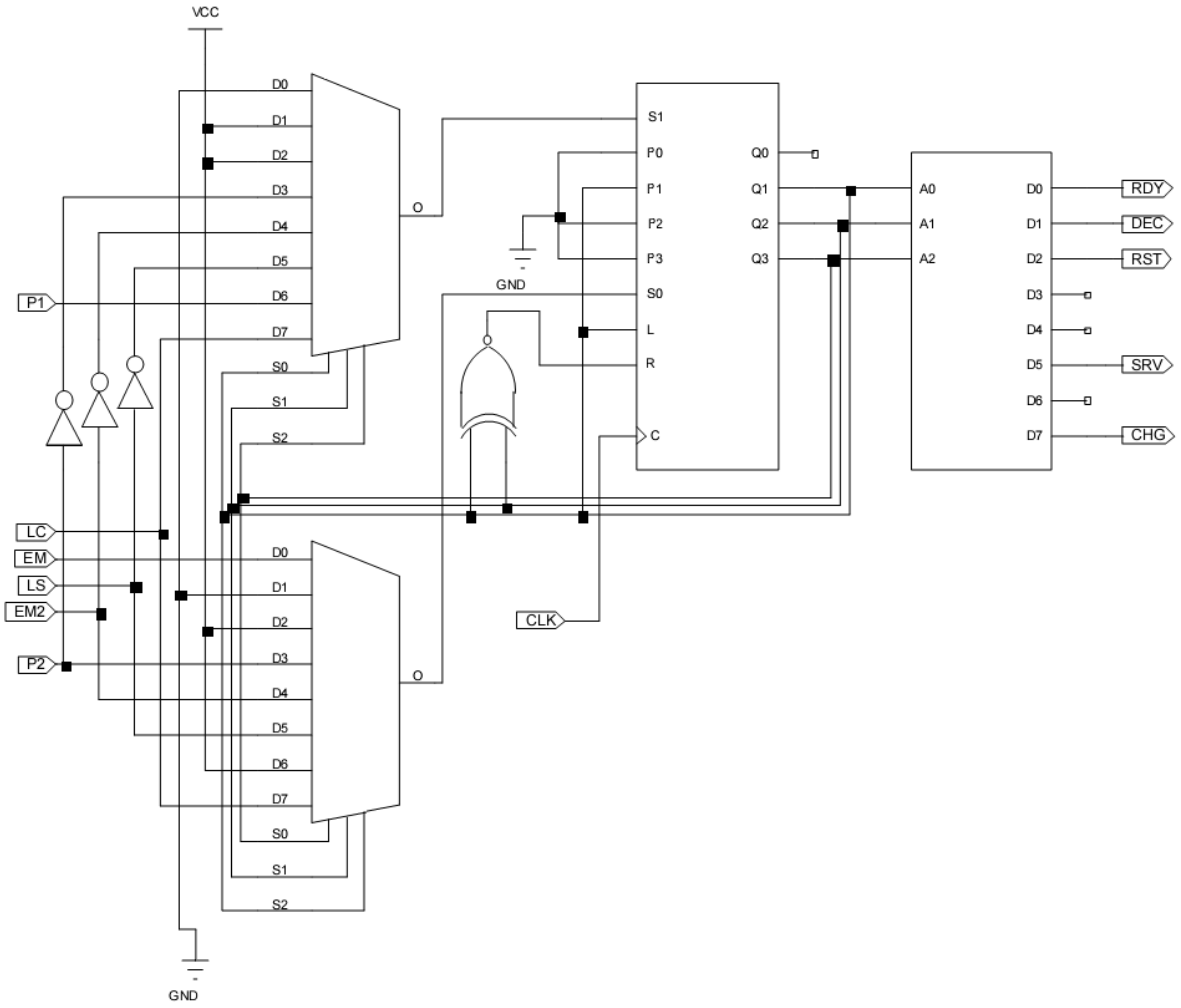


Fig. 5.2 Circuito solución

Lo que se hace a continuación es escribir un código en el cual se maneje de forma equivalente el circuito anterior.

En un proceso se hará la función del registro de corrimiento, en otro se tratarán las señales combinacionalmente. Es muy importante separar el código que debe ser combiacional y manejarlo como tal, por lo que la única parte síncrona del diseño será la que ejecuta las opciones del registro de corrimiento.

El código solución es el siguiente:

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

entity practica7 is port(
```

```
CLK, EM, EM2, P1, P2, LS, LC: in std_logic;
RDY, SRV, CHG, RST, DEC: out std_logic);
end practica7;
architecture Behavioral of practica7 is
signal P, Q: std_logic_vector(3 downto 1);
signal S: std_logic_vector(0 to 1);
signal L,R: std_logic;
begin
process(CLK, Q, P, L, R) begin
   if(CLK'event AND CLK='1') then
       case S is
            when "01"=> Q <= Q(2) & Q(1) & L;
            when "10"=> Q <= R & Q(3) & Q(2);
            when "11"=> Q <= P;
            when others => Q <= Q;
       end case;
 end if;
 if(Q="UUU") then
       Q<="000";
 end if;
end process;
process(Q, EM, EM2, P1, P2, LS, LC) begin
 case Q is                    --Multiplexores
       when "000"=>    S(0) <= EM;
                       S(1) <= '0';
       when "001"=>    S(0) <= '0';
                       S(1) <= '1';
       when "010"=>    S(0) <= '1';
                       S(1) <= '1';
       when "011"=>    S(0) <= P2;
                       S(1) <= NOT P2;
       when "100"=>    S(0) <= EM2;
                       S(1) <= NOT EM2;
       when "101"=>    S(0) <= LS;
                       S(1) <= NOT LS;
```

```
      when "110"=>    S(0) <= '1';
                      S(1) <= P1;
      when others =>  S(0) <=LC;
                      S(1) <=LC;
 end case;
 P(3)<= '0';
 P(2)<= '0';
 P(1)<= Q(1);

 R<= Q(2) XNOR Q(1);
 L<= Q(1);
                              --Decodificador
 RDY<= NOT(Q(3)) AND NOT(Q(2)) AND NOT(Q(1));
 SRV<= Q(3) AND NOT(Q(2)) AND Q(1);
 RST<= NOT(Q(3)) AND Q(2) AND NOT(Q(1));
 CHG<= Q(3) AND Q(2) AND Q(1);
 DEC<= NOT(Q(3)) AND NOT(Q(2)) AND Q(1);
end process;
end Behavioral;
```

En el código se muestra como las salidas Q del registro de corrimiento están regidas por un reloj, tal como se realizó en el circuito, también se inicializa la salida Q evitando que tenga valores indeterminados.

En el otro proceso se asignan todas las entradas que van al registro de corrimiento, y por medio de la estructura CASE se realiza el proceso que corresponde a la asignación de valores a S0 y S1 por medio de un multiplexor

Las salidas Q del registro de corrimiento pasan por compuertas para obtener las entradas que van a P, R y L; sin embargo, debido a la simplicidad que buscaba este diseño, a lo sumo se utiliza una compuerta XNOR y otra señal se toma directamente.

Las salidas RDY, SRV, RST, CHG, DEC son asignadas mediante ecuaciones booleanas para simplificar la lógica que se requiere para programarlas, si se hiciera de otro caso, tendrían que listarse los valores de los estados que provocan la activación de alguna de las mencionadas salidas, después de listarse, se asignaría cada una, pero necesitaríamos otra porción de código en la cual todas las demás se pongan en cero, esto

para cada una de las salidas y para todas en general, lo cual, lleva tiempo de programación y líneas extra de código que evidentemente son innecesarias.

Ha de ponerse especial atención en las listas sensitivas de los procesos, es decir, la lista en la que están todas las variables de entrada de las que depende él, en estas listas se puede ver más claramente cómo se usan las variables de entrada para cada proceso, distinguiendo notablemente la independencia de los procesos.

La simulación de esta máquina de estados es la siguiente:

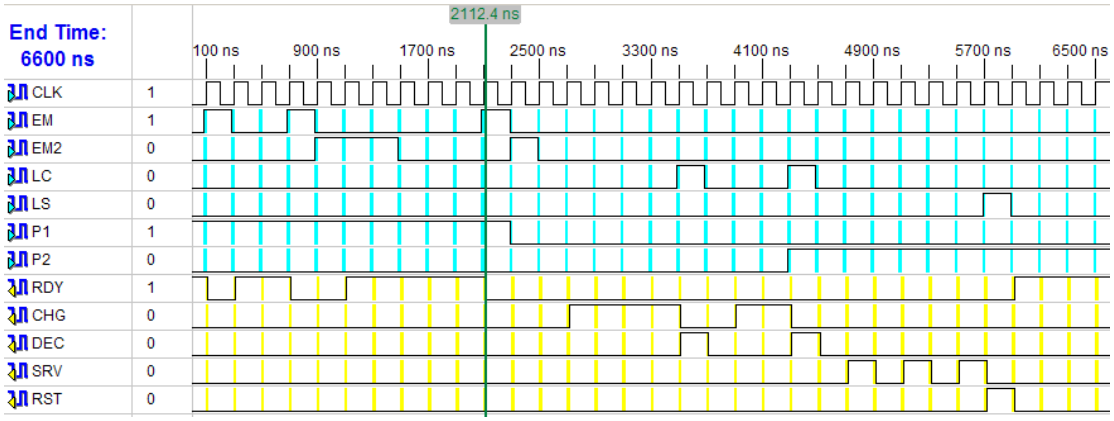


Fig. 5.3 Simulación de la máquina de estados

Es recomendable ver esta simulación junto con el diagrama de estados, puede verse que la máquina inicia con la salida RDY activa, esto significa que estamos en el estado 000, avanzando un poco en la línea de tiempo puede verse que cuando EM vale 1, RDY vale 0, eso significa que la máquina abandona el estado 000, sin embargo, vuelve a caer allí porque EM2 vale cero, lo que provoca un salto de 100 a 000, esto se repite hasta el indicador de tiempo, después del cual la máquina ya no cae en el estado 000 y RDY permanece en 0. Puede verse la activación de CHG, lo que indica que la máquina llega a 111, posteriormente se activa SRV tres veces, hasta que la entrada LS toma el valor de 1, cuando la máquina pasa al estado 010 activando RST y regresando al estado 0.

Ninguno de estos cambios ocurre si no se controlan las señales de entrada correctamente simulando la interacción de la máquina con sus dispositivos periféricos.

Si gusta, puede modificar el código y tomar Q como una entrada/salida para visualizar el cambio de un estado a otro en la simulación, con esto logrará comprender mejor cómo se hace todo el proceso dentro de la máquina de estados, pero ésta debe ser

siempre una caja negra y estar oculta, por lo que la señal Q y ninguna otra se muestran al exterior.

La simulación debe quedar como se ve a continuación, donde es totalmente claro el funcionamiento de la máquina acorde al diagrama de estados.

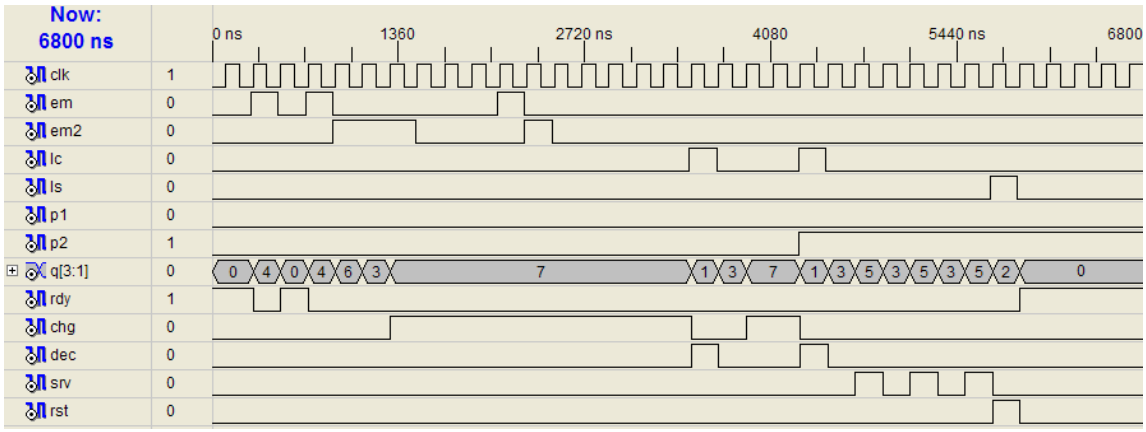


Fig. 5.4 Simulación con estados enteros

5.2 Práctica 8 - Máquinas de Estado Avanzadas

Objetivo:

En esta práctica se trabajará con los procedimientos habituales para crear máquinas de estado, que a diferencia de la máquina de la práctica anterior, no requieren de conocimientos previos sobre diseño lógico, y que al ser presentadas como *Cajas Negras* pueden seguir un estilo de diseño propio y diferente a todos los demás, lo único que se necesita saber es el lenguaje VHDL, y aún así Xilinx ISE lo facilita todo un poco más.

Antecedentes:

Esta práctica se desarrollará en dos partes, primero, se hará una máquina de estados como la del ejercicio anterior con ayuda del Asistente de Máquinas de estado de Xilinx ISE; en segundo lugar se hará una máquina de estados siguiendo un estilo propio de programación.

Desarrollo 1:

Inicie un nuevo proyecto en Xilinx ISE de click en *"Create new source"*, seleccione *State Diagram* y de un nombre a su diagrama, que será el nombre de la entidad de diseño, de *click* en Next y luego en *Finish* Aparecerá una nueva ventana de StateCAD como la siguiente:

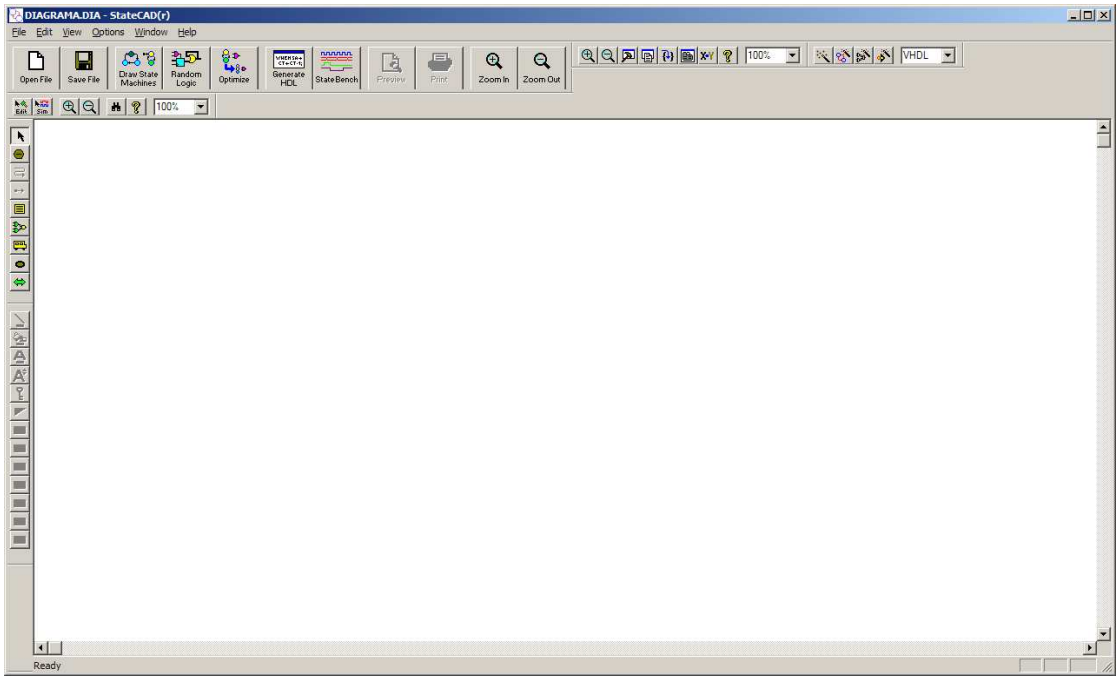


Fig. 5.5 Pantalla de StateCAD®

En esta ventana se crearán y editarán máquinas de estados completas en modo gráfico. Para comenzar, dé *click* en *"Draw state machines"*  y seleccione *Multi-Column* en el nuevo cuadro de diálogo, como haremos la máquina de estados de la práctica anterior, ésta es la que más se asemeja y ponemos el número de estados en 6.

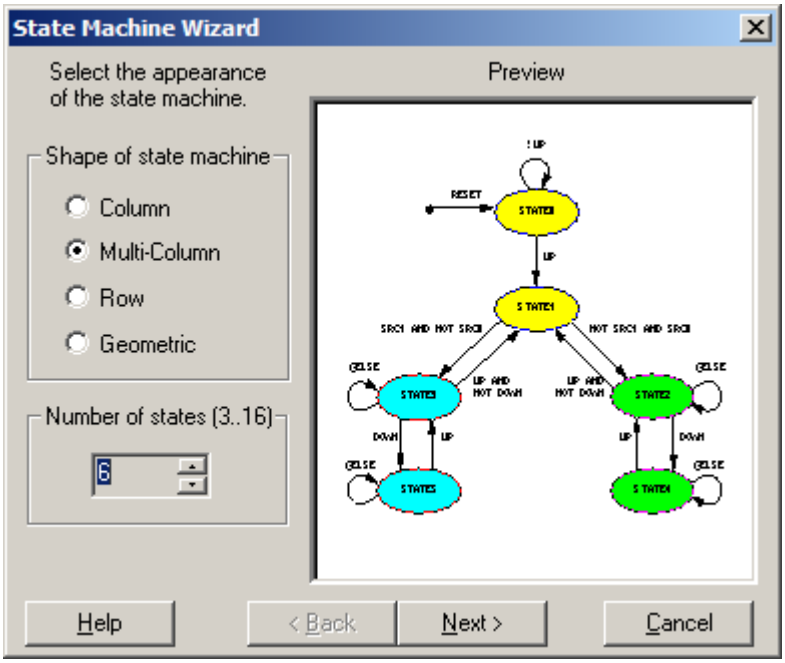


Fig. 5.6 Asistente de creación de máquinas de estados

El asistente nos pregunta cómo queremos que funcione el Reset General de la máquina de estados, que no fue incluido en la práctica anterior. Seleccionaremos un reset asíncrono.

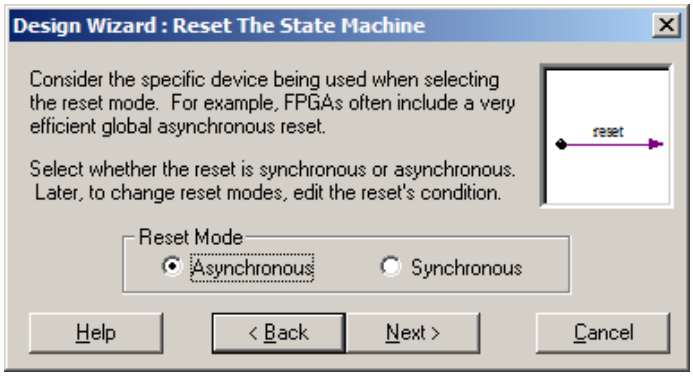


Fig. 5.7 Configurando el Reset de la máquina

A continuación, el asistente nos permite configurar la forma en que serán agregados los estados para comenzar a trabajar y también cómo funcionarán, por ahora no nos será

de mucha ayuda agregar un *Loop* a todos los estados desde el comienzo, por lo que desactivamos la opción *Loop Back*.

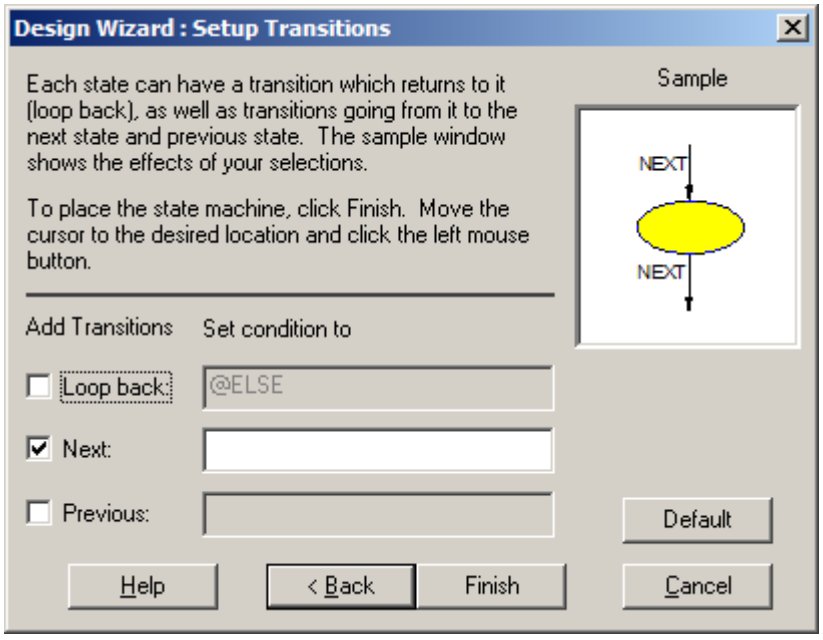


Fig. 5.8 Configurando las transiciones

Después de esto podemos terminar pulsando el botón *Finish*. El cursor del mouse es seguido de un cuadro amarillo, al dar click sobre el área en blanco de la página, podremos colocar el diagrama de estados.

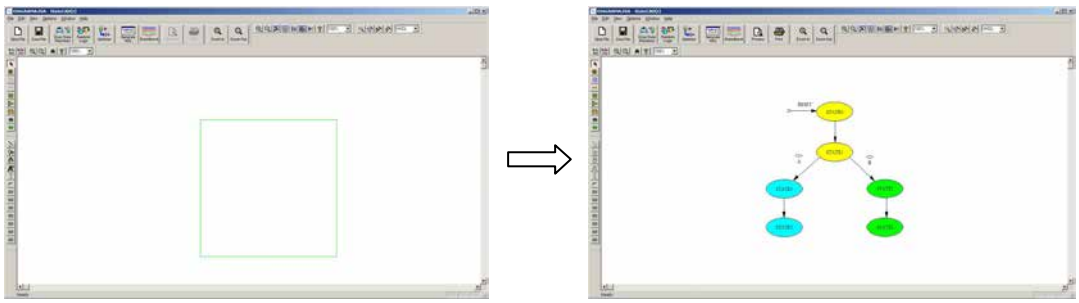


Fig. 5.9 Colocar el nuevo diagrama

Dos estados tienen transiciones con variables, las seleccionaremos y las eliminaremos. A continuación, seleccionamos los estados que están en color amarillo y los desplazamos hacia arriba, o seleccionamos los demás y los desplazamos hacia abajo con el fin de insertar más objetos.

En la barra que aparece a la izquierda damos click en el botón *Add State*  . Dibujamos los nuevos 2 estados.

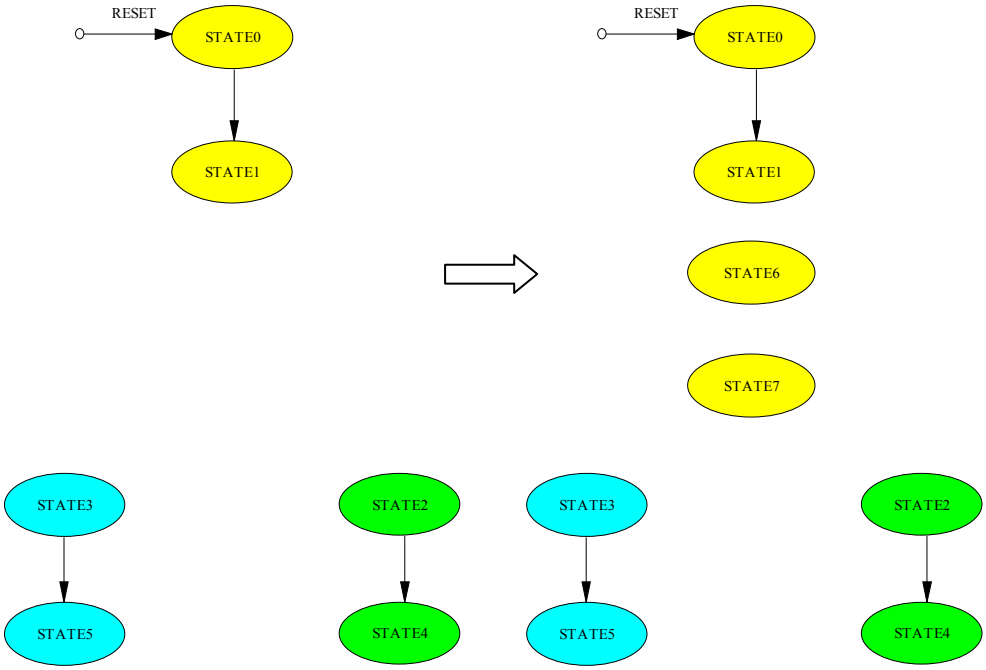


Fig. 5.10 Modificación del diagrama

Debajo del botón *Add State*, se encuentra el botón *Add Transtition*  , lo ocuparemos para agregar las transiciones, damos click en el estado de que nace la flecha que representa la transición y un segundo click en el estado al que va la transición, si damos click dos veces en el mismo estado al crear la transición, se hará un loop.

Las flechas tienen dos puntos de control, sirven para curvarlas, esto nos ayuda a crear todas las transiciones que saltan por varios estados.

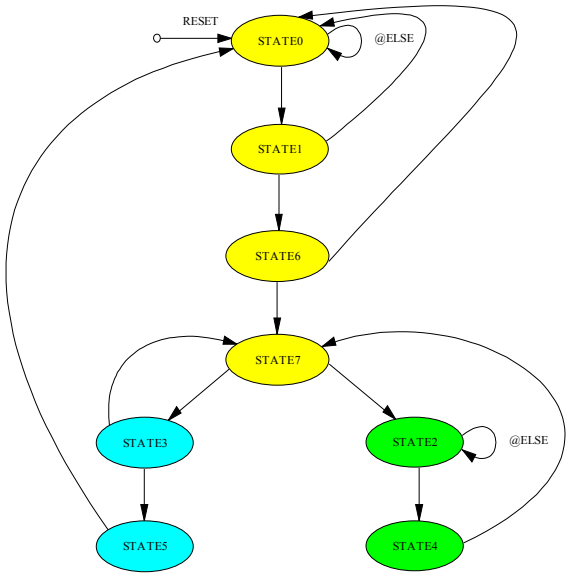


Fig. 5.11 Forma final del diagrama

El diagrama de estados ya tiene su forma final, puede nombrar los estados para corresponder como el ejemplo, pero si no lo hace, no habrá ningún problema, en este ejemplo lo haré.

El siguiente paso es asignar los valores de las transiciones, para esto daremos doble click en la transición. En este caso, dar *click* en la transición que va del estado 0 al 4 (000 a 100). El cuadro de dialogo es el siguiente:

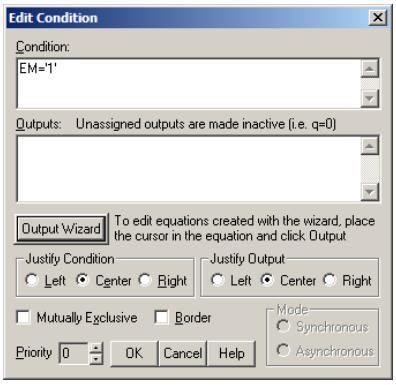


Fig. 5.12 Configurar transiciones

En donde dice condición, escribiremos la condición que debe cumplirse para que se haga la transición, en este caso, EM=1, nótese que debe usarse la misma forma que se usa en VHDL: **`EM='1'`**. Procedemos de la misma forma hasta completar todos los estados.

EL diagrama debe de verse así:

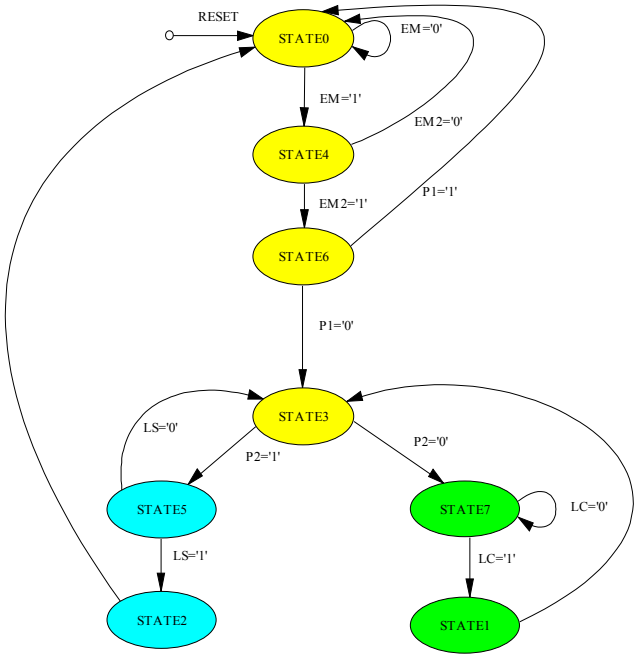


Fig. 5.13 Diagrama con transiciones configuradas

Finalmente, se asignan las salidas, para esto hay que dar doble click sobre el estado que dará una salida, recuerde usar la herramienta de selección: 

Para agregar una salida, después de dar *click* aparecerá el siguiente cuadro de diálogo, en el cual se puede cambiar el nombre del estado y también la salida que asigna. Comenzando con el estado 0:

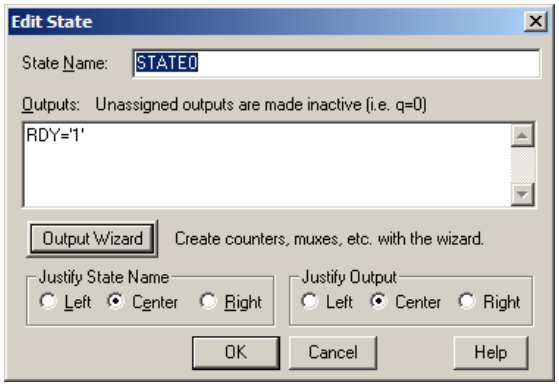


Fig. 5.14 Edición del estado

La salida asignada es: **`RDY='1'`**.

Se procede de igual forma para todos los estados que asignan una señal de salida, el diagrama terminado debe verse así:

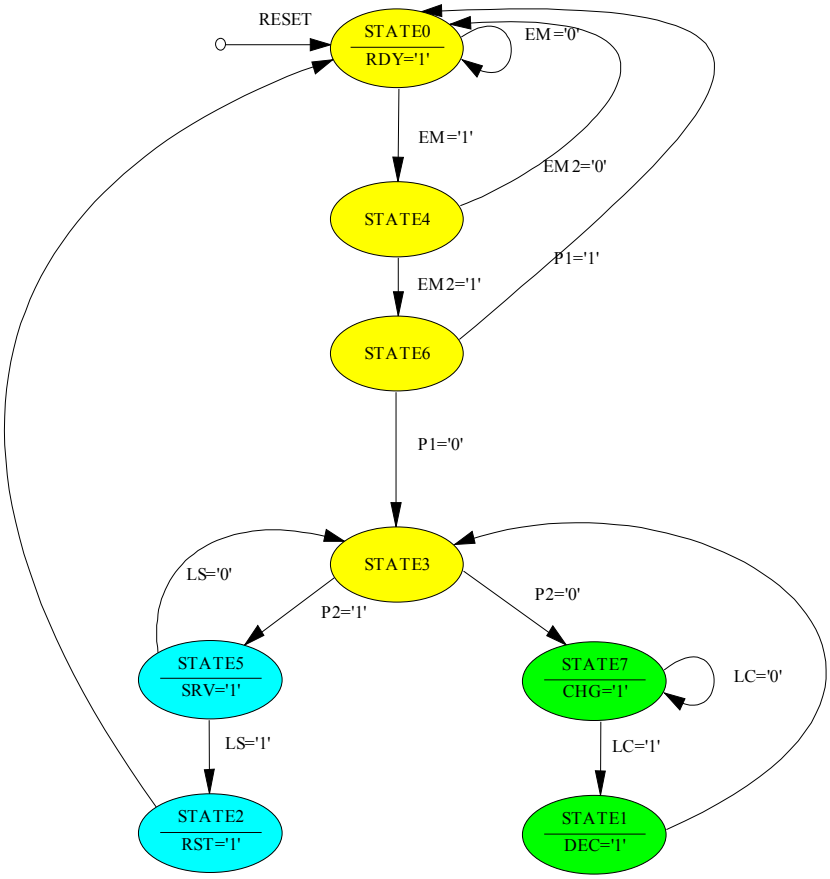


Fig. 5.15 Forma final del diagrama de estados

Ahora que el diagrama está terminado, el siguiente paso es obtener el código HDL que automáticamente se generará a partir de este diagrama, para obtenerlo hay que hacer click en el botón *Compile* 

.

Si todo se ha hecho correctamente, después de enviar un mensaje, StateCAD nos muestra el código solución de la máquina de estados, éste código tiene una extensión .vhd, por lo que puede ser incluido en el proyecto, siempre y cuando se abra un archivo con un nombre distinto o el diagrama no haya sido añadido anteriormente al proyecto.

Este código tiene más de cien líneas, razón por la cual esta práctica se divide en dos, a continuación, se ofrece otra forma de codificar máquinas de estados.

Desarrollo 2:

Mediante el uso de todas las técnicas de diseño en VHDL aquí vistas, es posible hacer un código de menos de la mitad de las líneas que el anterior que cumpla con la misma función sin diferencia alguna, para eso usaremos un poco de Diseño Lógico, Ecuaciones Booleanas, asignación CASE-WHEN y procesos, esto llevará el código a su mínima expresión, explotando la gran facilidad y versatilidad de VHDL en el diseño de sistemas de máquinas de estado.

Las asignaciones serán del estilo del decodificador de la Práctica 7, es decir del tipo:

```
RDY<= NOT(Q(3)) AND NOT(Q(2)) AND NOT(Q(1));
```

Esto nos facilitará mucho el trabajo, pues es algo que ya hemos hecho antes, esta parte del código, seguirá siendo combinacional. Dentro de una estructura WHEN-CASE, listaremos a todos los estados, aquí es donde se manejarán las transiciones.

Para el caso del estado 000 se presentan dos escenarios:

El primero consiste en señalar el estado y posteriormente decidir la transición por medio de una sentencia IF-THEN-ELSE, como se muestra a continuación:

```
case STA is
when "000"=>if(EM='1')
STA<= "100";
else
STA<= "000";
end if;
```

Donde STA es la variable en la que se almacena el estado actual y STA2 será el estado futuro.

Para el segundo escenario, podemos observar que cuando EM=1 el bit más significativo, por lo tanto podemos escribir un código equivalente al anterior de la siguiente manera:

```
case STA is
when "000"=> STA2<=EM & "00";
```

La razón por la cual el estado se asigna a una variable temporal es que si se hace de otra forma la entrada de la variable que provoca una transición debe coincidir con el pulso de reloj, si el pulso de reloj tiene intervalos de 15 nanosegundos y el pulso de la entrada 10 nanosegundos puede suceder que tal entrada no quede registrada, lo cual

provocaría un error en el funcionamiento de la máquina de estados, por lo tanto, registraremos el cambio de manera combinacional y asignaremos el cambio a un estado previamente seleccionado hasta el momento en el que se presente un cambio en el estado del reloj.

Por lo tanto, la sentencia *case STA is*, está contenida dentro del proceso pero fuera de la sentencia del cambio del estado del reloj, la única parte del funcionamiento, como en ejemplos anteriores, es en la que se asigna la variable temporal STA2 a STA.

Finalmente, solo queda inicializar la variable STA para evitar que la máquina de estados funcione con estados desconocidos.

El código resultante es de tan sólo 46 líneas, entre las que figuran las llamadas a librerías, la definición de entidad, puertos, arquitectura, proceso, por lo tanto, este método de programar máquinas de estados es sumamente sencillo, y puede o no requerir de conocimientos de diseño lógico, pues la sentencia de control IF-THEN-ELSE simplifica mucho el trabajo, aumentando el número de líneas de código.

Cabe señalar que Xilinx ISE optimiza el código que escribimos, por lo que no es de vital importancia hacer códigos pequeños, ya que serán simplificados, el resultado de un IF-THEN-ELSE comparado con el de una ecuación booleana bien puede ser el mismo una vez que la herramienta de síntesis ha hecho su trabajo.

Un código pequeño no es por regla más fácil de traducir o interpretar para el usuario, puede simplificar el trabajo, pero será menester del programador decidir cuándo usar el asistente y cuándo escribir todo el código, o modificar el código entregado por el asistente.

El código de ésta práctica, aunque ha sido proporcionado en partes, es el siguiente:

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

entity maquina2 is port (
STA: inout std_logic_vector(3 downto 1);
CLK, EM, EM2, P1, P2, LS, LC : in  STD_LOGIC;
RDY, CHG, DEC, SRV, RST : out  STD_LOGIC);
end maquina2;
```

```
architecture Behavioral of maquina2 is
signal STA2: std_logic_vector(3 downto 1);
signal BAN: std_logic;
begin
process(CLK, EM, EM2, P1, P2, LS, LC, STA, STA2, BAN) begin

 if(BAN='0') then
      BAN<='1';
      case STA is
                when "000"=> STA2<=EM & "00";
                when "001"=> STA2<="011";
                when "010"=> STA2<="000";
                when "011"=> STA2<='1' & NOT P2 & '1';
                when "100"=> STA2<=EM2 & EM2 & '0';
                when "101"=> STA2<="01" & NOT LS;
                when "110"=> STA2<='0' & NOT P1 & NOT P1;
                when others=>STA2<='0' & LC & '1';
      end case;
 end if;
 RDY<= NOT(STA(3)) AND NOT(STA(2)) AND NOT(STA(1));
 SRV<= STA(3) AND NOT(STA(2)) AND STA(1);
 RST<= NOT(STA(3)) AND STA(2) AND NOT(STA(1));
 CHG<= STA(3) AND STA(2) AND STA(1);
 DEC<= NOT(STA(3)) AND NOT(STA(2)) AND STA(1);

 if(CLK='1' and CLK'event) then
      STA<=STA2;
      BAN<='0';
 end if;
      if(STA="UUU") then
 STA<="000";
 end if;
end process;
end Behavioral;
```

5.3 Práctica 9 – Herramientas avanzadas de programación y memorias RAM

Objetivo:

Conocer y aplicar técnicas y estructuras avanzadas de programación que permiten incrementar la versatilidad de los diseños en VHDL y aplicar estas técnicas y estructuras en la creación de una memoria RAM sencilla.

Antecedentes:

Las herramientas a usar en esta práctica son:

- Definición de constantes.
- Definición de tipos de datos personalizados.
- Utilización de arreglos.
- Funciones de conversión.

Estas cuatro herramientas serán utilizadas en un sólo diseño para dar forma a una memoria RAM.

En primer lugar, la **definición de constantes**, cuando definimos una constante, podemos por ejemplo, si estamos programando un contador, definir esa constante como un valor máximo, para que al llegar a ese valor, el contador se reinicie, la ventaja de esto facilita escribir el código en el que el mismo valor va a repetirse, incluso, al hacer una modificación sólo se hace en donde está definida la constante y esto afectará a todo el código que haga referencia a ella.

Si una serie de buses van a definirse de un mismo ancho, puede utilizarse esta constante para escribirla en todos los buses.

Para definir una constante, en la entidad del diseño, y antes de describir los puertos, escribimos:

```
GENERIC( Nombre: Tipo:= Valor);
```

Esto nos permitirá crear una constante que podrá ser referenciada como un valor del tipo seleccionado en cualquier parte del código siempre que se escriba su nombre.

Para **definir tipos de datos personalizados** podemos listar los datos o bien hacer un tipo de dato complejo, muchos programadores prefieren listar los estados de sus máquinas de estado en un tipo de dato personalizado y trabajar sobre él, por la

simplicidad que representa y porque no necesita trabajo lógico extra para describir el diseño. Un ejemplo de cómo listar una variable de tipo estado es:

```
type ESTADO is (E1, E2, E3, E4, E5, E6, E7);
```

El sintetizador de Xilinx ISE se encargará de asignar un código a cada estado al implementar el diseño. Sin embargo; si deseamos usar valores específicos para cada estado podemos también listarlos de la siguiente forma:

```
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of ESTADO: type is "001 010 011 100 101 110 111";
```

Una forma distinta de manejar los tipos definidos por el usuario es usar estructuras de datos, un tipo de dato compuesto que puede almacenar varios datos en sí, para hacer esto se crea el tipo de dato y se listan los datos que lo compondrán dentro de un *record*, la sintaxis es la siguiente:

```
type NOMBRE is
 record
   variable1: tipo;
   variable2: tipo;
end record;
```

Bajo esa estructura, para acceder a una variable, primero declaramos una nueva variable del tipo que hemos definido y después, para leer o escribir uno de sus elementos usamos el nombre de la variable, un punto y luego el nombre del elemento que se quiere accesar.

Ejemplo:

```
type switch is record
      estado: std_logic;
      id: integer;
end record;
```

También se pueden hacer combinaciones de declaraciones de variables:

```
type prendido is (encendido, apagado);
type switch is record
        estado: prendido;
        id: integer;
end record;
```

Dentro de lo que es la declaración de tipos de datos personalizados, se encuentra la **definición de arreglos**; así como el tratamiento de las estructuras de datos en VHDL es una analogía a las estructuras de datos y punteros de otros lenguajes de programación, el funcionamiento de los arreglos es también análogo, los arreglos en VHDL nos permiten definir una colección ordenada de datos del mismo tipo que pueden ser accedidas mediante un índice.

La manera en que se define un arreglo es la siguiente:

```
type nombre is array(rango) of tipo;
```

Donde rango es un intervalo definido de la misma forma en que se define un vector.

A continuación, un ejemplo de la manipulación de arreglos:

```
type arreglo is array(7 downto 0) of std_logic;
variable A: arreglo;
variable B: std_logic;
B:= A(3);
```

Es muy simple manejar arreglos en VHDL, una ventaja es que se pueden hacer arreglos de vectores, lo cual puede darle al lector una idea de cómo se manejará el mapa de memoria de esta práctica.

Otra herramienta más que se utilizará es una de las **funciones de conversión** contenidas en el paquete *ieee.numeric_std.all, to_integer()*, la cual convierte un vector en un valor entero, si queremos guardar un vector de datos en una variable entera; por ejemplo, tenemos que transformar ese valor para que el sintetizador de Xilinx ISE pueda asignarlo sin problemas y no ocurra un error.

to_integer(dato) siendo dato un vector, devuelve el equivalente entero del valor especificado, es importante tener en cuenta que to_integer() puede devolver valores negativos, por lo que se recomienda, a menos que no se requiera así, que al llamar a esta función se llame también a la función unsigned().

Con estas herramientas fácilmente se puede construir una memoria, usaremos un arreglo de vectores para crear el mapa de memoria dentro del CPLD, los arreglos se acceden por medio de índices enteros, por lo que el bus de direcciones será convertido a un entero al momento de que seleccione un índice en el mapa de memoria. El arreglo de vectores será una variable personalizada, la conversión se hará mediante la función de conversión, la herramienta faltante es la que nos permite crear constantes, ésta la

usaremos para definir el ancho de las palabras y para definir el tamaño de los buses, cabe mencionar que el ancho de la palabra por el tamaño del bus nos da la capacidad de memoria.

Desarrollo:

Con las herramientas vistas en los antecedentes de esta práctica, crearemos una memoria de 64KB, el ancho de la palabra será de 8 bits, por lo que harán falta 8 bits más para el bus de direcciones, de tal forma que $8^2*8^2 = 65536 = 64KB$.

La lógica del programa es sencilla, si la escritura está habilitada, la memoria escribirá lo que tenga como datos de entrada, de no ser así, no lo hará.

En la salida siempre estará el dato al que la dirección de lectura haga referencia. Todo lo demás es simplemente la definición de las señales que actuarán.

En primer lugar se usarán constantes para definir los buses, después los puertos, dentro de la arquitectura se define un tipo de dato usando vectores y arreglos para que actúe como la RAM, después en el proceso, la lógica del programa. Con esos simples elementos estará lista la memoria.

Programación en VHDL:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ram IS GENERIC(
      palabra    : integer := 8;
      busDatos   : integer := 8);
 PORT(
      clock: IN  std_logic;
      data  : IN  std_logic_vector(busDatos - 1 DOWNTO 0);
      direcciónEscritura:  IN    std_logic_vector(palabra  -  1
DOWNTO 0);
      direcciónLectura :  IN    std_logic_vector(palabra  -  1
DOWNTO 0);
      writeEnable     : IN  std_logic;
      q    : OUT std_logic_vector(busDatos - 1 DOWNTO 0));
END ram;
```

```
ARCHITECTURE memoria OF ram IS
 TYPE RAM IS ARRAY(0 TO 2 ** palabra - 1) OF
std_logic_vector(busDatos - 1 DOWNTO 0);
 SIGNAL bloqueRAM : RAM;
BEGIN
 PROCESS (clock)
 BEGIN
 IF (clock'event AND clock = '1') THEN
 IF (we = '1') THEN
 bloqueRAM(to_integer(unsigned(direcciónEscritura))) <= data;
 END IF;
 q <= write_address(to_integer(unsigned(direcciónLectura)));
 END IF;
 END PROCESS;
END memoria;
```

Simulación:

A continuación se muestra una simulación del funcionamiento de la memoria RAM que se acaba de programar.

Se accederán las direcciones de escritura de la 0 a la 6 (números enteros), de ellas se escribirá en todas excepto la 2 y la 6. Posteriormente, se leerá la memoria en las direcciones 0 a 7, como es de esperarse, la memoria arrojará la información introducida anteriormente.

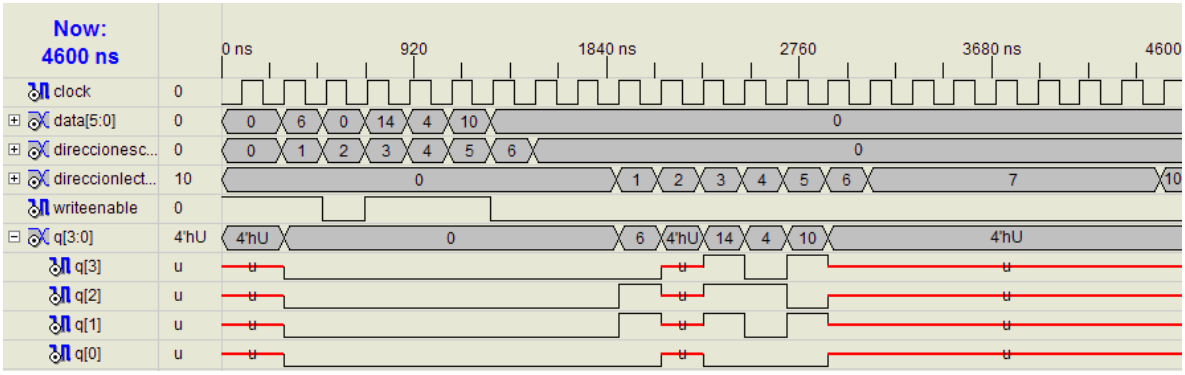


Fig. 5.16 Simulación de la RAM

Como puede verse en la imagen, para escribir se pone en alto la señal *writeEnable*, y al mismo tiempo se introduce la información, no es necesario poner nada al momento de acceder la dirección 2 puesto que no será escrita, pero es importante notar que hay un cero.

Como puede verse, en la dirección cero se introduce un cero, por eso a la salida primero se muestra un valor indeterminado y después un cero, porque antes no había nada allí, sino hasta el momento en que termina el ciclo de reloj en el que fue escrita esa dirección que se está leyendo constantemente, de allí que durante un largo periodo de tiempo solo exista cero en la salida, se cambia entonces la dirección de lectura, la dirección de escritura pasa a ser irrelevante mientras *writeEnable* esté deshabilitada.

Ahora observaremos lo que va a entregar la memoria a la salida, tenemos que mientras se lee la dirección cero existe cero en la salida, pero cuando se lee la dirección 1, a la salida hay un 6, ese fue introducido anteriormente, sin embargo, al leer la dirección 2 hay un valor indeterminado, esto es porque al acceder la dirección de escritura 2, el valor de *writeEnable* estaba en bajo. En las direcciones siguientes se introdujeron 14, 4 y 10 respectivamente, dichos valores son leídos a la salida al ser accedidas las direcciones correspondientes, finalmente, al leer la dirección 6 y siempre que se lee la dirección 7 no hay ningún dato, puesto que no se ingresó información allí.

Conclusiones

La tecnología de CPLDs está en crecimiento, ha logrado avanzar hacia los FPGA, este documento se basa en los CPLDs Xilinx porque son pioneros en el desarrollo de esta tecnología y porque son líderes mundiales en esta materia.

El trabajo cubre todos los aspectos de la programación de estos dispositivos, por lo que puede llevarse a cabo como un curso muy completo, no es un manual o una referencia, sino algo llevado por la teoría a la práctica, no hay literatura que ofrezca esto, mucho menos en español y mucho menos al alcance de estudiantes. Las prácticas son simples pero todas muy educativas, cada una cumple muchos aspectos orientándose a un solo objetivo. En un solo ejercicio se pone en práctica un recurso de VHDL y al mismo tiempo se tiene un conjunto de herramientas de carácter lógico que ayudarán al alumno a entender cómo ocuparlas en sus propios casos.

El desarrollo de este trabajo ofrece al lector una preparación integral y conjunta del software, el lenguaje y el hardware necesario para trabajar con CPLDs.

Para un estudio futuro queda el desarrollo de este tipo de programación en FPGAs y en FPGAs con DSP integrados, el campo de desarrollo de estas tecnologías y las propias tecnologías siguen creciendo y si no se comienza ahora, después será más difícil ponernos al corriente.

Bibliografía

Xilinx, Design Kit Programmable Logic Guide

MultiVideoDesigns, VHDL Training Manual

MultiVideoDesigns, CPLD Introduction Manual

Doulos.com, VHDL Designers Guide

http://www.doulos.com/knowhow/vhdl_designers_guide/

T.L. Floyd. *Fundamentos de Sistemas Digitales.* 7ª Edición. Ed. Pentice-Hall.