Sistemas Informáticos
Curso 2003-2004

Estudio de técnicas de reconfiguración parcial dinámica de FPGAs. Aplicación al procesamiento de imágenes digitales

David Corrales Pérez
César Granados García
Rafael Domínguez Abad

Dirigido por:
Prof. Hortensia Mecha López
Dpto. de Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de Madrid

ÍNDICE

1. AUTORIZACIÓN

Por la presente, los autores del presente trabajo, que abajo firman, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Madrid, 8 de Julio de 2004

David Corrales Pérez

César Granados García

Rafael Domínguez Abad

2. PALABRAS CLAVE

FPGA

Reconfiguración Parcial

Handel-C

Vhdl

RC1000

Virtex

3. RESUMEN DEL PROYECTO

El principal objetivo de nuestro proyecto era lograr la reconfiguración parcial dinámica sobre un FPGA Virtex 1000, es decir, poder reprogramar una determinada área de la FPGA para cambiar el comportamiento de la misma sin necesidad de resetear el dispositivo. Por tanto, se pretendía que la FPGA siguiera funcionando normalmente y con el comportamiento inicial hasta el momento en que el área en cuestión estuviera totalmente reprogramada, momento en el que la FPGA adoptaría el nuevo comportamiento derivado de la modificación de esa área.

Un ejemplo sencillo de esta reconfiguración parcial es programar la FPGA para que filtre las imágenes que el usuario le vaya indicando. Inicialmente se carga un primer filtro, pero posteriormente se decide que se desea utilizar un filtro distinto. Pues bien, con la reconfiguración parcial no haría falta resetear la FPGA y programarla entera de nuevo, ya que lo único que se desea es reprogramar la parte de la FPGA que "contiene" el filtro que se desea cambiar, sino que simplemente habría que cargar el nuevo filtro en la zona que ocupaba el anterior, dejando inalterada el resto de la FPGA y permitiéndola que siguiera funcionando mientras tanto.

Con este tipo de ejemplo es donde entra el otro objetivo de este proyecto: conseguir utilizar módulos diseñados en VHDL dentro de código Handel-C, lenguaje muy parecido a C pero con ciertas directivas para controlar la FPGA y comunicarse con ella.

4. PROJECT OUTLINE

The main goal of our project was to achieve dynamic partial reconfiguration on a FPGA Virtex 1000, it is said, to reprogramme an determinated area of the FPGA to change its behaviour without having to reset the device. Therefore, it was trying the FPGA continued working normaly and with the initial behavior until the mentioned area was reprogrammed completely, moment at which the FPGA would get the new behaviour derived from the modification of this area.

A simple example of this partial reconfiguration is to programme the FPGA to filter images the user was indicating. Initially it was load the first filter, but then it was decided it was wanted to use a different filter. With partial reconfiguration you don't need to reset FPGA and reprogramme it full again, since the only thing is wanted is to reprogramme the area of the FPGA that contains the filter that is wanted to change, but simply it would have to load the new filter on the area that the other filter was ocupping, letting untouched the rest of the FPGA and letting it to work meanwhile.

With this kind of example the other goal of this project enters: to achieve to use VHDL modules in Handel-C code, a lenguage similar to C but with directives to control the FPGA and to communicate with it.

5. OBJETIVO DEL PROYECTO

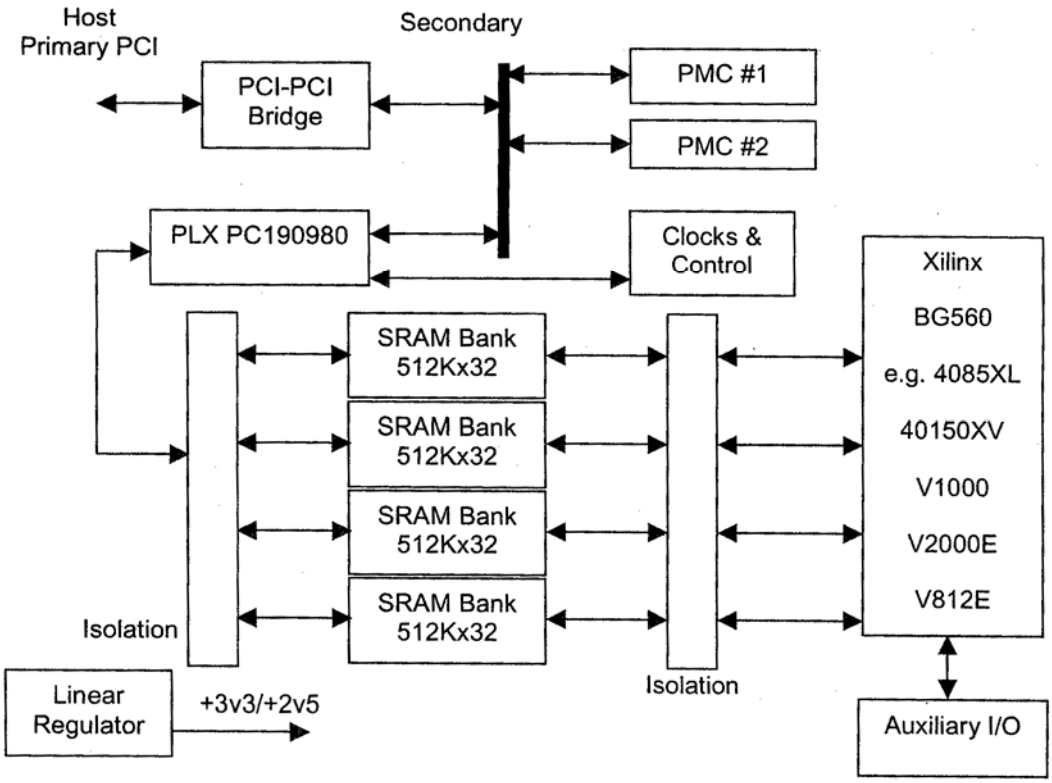
El objetivo del proyecto estaba dividido en tres partes bien diferenciadas:

- Unión de hardware programado en VHDL con programación en HandelC.
- Implementación de algunos de los filtros del proyecto de sistemas informáticos del año 2002-2003 con la unión de VHDL y HandelC.
- Reconfiguración parcial de la FPGA, implementación de un circuito simple en la FPGA con varias partes fijas y una parte sustituible por otra, la sustitución se debe hacer sin modificar las partes fijas.

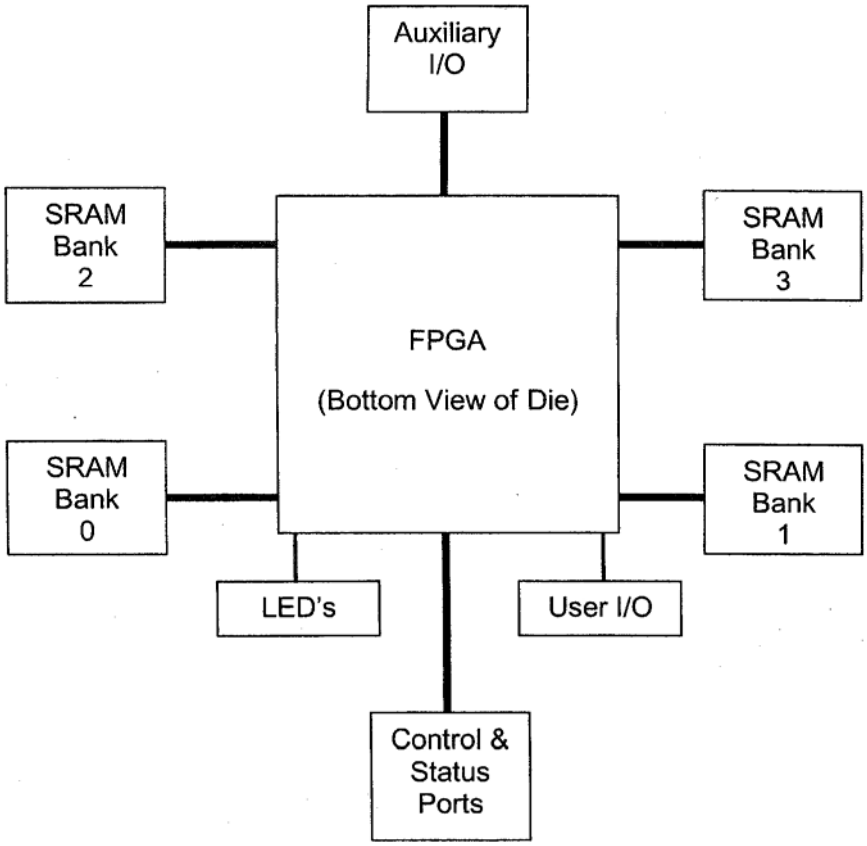
6. HARDWARE. TARJETA RC1000

La RC1000 es una tarjeta que se conecta al bus PCI del pc. En ella está la FPGA de Xilinx con cuatro bancos de memoria para operaciones de procesamiento de datos, y dos PMC Sites para I/O con el exterior.

Diagrama de bloques de la tarjeta RC1000



Grupo de pines funcionales



6.1. LA FPGA VIRTEX 1000

Las familias de FPGAs Virtex ofrecen un alto rendimiento y una alta capacidad de soluciones lógicas programables. Gracias a los avances en eficiencia del silicio se han optimizado y se ha creado una nueva arquitectura de 5 capas de metal de 0.22μm CMOS. Estos avances hacen que la reprogramación de las Virtex sea más potente y flexible. Existen 9 clases de Virtex como se muestra en la siguiente tabla, la que nosotros utilizamos es la XCV1000.

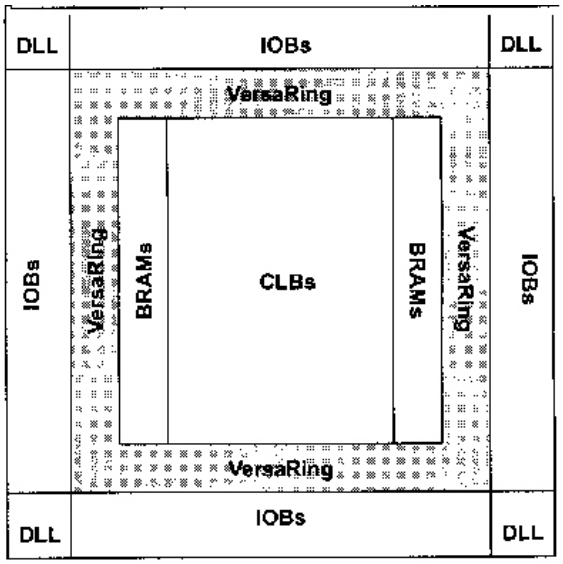
Familias Virtex

Device	System Gates	CLB Array	Logic Cells	Maximum Available I/O	Block RAM Bits	Maximum SelectRAM+™ Bits
XCV50	57,906	16x24	1,728	180	32,768	24,576
XCV100	108,904	20x30	2,700	180	40,960	38,400
XCV150	164,674	24x36	3,888	260	49,152	55,296
XCV200	236,666	28x42	5,292	284	57,344	75,264
XCV300	322,970	32x48	6,912	316	65,536	98,304
XCV400	468,252	40x60	10,800	404	81,920	153,600
XCV600	661,111	48x72	15,552	512	98,304	221,184
XCV800	888,439	56x84	21,168	512	114,688	301,056
XCV1000	1,124,022	64x96	27,648	512	131,072	393,216

Comparando con otras generaciones anteriores, la familia de las Virtex representa un revolucionario paso en diseño lógico programable.

Los dispositivos Virtex constan de una flexible y regular arquitectura que comprende un array de bloques lógicos configurables (CLBs) rodeados por bloques de entradas/salidas programables (IOBs). La abundancia de posibles rutados permite a las familias Virtex albergar diseños muy complejos.

Arquitectura Virtex



Las Virtex están basadas en SRAM y se personalizan leyendo los datos de configuración de unas celdas de memoria internas. En algunos modos la FPGA lee su propia configuración de datos desde una PROM externa (master serial mode). En otros casos, la configuración de los datos es escrita dentro de la FPGA (modos Select Map, slave serial y JTAG).

Los diseños de las Virtex pueden llegar hasta los 200 MHz de frecuencia de reloj incluyendo las I/O. Las entradas y salidas de la Virtex cumplen perfectamente con las especificaciones PCI, y las interfaces pueden ser implementadas para operar a 33MHz o 66 MHz.

Como ya hemos mencionado anteriormente las Virtex tienen dos principales elementos de configuración: bloques lógicos configurables (CLBs) y bloques de entrada/salida (IOBs).

- CLBs proporcionan los elementos funcionales para la construcción lógica.
- IOBs proporcionan la interface entre el paquete de pines y los CLBs.

Los CLBs se interconectan a través de una matriz de rutado general (GRM). La GRM contiene un array de conmutadores de rutado en las intersecciones de los canales horizontales y verticales.

La arquitectura de las Virtex también incluyen los siguientes circuitos para conectar con la GRM:

- Bloques de memorias dedicados de 4096 bits cada uno.
- Relojes DLLs para distribución de la compensación del retardo del reloj y un control del dominio del reloj.
- 3 “State buffers” (BUFTs) asociados con cada CLB que conducen el rutado horizontal segmentable dedicado.

6.2. MEMORIA SRAM

La memoria SRAM está compuesta de cuatro bancos de 2Mbytes cada uno. Todos los bancos son accesibles desde la FPGA y desde fuera de la tarjeta mediante el PCI.

Las cuatro interfaces SRAM son iguales. Cada una tiene cuatro pines de selección, dos de ellos se usan para leer y escribir, y los otros dos son el *"output enable"* y el "*write enable".* El interfaz consta también de 21 pines de direcciones y 32 pines de datos. Todas las señales de control son activas a baja.

La FPGA puede acceder a los cuatro bancos simultáneamente e independientemente.

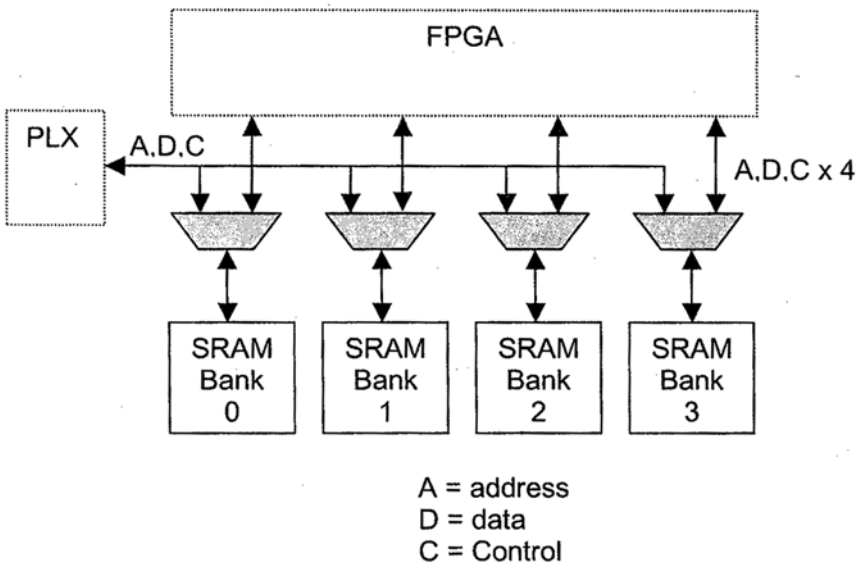
Todos los bancos están puestos en el espacio de direcciones PCI, con lo cual pueden ser accedidos por el Host y por cualquier otro mecanismo PCI. El software del Host permite el mapeo dentro del espacio de direcciones virtual de una aplicación Host. Solo un banco puede ser accedido por el Host y por cualquier otro mecanismo PCI.

6.2.1. ARQUITECTURA

El interfaz de cada banco de memoria SRAM entre la FPGA y el PCI está controlado por *"QuickSwitches"* que actúan como cuatro multiplexores independientes, uno por banco, para permitir que cada banco sea accedido por la FPGA y por el PCI, pero nunca ambos al mismo tiempo.

El arbitraje está implementado en un CPLD que conecta al bus local PCI9080 y a la FPGA. Cualquier mecanismo PCI puede solicitar los bancos de memoria vía el registro *H_ARB*, y el estado de cualquier arbitración por la FPGA se muestra en el registro *F_ARB*.

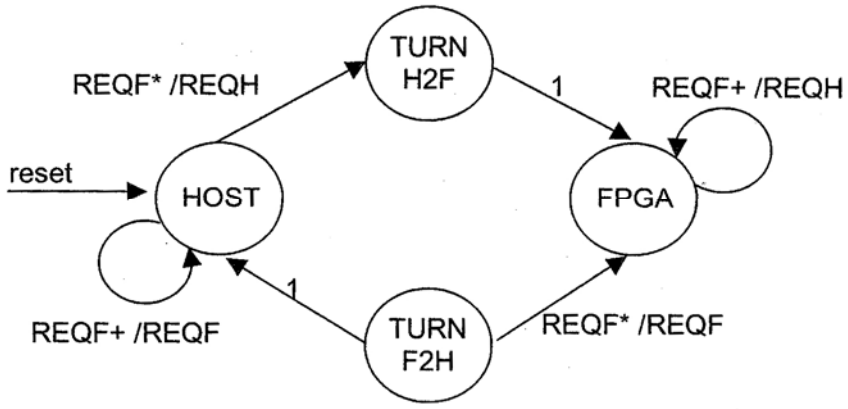
Arquitectura de los bancos de la SRAM



6.2.2. MÁQUINA DE ESTADOS

El protocolo de arbitraje es simple y se muestra con una simple máquina de 4 estados que es la siguiente:

Máquina de estados de arbitraje en el CPLD



6.2.3. ARBITRAJE

Para que la FPGA consiga el control de un banco de memoria ésta tiene que activar la señal *REQn* y esperar hasta que se le devuelva la señal *GNTn* que significará que ya puede acceder al banco de memoria. Al mismo tiempo, la FPGA tiene que conducir las señales de direcciones y de control a cada banco SRAM. Cuando se devuelve la señal *GNTn*, la FPGA tiene el control del banco y puede acceder a él el tiempo que sea necesario. Cuando la FPGA termina, la señal *REQn* puede ser desactivada para así permitir al Host vía PCI9080 que tome el control, pero esto sólo ocurrirá si el Host ha activado la señal de *REQn*.

Lo mismo ocurre cuando el Host quiere tomar el control de un banco de memoria usando el registro *H_ARB*, debe esperar hasta que le devuelvan la señal *GNTn* para acceder al banco de memoria.

6.3. TRANSFERENCIA DE DATOS Y COMUNICACIÓN CON LA FPGA

Hay tres métodos de transferencia de datos o comunicación entre la FPGA y cualquier mecanismo PCI las cuales la explicamos en los siguientes apartados.

6.3.1. TRANSFERENCIAS POR DMA

Estas transferencias se realizan entre la FPGA y cualquier otro mecanismo PCI a través de los bancos de memoria. La sincronización entre la FPGA y los otros mecanismos PCI se realiza mediante los métodos de comunicación “Control y Estado” y “Pines de Usuario I/O” los cuales los explicamos en los dos sigientes apartados. La FPGA no puede iniciar una transferencia por DMA, sólamente los mecanismos PCI pueden hacerlo.

6.3.2. PUERTOS CONTROL Y ESTADO

También denominada como comunicación *Single Byte*. Hay dos puertos unidireccionales de 8 bits, llamados *Control y Estado*. Estos puertos se utilizan para sincronizar las comunicaciones directas entre la FPGA y los mecanismos PCI. Indica cuando los datos se deben leer o escribir.

El registro *Control* se usa para que el Host envíe datos a la FPGA, y el registro *Estado* se usa para que la FPGA envíe datos al Host.

6.3.3. PINES DE USUARIO I/O

Hay otro método de comunicación que son los pines de usuario I/O del PLX o también llamados *USERI* y *USERO* que están ambos conectados a la FPGA para ofrecer un bit de comunicación. Este modo de comunicación es también denominada como comunicación *Single Bit.*

6.4. RELOJES

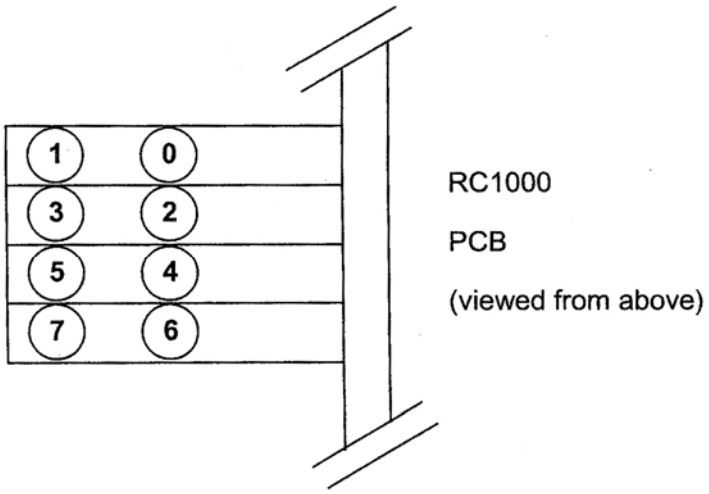
La FPGA tiene dos pines conectados a relojes. Uno de los pines está conectado a un reloj programable o a un reloj externo seleccionable mediante un jumper. El otro pin está conectado a otro reloj programable o al reloj del bus local PCI9080, también seleccionable por un jumper.

Los relojes programables se programan por desde el Host, y tienen una frecuencia que oscila entre 400kHz y 100MHz.

6.5. LED'S

El RC1000 tiene 8 LED los cuales los hemos usado para hacer pruebas de comunicación y depuración de la FPGA.

Visión desde arriba de los led's del RC1000



7. SOFTWARE Y SU CONFIGURACIÓN

7.1. INTRODUCCIÓN

Las principales herramienas software que hemos utilizado son:

- Visual C++: Esta herramienta la hemos utilizado para generar el programa Host el cual se encarga de inicializar la FPGA, cargar la imagen *".bit"* en la FPGA y comunicarse con la SRAM y la FPGA.

- DK2: Esta herramienta se utiliza para comunicar la FPGA con la SRAM y el Host.

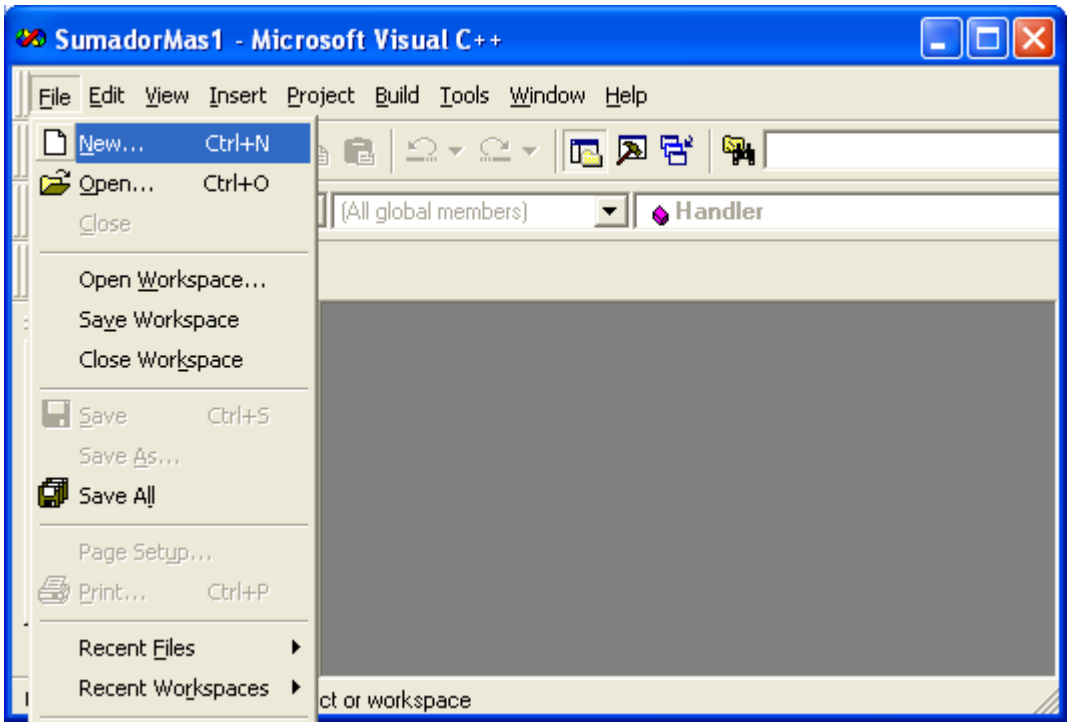
- Project Navigator: Con esta herramienta es con la que se crean los archivos imagen *".bit"* que posteriormente son cargados en la FPGA mediante el programa Host.

7.2. USO Y CONFIGURACIÓN DEL SOPORTE SOFTWARE PARA GENERAR EL PROGRAMA DEL HOST

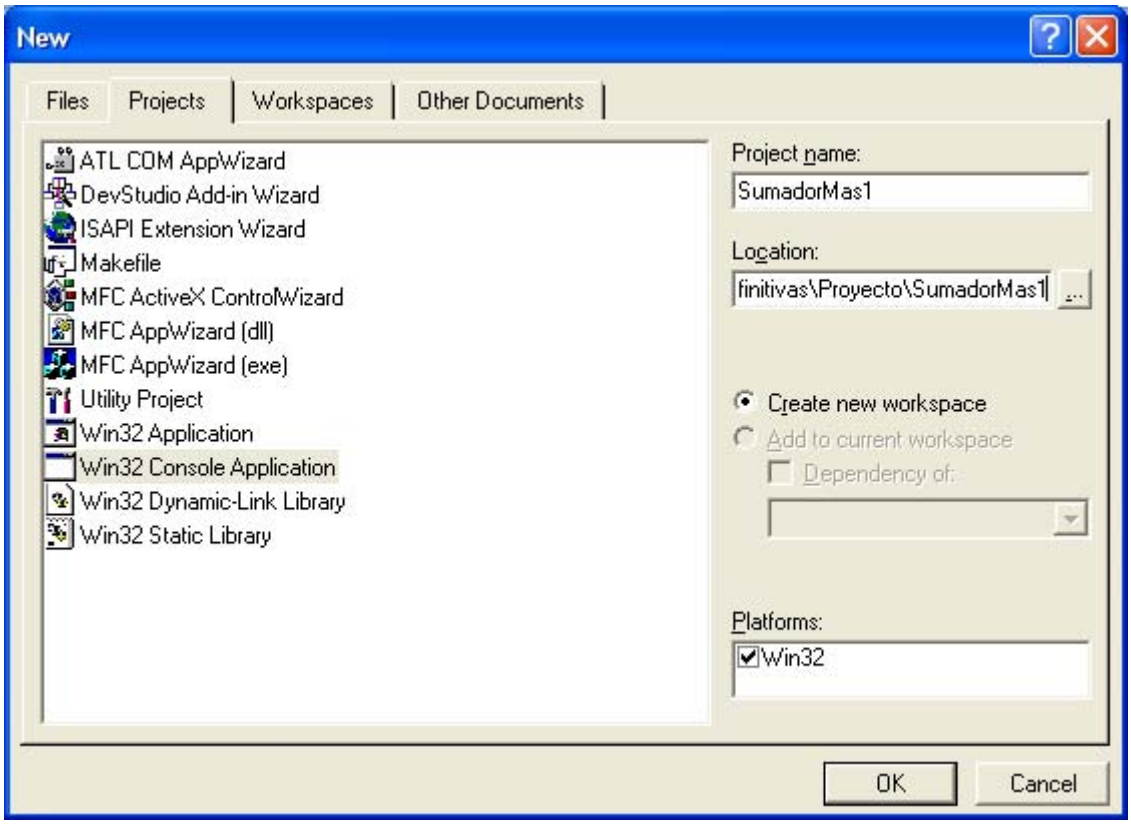
7.2.1. CONFIGURACIÓN DE VISUAL C++

Con el fin de que la configuración de la herramienta quede lo más clara posible a continuación mostramos los pasos a seguir con capturas de pantalla detalladas.

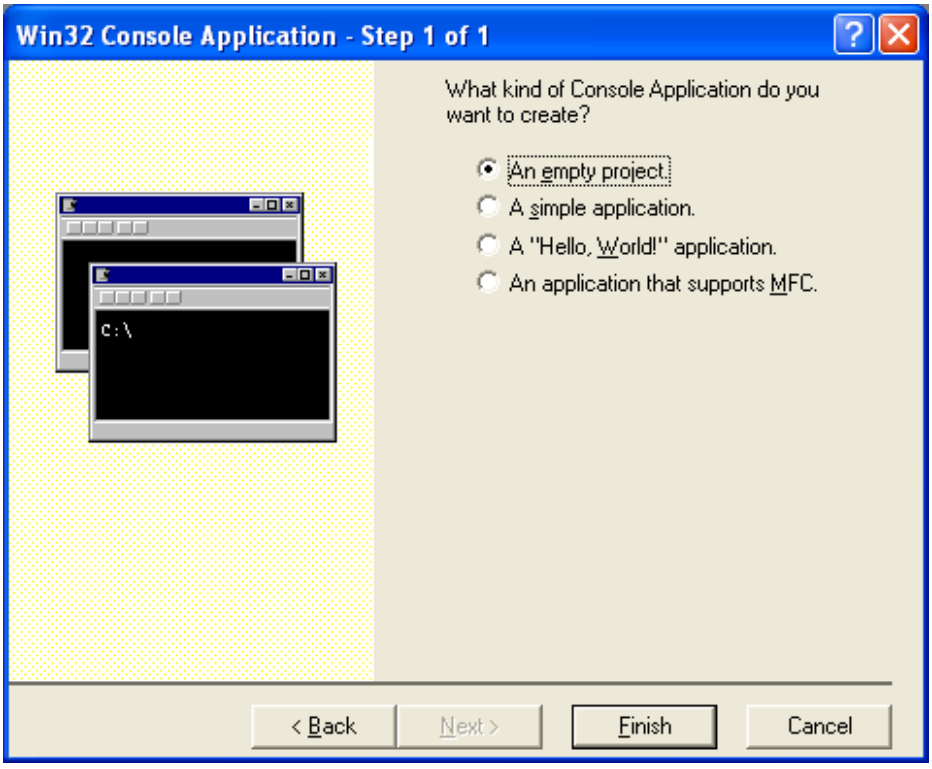
Lo primero que hay que hacer es abrir el Visual C++ y pinchar en el menú *"File/New"* como se muestra a continuación:



En la pantalla "*New*" hay que seleccionar la pestaña de "*Projects*" y la tarjeta "*Win32 Console Application*", se pone el nombre del proyecto en "*Project Name*" *y en "Location"* se pone la ruta donde se quiere poner el proyecto como se muestra en la siguiente imagen:



Se selecciona "*Empty proyect"* en la siguiente pantalla



Ahora es el momento de copiar el archivo ".c" *(Código fuente en lenguaje C)* y el archivo "*pp1000.h"* al directorio raíz del proyecto

ya que posteriormente van a ser utilizados y los vamos a necesitar en esta ubicación.

El archivo "*pp1000.h"* se tiene que incluir en el código fuente como:

#include "pp1000.h"

Nótese que *"pp1000.h"* está escrito entre comillas para indicar que el archivo al que hace referencia tiene que estar en el directorio raíz del proyecto.

El archivo *"pp1000.h"* que tenemos que copiar en la raíz del proyecto se encuentra en la carpeta "*include"* dentro de la carpeta de instalación de Celoxica, en nuestro caso está en:

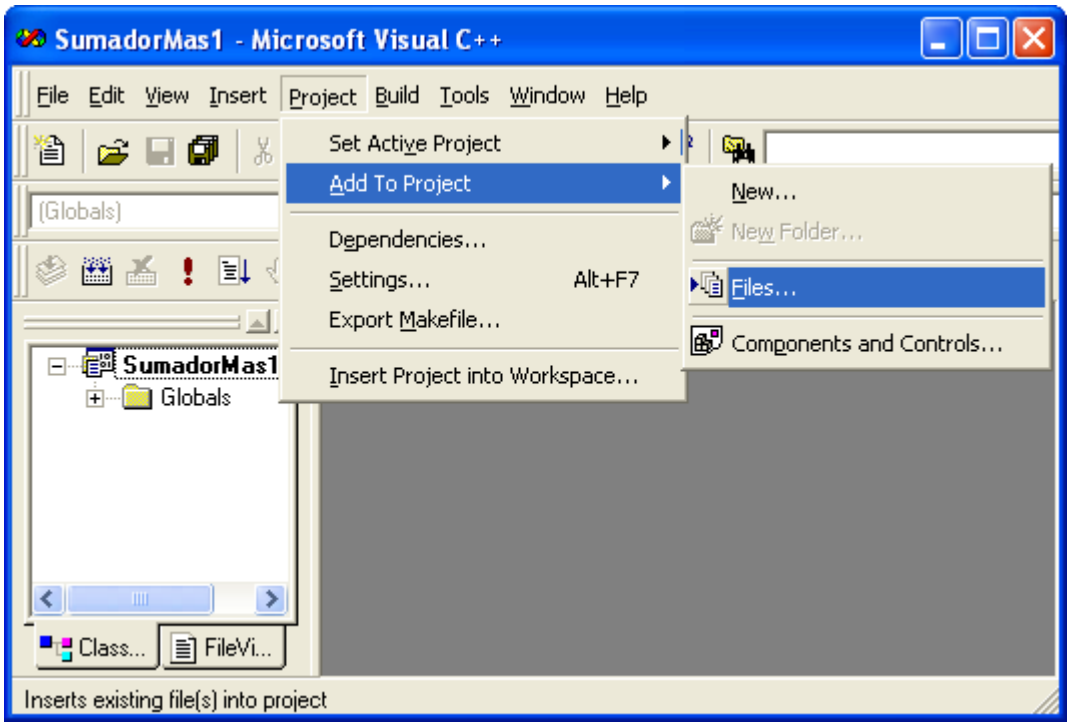
C :\Archivos de programa\Celoxica\RC1000\include

También se puede encontrar en nuestro proyecto en la carpeta:

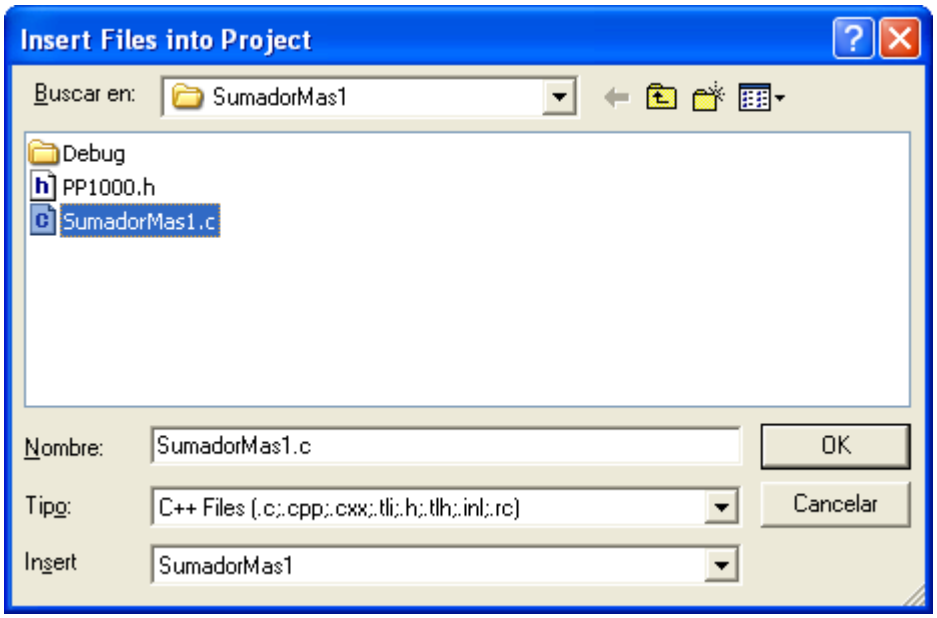
Librerias/HOST

A continuación hay que añadir al proyecto el archivo ".c", es muy importante que la extensión sea ".c" para que en el proceso de compilación no se generen errores. Los archivos con la extensión ".c" no se pueden crear desde el propio Visual C++ sino que hay que añadirlo desde fuera ya que al crear un archivo desde el Visual C++ lo crea con la extensión ".cpp".

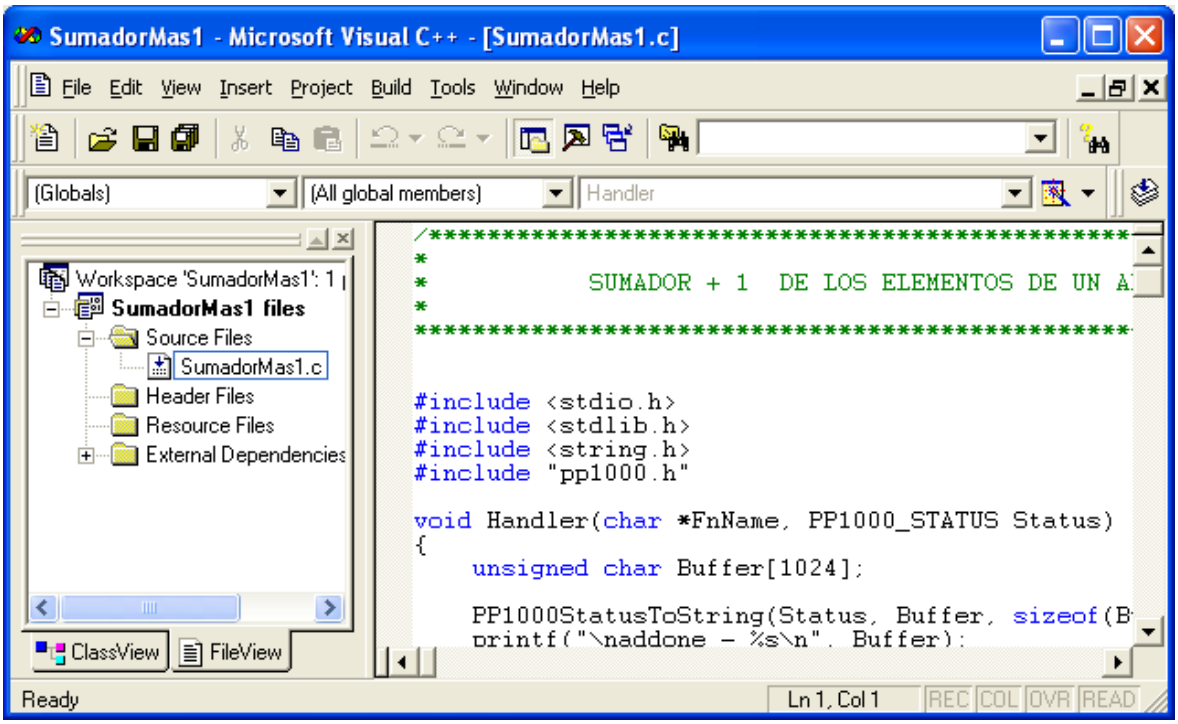
Para añadir el archivo ".c" al proyecto lo hacemos pinchando en la barra de herramientas en "*Proyect/Add To Proyect/Files..."* como se muestra a continuación:



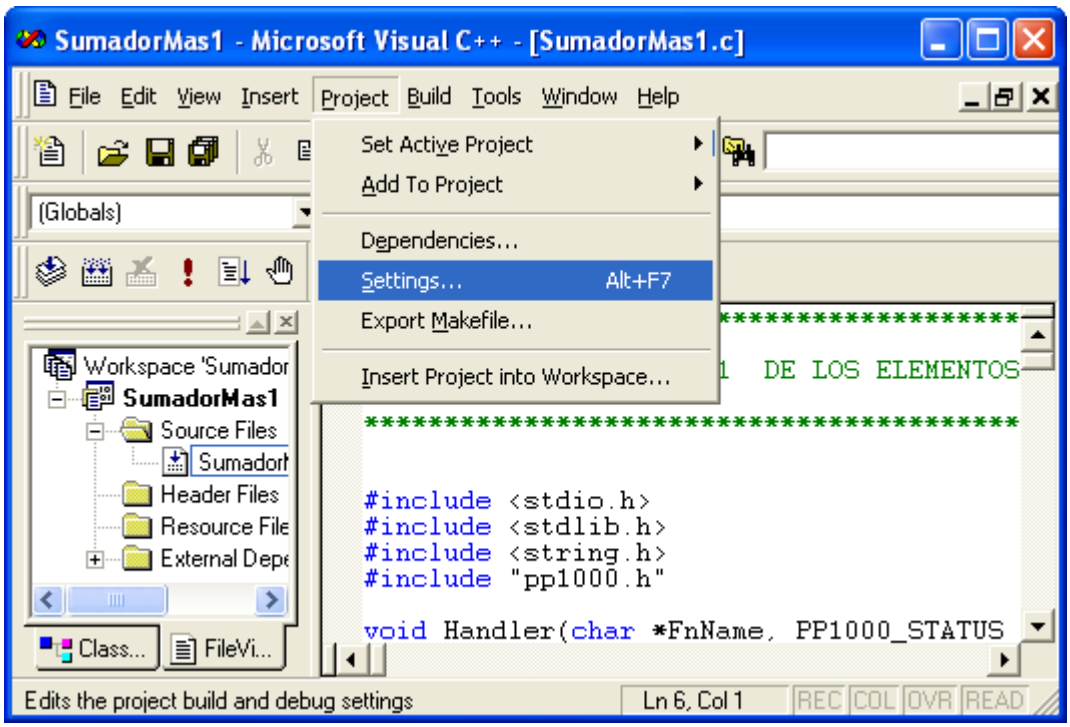
y ahora se añade el archivo



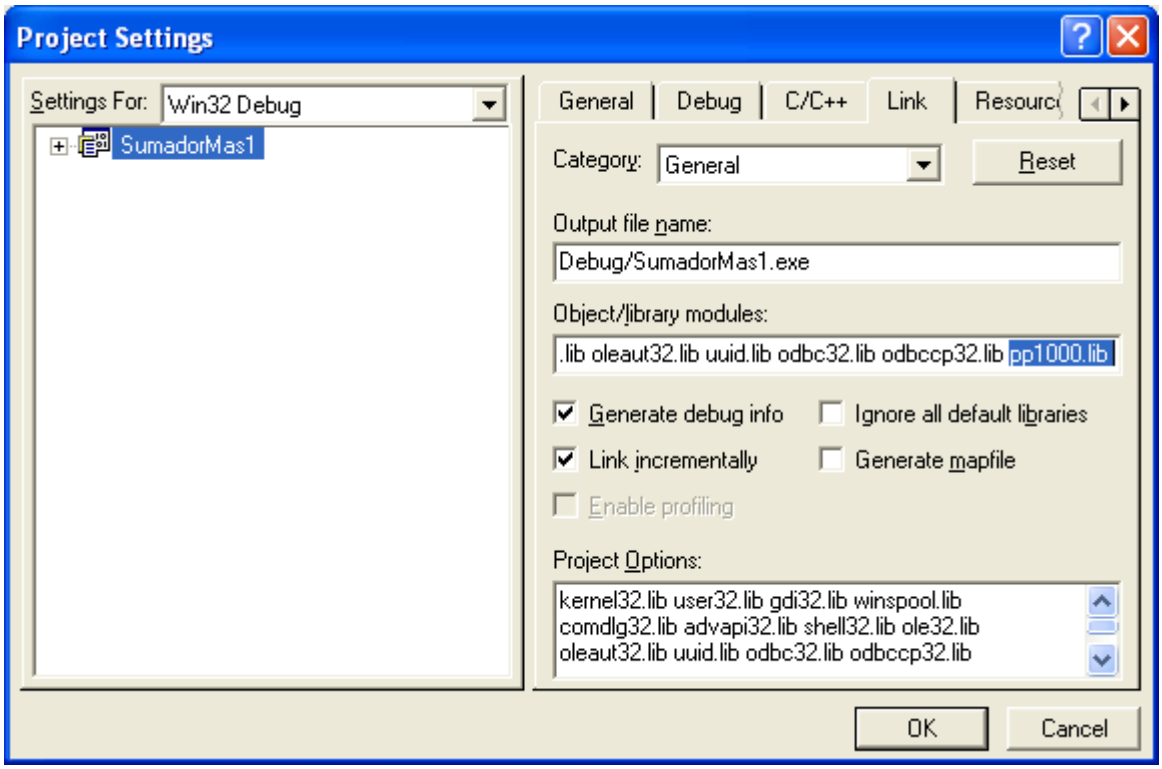
Por defecto está activada la pestaña "ClassView" en el visualizador de archivos que es el cuadro que está a la izquierda de la pantalla, pues hay que cambiarla a "*FileWiew*" para poder ver y abrir el archivo ".c" que acabamos de añadir al proyecto. Para abrirlo hacemos doble click en el nombre del archivo ".c" dentro de la carpeta "*Source Files*" como se muestra en la siguiente imagen:



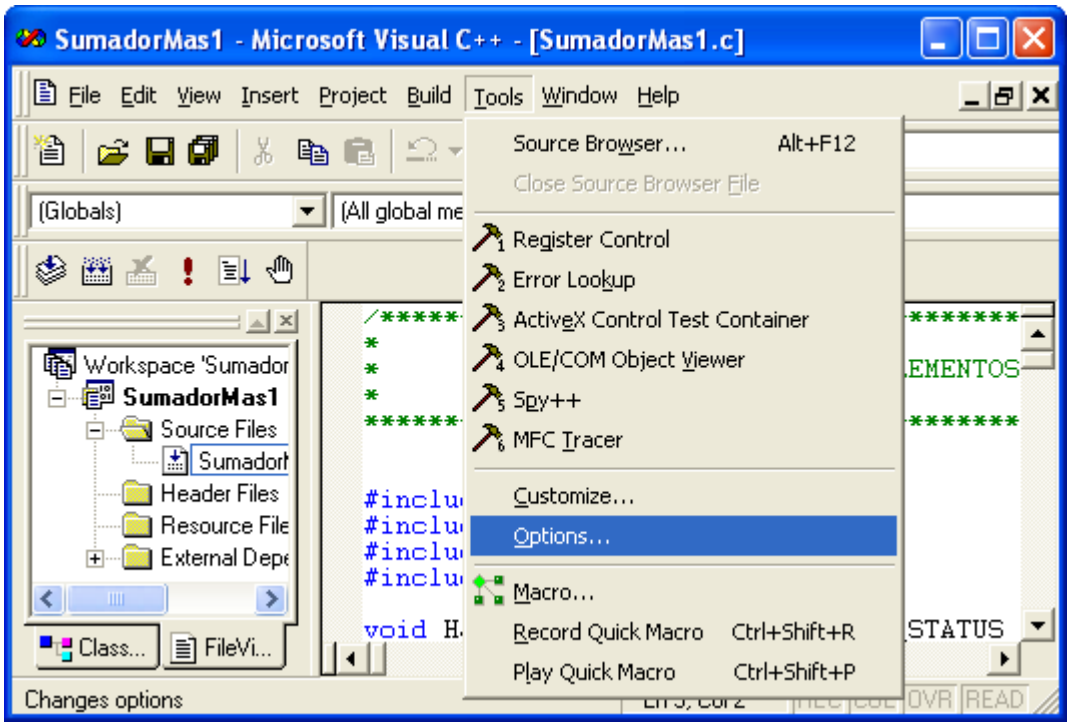
A continuación hay que configurar la herramienta para que funcione correctamente con la librería "*pp1000.lib*", para ello lo primero que hay que hacer es pinchar en "*Proyect/Settings...*" como en la siguiente imagen:



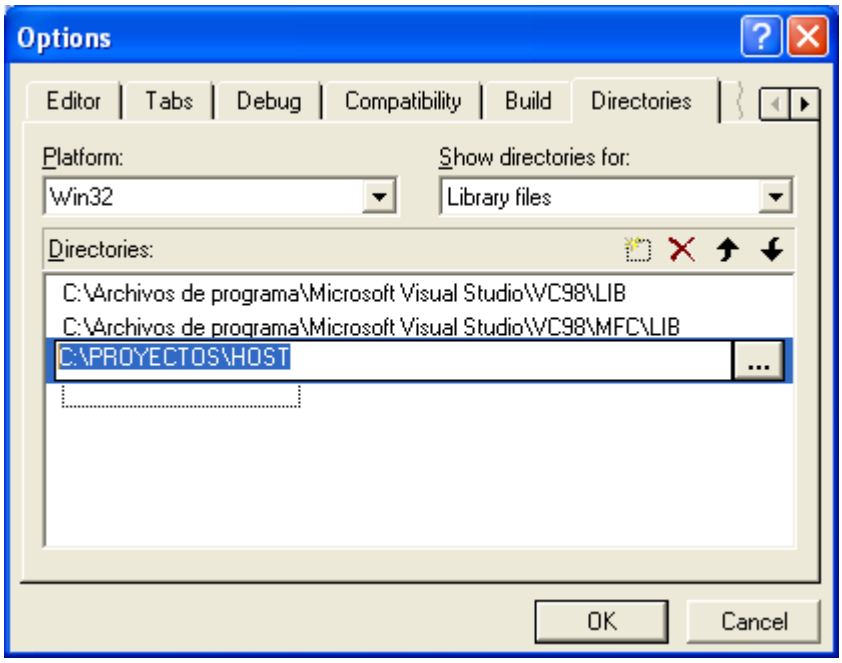
A continuación se muestra la siguiente imagen en la que hay que pinchar en "*Link*" e insertar en "*Objet/library modules*" la librería "*pp1000.lib*" como se muestra en la siguiente imagen:



Ahora hay que pinchar en "*Tool/Options...*"



En la siguiente pantalla hay que seleccionar la pestaña de "*Directories"*, en "*Show directories for*:" hay que seleccionar "*Library files"* y finalmente en la pantalla inferior se le añade la ruta donde se encuentra la librería "*pp1000.lib".*



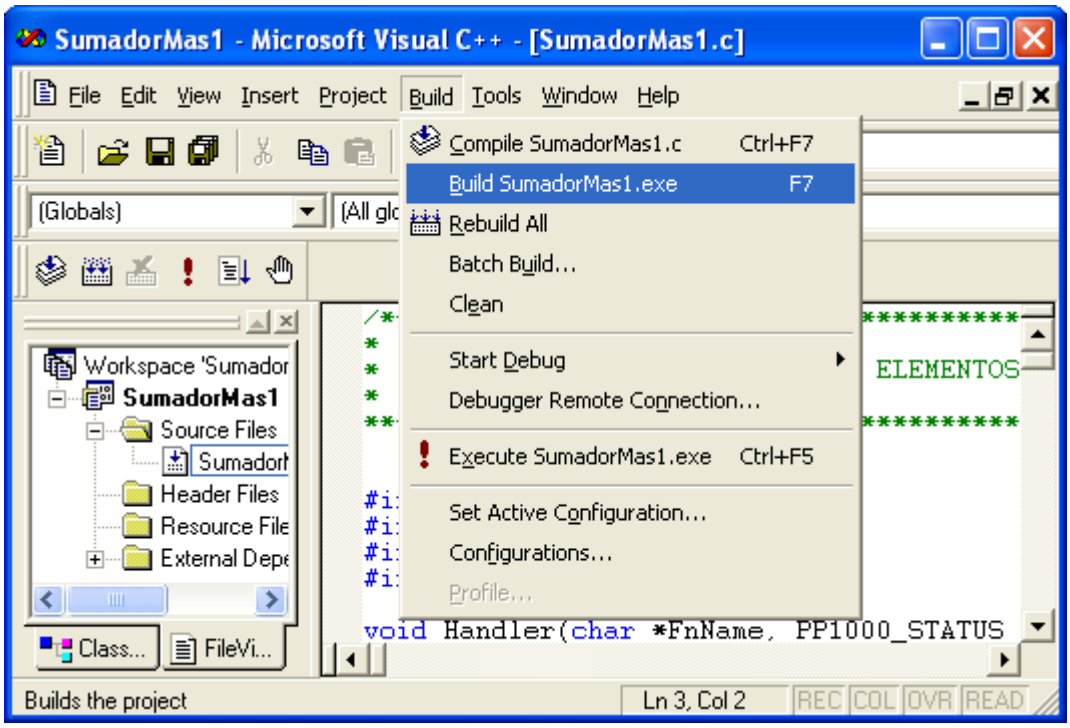
La librería "*pp1000.lib"* se encuentra en la carpeta "*lib"* dentro de la carpeta de instalación de Celoxica, en nuestro caso está en:

C :\Archivos de programa\Celoxica\RC1000\lib

También se puede encontrar en nuestro proyecto en la carpeta:

Librerias/HOST

Para compilar y ver errores después de haber desarrollado el código fuente hay que pinchar en la barra de herramientas en *"Built/Built (nombre_del_proyecto)" como se muestra a continuación:*



Una vez que ya hemos compilado sin errores se crea un archivo ejecutable con el nombre del proyecto en este caso *"SumadorMas1.exe"* en la carpeta *"debug"* del proyecto.

7.2.2. INICIALIZACIÓN DEL HARDWARE Y EL SOFTWARE

El primer paso para cualquier programa es inicializar el soporte software y hardware de la tarjeta RC1000. Generalmente esto se hace llamando a la función *PP1000OpenCard()* especificando la ID de la tarjeta. Esta función devuelve un *"handel"* el cual debe ser usado para identificar la tarjeta RC1000 en futuras llamadas al soporte software.

Como nosotros sólo tenemos una tarjeta por máquina la manera de inicializar el soporte software y hardware de la tarjeta RC1000 es más fácil ya que existe una función que abre cualquier tarjeta RC1000 que haya en la máquina al azar, y como nosotros sólo tenemos una, siempre nos abrirá la que tenemos.

La función que nosotros utilizamos es *PP1000OpenFirsTCard(),* con lo cual no nos tenemos que preocupar por la ID que tenga asignada la tarjeta.

7.2.2.1. CONFIGURACIÓN DE LA FRECUENCIA DEL RELOJ

El soporte software de la tarjeta RC1000 tiene una función para programar la frecuencia del reloj de la FPGA. La frecuencia del reloj tiene que ser configurada antes de configurar la FPGA para asegurar el buen funcionamiento del diseño.

Para configurar la frecuencia del reloj se usa la función *PP1000SetClockRAte()* especificando el *"handel"* que devuelve la función *PP1000OpenFirstCard()* y la frecuencia del reloj en Hercios.

Por ejemplo para poner la frecuencia de reloj a 10MHz se podría usar el siguiente código.

```
Status=PP1000SetClockRate(Handle, PP1000_MCLK; 10e6);
if(Status!=PP1000_SUCCESS)
{
    ReportError(Status);
}
```

Para más información acerca de las funciones utilizadas consultar el RC1000 Functional Reference Manual.

7.2.2.2. CONFIGURACIÓN DE LA FPGA

Una vez que la tarjeta ha sido inicializada y el reloj configurado, el siguiente paso es configurar los archivos imágenes de la FPGA es decir, los archivos *".bit"*. Hay tres formas de cargar las imágenes de la FPGA.

7.2.2.2.1. DIRECTA DESDE UN ARCHIVO

Configurándolo directamente desde un archivo usando la función *PP1000ConfigureFromFile().*

Esta opción requiere la lectura del archivo imagen cada vez que la FPGA es configurada, con lo cual esto puede suponer una pérdida de tiempo si frecuentemente se está reconfigurando la FPGA.

Por ejemplo, para configurar la FPGA directamente desde una archivo llamado prueba.bit se podría usar el siguiente código:

```
Status=PP1000ConfigureFromFile(Handle, "prueba.bit");
If(Status!=PP1000_SUCCESS)
{
    ReportError(Status);
}
```

7.2.2.2.2. CON CARGA EN LA MEMORIA

Cargando en la memoria el archivo imagen de la FPGA con la función *PP1000LoadFile()* dejándolo preparado para la configuración más tarde con la función *PP1000ConfigureFPGA().*

Con esta opción salvamos el problema de la opción anterior ya que cada vez que se reconfigura la FPGA no hay que

leer la imagen del archivo sino de la memoria, de esta manera se disminuye el tiempo de carga sustancialmente.

Por ejemplo, para cargar en la memoria un archivo llamado *"prueba.bit"* y configurarlo más tarde se podría usar el siguiente código:

```
PP1000_IMAGE Image;
Status=PP1000LoadImage("prueba.bit", &Image);
if(Status!=PP1000_SUCCESS)
{
    ReportError(Status);
}
…… //resto del código
Status=PP1000ConfigureFPGA(Handle, Image);
if(Status!=PP1000_SUCCESS)
{
    ReportError(Status);
}
```

7.2.2.2.3. MEDIANTE UN ARCHIVO *".H"* INCLUIDO EN EL EJECUTABLE

Incluyendo un archivo ".h" en el código del Host. Esto se puede hacer mediante la función del soporte software *PP1000RegisterImage()* para obtener un *"handel"* y usarlo más tarde con la función *PP1000ConfigureFPGA().*

Antes de usar las funciones anteriores en el código hay que hacer un procedimiento previo con el archivo imagen .bit. Hay que convertir el archivo .bit en un archivo ".h" para que lo pueda leer el Host. Esto se hace mediante la utilidad *"gencfg"* del

soporte software. Para dejar más claro el procedimiento a continuación mostramos un pequeño ejemplo de todo el procedimiento.

Supongamos que tenemos un archivo imagen *"archivoImagen.bit"* y queremos generar un archvio *"archivoSalida.h",* para esto habría que ejecutar la siguiente línea:

gencfg archivoImagen.bit archivoSalida.h

En el código fuente del Host es necesario incluir el archivo ".h" de la siguiente forma:

#include "archivoSalida.h"

Al estar incluido el archivo entre comillas, éste tiene que estar situado en la carpeta raíz del proyecto. Para más detalles de la utilidad *"gencfg"* se puede consultar el apartado 4.3.4.3.

Para registrar el archivo imagen *"archivoSalida.h"* y configurarlo más tarde se podría usar el siguiente código:

```
#include "archivoSalida.h"

PP1000_IMAGE Imagen;

Status=PP1000RegisterImage(configBuffer, configLength,
&Imagen);

if(Status!=PP1000_SUCCESS)

{

    ReportError(Status);

}

……..//resto del código

Status=PP1000ConfigureFPGA(Handle, Imagen);

if(Status!=PP1000_SUCCESS)
```

{

ReportError(Status);

}

Se ha utilizado la función *PP1000RegisterImage(configBuffer,configLength,&Imagen)* en la cual se usan los parámetros *"configBuffer"* y *"configLength",* estos parámetros de entrada se encuentran declarados en el archivo *"archivoSalida.h"* generado automáticamente con la utilidad *"gencfg"* del soporte software.

Esta opción permite una más rápida configuración sin la necesidad de múltiples archivos de imágenes .bit. También tiene la ventaja de que el mismo ejecutable contiene el programa Host y la información de la configuración de la FPGA, con lo que será mucho más cómodo el traslado de la aplicación ya que sólo habría que mover el ejecutable sin todos los archivos imagen .bit.

Para más información acerca de las funciones utilizadas consultar el RC1000 Functional Reference Manual.

7.2.3. COMUNICACIÓN DEL HOST CON LA FPGA.

Una vez que se ha configurado la FPGA, hay tres posibilidades de comunicación con ella, las cuales las explicamos en los tres siguientes apartados.

7.2.3.1. COMUNICACIÓN SINGLE BIT

En el soporte software existen dos funciones de control para los *"single pins".* La función *PP1000SetGPO()* que modifica el estado de uno de los pines y la función *PP1000ReadGPI()* que lee el estado del otro pin. Para que estos pines puedan servir de control

es necesario que haya un entendimiento entre el Host y la FPGA, por eso el Host modifica el pin *GPO* que es el que sólo se podrá leer desde la FPGA y lee el pin *GPI* que es el que modifica la FPGA.

Por ejemplo para poner el estado del pin *GPO* a 1 se podría utilizar el siguiente código:

```
Status=PP1000SetGPO(Handle, 1);
If(Status!=PP1000_SUCCESS)
{
    ReportError(Status);
}
```

Por ejemplo para leer el estado del pin *GPI* se podría utilizar el siguiente código:

```
Status=PP1000ReadGPI(Handle, &Value);
If(Status!=PP1000_SUCCESS)
{
    ReportError(Status);
}
```

Consulta el apartado 4.3.2.1 para ver los detalles de cómo se modifica y se lee el estado de estos pines desde la FPGA.

Para más información acerca de las funciones utilizadas consultar el RC1000 Functional Reference Manual.

- **El pin *GOP* es equivalente al pin *USERO* y el pin *GPI* es equivalente al pin *USERI* de los que hemos hablado en el apartado anterior 3.3.3.**

7.2.3.2. COMUNICACIÓN SINGLE BYTE

La tarjeta RC1000 tiene un puerto de un byte de ancho entre el Host y la FPGA. Este puerto puede ser usado para enviar pequeños mensajes de control o de estado entre las dos partes. La función *PP1000WrteControl()* envía un byte del Host a la FPGA y la función *PP1000ReadStatus()* espera a que la FPGA escriba en el puerto capturando el dato que esta envía.

Por ejemplo para enviar un 1 por el puerto a la FPGA y esperar hasta que la FPGA escriba en el puerto se podría usar el siguiente código:

```
char Buffer[256];
unsigned char Number;
unsigned char ReturnVal;
gets(Buffer);   //Se captura el número
Number = (unsigned char)(atol(Buffer) & 0xff);
PP1000WriteControl(Handle, Number);  //Se envía el dato
PP1000ReadStatus(Handle, &ReturnVal);  //Se espera
```

Es muy recomendado que se chequeen los códigos devueltos por las funciones para confirmar posibles errores. Nosotros los hemos obviado aquí para mayor legibilidad del código.

Consulta el apartado 4.3.2.2 para ver los detalles de cómo se envían mensajes y se leen estos puertos desde la FPGA.

Para más información acerca de las funciones utilizadas consultar el RC1000 Functional Reference Manual.

7.2.4. TRANSFERENCIA POR DMA ENTRE EL HOST Y LA SRAM

Para realizar una transferencia por DMA hay que seguir 5 pasos:

1. Crear un canal DMA.
2. Solicitar el acceso al banco de memoria requerido.
3. Realizar la transferencia DMA.
4. Liberar los bancos de memoria solicitados.
5. Liberar el canal DMA.

La transferencia DMA se puede realizar de dos formas:

1. La transferencia 1D que son series de pequeñas transferencias con saltos en medio. Esta transferencia es la que usamos nosotros y se realiza con la función *PP1000SetupDMAChannel().*
2. La transferencia 2D al cual se usa para transferencias de imágenes de datos con un ancho y un alto de imagen. Esta transferencia se realiza con la función *PP1000Setup2DDMAChannel().*

Los bancos de memoria requeridos pueden ser solicitados desde el Host con la función *PP1000RequestMemoryBank().* Esta función espera hasta que los bancos de memoria requeridos hayan sido cedidos al Host. Hay que asegurarse que la FPGA ha liberado los bancos de memoria que haya solicitado para que el Host pueda tener acceso a ellos.

Para realizar la transferencia DMA hay que llamar a la función *PP1000DoDMA().* Hasta que no se termine la transferencia no se pasará a la siguiente instrucción del código.

Para liberar los bancos de memoria solicitados hay que llamar a la función *PP1000ReleaseMemoryBanck()* . Una vez que se han liberado los bancos por el Host, la FPGA ya puede acceder a ellos.

Consulta el apartado 4.3.3 para ver los detalles de cómo se solicitan y se liberan los bancos de memoria desde la FPGA.

Para liberar los bancos asociados con el canal DMA hay que llamar a la función *PP1000CloseDMAChannel().*

En el siguiente ejemplo se transfiere por DMA 1MB desde un buffer a la dirección 0x1000 del banco 1 de memoria:

```
//Paso 1: Creación del Canal DMA
PP1000SetupDMACHannel(Handle,   //Handle de la tarjeta
    Buffer,          //dirección Host, puntero a un array
    0x201000,   //dirección donde se va hacer la
                transferencia, se cuenta la distancia
                desde el banco 0 hasta el destino.
    0x100000,    //longitud de la transferencia en bytes
    PP1000_PCI2LOCAL, //Marca la dirección de la
                transferencia en este caso es del Host a
                la FPGA
    &Channel);   //devuelve el Handel

/*Paso 2: Se solicita el banco 1 de meoria, se hace por
medio de una máscara de bits con lo cual 0x1 = banco 0;
0x2 = bancos 1; 0x3 = bancos 0 y 1; 0x4 = banco 2; y así
sucesivamente.*/
```

```
PP1000RequestMemoryBanck(Handle, 0x2);
//Paso 3: Se realiza la transferencia DMA
PP1000DoDMA(Channel);
//Paso 4: Se libera el banco 1
PP1000ReleaseMemoryBanck(Handle, 0x2);
//Paso 5: Se libera el canal DMA
PP1000CloseDMAChannel(Channel);
```

Cuando sean requeridas múltiples transferencias del mismo canal DMA, el canal se debe dejar abierto, es decir, no llamar a la función *PP1000CloseDMAChannel().* Los canales DMA deben de ser cerrados lo antes posible después de las transferencias ya que dejar los canales abiertos tiene efectos adversos para el sistema como la reducción de memoria disponible para la paginación de memoria virtual.

Es muy recomendable que se chequeen los códigos devueltos por las funciones para confirmar posibles errores. Nosotros los hemos obviado aquí para mayor legibilidad del código.

Para más información acerca de las funciones utilizadas consultar el RC1000 Functional Reference Manual.

7.2.5. LIBERACIÓN DE LA PLACA

Antes de salir del programa Host, se debe liberar cualquier imagen FPGA que se haya cargado con las funciones *PP1000LoadFile()* o *PP1000Registerimage().* La liberación se realiza con la función *PP1000FreeImage()* una vez con cada imagen que se haya cargado.

Por ejemplo, para liberar la imagen *"Imagen"* se podría usar el siguiente código:

```
Status=PP1000FreeImage(Imagen);
If(Status!=PP1000_SUCCESS))
{
    ReportError(Status);
}
```

Una vez liberadas todas las imágenes FPGA es cuando se debe cerrar la tarjeta con el siguiente código:

```
Status=PP1000CloseCard(Handle);
if(Status!=PP1000_SUCCESS))
{
    ReportError(Status);
}
```

Para más información acerca de las funciones utilizadas consultar el RC1000 Functional Reference Manual.

7.3. USO Y CONFIGURACIÓN DEL SOPORTE SOFTWARE PARA GENERAR EL CIRCUITO SOBRE LA FPGA

7.3.1. CONFIGURACIÓN DEL DK1

El paquete del soporte software de la tarjeta RC1000 proporciona una librería y un archivo ".h" para el DK que contienen un número de macros para permitir la comunicación entre la FPGA y

el Host. Estos archivos se encuentran en la carpeta *DK1* dentro de la carpeta de instalación de Celoxica que en nuestro caso está en:

C :\Archivos de programa\Celoxica\RC1000\fpga\DK1

O también en nuestro proyecto en la carpeta:

Librerias/FPGA

El soporte software puede ser personalizado usando las directivas de precompilación *"#define"* antes de incluir el archivo *pp1000.h.* Esta personalización la explicamos en las siguientes líneas:

Primero se define el tipo de placa que se está utilizando que en nuestro caso es:

#define PP1000_BOARD_TYPE PP1000_V2_VIRTEX

Ahora hay que definir la división del reloj en el que hay estas 3 posibilidades:

Código Fuente	*Efecto*
#define PP1000_DIVIDE1	Establece la división del reloj en 1. El pulso RAM WE se efectúa en la segunda mitad del ciclo de reloj del DK
#define PP1000_DIVIDE3	Establece la división del reloj en 3. El pulso RAM WE se efectúa la mitad del ciclo de reloj del DK
#define PP1000_DIVIDE4	Establece la división del reloj en 4. El pulso RAM WE se efectúa desde 1/2 a los ¾ del ciclo de reloj del DK

Las siguientes directivas de compilación permiten elegir la fuente para el reloj:

Código Fuente	*Efecto*
#define PP1000_CLOCK PP1000_VCLK	Se asigna la entrada VCLK de reloj
#define PP1000_CLOCK PP1000_MCLK	Se asigna la entrada MCLK de reloj
#define PP1000_CLOCK PP1000_NOCLOCK	No se asigna ninguna entrada de reloj. Esta directiva se utiliza para los diseños que no son el nivel superior de la jerarquía

Hay veces que se necesita establecer una frecuencia de reloj determinada del reloj para asegurar el buen funcionamiento del diseño. Para realizar esto usando el lenguaje de programación del DK existe una directiva de precompilación específica. El DK automáticamente pone la frecuencia de reloj a 20MHz aunque puede ser cambiada mediante la siguiente directiva:

Código Fuente	*Efecto*
#define PP1000_CLOCKRATE nn	Donde nn es la frecuencia del reloj en MHz.

A menudo los diseños grandes son diseñados en más de un archivo. Existe una directiva de precompilación para el caso de archivos que no sean el nivel superior de la jerarquía. La configuración de la RAM usada en un archivo que no sea del nivel superior de la jerarquía, tiene que ser la misma que la usada en el nivel superior. La directiva para un archivo que no es el nivel superior de la jerarquía es este:

Código Fuente	*Efecto*
#define PP1000_NOT_TOP_LEVEL	Hay que incluir esta definición para arribos que no sean el nivel superior de la jerarquía. También en estos archivos hay que usar la definición PP1000_NOCLOCK para prevenir la formación de múltiples dominios de reloj

Las macros que acceden a la RAM pueden ser configuradas para acceder a la SRAM con un ancho de 8 bits o 32 bits. Una de las siguientes definiciones debe aparecer al principio del código fuente del DK:

Codigo Fuente	*Efecto*
#define PP1000_8BIT_RAMS	Accede a la SRAM externa con 8 bits x 2-8Mbytes
#define PP1000_32BIT_RAMS	Accede a la SRAM externa con 32 bits x 512Kpalabras-2Mpalabras.

A continuación vamos a mostrar algunos ejemplos de la configuración de la librería de la tarjeta RC1000 para el DK:

En este ejemplo se muestra la configuración de la frecuencia del reloj a 25Mhz, acceso a la SRAM de 32 bits, con fuente VCLK de reloj, un factor 3 de división del reloj, con una placa Virtex:

#define PP1000_DIVIDE3

#define PP1000_CLOCK PP1000_VCLK

#define PP1000_CLOCKRATE 25

#define PP1000_BORARD_TYPE PP1000_V2_VIRTEX

#define PP1000_BOARD_32BIT_RAMS

#include “pp1000.h”

Si el archivo no es el nivel superior de la jerarquía se podría configurar de la siguiente forma:

#define PP1000_NOT_TOP_LEVEL

#define PP1000_CLOCK PP1000_NOCLOCK

#define PP1000_32BIT_RAMS

#include “pp1000.h”

Si el archivo es el nivel superior de la jerarquía pero no usa entrada de reloj se podría configurar de la siguiente forma:

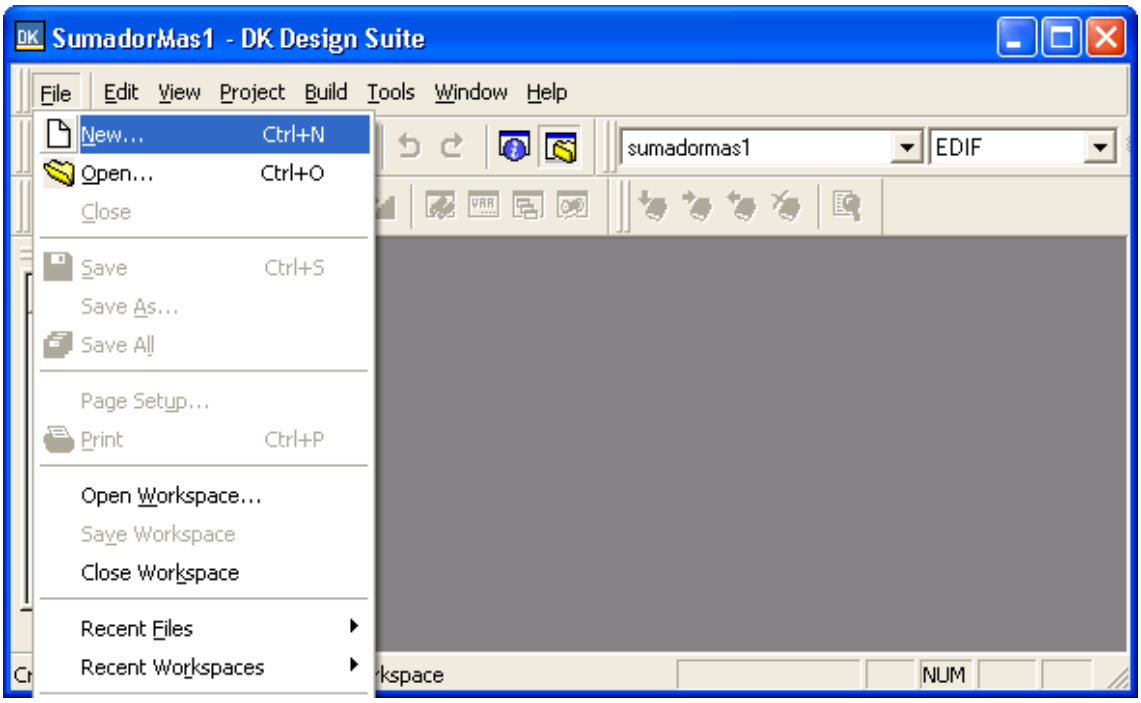
#define PP1000_CLOCK PP1000_NOCLOCK

#define PP1000_32BIT_RAMS

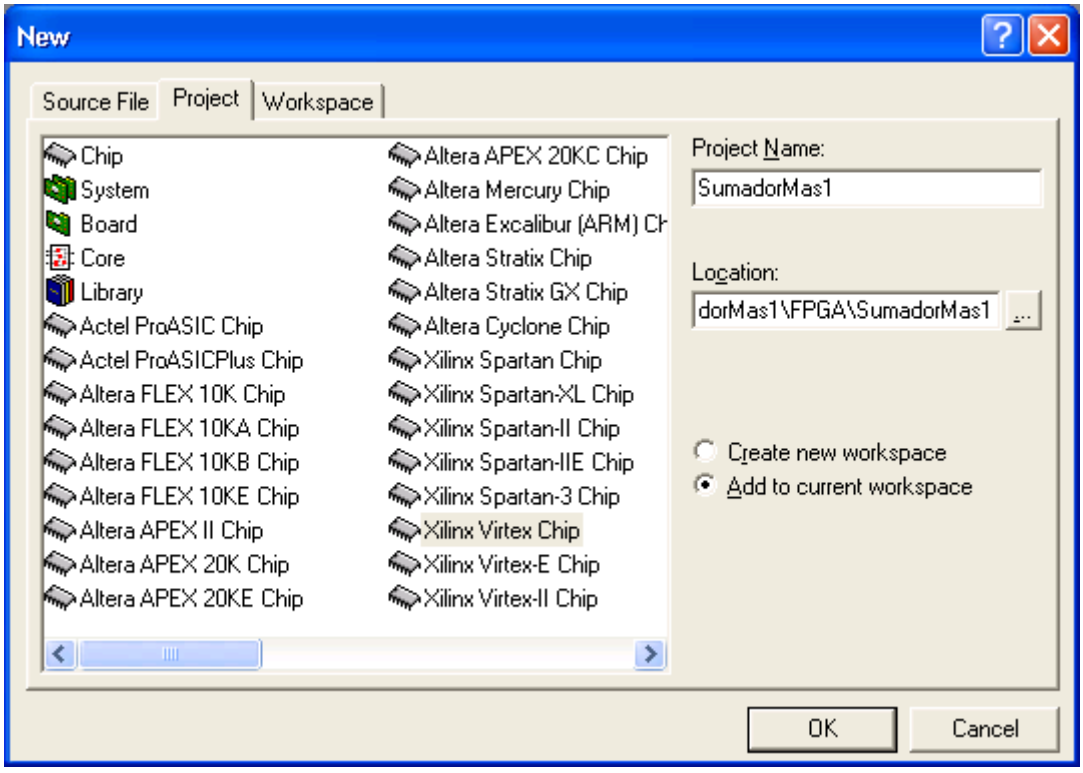
#include “pp1000.h”

Con el fin de que la configuración de la herramienta quede lo más clara posible a continuación mostramos los pasos a seguir con capturas de pantalla lo más detalladas posible.

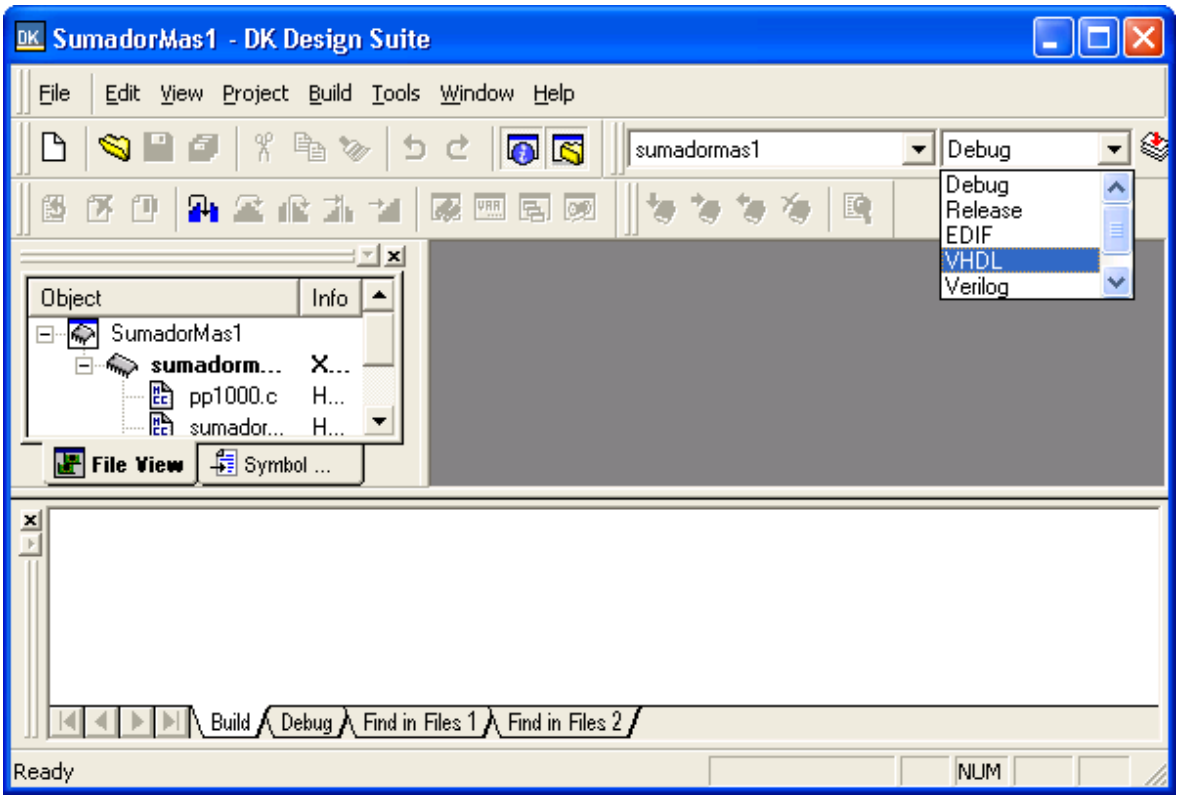
Lo primero que hay que hacer es abrir el DK, una vez abierto hay que pinchar en el menú “*File/New”* como se muestra en la imagen:



En la pantalla "*New*" hay que seleccionar la pestaña de "*Project*" y la tarjeta "*Xilinx Viertex Chip*", se pone el nombre del proyecto en "*Project Name*" *y en "Location"* se pone la ruta donde se quiere poner el proyecto como se muestra a continuación:



Ahora en la barra de Herramientas hay que quitar la opción "*Debug"* que aparece por defecto y cambiarla por "*Edif."*, ya que siempre que usemos el DK vamos a crear un archivo ".edf".



Ahora es el momento de copiar los archivos "*pp1000.h"* y "*pp1000.c"* al directorio raíz del proyecto ya que posteriormente van a ser utilizados y los necesitamos en esta ubicación. Estos archivos se encuentran en la carpeta "*DK1"* dentro de la carpeta de instalación de Celoxica que en nuestro caso está en:

C :\Archivos de programa\Celoxica\RC1000\fpga\DK1

O también en nuestro proyecto en la carpeta:

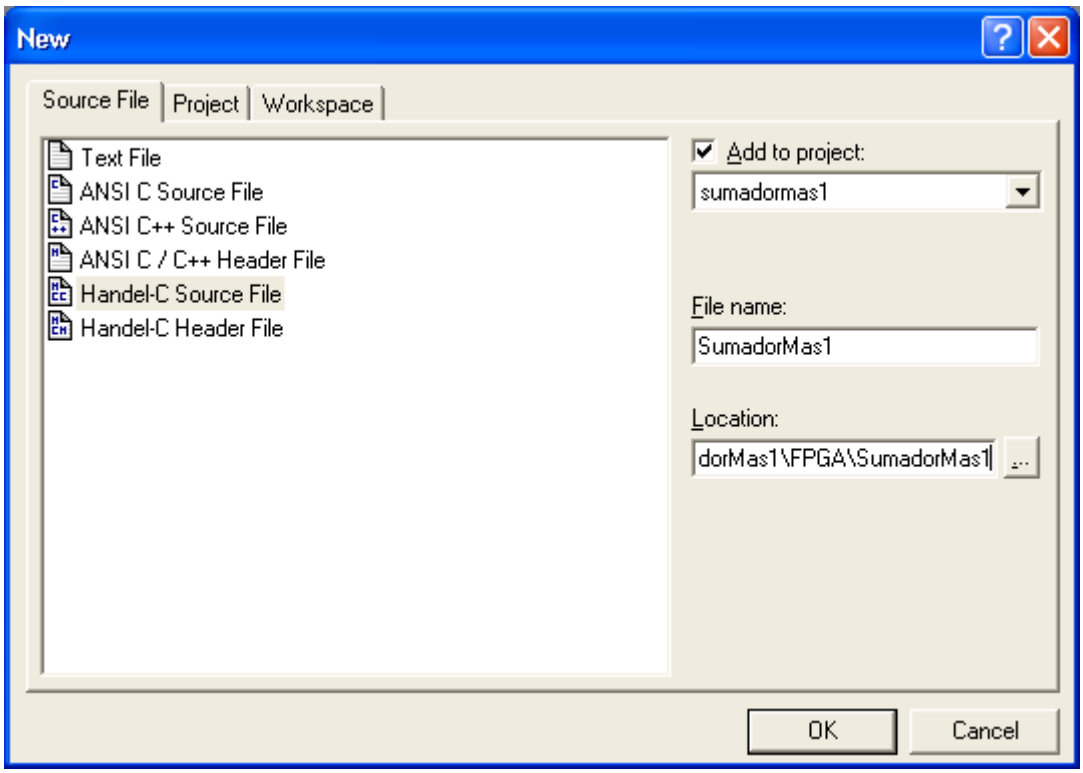
Librerias/FPGA

El archivo "*pp1000.h"* se tiene que incluir en el código como *#include "pp1000.h"*, hay que fijarse que está escrito entre comillas para hacer referencia al directorio raíz del proyecto.

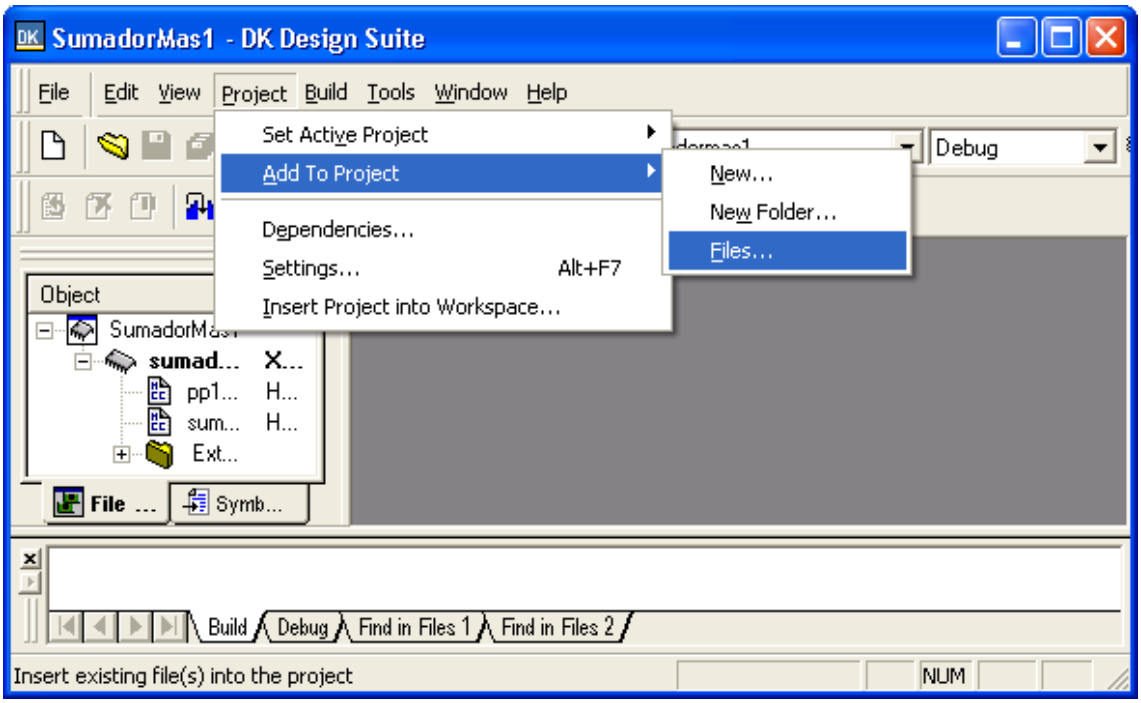
El archivo “*pp1000.c*” hay que incluirlo dentro del proyecto para que compile. No es estrictamente necesario que se copie el archivo en el directorio raíz del proyecto para que funcione correctamente, pero nosotros lo hacemos para no tener que ir a buscarlo a otra ubicación y para tener todos los archivos necesarios para el proyecto perfectamente localizados dentro de la propia carpeta del proyecto.

Llegado a este punto hay dos maneras diferentes de seguir con el proyecto, una es crear el archivo HandelC desde cero y la otra es añadirlo al proyecto si ya lo has escrito anteriormente. La extensión de un archivo HandelC es “.hcc”.

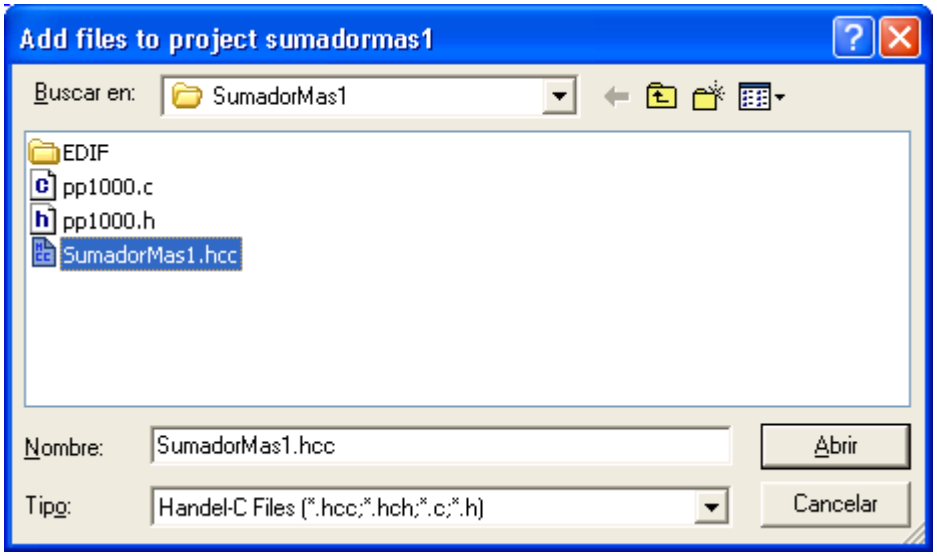
Si se ha elegido la opción de crear el archivo desde cero, entonces hay que pinchar en el menú “*File/New*”, en la pantalla que sale hay que seleccionar la pestaña de “*Source File*” y *dentro “Handel-C Source File”* y se pone el nombre del archivo en “*File Name*” como se muestra en la siguiente imagen:



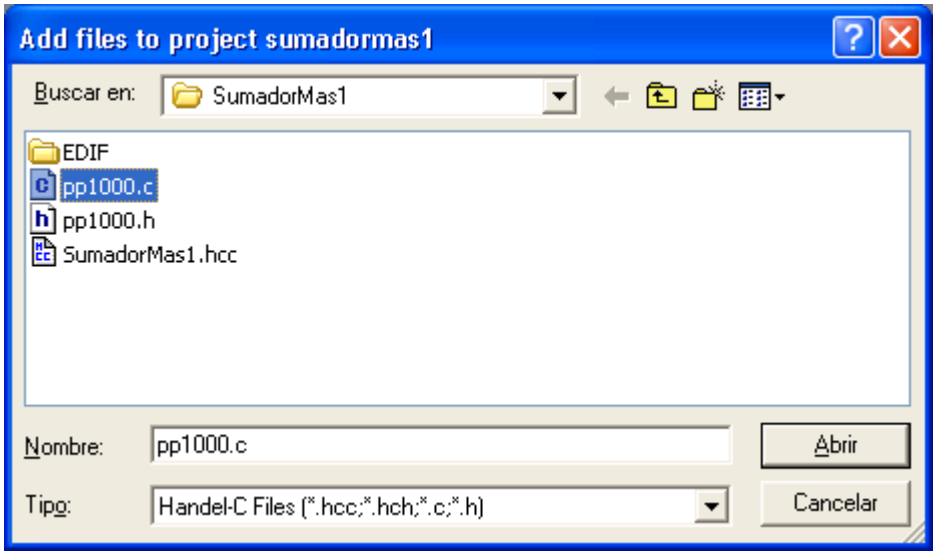
Si por el contrario ya se tiene el archivo HandelC de antemano, lo que hacemos es añadirle al proyecto pinchando en la barra de herramientas en "*Proyect/Add To Proyect/Files...*" como se muestra a continuación:



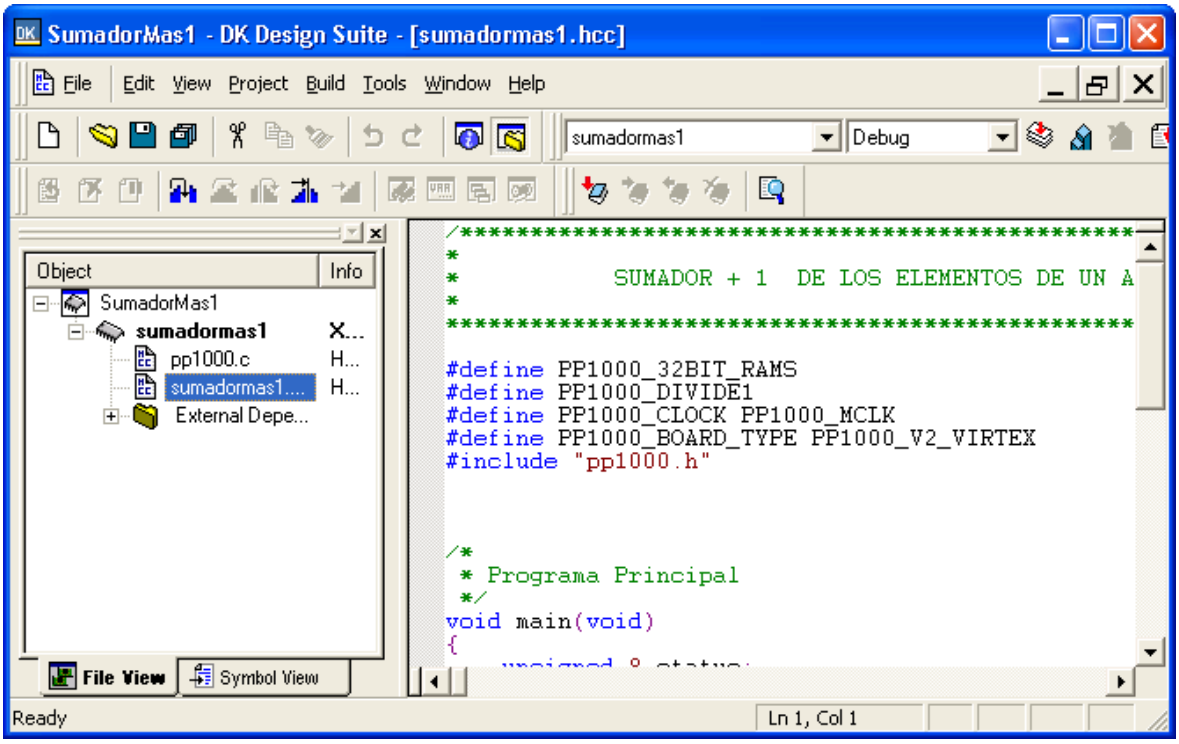
y ahora se añade el archivo HandelC



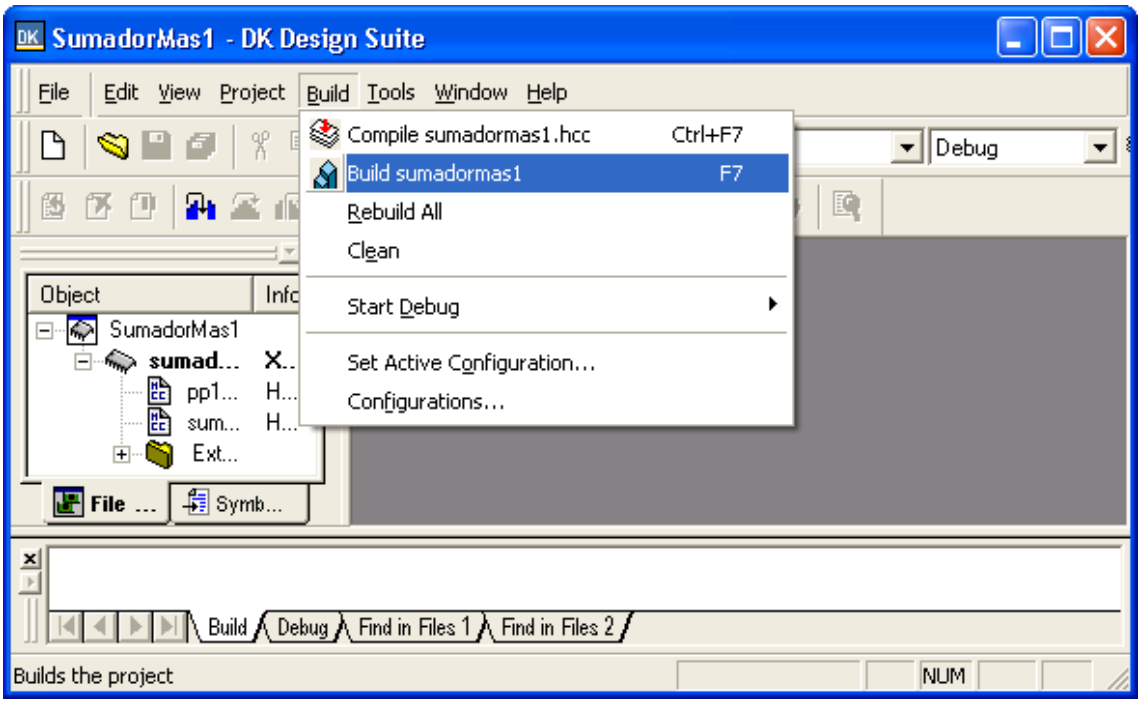
A continuación hay que añadir al proyecto el archivo "*pp1000.c*" de la misma manera que se ha añadido anteriormente el archivo HandelC.



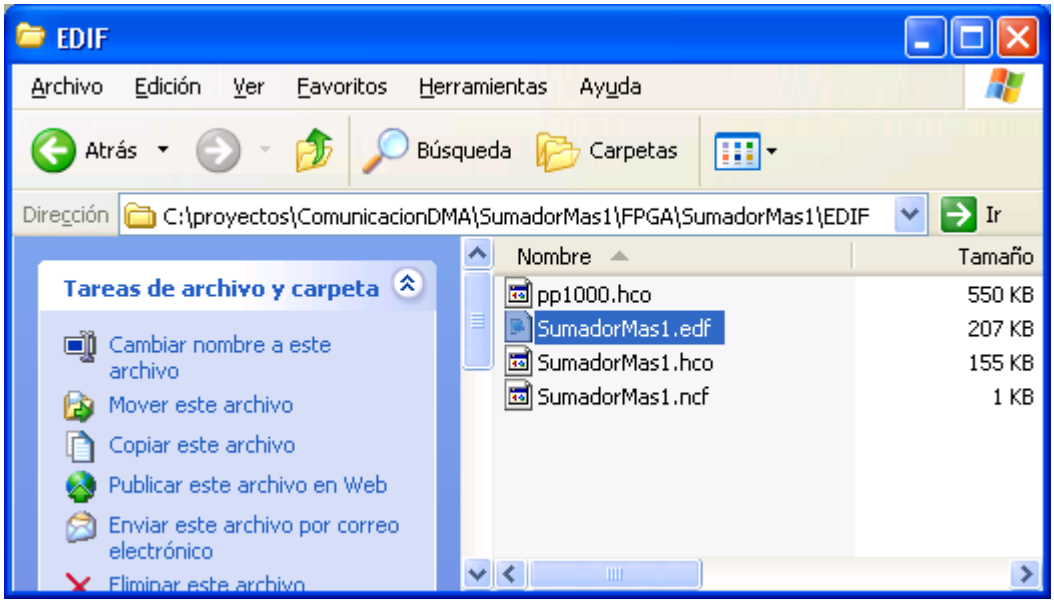
A continuación se abre el archivo HandelC pinchando sobre él para hacer las modificaciones oportunas sobre el código.



Para construir el archivo ".edf" después de haber desarrollado el código fuente en HandelC hay que pinchar en la barra de herramientas en "*Built/Built/(nombre_del_proyecto)".*



Si no hay errores en el código ya estará construido el ".edf" en la carpeta "*EDIF"* del proyecto.



7.3.2. COMUNICACIÓN DE LA FPGA CON EL HOST

Hay dos posibilidades de comunicación con el Host, la cuales las explicamos en los dos siguientes apartados.

7.3.2.1. COMUNICACIÓN SINGLE BIT

En el soporte software existen dos funciones de control para los “single pins”. La función *PP1000ReadGPO()* que modifica el estado de uno de los pines y la función *PP1000SetGPI()* que lee el estado del otro pin. Para que estos pines puedan servir de control es necesario que haya un entendimiento entre el Host y la FPGA, para ello, la FPGA modifica el pin *GPI* que es el que sólo se podrá leer desde el Host y lee el pin *GPO* que es el que modifica el Host.

Por ejemplo para leer el estado del pin *GPO* se podría utilizar el siguiente código:

```
Unsigned 1 Valor;
Valor = PP1000ReadGPO();  //1 ciclo de reloj
```

Por ejemplo para establecer el estado del pin GPI a 1 se podría utilizar el siguiente código:

```
PP1000_SetGPI(1); // 1 ciclo de reloj
```

Consulta el apartado 4.2.3.1 para ver los detalles de cómo se modifica y se lee el estado de estos pines desde el Host.

Para más información acerca de las funciones utilizadas consultar el RC1000 Functional Reference Manual.

7.3.2.2. COMUNICACIÓN SINGLE BYTE

La tarjeta RC1000 tiene un puerto de un byte de ancho entre el Host y la FPGA. Este puerto puede ser usado para enviar pequeños mensajes de control o de estado entre las dos partes. La función *PP1000WriteStatus()* envía un byte desde la FPGA al Host y la función *PP1000ReadControl()* espera a que el Host escriba en el puerto capturando el dato que éste envía.

En el siguiente ejemplo se espera hasta que el Host escribe en el puerto y luego la FPGA escribe el dato recibido de el Host multiplicado por 2:

```
unsigned 8 Val;
PP1000ReadControl(Val);      //Lee el byte del Host
PP1000WriteStatus(Val*2);  // Escribe un valor al Host
```

Consulta el apartado 4.2.3.2 para ver los detalles de cómo se envían mensajes y se leen estos puertos desde el Host.

Para más información acerca de las funciones utilizadas consultar el RC1000 Functional Reference Manual.

7.3.3. TRANSFERENCIA POR DMA ENTRE LA FPGA Y LA SRAM

Antes de acceder a la memoria externa SRAM, los bancos de memoria requeridos tienen que ser solicitados y concedidos. La macro *PP1000RequestMemoryBank()* es la encargada de solicitar los bancos de memoria. La macro *PP1000ReleaseMemoryBank()* se encarga de liberar los bancos de memoria que han sido concedidos.

La externa SRAM puede ser accedida mediante las macros *PP1000ReadBank#()* y *PP1000WriteBank#().* Donde en # se especifica el banco de memoria al que se refiere.

Por ejemplo para copiar una palabra de la dirección 0x1000 a la dirección 0x2000 en el banco 2 se podría usar el siguiente código:

```
unsigned 32 Dato;

/*Paso 1: Se solicita el banco 2 de memoria, se hace por medio de una máscara de bits con lo cual 0x1 = banco 0; 0x2 = bancos 1; 0x3 = bancos 0 y 1; 0x4 = banco 2; y así sucesivamente.*/

PP1000RequestMemoryBank(0x4);

//Paso 2: Se lee el dato de la dirección 0x1000

PP1000ReadBank2(Dato, 0x1000);

// Paso 3: Se escribe el dato a la dirección 0x2000

PP1000WriteBanck2(0x2000, Dato);

//Paso 4: Se libera el banco 2

PP1000ReleaseMemoryBank(0x4);
```

Consulta el apartado 4.2.4 para ver los detalles de cómo se solicitan y se liberan los bancos de memoria desde el Host..

Para más información acerca de las funciones utilizadas consultar el RC1000 Functional Reference Manual.

7.3.4. UTILIDADES

El soporte software del RC1000 contiene varias utilidades de ayuda las cuales listamos a continuación.

7.3.4.1. UTILIDAD list

La utilidad *"list"* simplemente lista todas las tarjetas presentes en el sistema detallando la ID de la tarjeta y el número de serie.

Por ejemplo, en el caso de tener tres tarjetas y teclear *"list"* en el *prompt* nos debe devolver algo así por pantalla:

Number of cards in system: 3

Card ID : 5 *Serial Number : 0x00000001*

Card ID : 15 *Serial Number : 0x0000001b*

Card ID : 22 *Serial Number : 0x0000005f*

7.3.4.2. UTILIDAD setid

La utilidad *"setid"* se usa para camibar la ID de una tarjeta. El formato general de la línea de comando es:

setid NumeroSerie IDTarjeta

El número de serie es requerido para identificar unívocamente la tarjeta, pero dos tarjetas sí pueden tener la misma ID.

Por ejemplo, para reprogramar una tarjeta con un número de serie 0x12 y una ID de 5, se tendría que escribir la siguiente línea:

setid 0x12 5

En windows 98 será necesario reiniciar la máquina para que los cambios tengan efecto.

7.3.4.3. UTILIDAD gencfg

Como se describió anteriormente en el apartado 4.2.2.2.3, el soporte software de la RC1000 permite que imágenes FPGA (archivos .bit) sean incluidos dentro del archivo ejecutable del Host para evitar múltiples archivos imágenes *".bit"* en una aplicación. Esta utilidad coge una imagen FPGA y genera un archivo ".h" de salida. La sintaxis sería esta:

gencfg archivoEntrada.bit archivoSalida.h

La entrada debe de estar en formato binario de Xilinx. (extensión .bit).

Si se edita el archivo de salida *"archivoSalida.h"* con cualquier editor de texto se puede observar dos definiciones:

```
Static unsigned long warpLength = 0x0003abf0;  //Longitud
                                               //de la
                                               //imagen

Static unsigned char warpBuffer[]={            //Datos de

0xff, 0x20, 0x2b, 0xff, 0x20, 0x2b, 0xff, 0x20,   //la imagen

0x20, 0x2b, 0xff, 0x20, 0x2b, 0xff, 0x20, 0x35,

……..}
```

Las dos definiciones del archivo de salida *"warpLength"* y *"warpBuffer"* son requeridas en la función *PP1000RegisterImage()* que se utiliza en el código del Host para registrar la imagen.

En el código fuente del Host es necesario incluir el archivo ".h" de la siguiente forma:

```
#include "archivoSalida.h"
```

Nótese que *"archivoSalida"* está escrito entre comillas para indicar que el archivo al que hace referencia tiene que estar en el directorio raíz del proyecto.

Para registrar el archivo imagen *"archivoSalida.h"* se podría usar el siguiente código:

```
#include "archivoSalida.h"

PP1000_IMAGE Imagen;

Status=PP1000RegisterImage(warpBuffer, warpLength, &Imagen);

if(Status!=PP1000_SUCCESS)

{

    ReportError(Status);

}
```

Para configurar la FPGA se podría hacer del siguiente modo:

```
Status=PP1000ConfigureFPGA(Handle, Imagen);

if(Status!=PP1000_SUCCESS)

{

    ReportError(Status);

}
```

Para más información acerca de las funciones utilizadas consultar el RC1000 Functional Reference Manual.

7.3.4.4. UTILIDAD loadfpga

La utilidad *"loadfpga"* se usa para configurar la FPGA desde la línea de comandos del *prompt* sin necesidad de tener una aplicación Host. El formato general de la línea de comandos es:

loadfpga {-i IDTarjeta} {-c FrecReloj} NombreArchivo

"IDTarjeta" es el ID de la tarjeta a configurar. Si este valor no se pone se configurará la primera tarjeta libre del sistema.

"FrecReloj" especifica la frecuencia a la que se quiere poner el reloj programable. Esta opción es opcional para diseños que usan el reloj programable. Para más detalles sobre los relojes consultar el RC1000 Hardware Reference Manual.

"NombreArchivo" especifica el nombre del archivo imagen FPGA. Este archivo tiene que estar en el formato binario de Xilinx con la extensión .bit.

7.3.4.5. UTILIDAD diag

La utilidad *"diag"* permite hacer diagnósticos de la placa sin necesidad de un programa Host. El formato general de la línea de comandos es:

diag {Script}

Cuando se ejecuta sin un *script* como parámetro, la utilidad entra en un modo interactivo que permite escribir comandos en el *prompt*. Si se escribe con un *script* como parámetro entonces la utilidad ejecutará los comandos escritos en el *script*.

Por ejemplo, para ejecutar el programa de prueba que hicimos nosotros y que está en el apartado 4.6.1 que consiste en

una prueba de comunicación *Single Bit*, habría que escribir las siguientes líneas de comando:

C:\Archivos de Programa\Celoxica\RC1000\util\diag

→ RC1000-PP Diagnostics Utility v1.00

> list //listamos las tarjetas para saber el número de serie //y utilizarlo más adelante con el comando "card" //para configurar la tarjeta.

→ Number of cards in system : 1

→ Card ID : 5 Serial : 0x00000065

> card 0x00000065 //se configura la tarjeta con ese //número de serie.

> clock mclk 1000000 //se configura la frecuencia de //reloj.

>config contadorleds.bit //se configura la imagen de la //FPGA.

>setgpo 0 //se pone a cero el estado del pin GPO de la //FPGA. Pararía de contar el contador.

>setgpo 1 //se pone a uno el estado del pin GPO de la //FPGA. Seguiría contando el contador.

A continuación vamos a listar todos los comandos que están integrados dentro de la utilidad *"diag":*

- help: Lista todos los comandos que se pueden usar con la utilidad *"diag"* .

- quit: Para salir de la aplicación *“diag”* .
- list: Lista todas las tarjetas RC1000 que haya en el sistema.
- card: Abre una de las tarjetas del sistema.
- info: Lista los detalles del hardware de la tarjeta.
- config: Configura la FPGA desde un archivo imagen .bit.
- clock: Programa la frecuencia del reloj.
- setgpo: Modifica el estado del pin *GPO* de la FPGA.
- readgpi: Lee el estado del pin *GPI* o *USERRI*.
- wc: Escribe un byte en el puerto *“Control”* de la placa.
- rs: Lee el byte del puerto *“Control”* de la placa.
- req: Solicita los bancos de memoria.
- rel: Libera los puertos de memoria.
- bs: Devuelve los detalles de quien tiene el control de los bancos de memoria.
- dm: Muestra por pantalla los datos de un rango de la memoria.
- em: Escribe una palabra de 32 bit en la memoria.
- testdma: Hace un test de los bancos de memoria.
- script: Ejecuta los comandos de un archivo especificado.

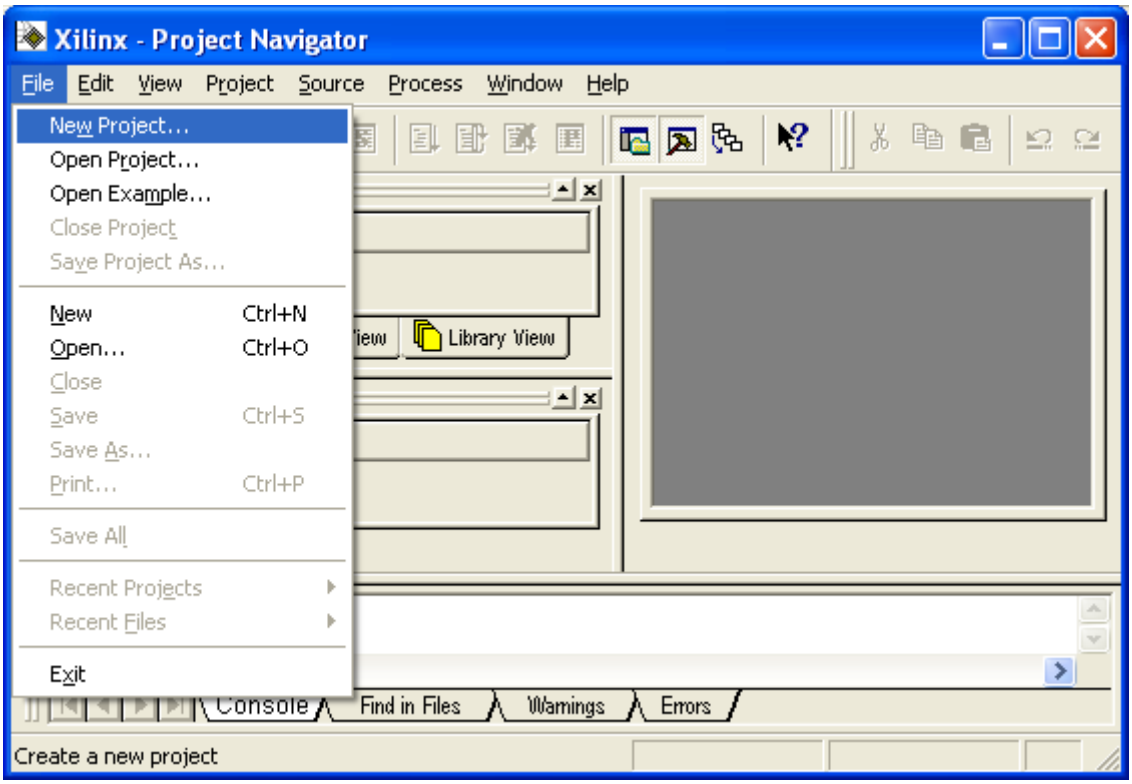
7.4. CONFIGURACIÓN DEL PROJECT NAVIGATOR PARA GENERAR CIRCUITOS *".BIT"*

Con el fin de que la configuración de la herramienta quede lo más clara posible a continuación mostramos los pasos a seguir con capturas de pantalla detalladas.

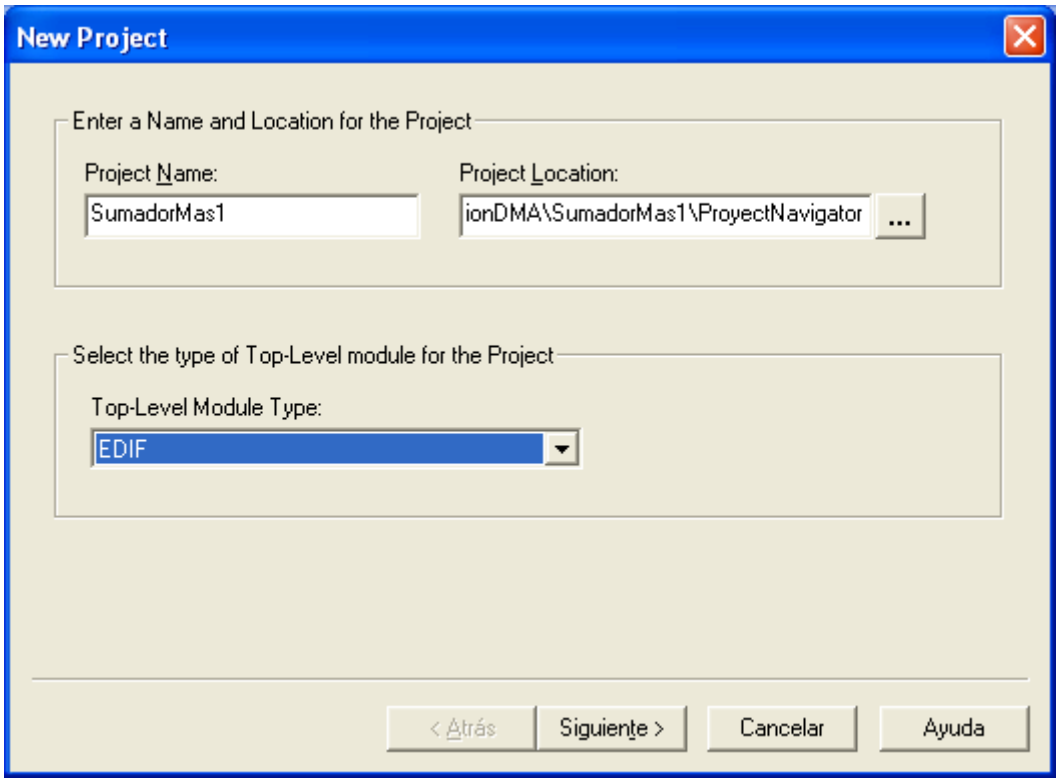
Esta herramienta la hemos utilizado para crear los archivos *".bit".* Nosotros creamos archivos *".bit"* desde dos formatos de archivos diferentes *"edif"* ó *"vhdl". L*a configuración de la herramienta es diferente con cada uno de ellos, primero vamos a explicar la configuración mediante archivos *"edif"* y luego con archivos *"vhdl".*

Mediante archivos "edif"

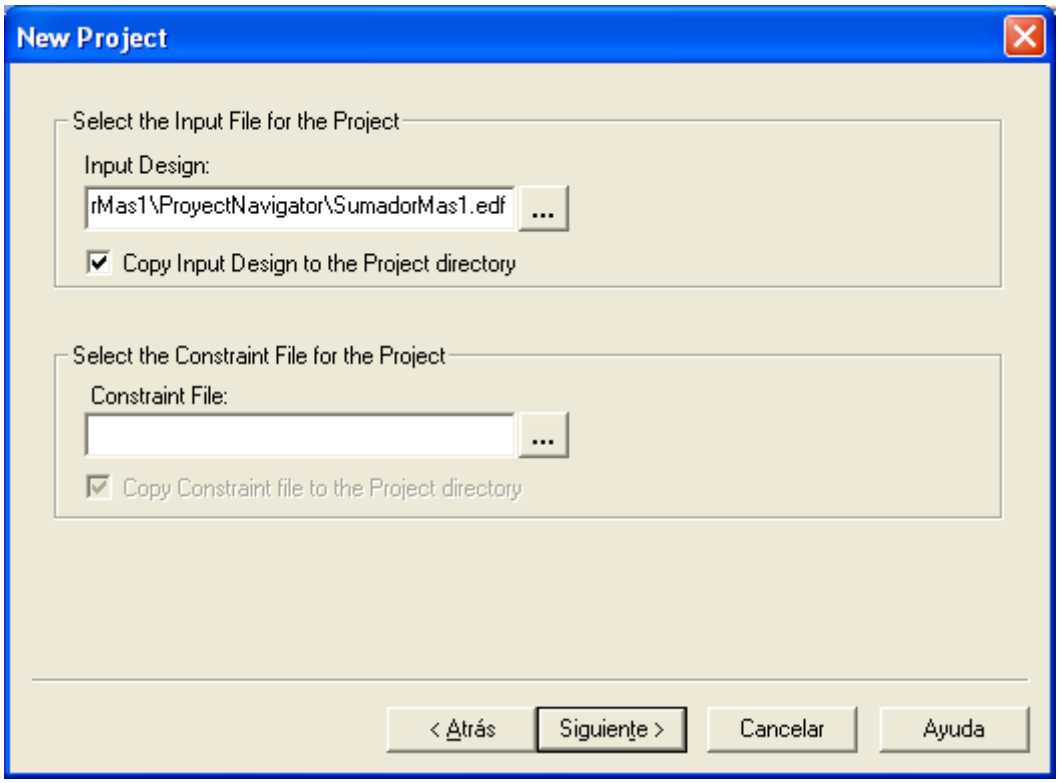
Lo primero que hay que hacer es copiar el archivo *"edif"* a la carpeta raíz donde se vaya a tener alojado el proyecto, luego hay que abrir el Project navigator y crear el proyecto pinchando en el menú *"File/New Project..."* como se muestra a continuación:



En "*Project Name*" se pone el nombre del proyecto, en "*Project Location*" se pone la ruta donde se quiere construir el proyecto y en "*Top-Level Module Type*" se pone "EDIF" como se muestra en la siguiente imagen:

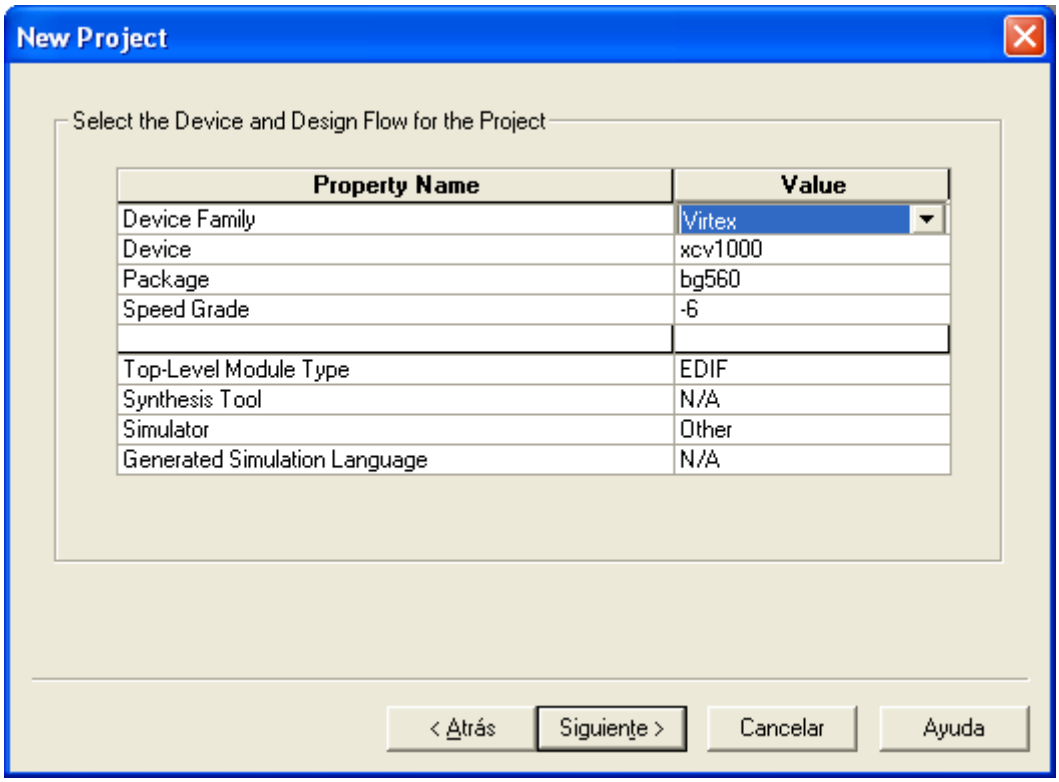


En la siguiente pantalla en *"Input Design"* se pone la ruta donde se encuentra el archivo *".edf"* y en *"Constraint File"* se pone la ruta donde se encuentra el archivo de restricciones *".ucf"* que en nuestro caso no tenemos y se deja vacío como se muestra a continuación:

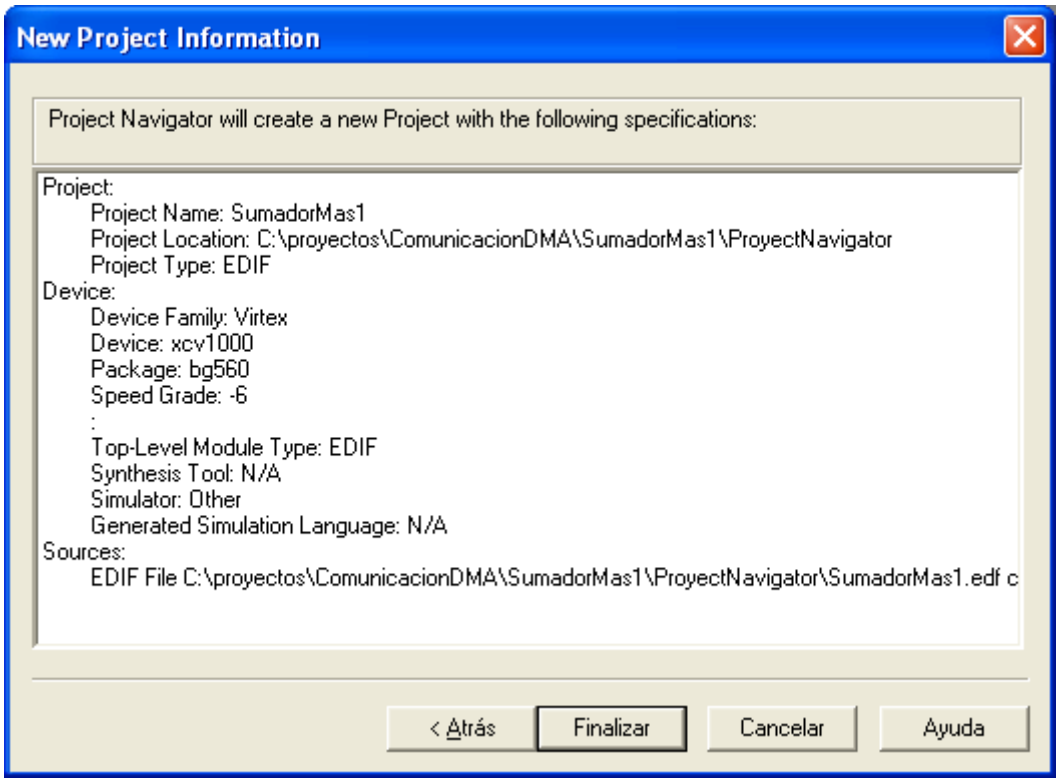


Al pulsar en *"siguiente"* nos saldrá un mensaje de aviso que es debido a que hemos dejado *"Constraint File"* vacío.

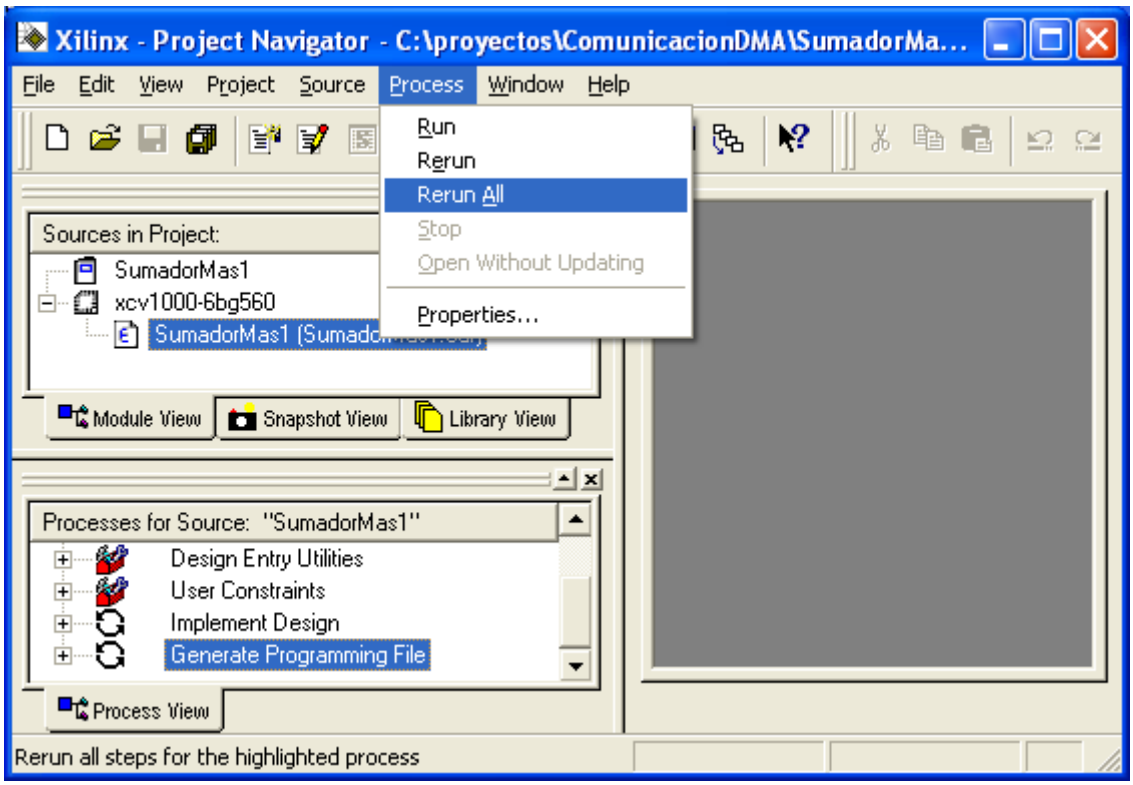
Ahora se definen algunas de las propiedades del proyecto:



Se genera un resumen del proyecto que se va a crear



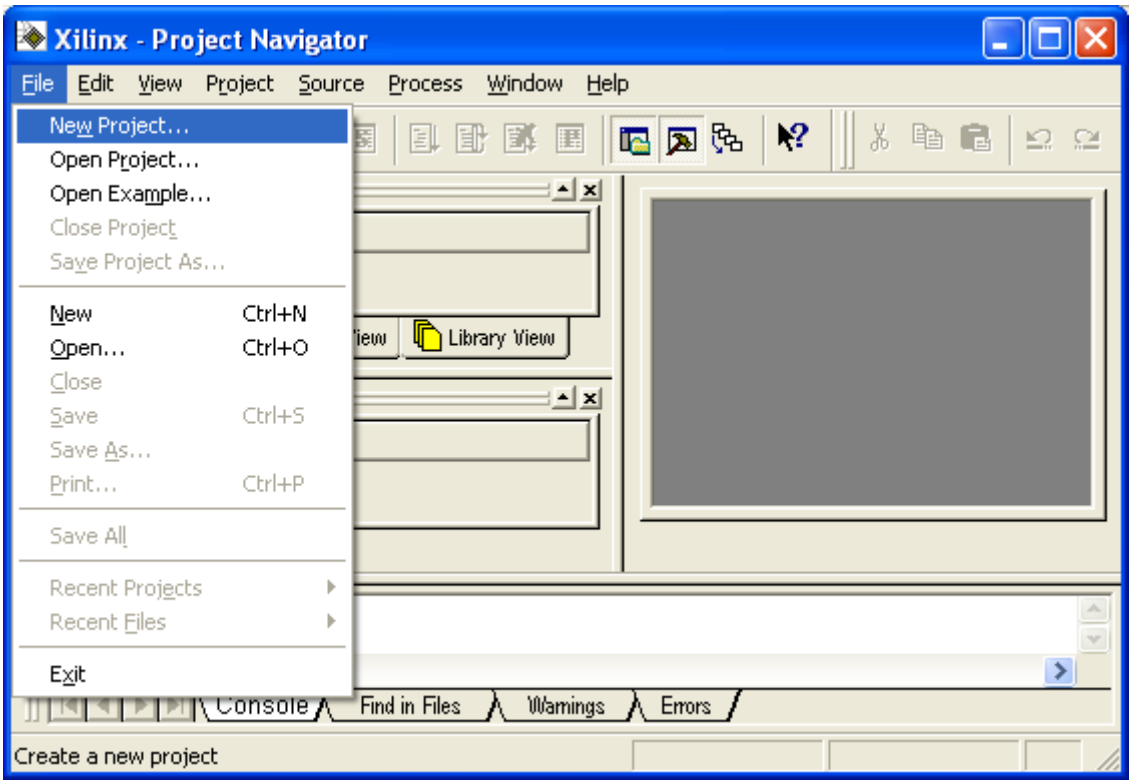
Para generar el archivo *".bit"* se pincha sobre el archivo *".edf"* *(*en nuestro caso *"SumadorMas1.edf")* hasta que se ponga de color azul, luego hay que pinchar en *"Generate Programming File"* hasta que se ponga también en azul y luego hay que ir a la barra de herramientas y pinchar en *"Process/Rerun All" como se muestra a continuación:*



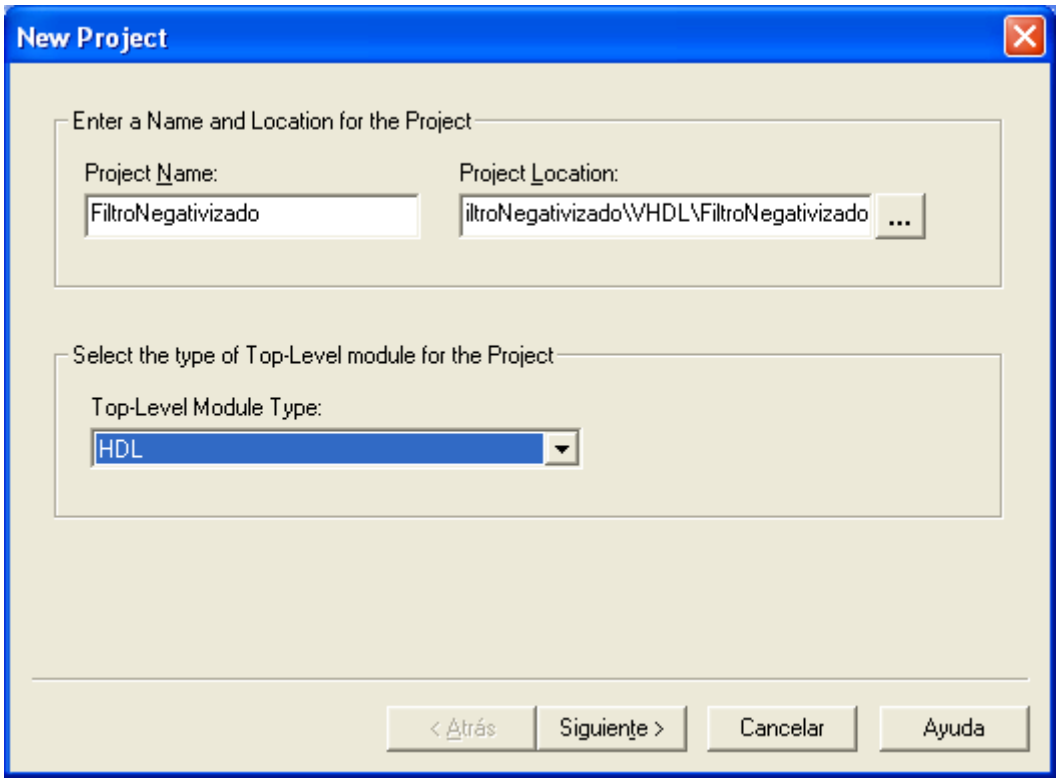
Si todo ha ido bien y no hay errores el archivo *".bit"* estará en la carpeta raíz del proyecto.

Mediante archivos "vhdl"

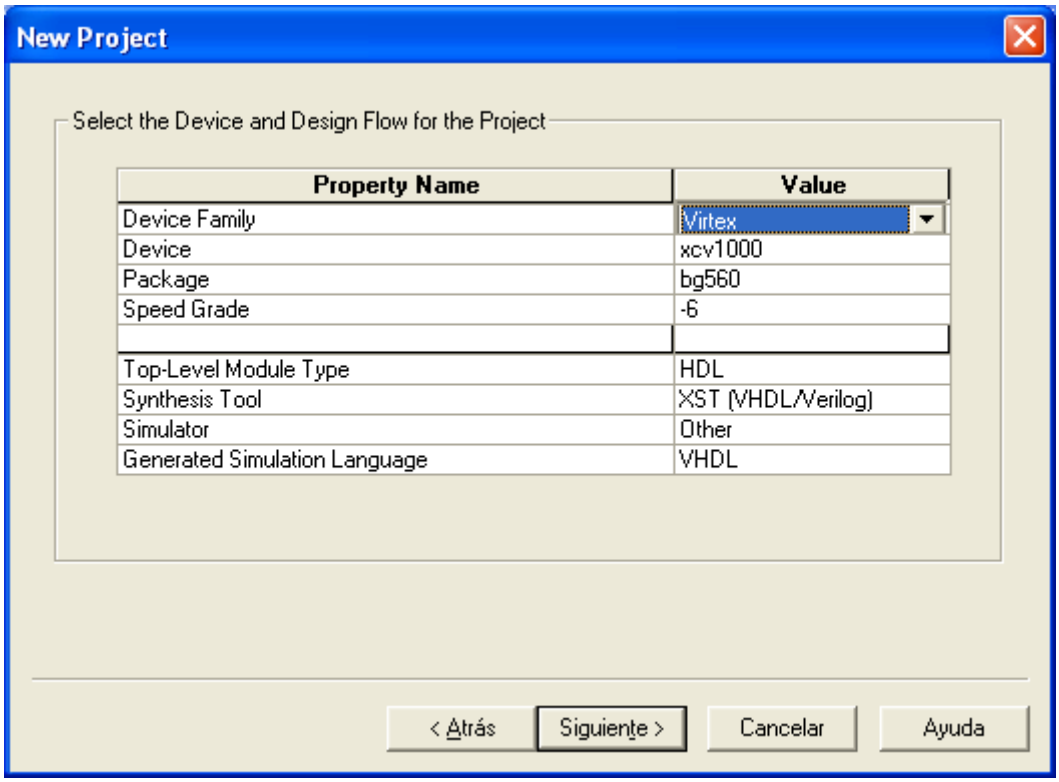
Lo primero que hay que hacer es copiar el archivo *".vhd"* a la carpeta raíz donde se vaya a tener alojado el proyecto, luego hay que abrir el Project navigator y crear el proyecto pinchando en el menú *"File/New Project..."* como se muestra a continuación:



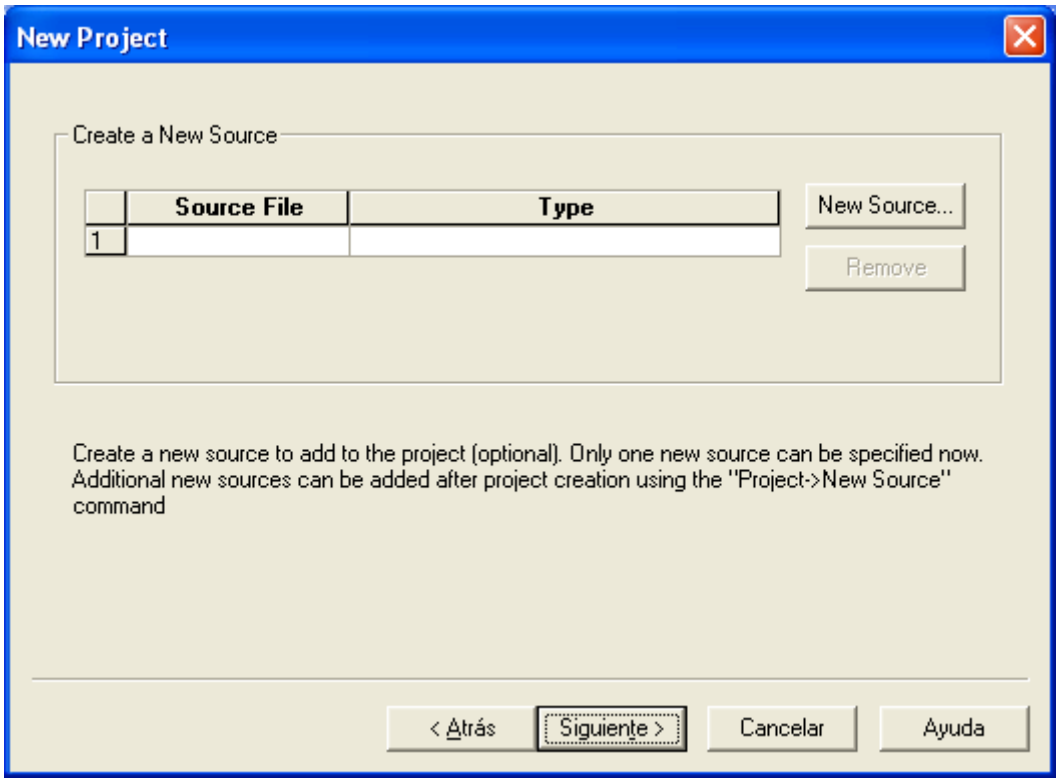
En "*Project Name*" se pone el nombre del proyecto, en "*Project Location*" se pone la ruta donde se quiere construir el proyecto y en "*Top-Level Module Type*" se pone "HDL" como se muestra en la siguiente imagen:



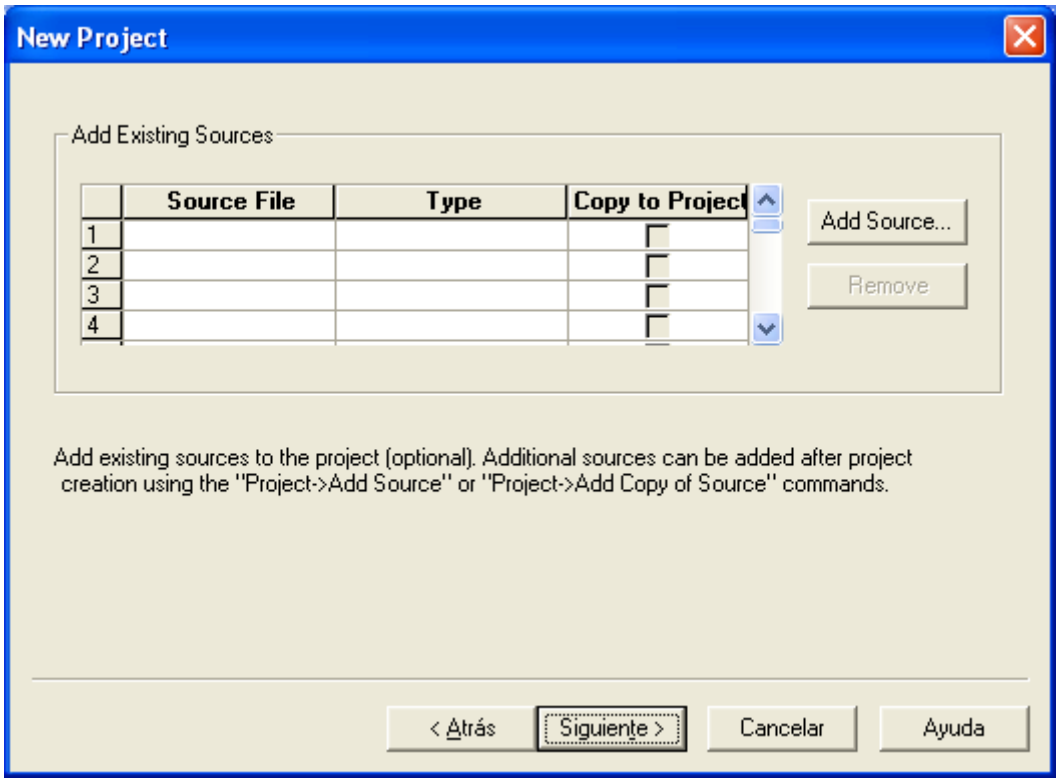
Ahora se definen algunas de las propiedades del proyecto:



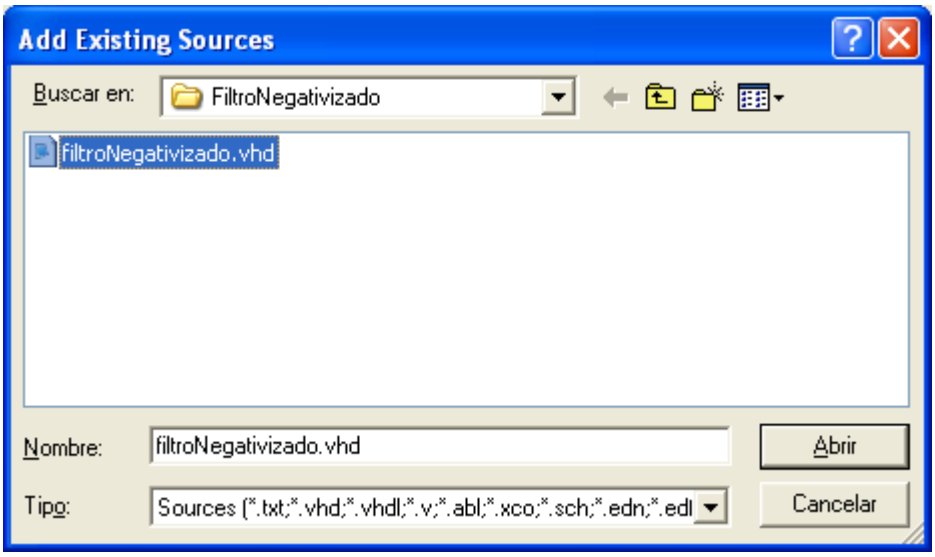
La siguiente pantalla se usa para generar nuevas fuentes automáticamente pero nosotros no lo usamos con lo cual pulsamos en *"siguiente"* sin hacer nada:



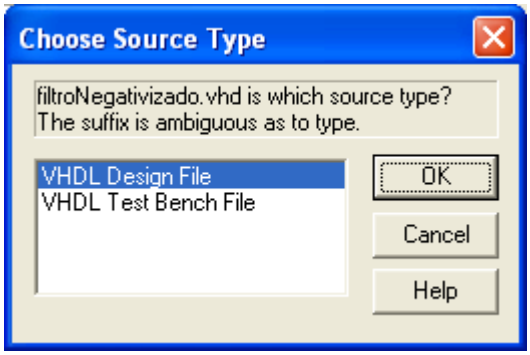
En la siguiente pantalla se añaden las fuentes *".vhd"* necesarias pinchando en *"Add Source..."*



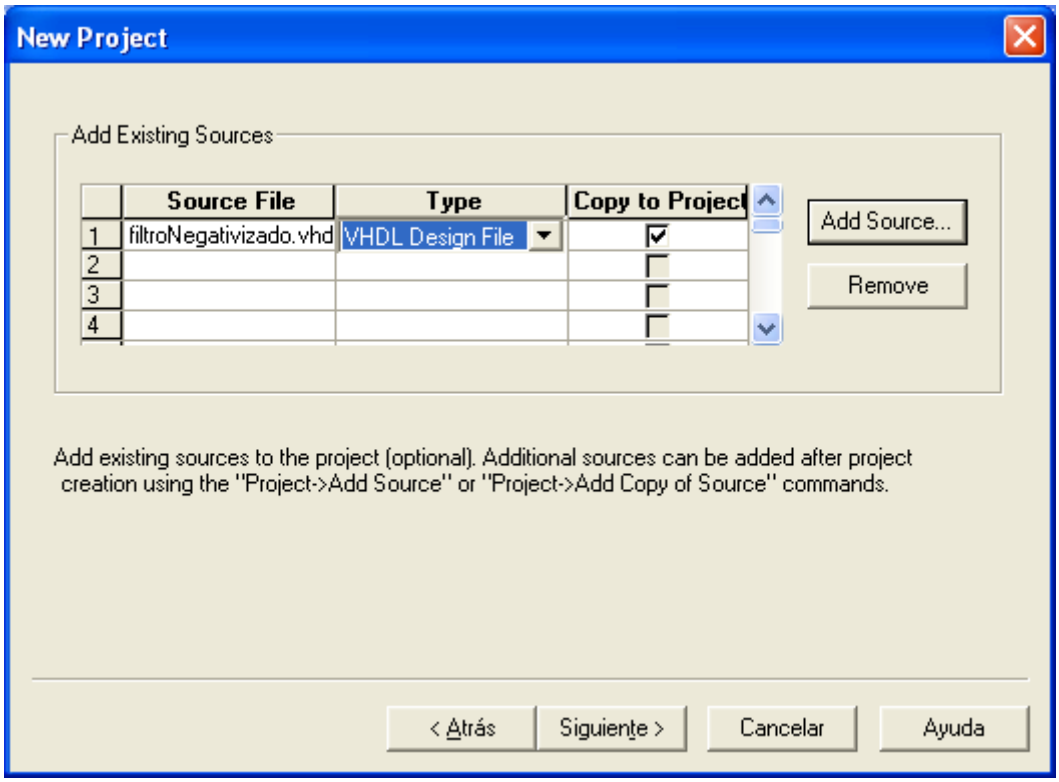
Se selecciona el archivo



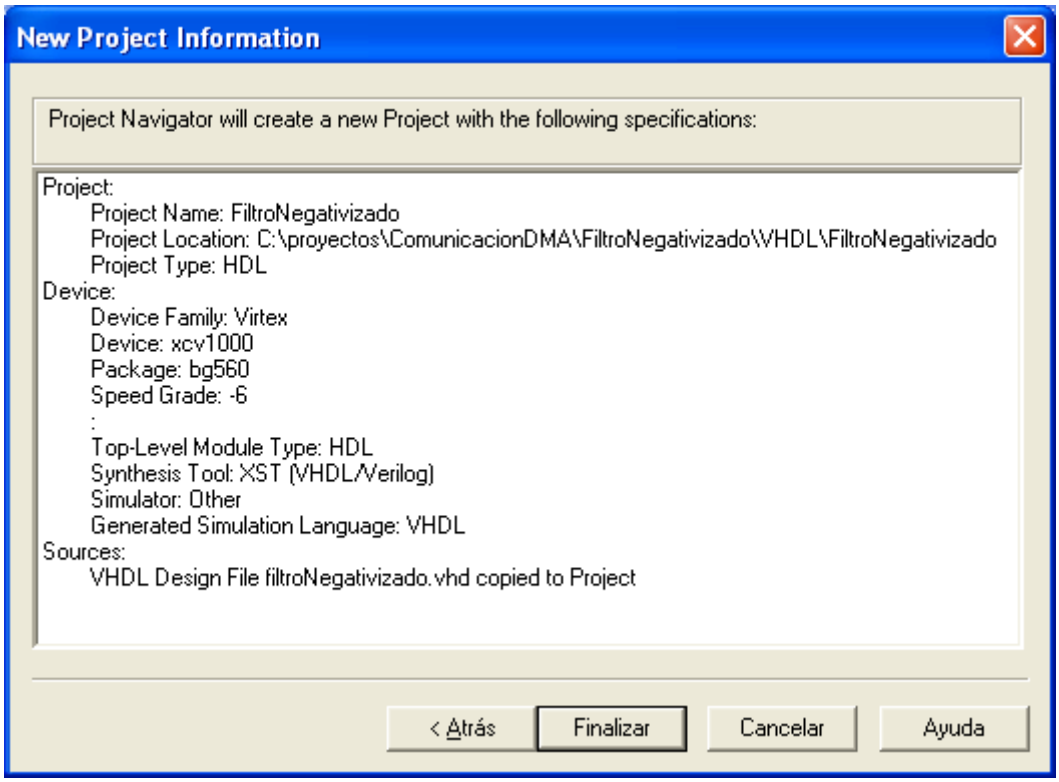
En la siguiente pantalla se selecciona “VHDL Design File”



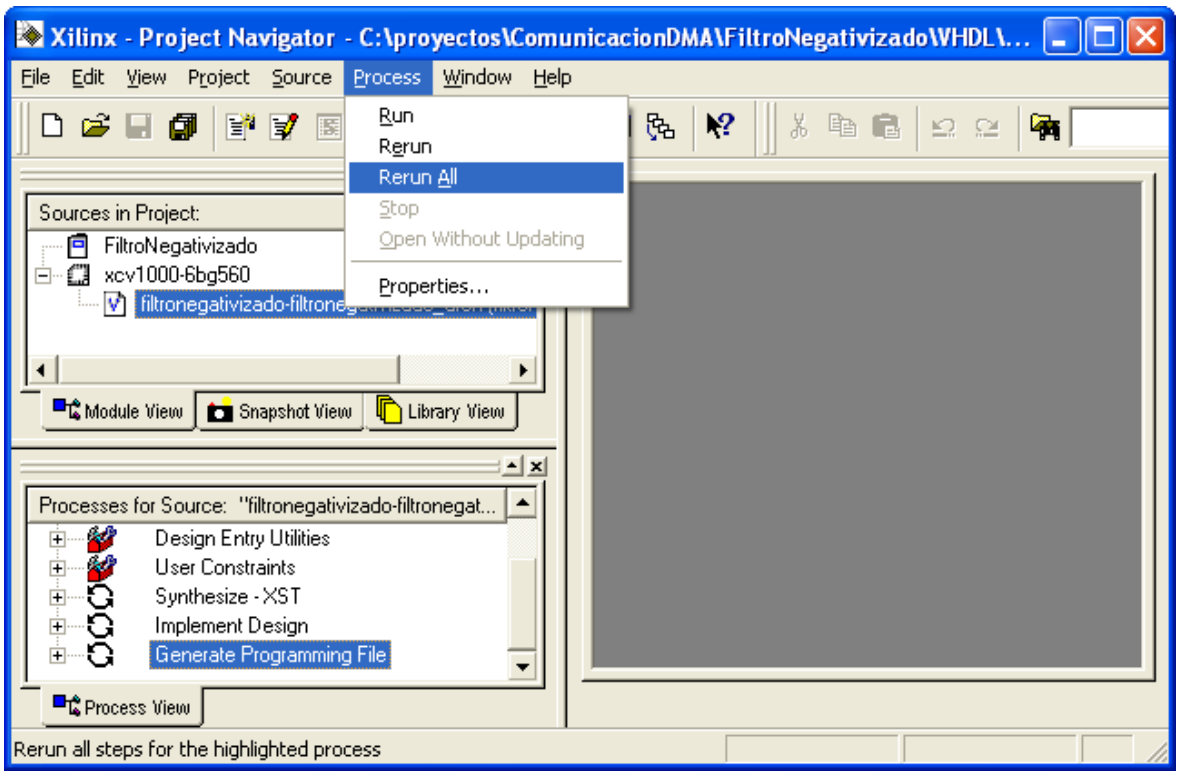
Se añade el archivo *".vhd"* a la lista



Se genera un resumen del proyecto que se va a crear



Para generar el archivo *".bit"* se pincha sobre el archivo *".vhd"* (en nuestro caso *"FiltroNegativizado.vhd")* hasta que se ponga de color azul, luego hay que pinchar en *"Generate Programming File"* hasta que se ponga también en azul y luego hay que ir a la barra de herramientas y pinchar en *"Process/Rerun All" como se muestra a continuación:*



Si todo ha ido bien y no hay errores el archivo *".bit"* estará en la carpeta raíz del proyecto.

7.5. HERRAMIENTAS ADICIONALES

7.5.1. USO DEL SOFTWARE BMP2PACA

Este software le hemos heredado del proyecto de sistemas informáticos del año 2002-2003. La aplicación convierte una imagen en formato *BMP* a formato *PACA* para cargar la imagen en la memoria de la FPGA.

Los archivos *PACA* son archivos donde cada entrada está en una línea distinta, acabada en un salto de línea. Hay dos posibles tipos de entrada, de dirección o de datos:

- Una entrada de dirección es un número en hexadecimal terminado en dos puntos.
- Una entrada de datos consiste en un número binario de 16 bit. En este formato los colores están agrupados de 5 bits en 5 bits. Por ejemplo para este número 0111110000011111 los grupos de colores empezando por los bits más significativos serían los siguientes:

  0 → Se ignora.

  11111 → Código de bits del rojo.

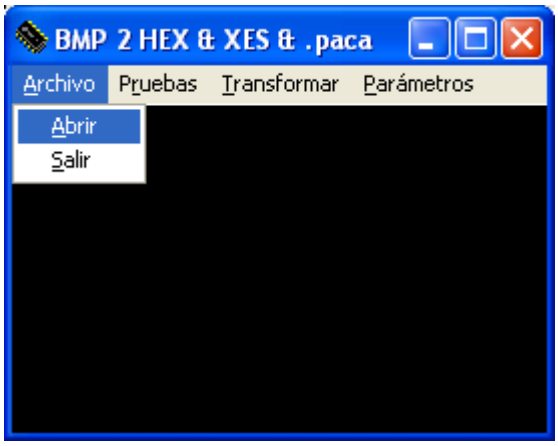
  00000 → Código de bits del verde.

  11111 → Código de bits del azul.

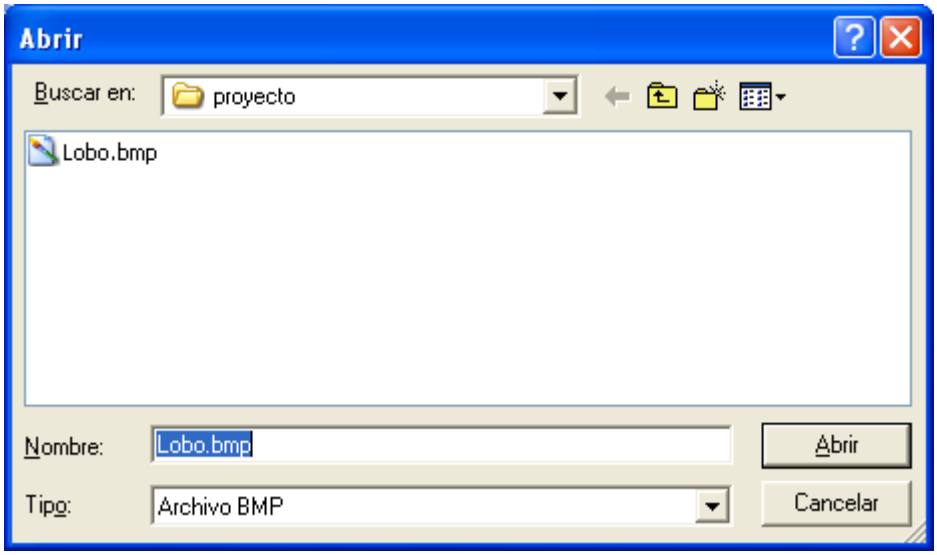
Para dejar más claro el funcionamiento de la herramienta a continuación mostramos su funcionamiento mediante capturas de pantalla.

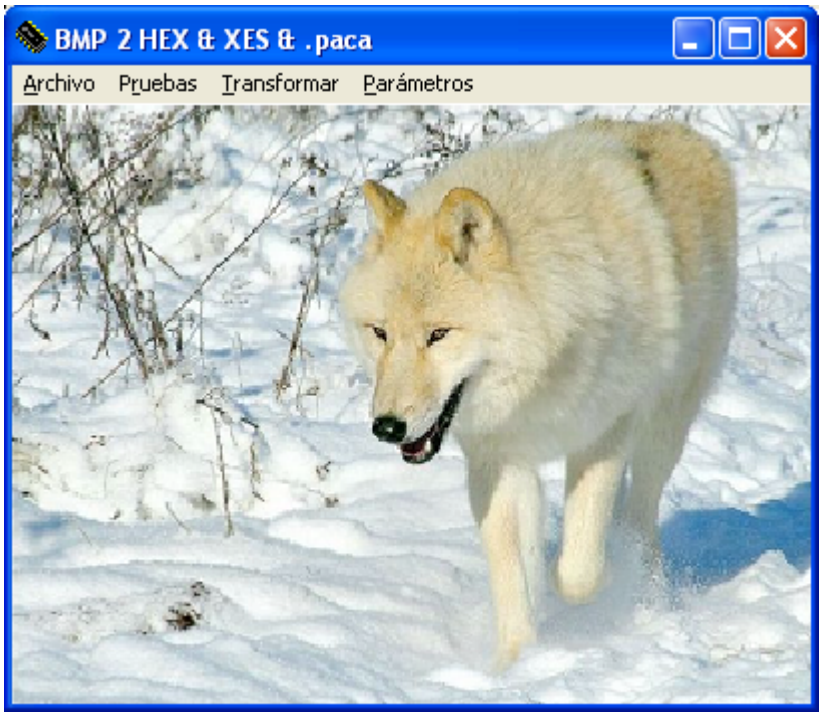
La herramienta ofrece la conversión a más formatos pero nosotros sólo nos vamos a centrar en la transformación de formato *BMP* a *PACA* ya que es el único que usamos. Por ejemplo si se quiere pasar el archivo *"imagen.bmp"* a *"imagen.paca"* el proceso a seguir sería el siguiente:

Se ejecuta el archivo *"BMP2PACA.exe"* y sale la siguiente pantalla en la que hay que pinchar en "*Archivo/Abrir"* como se muestra en la siguiente imagen:

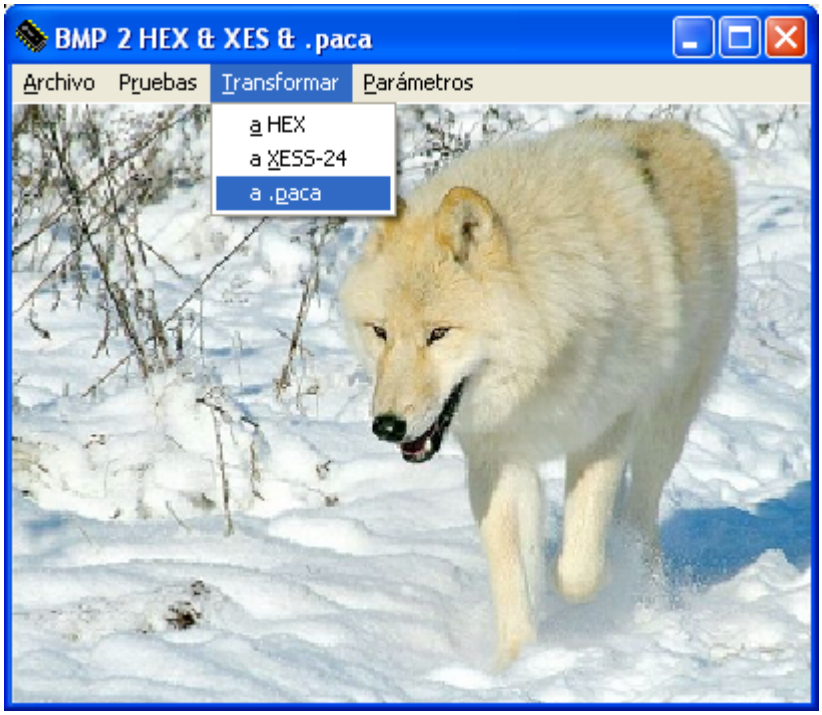


A continuación elegimos la imagen que queremos convertir, pinchamos en abrir:

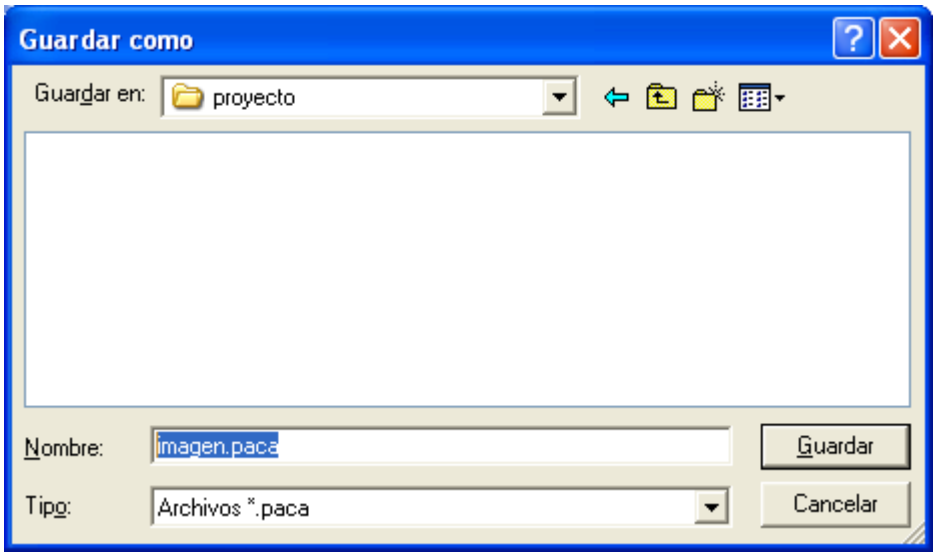




A continuación hacemos la transformación pinchando en "*Transformar/a .paca*" como se muestra a continuación:



Se guarda con el nombre que se quiere dar a la imagen con el formato *PACA*



Ya tendríamos la imagen en formato *PACA*, la cual se podría visualizar con la aplicación *DIBUPACA* que explicamos en el siguiente apartado.

Este software se puede encontrar en nuestra carpeta *"Proyectos/BMP2PACA"* y su código fuente se puede consultar en el proyecto de sistemas informáticos del año 2002-2003.

7.5.2. USO DEL SOFTWARE DIBUPACA

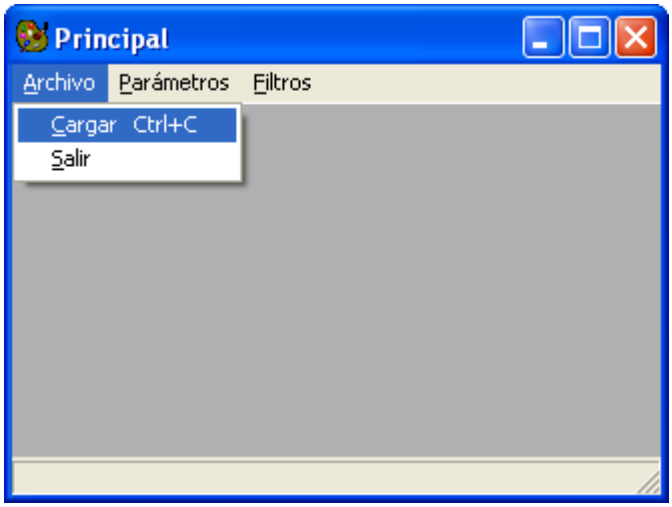
Esta herramienta también es heredada del proyecto de sistemas informáticos del año 2002-2003 y sirve para visualizar las imágenes de formato *PACA*.

Nosotros usamos esta herramienta en los filtros blanco negro y negativizado ya que además de hacer la visualización mediante la aplicación propia del filtro, también se crea un archivo *PACA* con el que se puede visualizar la misma imagen con esta herramienta, esto lo hicimos por motivos de depuración, ya que al principio tuvimos problemas con la visualización de la imagen desde la aplicación y utilizamos este método para depurar, ya que el archivo PACA es muy

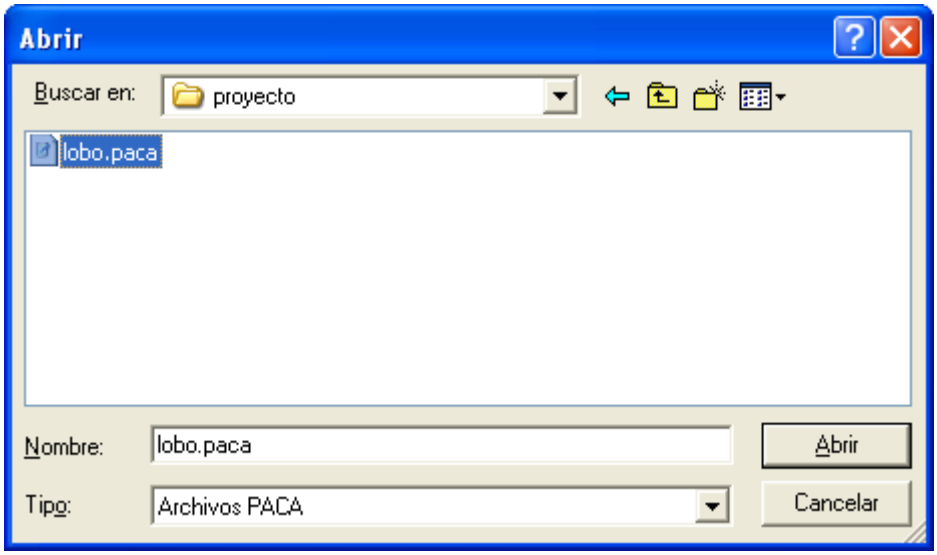
sencillo y se puede ver el formato de la imagen en binario simplemente editando el archivo PACA con cualquier editor de texto. Finalmente lo dejamos como complemento del ejercicio.

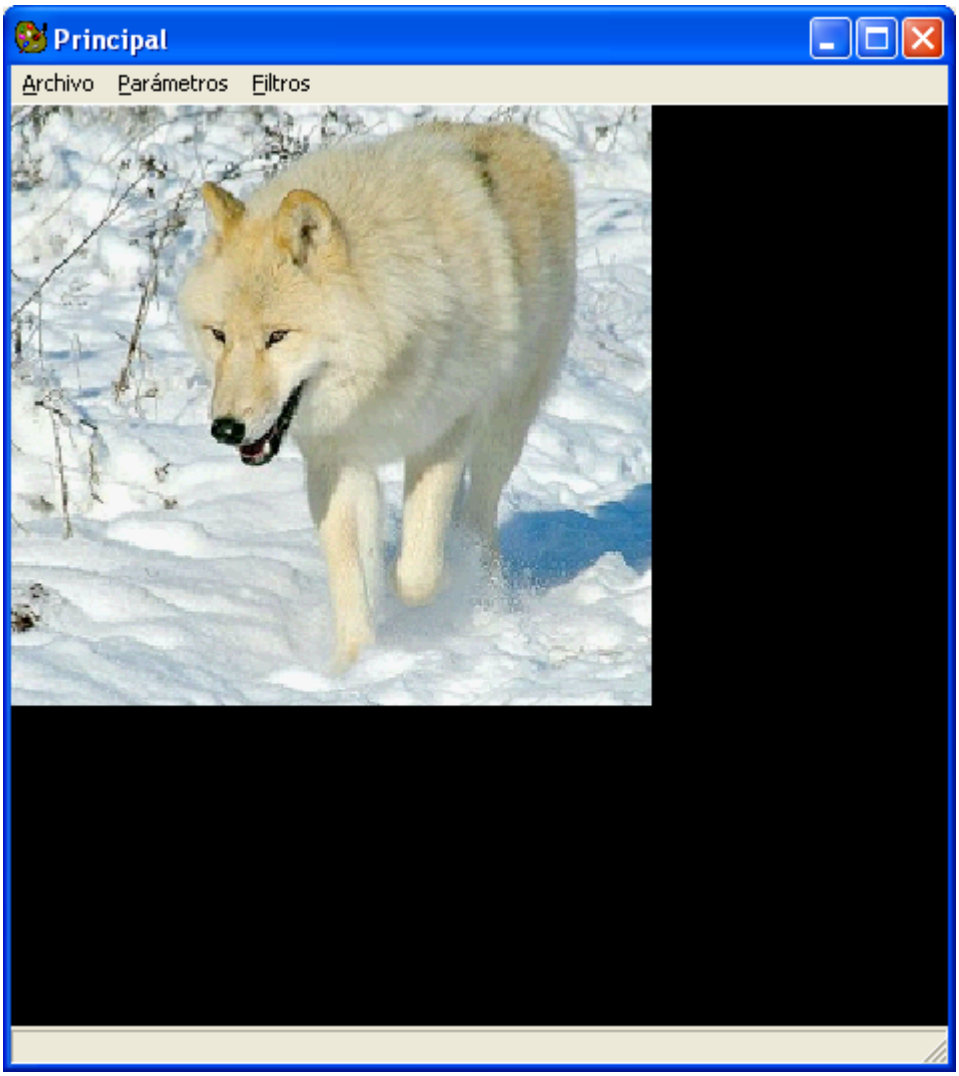
Para dejar más claro el manejo de la herramienta a continuación mostramos su funcionamiento mediante capturas de pantalla.

Se ejecuta el archivo *"DIBUPACA.exe"* y sale la siguiente pantalla en la que hay que pinchar en *"Archivo/Cargar"* como se muestra en la siguiente imagen:



A continuación elegimos la imagen *PACA* que queremos visualizar , pinchamos en abrir y se muestra la siguiente imagen.





Y así tendríamos la imagen *PACA* visualizada.

Este software se puede encontrar en nuestra carpeta *"Proyectos/dibuPaca"* y su código fuente se puede consultar en el proyecto de sistemas informáticos del año 2002-2003.

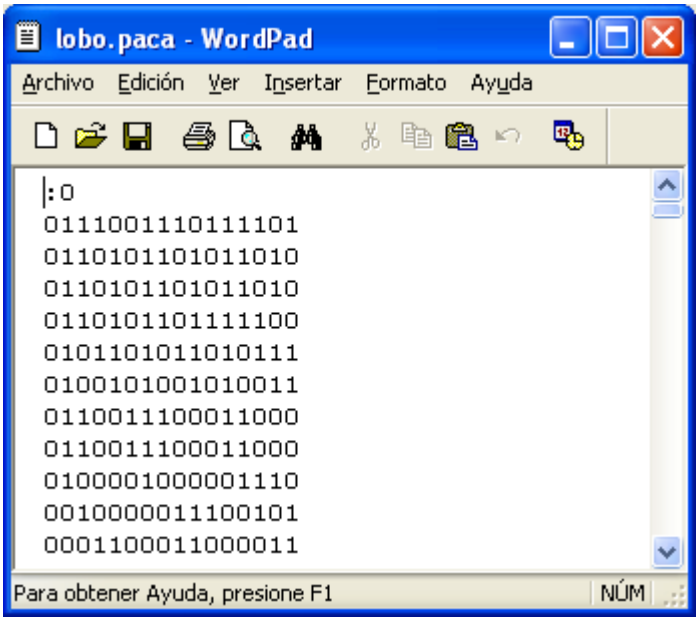
7.5.3. USO DEL SOFTWARE PACATOHEADERHEX

El motivo de implementar esta herramienta fue porque en los filtros blanco negro y negativizado la imagen se carga mediante archivos ".h" (explicado en el punto 4.1.2.2.3), los cuales están formados por un array en formato hexadecimal, es decir, que necesitábamos pasar los números binarios que genera la herramienta

BMP2PACA a un array hexadecimal. Y esto es justamente lo que hace esta herramienta.

Para dejar más claro el manejo de la herramienta a continuación mostramos su funcionamiento mediante capturas de pantalla.

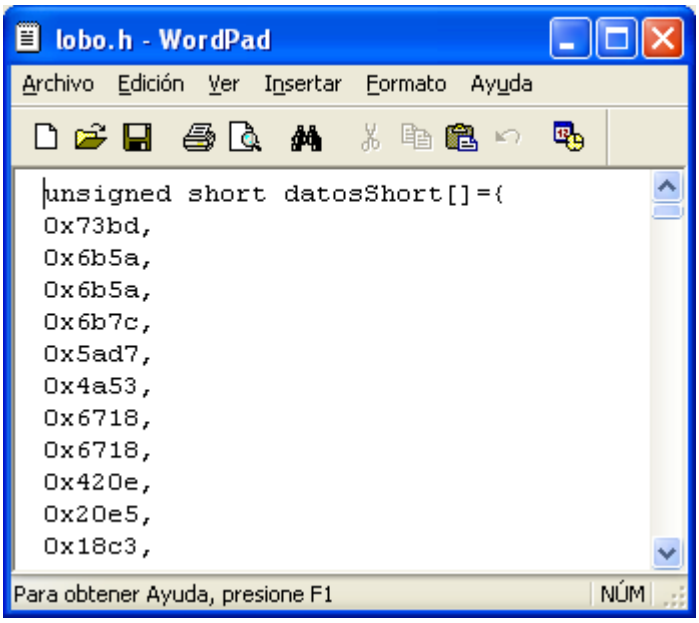
Vamos a pasar un archivo *PACA* a formato ".h". Si abrimos por ejemplo el archivo *"lobo.paca"* con un editor de textos lo que vemos es justamente esto:



Bien, pues ahora procedemos a la conversión, para ello ejecutamos *"PacaToHeaderHex.exe"* con dos argumentos, el primero es la imagen .paca y el segundo el nombre que se le quiere dar al archivo convertido ".h", el comando de ejecución quedaría así:

PacaToHeaderHex lobo.paca lobo.h

Una vez hecho esto, si abrimos el archivo *"lobo.h"* con un editor de textos veríamos la siguiente conversión:



Se puede observar que son los mismos números que había en el archivo *"lobo.paca"* pero en formato hexadecimal necesario para el archivo ".h".

Este software se puede encontrar en nuestra carpeta *"Proyectos/PacaToHeaderHex".*

Este software se puede encontrar en nuestra carpeta *"Proyectos/ PacaToHeaderHex"* y su código fuente se puede consultar en el apartado 7.1.1.

7.6. EJEMPLOS REALIZADOS DE COMUNICACIÓN ENTRE EL HOST Y LA FPGA

Los siguientes ejemplos los hicimos para familiarizarnos con los diferentes tipos de comunicación y la transferencia de datos entre la FPGA y el Host ya que era la primera vez que trabajábamos con estas herramientas.

7.6.1. CONTADORLEDS (COMUNICACIÓN SINGLE BIT)

Este fue el primer ejemplo que hicimos ya que la comunicación *Single Bit* es la más fácil.

El programa hace un contador cuya salida se muestra en 4 de los led's y el bit de comunicación del método *Single Bit* se usa para parar y continuar la cuenta del contador de leds. Con este programa no utilizamos un Host sino que usamos la utilidad *"diag"* del soporte software. Para ver los pasos a seguir para ejecutar esta prueba dirigirse al apartado 4.3.4.5 donde se utiliza justamente esta prueba como ejemplo de explicación.

Esta prueba se puede encontrar en nuestra carpeta *"Proyectos/ComunicacionSingleBit/ContadorLeds" y su* código fuente en el apartado 7.2.1.

7.6.2. MULTIPLICADORX2 (COMUNICACIÓN SINGLE BYTE)

En este ejercicio ya hicimos un Host para comunicarnos con la FPGA ya que es mucho más cómodo que hacerlo con la utilidad *"diag"* del ejercicio anterior; además era la manera con la que teníamos que trabajar con el resto del proyecto y cuanto antes lo hiciéramos mejor.

El Host inicializa la tarjeta FPGA y carga la imagen ".bit" en la FPGA.

El programa Host implementado en Visual C++ es un bucle que pregunta un número, se lo pasa a la FPGA, espera a que la FPGA le mande el dato procesado (multiplicado por 2), imprime el dato en pantalla y vuelve a preguntar otro número hasta que se le introduce un cero y se sale del programa.

El programa de la FPGA implementado en Handel-C es un bucle que primero espera a que el Host le mande un dato, toma el dato, lo multiplica por 2, lo envía de vuelta al Host y vuelve a esperar a otro dato del Host.

Esta prueba se puede encontrar en nuestra carpeta *"Proyectos/ ComunicacionSingleByte/Multipllicadorx2" y su* código fuente en el apartado 7.2.2.

El archivo ejecutable *"Multiplicadorx2.exe"* y el archivo imagen *"multiplicador.bit"* los hemos metido en la carpeta "EJECUTABLE" ya que para que funcione es necesario que ambos archivos estén en la misma carpeta.

7.6.3. SUMADORMAS1 (TRANSFERENCIA DMA)

Esta prueba se hizo para aprender el funcionamiento de la transferencia por DMA tanto entre el Host y la SRAM como entre la FPGA y la SRAM.

En el programa Host implementado en Visual C++ hay 2 arrays de 16 posiciones cada uno, uno de ellos lo inicializamos con números consecutivos desde el 0 hasta el 15 y el otro todo a 0. Antes de hacer nada mostramos por pantalla los valores de ambos arrays, luego se pasa por DMA los valores del array de números consecutivos desde el Host al banco 0 de la memoria, se espera mediante el puerto *"Control y Estado"* a que la FPGA termine de procesar los datos*, y una vez que la FPGA ya ha dado la señal de que ha terminado,* se escriben los valores del banco 1 de memoria en el array que inicialmente estaba inicializado con todas sus celdas a 0 y finalmente se muestra por pantalla los datos del array que acabamos de modificar.

El programa de la FPGA inicialmente espera a que el Host haya terminado de escribir los datos en la memoria mediante el puerto “Control y Estado”, luego lee el banco 0 de memoria sumando en 1 cada valor y escribiéndolo en el banco 1 y finalmente mediante el puerto “Control y Estado” le indica al Host de que ya ha terminado.

El resultado que se aprecia es que el array que inicialmente estaba con todas las celdas a 0 ahora es igual que el que tenía números consecutivos pero sumado en uno cada celda.

Esta prueba se puede encontrar en nuestra carpeta *“Proyectos/ComunicacionDMA/SumadorMas1” y su* código fuente en el apartado 7.2.3.

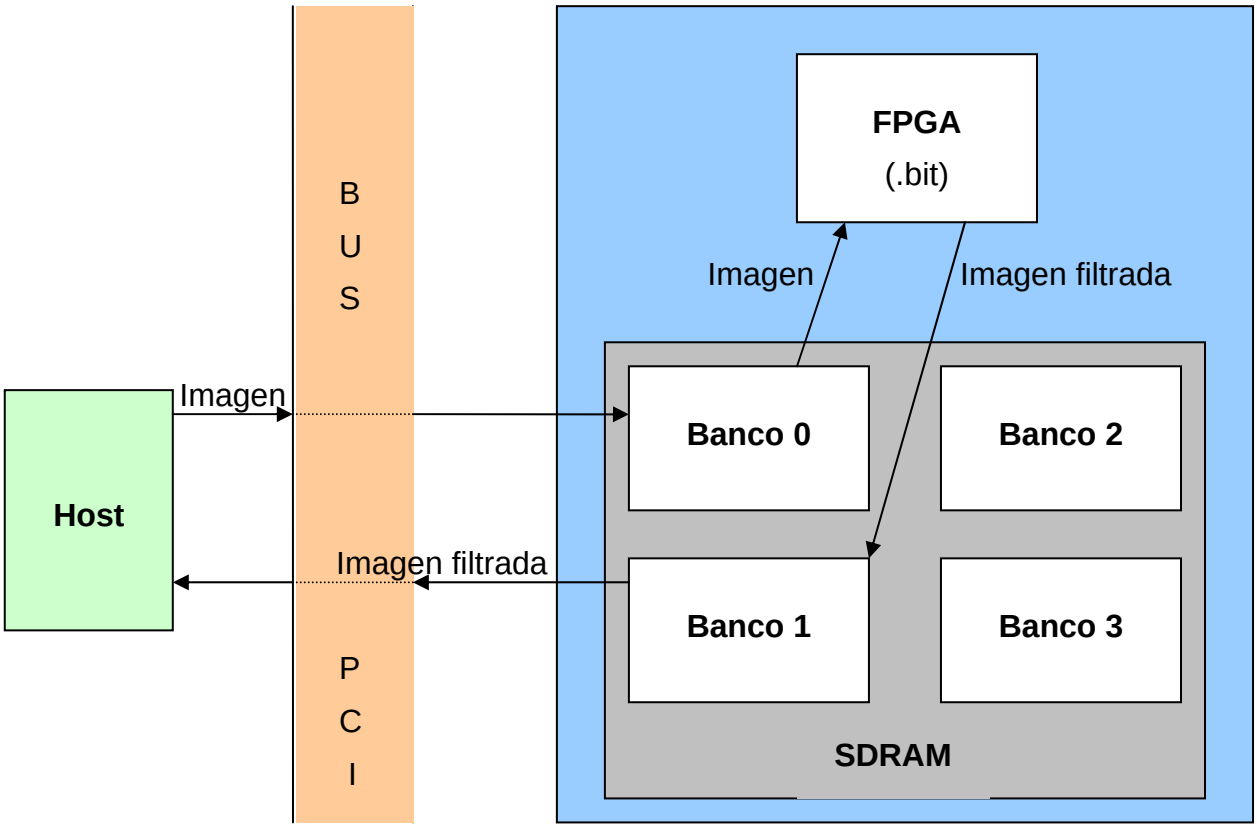
El archivo ejecutable *“SumadorMas1.exe”* y el archivo imagen *“sumadormas1.bit”* los hemos metido en la carpeta “EJECUTABLE” ya que para que funcione es necesario que ambos archivos estén en la misma carpeta.

8. UNIÓN DE VHDL CON HANDEL-C

8.1. IDEAS GENERALES

En este apartado pretendemos explicar la manera de utilizar módulos diseñados en VHDL dentro de código Handel-C que está siendo escrito y compilado con DK2 (y por tanto se ejecutará en la FPGA al programarla con el *".bit"* resultado de todo este proceso). De esta manera podremos pasar como entrada de un módulo valores obtenidos en el código Handel-C y utilizar los valores de salida de dicho módulo para realizar otros cálculos en el código escrito en Handel-C. La documentación utilizada para integrar VHDL en Handel-C procede del documento "*Integrating Handel-C with VHDL, Verilog and EDIF*" de Celoxica, proporcionado por la profesora.

En concreto, nosotros hemos tenido que integrar en el código Handel-C filtros desarrollados en VHDL. El programa "host" (con el que interacciona el cliente y que se ejecuta en la CPU del PC) envía una imagen al banco 0 de memoria de la FPGA por comunicación DMA a través del bus PCI. El programa cargado en la FPGA (*".bit"*) es el resultado de la implementación de un filtro, diseñado en VHDL, y del código Handel-C, desarrollado con el programa DK2, que se encarga de leer esa imagen del banco de memoria de la FPGA mediante DMA, ir enviando estos datos como entrada del filtro y escribiendo por DMA en el banco 1 de memoria la imagen filtrada.



Para comunicar el código Handel-C con el filtro VHDL ha sido necesario utilizar, en el DK2, una definición de interface que se ajustara a la entidad del filtro VHDL, con lo que conseguimos una instancia de esta interface así como definir los datos que serán transmitidos, tanto hacia la entidad VHDL como desde ella. La **definición de un interface** sigue el siguiente formato:

```
interface nombre_entidad_VHDL
        (puerto_salida_entidad [with especificación_puerto]
         {, puerto_salida_entidad [with especificación_puerto]})
nombre_instancia
        (puerto_salida_entidad  [with especificación_puerto]
         {, puerto_salida_entidad [with especificación_puerto]})
        with {busformat = "tipo_busformat"};
```

donde tipo_**busformat** puede tomar los siguientes valores: "B", "B[I]", "B(I)", "B<I>", "BI", "B_I" , "B[N:M]" , "B(N:M)" , "B<N:M>" , "B[N]" , "B(N)" y "B<N>"

Así, tendremos que hacer tantas definiciones de interface de una entidad VHDL como instancias de dicha entidad queramos utilizar en

nuestro código Handel-C, es decir, si quisiéramos tener dos instancias de un determinado tipo de filtro, tendríamos que hacer dos definiciones de su interface. A continuación mostramos un ejemplo donde declararemos dos instancias del filtro blanco y negro, llamadas filtroBN0 y filtroBN1 respectivamente.

```
interface filtroBlancoNegro
        (unsigned 16 datosSalida,
         unsigned 1 terminado)
  filtroBN0
        (unsigned 1 comienzo = auxComienzo1,
         unsigned 16 datosEntrada = Dato0)
        with {busformat= "B(I)"};

interface filtroBlancoNegro
        (unsigned 16 datosSalida,
         unsigned 1 terminado)
  filtroBN1
        (unsigned 1 comienzo = auxComienzo2,
         unsigned 16 datosEntrada = Dato1)
        with {busformat= "B(I)"};
```

A partir de este momento, cualquier cambio en las variables que hemos asociado con los puertos de entrada de los filtros filtroBN0 y filtroBN1 hará que cambie la entrada asociada en el filtro correspondiente y que dicho filtro actúe en consecuencia. Además, ahora podríamos en nuestro código Handel-C asignar a nuestras variables los valores de los puertos de salida de cualquiera de las dos instancias de interface utilizando dichas instancias como si de cualquier objeto normal y corriente se tratara:

```
auxDato0=filtroBN0.datosSalida;
```

Hay que tener muy en cuenta que el **nombre** de la interface en el código Handel-C y de la entidad en el VHDL, los nombres de los puertos en ambos sitos y el orden de los mismos **debe ser idéntico**.

Otra cosa en la que hay que fijarse mucho (si no, nos dará error al generar el *".bit"* en el Project Navigator) es en el valor que pongamos en el "**busformat**" al crear el EDIF con el DK2.

El “busformat” sirve para indicar el nombre que se va a generar en el EDIF para cada componente de un vector de la entidad VHDL definida en la interface Handel-C. Por ejemplo, si uno de los puertos de nuestra entidad VHDL es de la forma *salida std_logic_vector **(**3 downto 0**)***, para que el EDIF sea generado con los nombres correctos (*salida**(**0**)***, *salida**(**1**)***, ...) es necesario poner en la definición del interface *with {bus_format = **“B(I)”**}*

```
interface filtroBlancoNegro
        (unsigned 16 datosSalida,
         unsigned 1 terminado)
  filtroBN0
        (unsigned 1 comienzo = auxComienzo1,
         unsigned 16 datosEntrada = Dato0)
         with {busformat= "B(I)"};
```

De esta manera renombra al formato correcto las salidas generadas por el DK2 en el EDIF. En nuestro caso se podría ver en el EDIF que se generó la línea :

*(port (rename salida0 "salida**(**0**)**") (direction OUTPUT))*

A continuación mostramos una tabla en la que se indica el valor que hay que dar al busformat en todas las situaciones posibles (dado un vector de ejemplo):

Queremos que genere...	Bus_format adecuado
nombreVector**[**Indice**]**	B**[**I**]**
nombreVector**(**Indice**)**	B**(**I**)**
nombreVector**<**Indice**>**	B**<**I**>**
nombreVector_Indice	B_I
nombreVectorIndice	BI

8.2. PASOS QUE SEGUIMOS

8.2.1. UN VHDL CON SEÑALES DE 1-BIT DENTRO DE HANDEL-C

Como primera prueba decidimos hacer un ejemplo sencillo: aún no sabíamos utilizar bien las interfaces ni el busformat, por lo que decidimos comenzar por una entidad VHDL que tuviera una única señal de entrada y una de salida. Ambas señales eran de un sólo bit para así no tener que preocuparnos del bus fomat que, como hemos dicho anteriormente, sólo sirve cuando tenemos señales que son un vector de bits. En concreto, diseñamos un *inversor* de un bit.

Para comprobar que funcionaba, creamos código Handel-C con el DK2 que asigna siempre el valor 0 a la entrada del *inversor* y, dependiendo de si el *inversor* devuelve 0 ó 1, muestra en los leds el valor 2 ó el valor 3 respectivamente. Con esta prueba es obvio que siempre mostraba el valor 3. También realizamos la prueba asignando 1 a la entrada del *inversor*, mostrándose el valor 2, como cabía esperar, en los leds.

En este primer paso perdimos muchísimo tiempo, ya que intentábamos hacerlo con una metodología que no era válida.

Nuestra intención era que el DK2 generara como resultado un archivo VHDL que contuviera el resultado de haber mezclado el código Handel-C con nuestro módulo VHDL (en este caso el *inversor*) y utilizar este VHDL resultado en el Project Navigator para generar su correspondiente *".bit"*. Nos aparecieron cantidad de problemas en el Project Navigator que fuimos resolviendo poco a poco. Finalmente, conseguimos generar el *".bit"* pero al cargarlo en la FPGA no obteníamos ninguna respuesta. Después de muchas pruebas intentando descubrir el fallo llegamos a la conclusión de que el problema estaba en una librería de Celoxica que había que incluir con un VHDL en el Project Navigator al generar el *".bit"*. Enviamos un mail a la gente de soporte de Celoxica explicando nuestra situación. Y no andábamos tan desencaminados ... aunque no en el sentido que nosotros creíamos. El archivo VHDL de Celoxica que pensábamos que estaba mal no es que fuera incorrecto, simplemente que Celoxica no daba soporte a nuestra herramienta de síntesis (XST), que era con la que trabajaba Project Navigator, si utilizábamos VHDL.

Así que tuvimos que cambiar de estrategia. Buscamos información y, finalmente, terminamos encontrando un foro en el que nos dieron una idea. En lugar de que DK2 generara un VHDL y trabajar con VHDL como fuente en Project Navigator, generaríamos un **EDIF** con DK2 como resultado de compilar el Handel-C que utilizaba nuestro *inversor* VHDL. Y en lugar de utilizar un VHDL como fuente para Project Navigator, utilizaríamos ese EDIF.

En este punto nos surgió otro problema: Proyect Navigator no permite incluir módulos VHDL (que eran en lo que teníamos nuestro *inversor*) en un proyecto que tuviera EDIF como fuente. La solución que terminamos encontrando fue sintetizar (con la opción "Add I/O buffers" deshabilitada) nuestro *inversor* VHDL en un proyecto Project Navigator distinto para generar así un archivo **NGC**. Al crear el otro

proyecto Project Navigator en el que la fuente era el EDIF, copiamos el NGC al mismo directorio en el que situamos el EDIF, pero no añadimos dicho NGC al proyecto. Con esto, generamos un *".bit"* que ya funcionó correctamente.

8.2.2. VARIOS VHDL CON SEÑALES DE 1-BIT DENTRO DE HANDEL-C

Lo que nos planteamos llegados a este punto es si habría algún problema en utilizar varios módulos VHDL distintos, dado que nos habíamos encontrado tantos problemas inesperados. Sin embargo, no fue así y todo funcionó correctamente: creamos un proyecto Project Navigator por cada módulo que había que sintetizar (en nuestro caso, una *and* y un *inversor*), generando así su NGC, generamos un EDIF a partir de código Handel-C en el que se utilizaban ambos módulos y generamos un *".bit"* a partir del EDIF y los dos NGC's.

8.2.3. VHDL CON SEÑALES DE TIPO VECTOR DENTRO DE HANDELC

Comentar que antes de hacer esta prueba, hicimos otra análoga en la que los VHDL tenían varias señales de salida de 1 bit.

Ahora ya sólo nos quedaba averiguar cómo utilizar el busformat. Esta parte resulto pesada ya que la documentación al respecto no explicaba nada. El busformat sirve para decir cómo se deben renombrar los vectores de bits, declarados en las interfaces para utilizar entidades VHDL dentro de Handel-C, al generar los EDIF.

En esta prueba, hicimos que la salida del *inversor* fuera un vector de 4 bits en lugar de un único bit. Así, en la declaración de la interface inversor en el código Handel-C hubo que especificar un

valor para el busformat de modo que se renombraran los elementos del vector de salida del interface de la misma forma que se usaban en la entidad VHDL *inversor*, es decir, indexándolos con paréntesis:

En inversor.vhd:

```
salida(0) <= not entrada;
```

En archivohandelc.hcc:

```
interface inversor
        (unsigned 4 salida)
i1
        (unsigned 1 entrada = 0) with {busformat= "B(I)"};
```

8.2.4. CARGAR IMAGEN EN UN BANCO DE MEMORIA Y VISUALIZARLA SIN MODIFICAR

A continuación queríamos dar un paso más: utilizar módulos VHDL dentro de código Handel-C y trabajar con datos de los bancos de memoria de la FPGA. Para ello, teníamos que aprender a comunicarnos con los bancos de memoria a través de DMA.

La idea era que el programa "Host" (el que se ejecuta en la CPU del PC y con el que puede interactuar el usuario) cargara una imagen en el banco 0 de la memoria de la FPGA, el *".bit"* cargado en la FPGA leyera la imagen de dicho banco, la modificara y la almacenara en el banco 1 y que el "Host" la leyera y la visualizara. El *".bit"* sería el generado por Proyect Navigator a partir del NGC del filtro en cuestión a aplicar y el EDIF resultante de compilar con el DK2 el código Handel-C en el que se utiliza el filtro VHDL.

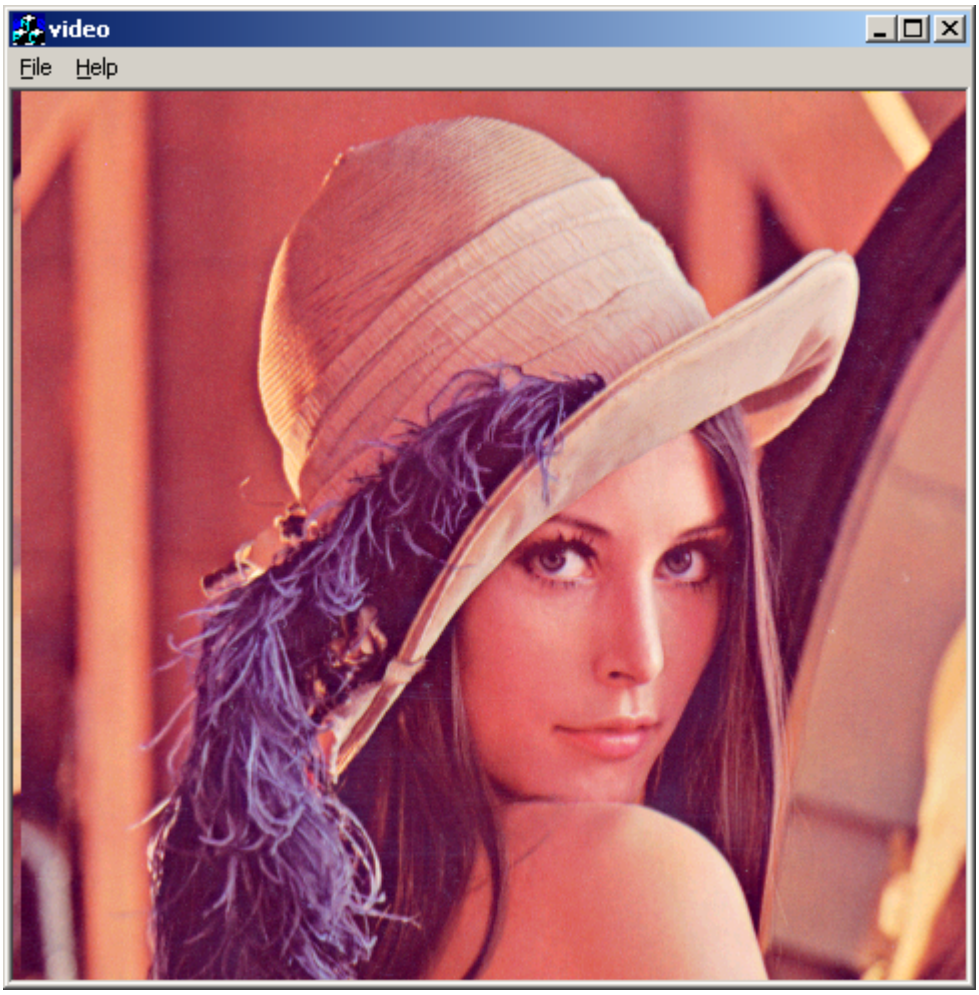
Además, hay que señalar que hacía falta **sincronizar** el programa "Host" con la FPGA: cuando el "Host" acababa de cargar la imagen en el banco 0, se lo indicaba a la FPGA enviándole algo mediante *PP1000WriteControl*. La FPGA estaba esperando mediante *PP1000ReadControl* a que el "Host" le indicara que había finalizado la

carga del banco 0 con la imagen. En ese momento, comienza todo el tratamiento de la FPGA. Cuando finaliza y ya ha dejado la imagen modificada en el banco 1, se lo indica al “Host” mediante *PP1000WriteStatus(0)*. El “Host” se había quedado esperando esta indicación mediante *PP1000ReadStatus*. En este momento, lee la imagen del banco 1 y la visualiza.

También hay que advertir que en Handel-C, cada instrucción tarda un ciclo de reloj en ejecutarse. La duración del ciclo de reloj será la duración de la instrucción más larga. En cuanto a los bancos de memoria, comentar que sólo se puede acceder a una dirección de memoria en cada banco por cada ciclo de reloj. Para más información referente a estos aspectos, consultar “Handel-C Language Reference”.

En esta primera prueba, no introdujimos filtro, la imagen se visualizaría tal cual era. La forma de saber si la prueba estaba funcionando era fácil, ya que la imagen visualizada era lo contenido en el banco 1 de la memoria de la FPGA y la imagen se cargaba inicialmente el banco 0, con lo que si se visualizaba correctamente era porque se había cargado de un banco a otro.

Como no había filtro VHDL, el *“.bit”* era generado a partir únicamente del EDIF generado por el DK2, que no utilizaba ningún filtro VHDL si no que simplemente se encargaba de leer por DMA los datos del banco 0 y llevarlos al banco 1. La imagen visualizada es:



Utilizamos como base para el programa “Host” un ejemplo de Celoxica disponible al instalar DK2 y que nos proporcionaba una interface gráfica y la visualización de la imagen. Así, nosotros sólo teníamos que introducir la parte en la que se enviaba la imagen a la memoria de la FPGA y se leía una vez que estaba modificada. De esta forma no nos teníamos que preocupar de crear código para visualizar la imagen por la pantalla. Los archivos que nos importan para el host son:

- *warp.h*: resultado de utilizar la utilidad gencfg sobre el *“.bit”* generado por Proyect Navigator.

- *lena.h*: contiene un vector de short llamado *ImageData* que es el que contiene la imagen original.

- *videoView.cpp:* nos interesa la parte del *constructor* (donde se envía la imagen original al banco de memoria 0 de la FPGA y se programa la misma a partir del archivo warp.h) y del método *ProcessImage* (donde se carga la imagen el banco 0 de memoria de la FPGA y se lee la imagen modificada del banco 1).

Uno de los problemas que tuvimos fue que al visualizar la imagen nos encontrábamos con líneas verticales que aparecían regularmente que parecían pertenecer a la imagen pero que no se correspondían con lo que se tenía que ver. Advertimos que aparecían regularmente, es decir, tenían todas el mismo ancho y éste coincidía con la línea vertical que sí se correspondía con lo que había que ver. Es decir, era como si tuviéramos una línea vertical que sí se veía bien y otra que no, una que sí y otra que no... Nos dimos cuenta que el problema era la forma de enviar los datos desde el "Host". En el constructor de videoView, al leer la imagen original desde el vector ImageData, se intercambiaban los 16 bits menos significativos por los 16 más significativos cada 32 bits, es decir, el short menos significativo con el más significativo de cada palabra de 32 bits. Así, al leer cada palabra de 32 bits del banco de memoria 0 en la FPGA, se había intercambiado el píxel menos significativo con el más significativo. Para corregirlo, le volvíamos a dar la vuelta a los píxeles al escribirlos en el banco 1, para que así se visualizaran correctamente. Posteriormente, nos dimos cuenta de que bastaba con no dar la vuelta en ninguno de los dos sitios, ya estaban bien alineados. Pero esta optimización no la hicimos hasta el proyecto del filtro de negativizado, en el que ya sabíamos antes de empezarlo que todo funcionaba correctamente incluso utilizando el filtro.

8.2.5. MODIFICAR LA IMAGEN CON UN FILTRO BLANCO Y NEGRO

Llegados a este punto, teníamos que filtrar la imagen original. Para ello, en el código Handel-C que se ejecutaría en la FPGA había que utilizar un filtro. Para ello, seguimos los pasos necesarios para utilizar un módulo VHDL (el filtro blanco y negro) en el código Handel-C, pasos que ya teníamos identificados por las pruebas anteriores.

El filtro (filtroBlancoNegro.vhd) está basado en el análogo del proyecto del año pasado ("Procesado de imágenes digitales sobre un sistema hardware dinámicamente reconfigurable"). En nuestro caso tiene tres señales de entrada (comienzo, reloj y datosEntrada, que será un vector que contenga los dos píxeles que se van a filtrar) y dos de salida (terminado y datosSalida, que es un vector con los dos píxeles filtrados).

Tuvimos un problema algo tonto al hacer esta prueba, pero que tardamos un poco en encontrarlo, debido a que los resultados que obteníamos eran algo extraños y nos despistaron bastante, haciendo que pensáramos en problemas bastante diferentes del que era: se nos había olvidado tener en cuenta la señal de reloj en el filtro. La señal de reloj puede ser referenciada en el código Handel-C como ***__clock***. Ejemplos de pruebas que realizamos fueron: poner varios filtros y que cada uno tratará determinados píxeles, poner un filtro que dejaba pasar, otro que filtraba en azul,... Además, dado que el PC donde funcionaba la transferencia DMA no se podía instalar el DK2 y no disponíamos de sus facilidades de depuración, tuvimos que añadir código para escribir el resultado en un fichero y visualizarlo con un programa propio.

A continuación mostramos el resultado de esta prueba (filtro blanco y negro de la imagen original que podemos ver en la prueba anterior):

8.2.6. MODIFICAR LA IMAGEN CON UN FILTRO DE NEGATIVIZADO

Dado que ya habíamos hecho funcionar un filtro, quisimos añadir otro, basado como el anterior en uno de los realizados en el proyecto del año pasado. Además se añadieron un par de optimizaciones.

En cuanto a las optimizaciones, fueron dos: quitar el intercambio de los dos píxeles de cada palabra de memoria en la FPGA y en el "Host" (mencionado anteriormente) y añadir una

instrucción Handel-C para paralelizar una parte del código y mejorar así el rendimiento.

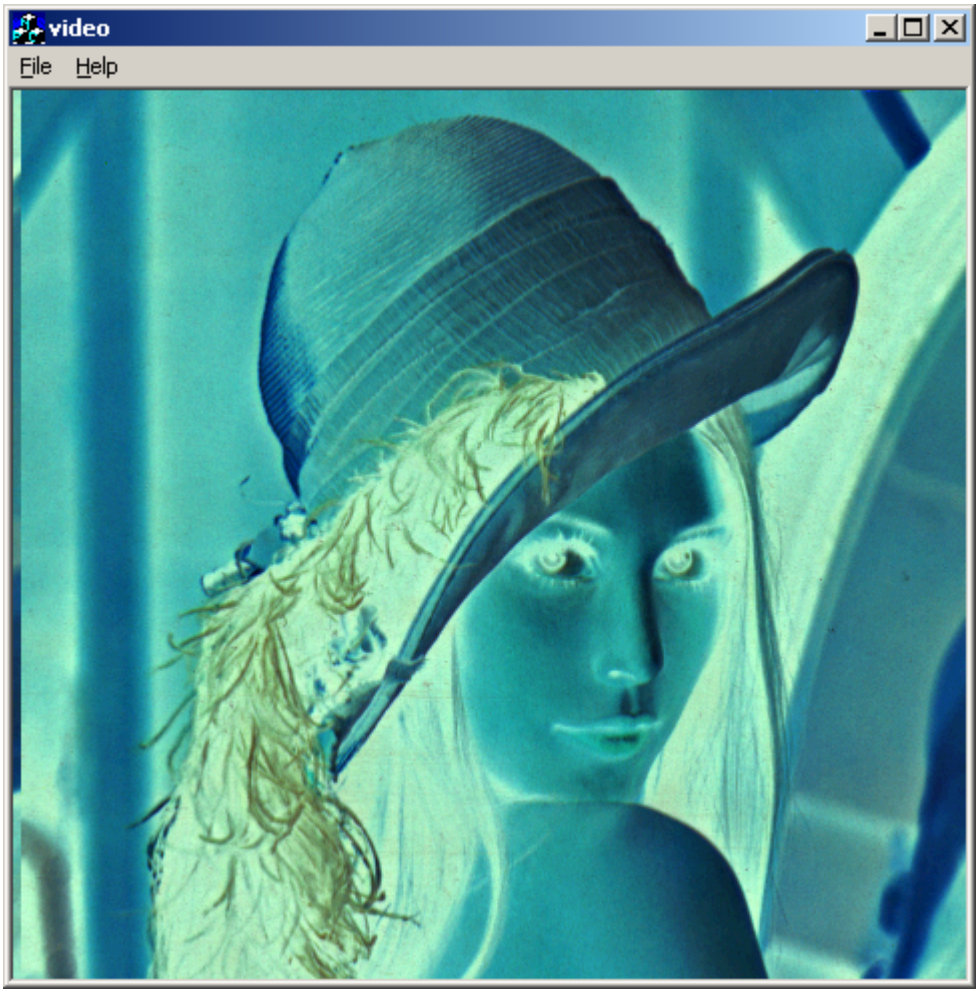
En el código del ejemplo de Celoxica en el que nos basamos para crear el “Host”, al coger los datos (la imagen) del vector de lena.h, los guardaba en un vector auxiliar intercambiando los dos short (16 bits) de cada 32 bits. Al parecer lo hacían por cuestiones de alineación de memoria. Por lo tanto, hasta ahora nosotros habíamos tenido que intercambiar también los píxeles al escribirlos en el banco 1 (en el que se escribían una vez que habían sido filtrados). Sin embargo nos dimos cuenta de que a nosotros no nos influía esa cuestión de alineación de memoria, ya que nosotros filtrábamos píxel a píxel y tal y como venían los datos no teníamos ningún problema. Por tanto, si quitábamos los dos intercambios todos los programas se seguirían comportando correctamente. Y efectivamente así fue.

Por otra parte, existe una palabra reservada en Handel-C, ***par***, que sirve para ejecutar instrucciones en paralelo. Este paralelismo es real, las dos instrucciones se ejecutan en el mismo instante de tiempo ya que al compilar con DK2 este código Handel-C “se genera” el hardware adecuado para que esto sea posible. En un bloque de instrucciones paralelas, todas las ramas que hayan acabado su ejecución esperan a que acabe la más lenta para continuar. Por tanto, la instrucción siguiente al bloque *par{...}* no se ejecutará hasta que todas las ramas del bloque *par{...}* hayan acabado. Si queremos incluir código secuencial dentro de un bloque *par{...}*, debemos encerrarlo entre {}. La sintaxis es la siguiente:

```
par
{
    instrucción_A;
    {
        bloque_de_instrucciones secuenciales;
    }
```

```
        instrucción_C;
        ...
}
```

Con esto mejorábamos el rendimiento, leyendo una palabra de 32 bits del banco 0 y escribiendo en el banco 1 la palabra que habíamos tratado en la iteración anterior, todo ello en el mismo ciclo.



8.3. PASOS PARA HACER UN FILTRADO

A continuación explicaremos los pasos que hay que seguir para utilizar código VHDL, en este caso un VHDL que filtre, “dentro” de código Handel-C y la estructura de directorios que hemos utilizado para conseguirlo.

En cuanto a los directorios, tendremos los siguientes:

- *VHDL*: en este directorio tendremos el código **VHDL** de la entidad que queramos utilizar en nuestro Handel-C, así como un proyecto Project Navigator con el que sintetizar dicha entidad y generar su **NGC**.

- *DK*: aquí es donde generaremos el proyecto **Handel-C** en el que queremos utilizar la entidad VHDL del directorio anterior. Al compilar este proyecto, obtendremos un **EDIF** para el código Handel-C.

- *Project Navigator*: generaremos un proyecto para el EDIF anterior y obtendremos el ***".bit"*** con el que programaremos la FPGA.

- *Imagen*: aquí copiaremos el *".bit"* generado en el directorio anterior y a partir suyo y mediante la herramienta *gencfg* generaremos un archivo **.H**, que será el que carguemos en la FPGA mediante el programa "Host".

- *Host*: en este directorio ubicaremos nuestro proyecto Visual C++, con el que programamos la aplicación con interface gráfica que interactúa con el usuario, envía la imagen (almacenada en el vector de lena.h) al banco 0 de memoria de la FPGA, espera a que la FPGA termine de procesarla, lee del banco 1 la imagen filtrada y la visualiza por pantalla.

Para realizar cualquiera de los pasos anteriores hay que seguir las siguientes instrucciones:

1. Diseñar en **VHDL** el filtro (.vhd) que queremos utilizar en nuestro código Handel-C.

2. Sintetizar dicho filtro con Project Navigator generando así un archivo **NGC** (.ngc). Es necesario deshabilitar en las opciones de Síntesis la opción “Add I/O buffers”.

3. Escribir el código **Handel-C** (.hcc) y compilarlo con DK2 generando un archivo **EDIF** (.edf).

4. Crear un proyecto Project Navigator a partir del EDIF anterior. Copiar el NGC del paso 2 al mismo directorio donde hayamos traído el EDIF, pero no añadir, con la opción correspondiente del Proyect Navigator, dicho NGC al proyecto. Sólo copiarlo al mismo directorio donde esté el EDIF.

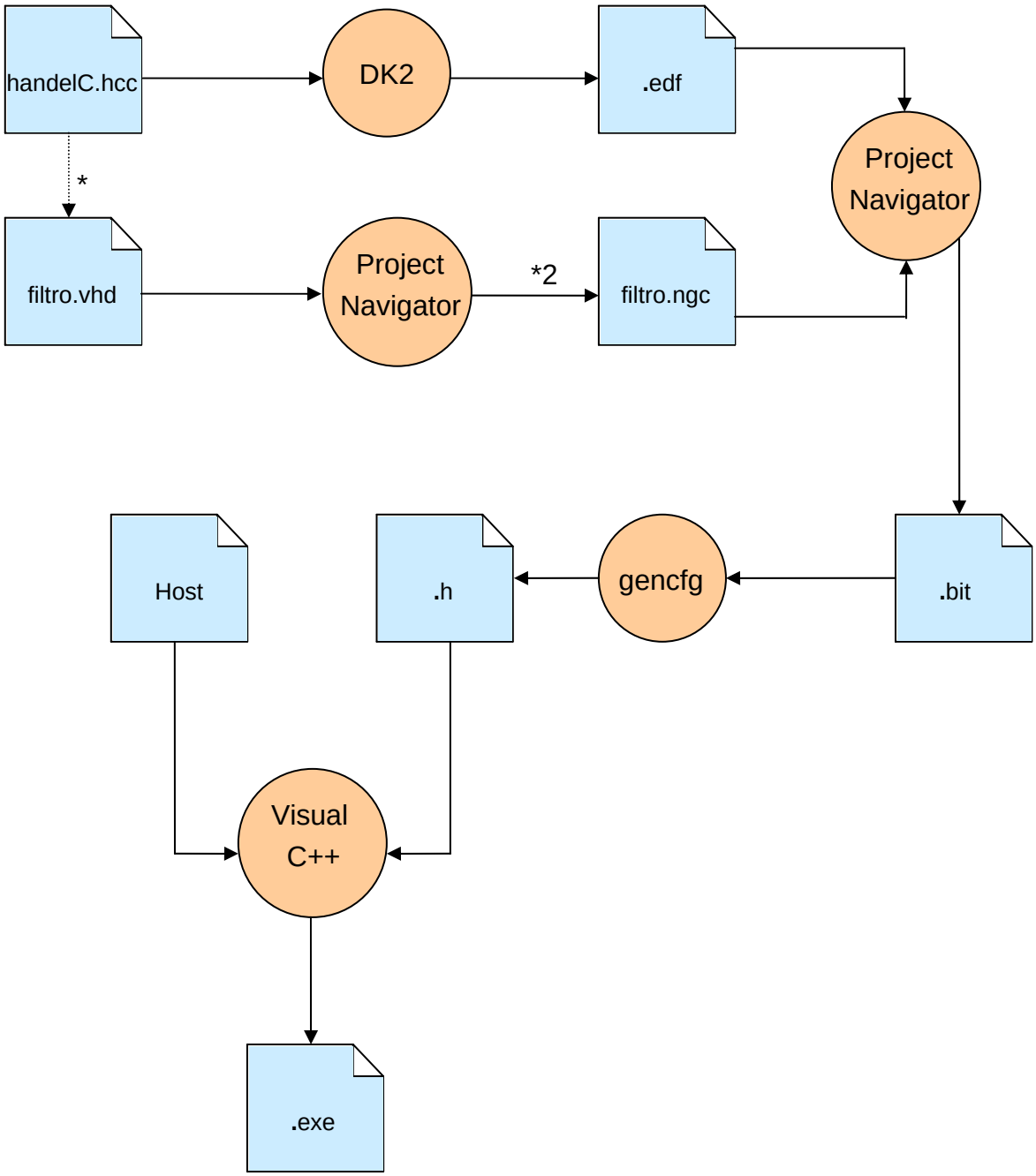
5. Sintetizar el proyecto anterior y generar el ***“.bit”***.

6. Copiar el *“.bit”* generado al directorio Imagen. Utilizar en dicho directorio la utilidad *gencfg* para generar warp**.h** a partir del *“.bit”* que acabamos de copiar.

7. Crear la aplicación “Host”. No olvidar incluir el archivo warp.h (o como lo queramos llamar: el archivo .h resultado de aplicar gencfg sobre el *“.bit”*). Para mantener dicho .h en el directorio Imagen y no tener que estarlo copiando, incluir en videoView**.cpp** (donde se hará la carga de la imagen a partir de un archivo .h, la carga de warp.h en la FPGA para programarla, y la comunicación con la misma y sus banco de memoria y la visualización de la imagen filtrada) la siguiente línea:

```
#include "../Imagen/warp.h"
```

8. Crear el **ejecutable** para la aplicación anterior.



*: define tantos *interface* como instancias quiere utilizar de la entidad VHDL.

*2: Sintetiza deshabilitando la opción “Add I/O Buffers”.

9. RECONFIGURACIÓN PARCIAL DINÁMICA DE LA FPGA

9.1. INTRODUCCIÓN

Una importante característica de la arquitectura Xilinx Virtex es la capacidad de reconfigurar una parte de la FPGA mientras que el resto del diseño permanece operacional. La reconfiguración parcial dinámica es útil para aplicaciones que requieren la carga de diferentes diseños en la misma área del dispositivo o flexibilidad para cambiar partes de un diseño sin tener que resetear o reconfigurar completamente el dispositivo entero. De esta forma, es posible reconfigurar ciertas áreas del dispositivo mientras otras áreas permanecen inalteradas y operacionales.

Hay dos formas distintas para llevar a cabo la reconfiguración parcial dinámica: reconfiguración parcial basada en diferencia (*Small bit manipulation* o *Difference-Based*) y reconfiguración parcial modular (*Module-Based*).

La reconfiguración parcial basada en diferencia consiste en hacer pequeños cambios en un diseño (como por ejemplo, cambiar una puerta OR por una puerta AND), cambios que serán normalmente realizados mediante el FPGA_Editor. A continuación, se creará un *".bit"* con sólo las diferencias entre el diseño original (el *".bit"* original) y el nuevo diseño con el cambio.

Por su parte, la reconfiguración parcial modular se utiliza cuando los cambios son algo mayores y no podemos estar realizándolos "a mano" con el FPGA_Editor. Normalmente tendremos una serie de módulos interrelacionados donde querremos cambiar uno de ellos para que cambie el comportamiento del sistema. Por tanto, querremos reconfigurar solamente la parte del diseño, el área, que contiene a dicho módulo, dejando inalterados al resto de módulos para que el

sistema permanezca operativo. Dado que éste es el método que utilizamos para reconfigurar nuestro diseño, lo explicaremos un poco más detenidamente en el siguiente apartado.

9.2. RECONFIGURACIÓN PARCIAL MODULAR

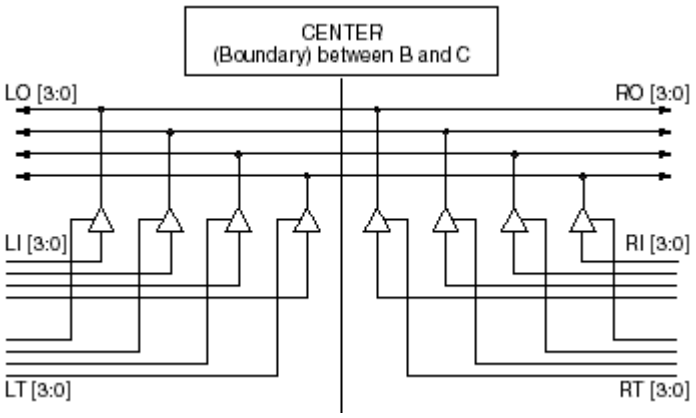
Como acabamos de explicar, cuando utilicemos esta técnica normalmente tendremos un conjunto de módulos interconectados entre sí y querremos reconfigurar uno de ellos. Aquí es donde debemos introducir un concepto importante en esta técnica de reconfiguración: las bus macro.

9.2.1. BUS MACRO

Mediante las bus macro conseguiremos comunicar los módulos reconfigurables con los módulos fijos, de forma que el sistema permanezca operativo durante la reconfiguración. Esto es así debido a que gracias a las bus macro podemos establecer rutas fijas de comunicación entre un módulo fijo y uno reconfigurable: asociaremos las señales que comunicar ambos módulos con un bus macro y así, aunque se reconfigure un módulo, cuando ya esté reconfigurado de nuevo va a seguir “conectando” esa señal a la bus macro, la cual permanece siempre inalterable, no la afectan las reconfiguraciones de los módulos.

La implementación de las bus macro utiliza 8 buffers triestado (TBUFs). La disposición de los TBUFs es de 4 columnas de TBUFs en el lado del módulo de la derecha y otras 4 en el lado del módulo de la izquierda. De esta forma cada bus macro proporciona 4 líneas de comunicación, cada una de las cuales puede ser utilizada para enviar un bit de derecha a izquierda o de izquierda a derecha. Por dejarlo más claro, cada bus macro nos proporciona 4 bits de comunicación entre dos módulos, donde cada uno de estos cuatro

bits puede ser utilizado para “viajar” de derecha a izquierda o de izquierda a derecha pero nunca en ambos, no es bidireccional. Una vez que se decida el sentido, quedará fijado para siempre. A continuación se muestra una figura ilustrativa.



- LO (RO) son las líneas de las que puede leer los datos el módulo de la izquierda (derecha) provenientes del de la derecha (izquierda).

- LI (RI) son las líneas que contienen los datos que se quieren enviar al módulo de la derecha (izquierda).

- LT (RT) son las líneas que controlan si la LI (RI) asociada será enviada al módulo de la derecha (izquierda) o si será desechada, bien porque no se vaya a usar ese bit (puede que lo que queramos comunicar entre los módulos sean menos de los 4 bits proporcionados por la bus macro) o bien porque la línea LO (RO) asociada vaya a contener datos enviados desde el módulo de la derecha (izquierda).

Las líneas de control (LTs y RTs) son activas a baja.

En cuanto a cómo obtener los archivos que implementan las bus macro para incluirlas en nuestros proyectos, nosotros lo

obtuvimos del ejemplo que indican en el XAPP290(V1.1). Podemos descargar este ejemplo en

http://www.xilinx.com/pub/applications/xapp/xapp290.zip .

Al descomprimirlo, podemos observar que tenemos varios archivos ".nmc", que son los que nos interesan. Cada uno sirve para un modelo de Virtex:

- bm_4b.nmc: bus macro para las Virtex. Es el que utilizamos nosotros.
- bm_4b_ve.nmc: bus macro para las Virtex-E.
- bm_4b_v2.nmc: bus macro para las Virtex-II.

Para declarar la bus macro y poder utilizarla en nuestra arquitectura, habrá que incluir en la misma algo que se ajuste al siguiente esquema:

```
COMPONENT bm_4b
PORT(
        LI : in  std_logic_vector (3 downto 0);
        LT : in  std_logic_vector (3 downto 0);
        RI : in  std_logic_vector (3 downto 0);
        RT : in  std_logic_vector (3 downto 0);
        O  : out std_logic_vector (3 downto 0)
        );
END COMPONENT;
```

Por último ya sólo quedaría indicar qué bits de la bus macro van a comunicar el módulo de la derecha con el de la izquierda y cuáles el de la derecha con el de la izquierda. A continuación mostramos un ejemplo de instanciación para la bus macro, donde los bits O(3) y O(2) comunicarán la izquierda con la derecha y O(1) y O(0) comunicarán la derecha con la izquierda. Recordemos que las líneas de control LT y RT con activas a baja.

```
bm_4b PORT MAP(
        LI(3) => dato_de_la_izq,
        LI(2) => dato_de_la_izq,
        LI(1) => GND,  -- dato desechado
        LI(0) => GND,
         --
        LT(3) => GND,  --Activa
        LT(2) => GND,
        LT(1) => VCC,  --Desactiva
        LT(0) => VCC,
         --
        RI(3) => GND,  -- dato desechado
        RI(2) => GND,
        RI(1) => dato_de_la_dch,
        RI(0) => dato_de_la_dch,
         --
        RT(3) => VCC,  --Desactiva
        RT(2) => VCC,
        RT(1) => GND,  --Activa
        RT(0) => GND,
        --
        O(3) => hacia_la_dch,
        O(2) => hacia_la_dch,
        O(1) => hacia_la_izq,
        O(0) => hacia_la_izq,
        );
```

9.2.2. PASOS GENERALES EN UN PROYECTO

A la hora de reconfigurar parcialmente la FPGA seguimos dos metodologías para obtener este resultado, pero ninguna de las dos funcionó completamente. Cada una tuvo un punto en el que el resultado no era el deseado. Y juntando lo que funcionaba correctamente de cada una tampoco conseguimos que la reconfiguración fuera satisfactoria.

Las dos maneras por las que intentamos conseguir la reconfiguración parcial son las siguientes:

1. Utilizando Proyect Navigator y configurándolo todo mediante su interface gráfica. De esta forma creímos en un principio que habíamos conseguido la reconfiguración parcial: cargábamos el *".bit"* de uno de los tops y a continuación cargábamos el *".bit"* del top que utilizaba el otro módulo reconfigurable. Ahora está claro que eso no era reconfiguración. La segunda que reprogramábamos la FPGA lo estábamos haciendo completamente con otro *".bit"* (por mucho que creyéramos que no era así al haber puesto las opciones de reconfiguración al crearlo), ya que utilizábamos el *".bit"* del segundo top cuando lo que había que hacer era cargar sólo el *".bit"* del otro módulo reconfigurable (con lo que el resto del diseño, el resto del top inicialmente cargado, no sería programado de nuevo, como pasaba en nuestro caso al cargar el *".bit"* del otro top).

2. Siguiendo toda la estructura y scripts del ejemplo que descargamos de http://www.xilinx.com/pub/applications/xapp/xapp290.zip . De esta forma conseguíamos crear *".bit"* de los módulos reconfigurables, que era lo que no conseguíamos de la manera anterior, pero lo que aquí no se generaba correctamente eran los *".bit"* de los top, con lo que de esta segunda manera no pudimos ni siquiera hacer una prueba.

Viendo que de la primera forma conseguíamos un top que funcionaba (tanto que de hecho, como he dicho antes, llegamos a pensar que lo habíamos conseguido) pero no podíamos crear los *".bit"* de los módulos reconfigurables, y de la segunda forma era esto último lo que sí conseguíamos y el *".bit"* de los top lo que no éramos

capaces de generar, decidimos hacer una prueba "híbrido". Generamos el *".bit"* del top de la configuración inicial de la primera manera (Proyect Navigator) y el *".bit"* del módulo reconfigurable con el que queríamos reprogramar la FPGA lo generábamos de la segunda forma (con los scripts). Así tampoco funcionó, ya que los leds que se encendían al reconfigurar eran siempre los mismos y no se correspondían con la respuesta que tenían que dar en función de la entrada que le pasábamos.

9.2.2.1. PASOS GENERALES UTILIZANDO PROJECT NAVIGATOR

1. Creamos un ".ucf" por cada combinación posible de diseño completo, es decir, por cada "top". Es posible (como en nuestro caso) que este ".ucf" sea el mismo en los dos diseños completos. De hecho, sólo serán distintos si los módulos reconfigurables de cada uno añaden alguna restricción diferente.

2. Creamos un proyecto con Project Navigator para cada módulo y copiamos en dicho proyecto el ".ucf" anterior. Sintetizamos cada uno de estos proyectos, teniendo deshabilitada la opción *"Add I/O Buffers"*. Se explicará más adelante.

3. Creamos un proyecto "top" de la jerarquía por cada combinación posible de módulos fijos y reconfigurables y copiamos en dicho proyecto el ".ucf" anterior y los ".ngc" resultado de la síntesis de los proyectos de cada módulo (punto anterior) que pertenezca a esta jerarquía. Sintetizamos estos proyectos "top", pero esta vez con la opción *"Add I/O Buffers"* habilitada. Creamos sus ".bit", utilizando las

opciones *-g ActiveReconfig:Yes* y *-g Persist:Yes* de BitGen.

4. Comprobar viendo el Map report y con el floorplanner que todo ha ido bien, que se han ubicado los módulos en las áreas que indicamos.

Más adelante se explicará qué más hay que configurar y hacer, utilizando nuestro proyecto como ejemplo para los pasos completos a seguir.

Señalemos que en el paso 3 realmente sólo haría falta realizar ese paso con el "top" elegido como configuración inicial. La idea de hacerlo para los demás es poder cargar de forma normal (sin reconfiguración parcial) sus ".bit" y comprobar que funcionan de por sí.

9.2.2.2. PASOS GENERALES UTILIZANDO SCRITPS

1. Escribir y sintetizar el código VHDL de los módulos y de los top. Así, generamos el EDIF (o el NGC) de cada módulo y cada top.

2. Fase Initial Budgeting. En ella se crea el UCF que describe el área de cada módulo, las restricciones de ubicación para las instancias de bus macro y para la lógica del top-level. Por otra parte, se genera un NGO y un NGD para el top-level.

3. Fase Active Module Impementation. Aquí comienza la implementación ("place and route") de todos los módulos (tanto fijos como reconfigurables), generando un *".bit"* para todos los módulos reconfigurables (dentro del contexto de la lógica y las restricciones del top-level) y el NGD y el PIM

(Physically Implemented Module) de cada módulo (tanto fijo como reconfigurable).

4. Fase Final Assembly. En este punto se combina cada uno de los módulos individuales en un diseño completo para la FPGA, generando un *".bit"* para la configuración inicial.

5. Comprobar el diseño utilizando el FPGA_Editor.

6. Programar la FPGA con el *".bit"* de la configuración inicial.

7. Reprogramar el módulo reconfigurable con el *".bit"* parcial.

En nuestro caso, el punto 6 y el 7 se traducen en crear un programa "Host" que es el que realiza ambos pasos. Los puntos 2, 3 y 4 serán en los que utilizaremos scripts para llevarlos a cabo.

9.2.3. CREANDO LOS ".UCF"

En los ".ucf" debemos definir áreas de la FPGA y establecer en qué área se ubicará cada módulo. Además hay que fijar también dónde estará cada bus macro. Finalmente, hay que indicar qué módulos serán reconfigurables.

Para definir un área, introduciremos líneas que se ajusten al siguiente esquema:

```
AREA_GROUP "<nombre_area>" RANGE = CLB_R20C12:CLB_R35C35;
AREA_GROUP "<nombre_area>" RANGE = TBUF_R20C12:TBUF_R35C34;
AREA_GROUP "<nombre_area>" MODE = RECONFIG;
INST "<instancia_modulo>" AREA_GROUP = "<nombre_area>" ;
```

En el rango de los CLB hay que recordar que la primera columna debe ser múltiplo de 4 y que la anchura debe ser también múltiplo de 4 y mayor que 1.

La línea *AREA_GROUP "<nombre_area>" MODE = RECONFIG;* sólo hace falta para las áreas de los módulos reconfigurables.

Para definir la localización específica de una bus macro, introduciremos líneas que se ajusten al siguiente esquema:

```
INST "instancia_bus_macro" LOC = "TBUF_R20C12.0";
```

Recordemos que las bus macro ocupan 8 columnas y que tiene que estar justo en medio del módulo fijo y el reconfigurable a los que comunica, con lo que tendrá que tener 4 columnas en uno y otras 4 en el otro.

Hay que señalar que los ejemplos anteriores siguen la sintaxis para las placas Virtex, en la que se indica la fila con la “R” y la columna con la “C” y se especifica primero la fila y después la columna. Con las Virtex-II, las filas se denotan con “Y” y las columnas con “X” y se especifica primero la columna y después la fila. Por ejemplo, para una Virtex (de hecho, para todas las placas de la familia excepto para la Virtex-II) pondríamos TBUF_R2C4 para indicar la fila 2 y columna 4, mientras que para una Virtex-II pondríamos TBUF_X4Y2.

9.2.4. CREANDO LOS “.BIT”

Cuando queramos hacer reconfiguración parcial necesitaremos un “top” para la configuración inicial y otros tantos “top” por cada reconfiguración distinta de la configuración inicial que queramos tener. Así, en nuestro proyecto tendremos un “.bit” para el “top” que consideramos que será el correspondiente a la

configuración inicial y otro ".bit" para el segundo "módulo reconfigurable" con el que reconfiguramos la FPGA para que se comporte de modo diferente. Para generar estos ".bit" hay que especificar las siguientes opciones:

- *"-g ActiveReconfig:Yes"*: con ella indicamos que el dispositivo debe permanecer operacional mientras que se esté cargando el nuevo ".bit" parcial.

- *"-g Persist:Yes"*: esta opción es necesaria cuando la reconfiguración parcial la estamos haciendo bajo el modo SelectMAP. Permite que los pins de SelectMAP se mantengan persistentes después de que el dispositivo sea configurado, lo cual permite que la interface SelectMAP sea utilizada para la reconfiguración.

9.2.5. CREANDO UN "HOST"

Para comunicarnos con la FPGA, darle datos y que nos devuelva los resultados, necesitamos crear un programa que se ejecutará en la CPU del PC. A este programa es al que nos referimos en todos los puntos de esta memoria como "HOST".

En el caso de la reconfiguración parcial, este programa debe, básicamente, cargar la configuración inicial para la FPGA mediante el ".bit" resultante del proyecto "top" que hayamos elegido como configuración inicial y posteriormente y después de realizar las operaciones que consideremos oportunas en función de nuestros objetivos, cargar el ".bit" del módulo reconfigurable que deseemos utilizar en ese momento. Podremos realizar este último paso tantas veces cómo queramos, es decir, podremos reconfigurar la FPGA un número ilimitado de veces si ese es nuestro deseo.

En cuanto a la forma de cargar los “.bit”, lo normal es cargar los “.bit” que van a reconfigurar al inicial mediante transferencias DMA. El “.bit” inicial se puede cargar de ésta misma forma o bien con la primitiva *PP1000ConfigureFromFile(Handle, "nombre.bit")*, que es lo que hemos hecho nosotros. La manera de cargar por DMA durante la reconfiguración los “.bit” de los módulos reconfigurables es la siguiente:

1. Generar un archivo “.h” a partir del “.bit” que va a producir la reconfiguración. Para esto utilizamos la utilidad *“gencfg”*, ejecutando la siguiente orden en una ventana de comandos:

   - `gencfg nombre.bit resultado.h`

2. Hacer un “include” en el Host (en el “.c”) del archivo “.h” resultante de la acción anterior. En dicho “.h” se habrá creado un *array* cuyo contenido es el “.bit” a cargar. El nombre de este *array* le utilizaremos en la instrucción de transferencia DMA.

3. Hacer la reconfiguración parcial cargando mediante DMA el “.bit”. Para ello, hay que transferir desde el Host hacia la FPGA el *array* mencionado previamente y contenido en el “.h” y que “contiene” a fin de cuentas el “.bit”. Hay que señalar que la *longitud* la tenemos también generada automáticamente en el “.h”. Las líneas en cuestión para esta transferencia DMA serían de la forma:

```
PP1000SetupSelectMapChannel(Handle, array_del_punto_h,
longitud_del_array, PP1000_PCI2LOCAL, &canal);
//Devuelve “canal”

PP1000DoDMA(canal);
```

9.3. FUNCIONAMIENTO DE NUESTRO PROYECTO DE RECONFIGURACIÓN PARCIAL

El proyecto de Reconfiguración parcial que hemos hecho consta de 3 módulos bien diferenciados (izquierda, reconfigurable y derecha), de los cuales dos de ellos (izquierda y derecha) son fijos y uno (reconfigurable) es el reconfigurable como su propio nombre indica.

Modulo Izquierdo: El usuario introduce un número por teclado y se le pasa al módulo izquierdo en dos líneas de 4 bits, éste módulo suma 5 a los 4 bits menos significativos y 2 a los más significativos. Al igual que a las entradas tiene dos salidas de 4 bits que van hacia las entradas del módulo reconfigurable.

Módulo Reconfigurable: Le llegan las dos líneas de 4 bits del módulo izquierdo, si no se ha hecho la reconfiguración parcial todavía, el módulo sumará ambas entradas, y si ya se ha hecho la reconfiguración parcial las restará. Tiene una única salida, que son los 4 bits menos significativos resultantes de la operación realizada (suma ó resta) que van hacia la entrada del módulo derecho.

Módulo Derecha: Le llegan los 4 bits de la salida del módulo reconfigurable, éste módulo suma 3 a los bits de su entrada y la salida la lleva a los led's de la placa.

Diagrama de módulos: En este diagrama se puede observar claramente el comportamiento del nuestro proyecto de reconfiguración parcial.

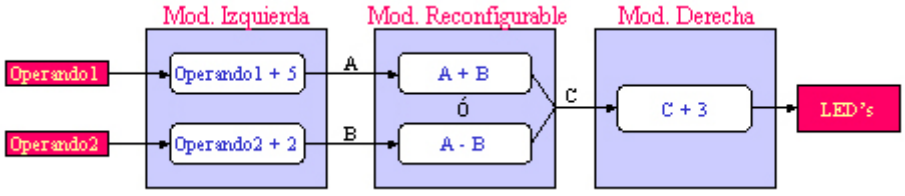


Tabla de entradas/salidas: En esta tabla se muestran las salidas que se tienen que dar frente determinadas entradas.

Teclado	Entradas		Salida con ModReconfigurable		Salida con ModReconfigurable2	
Número	Operando2	Operando1	(Op1+5)+(Op2+2)+3	Led's	(Op1+5)-(Op2+2)+3	Led's
0	0	0	10	1010	6	0100
1	0	1	11	1011	7	0111
2	0	2	12	1100	8	1000
3	0	3	13	1101	9	1001
4	0	4	14	1110	10	1010
5	0	5	15	1111	11	1011
6	0	6	16	0000	12	1100
7	0	7	17	0001	13	1101
8	0	8	18	0010	14	1110
9	0	9	19	0011	15	1111
10	0	10	20	0100	16	0000

Esta prueba se puede encontrar en nuestra carpeta "Proyectos/ReconfiguracionParcial" y su código fuente en el apartado 7.4.

9.4. PASOS PARA HACER UNA RECONFIGURACIÓN PARCIAL (MODO PROJECT NAVIGATOR)

Para que todo quede lo más claro posible vamos a ir mostrando todos los pasos necesarios para crear un proyecto de reconfiguración parcial basándonos en el que nosotros hemos hecho.

Debido a la dificultad del proyecto hemos dividido este apartado en varios puntos con el objetivo de que la explicación quede lo más estructurada posible. Para construir un proyecto de reconfiguración parcial se aconseja que se sigan secuencialmente los pasos siguientes:

9.4.1. CREAR ARCHIVOS FUENTE

Lo primero que tenemos que hacer es crear los archivos ".vhd", ".ucf" y ".c" necesarios para el funcionamiento del proyecto, en nuestro caso los archivos son los siguientes:

- TopJererquia.vhd
- TopJerarquia2.vhd
- ModDerecha.vhd
- ModReconfigurable.vhd
- ModReconfigurable2.vhd
- ModIzquierda.vhd
- TopJerarquia.ucf
- HOST.c

9.4.2. CREAR LA ESTRUCTURA DE DIRECTORIOS PARA EL PROYECTO

Es aconsejable hacer una estructura de directorios como la que se muestra a continuación:

En la carpeta *"EJECUTABLE"* metemos el archivo ejecutable del Host *"HOST.exe"* más los archivos que sean imprescindibles para su ejecución, que en nuestro caso es la primera imagen que se carga en la fpga "topjerarquia.bit".

En la carpeta *"HOST" metemos el proyecto en Visual C++ del Host.*

En la carpeta "*Imagen*" se encuentra el archivo ejecutable *"gencfg.exe"* y "*Ejecutar gencfg.bat"* que es el que se encarga de crear el archivo "*parcial.h*" a partir del *"topjerarquia2.bit".* El archivo *"parcial.h"* debe estar en esta ubicación cuando se cree el archivo ejecutable "*HOST.exe"* con el Visual C++ ya que éste está configurado para que lo lea de esta ubicación.

En el resto de carpetas se meten los proyectos creados con el Project Navigator para cada uno de sus correspondientes módulos.

Ahora metemos cada archivo fuente en la carpeta que le corresponde:

- TopJererquia.vhd → top
- TopJerarquia2.vhd → top2
- ModDerecha.vhd → ModuloDerecha
- ModReconfigurable.vhd → ModuloReconfigurable
- ModReconfigurable2.vhd → ModuloReconfigurable2
- ModIzquierda.vhd → ModuloIzquierda
- TopJerarquia.ucf → En todas las carpetas anteriores
- HOST.c → HOST

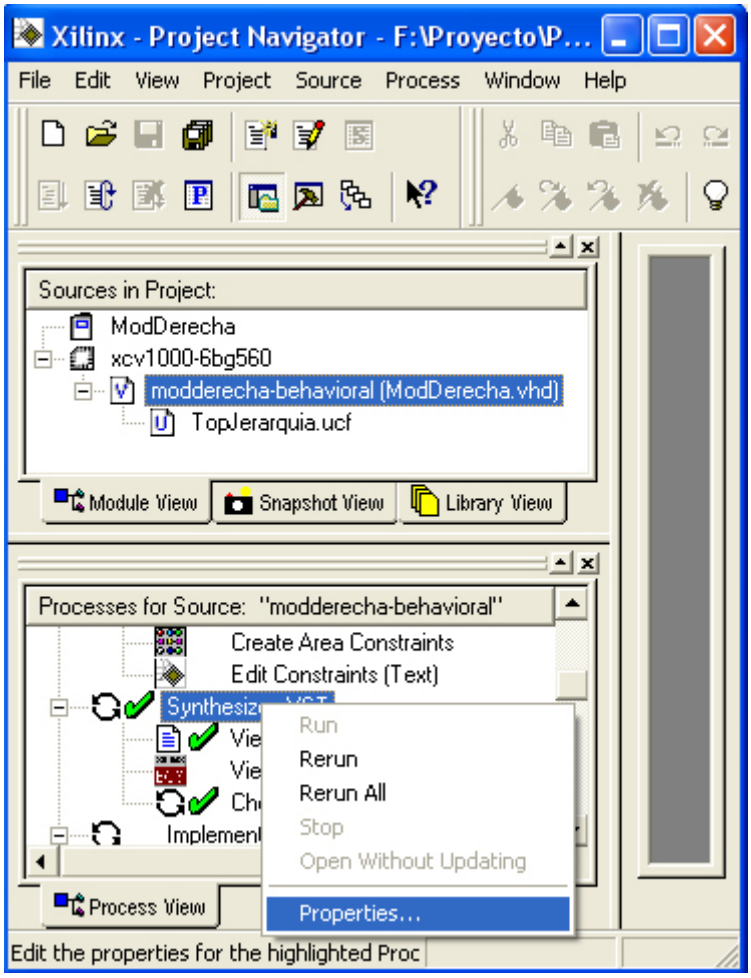
A continuación metemos el archivo *"bm_4b.nmc"* en la carpeta "*top*" y "*top2*", (este archivo es necesario para que funcionen los Bus Macro que se utilizarán más adelante).

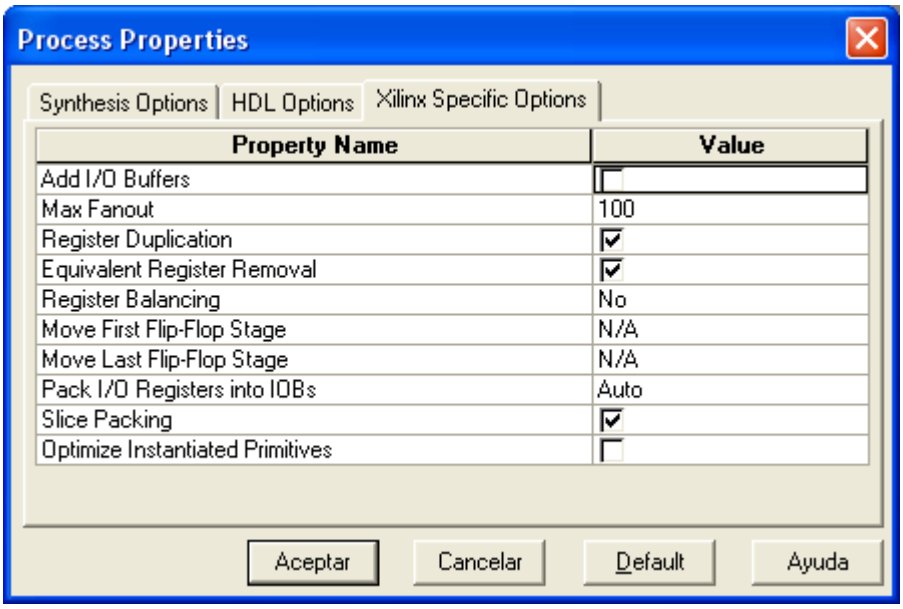
9.4.3. SINTETIZAR LOS MÓDULOS INTERNOS (NO TOP)

Las siguientes instrucciones se tienen que hacer para todos los módulos que no sean "*top*", en nuestro caso son ModDerecha, ModReconfigurable, ModReconfigurable2 y ModIzqierda.

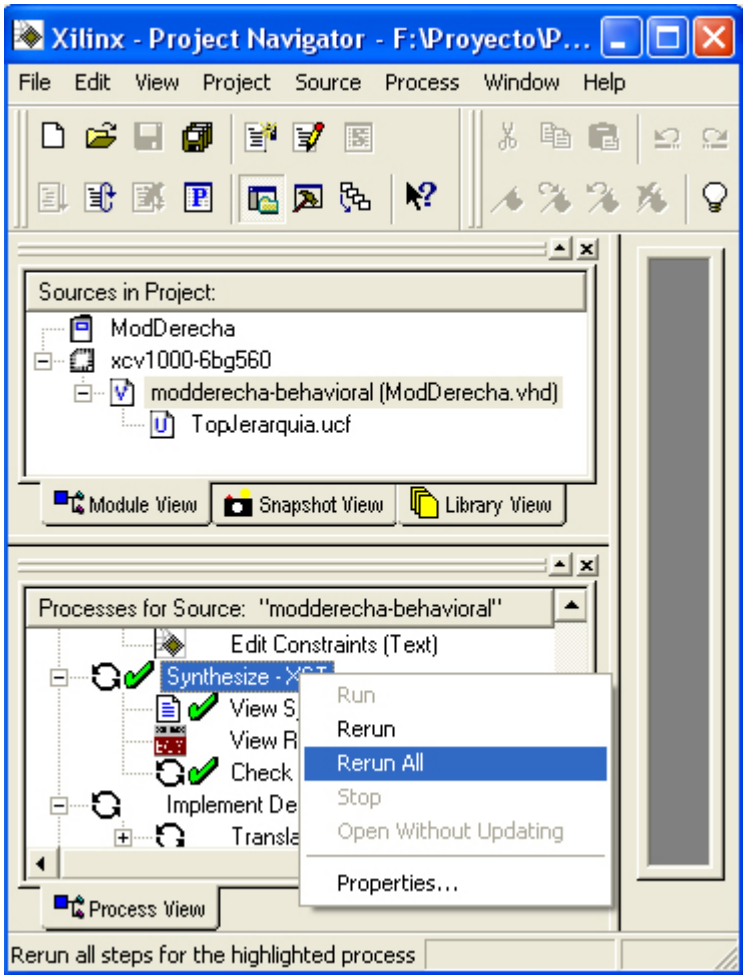
Se crea un proyecto con la herramienta Project Navigator en la que se añaden los archivos "*.vhd*" y "*.ucf*" correspondiente. Los pasos para configurar la herramienta están explicados en el apartado 4.4.

Una vez que ya se ha creado el proyecto y se han añadido los archivos correspondientes hay que eliminar la opción de añadir los *"I/O Buffers"* en las propiedades de "*Synthesize*" como se muestra a continuación:





A continuación ya podemos sintetizar pinchando con el botón derecho sobre "*Synthesize*" y haciendo click en *"Rerun all"* como se muestra a continuación.



Una vez sintetizados todos los proyectos hay que coger los archivos *".ngc"* creados cada uno en la raíz de su proyecto y copiarlos en sus respectivas carpetas top ó top2 como se muestra a continuación:

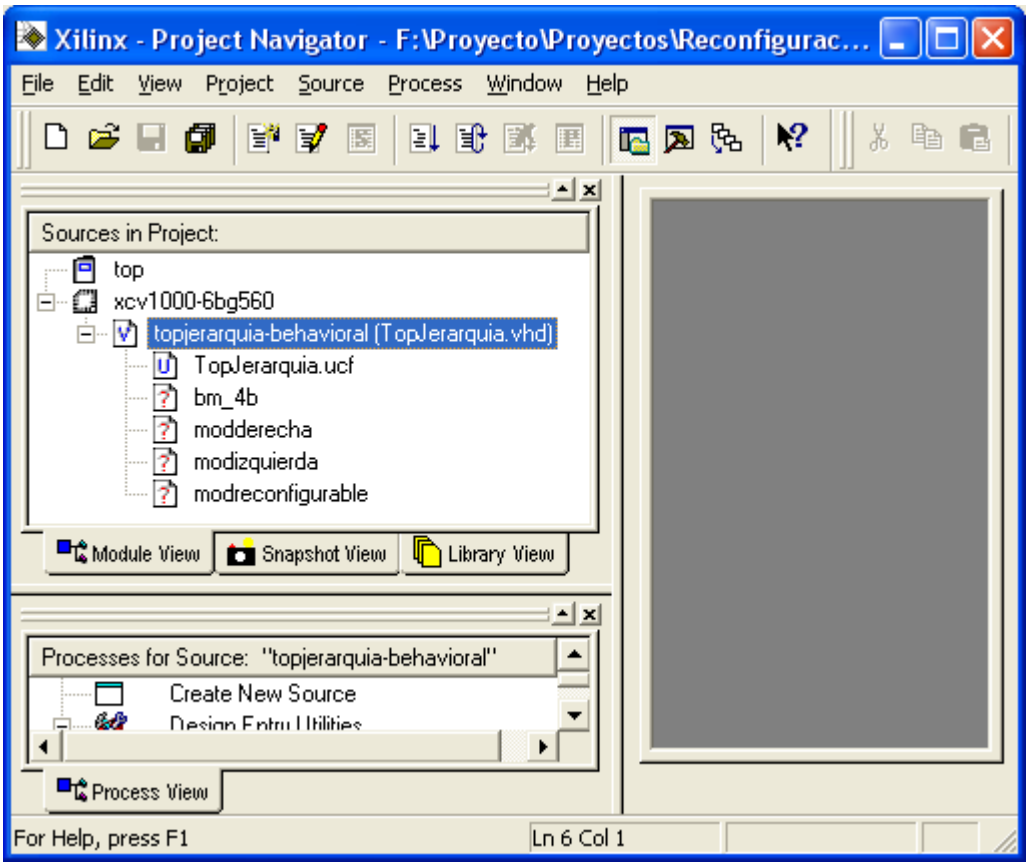
- modderecha.ngc → top y top2
- modreconfigurable.ngc → top
- modreconfigurable2.ngc → top2
- modizquierda.ngc → top y top2

9.4.4. CREAR LOS ARCHIVOS "*.BIT*" DE LOS TOP

Las siguientes instrucciones se tienen que hacer para todos los módulos que sean "*top*", en nuestro caso son "*top*" y "*top2*".

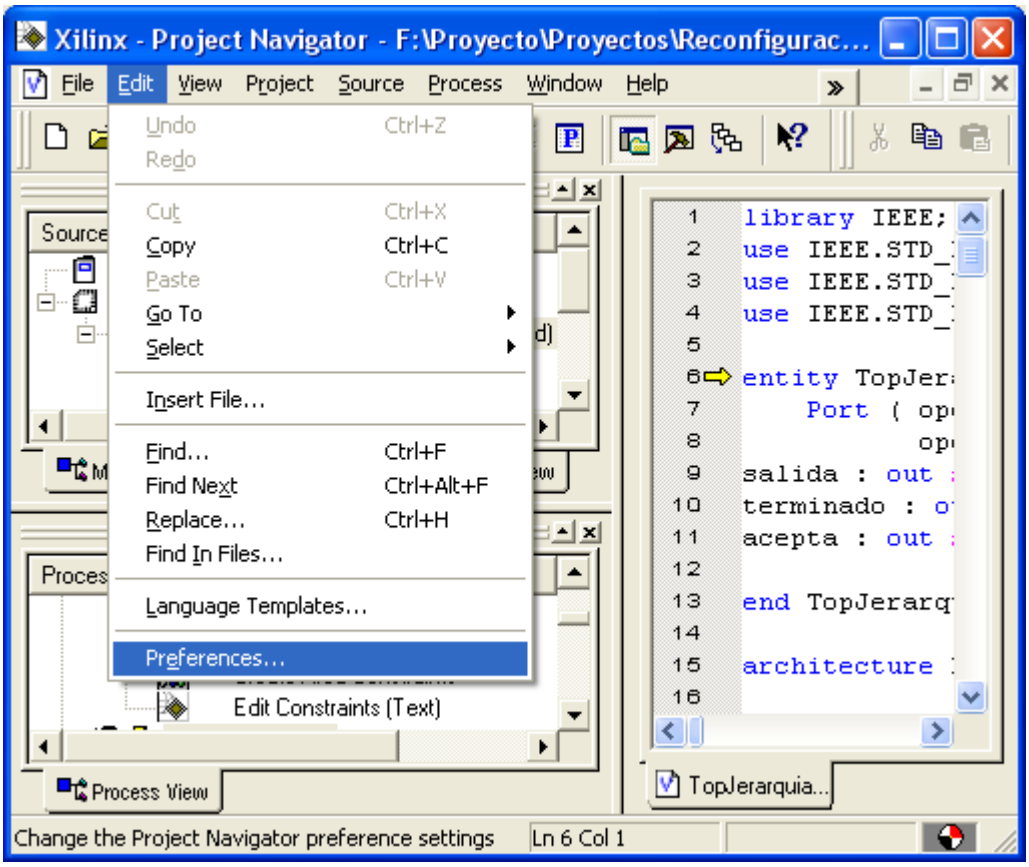
Se crea un proyecto con la herramienta Project Navigator en la que se añaden los archivos ".*vhd*" y ".*ucf*" correspondientes. Los pasos para configurar la herramienta están explicados en el apartado 4.4.

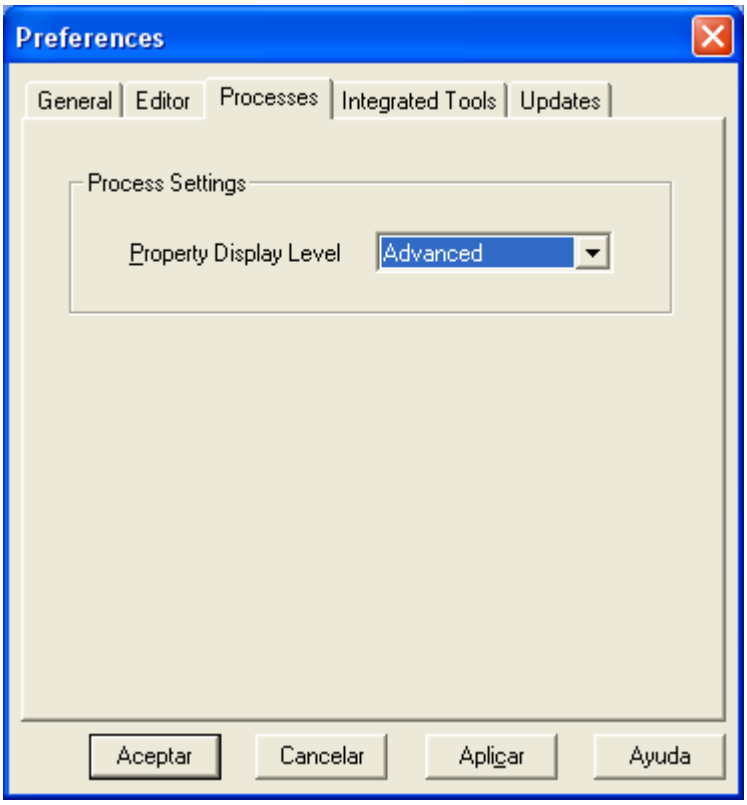
Una vez que ya se ha creado el proyecto y se han añadido los archivos correspondientes, la herramienta mostrará unas interrogaciones por cada módulo al que se haga referencia en el top; esto no es un problema ya que hemos copiado anteriormente los archivos ".*ngc*" de los módulos dentro de la raíz del proyecto.



La razón por la que copiamos en la raíz del proyecto los archivos "*.ngc*" y no añadimos en este proyecto los archivos "*.vhd*" de los módulos internos a los que se hace referencia con las interrogaciones es porque éstos tienen que ser sintetizados con la opción de *"I/O Buffers"* desactivada y los módulos top la tienen que tener activada, por lo tanto no se pueden sintetizar en el mismo proyecto ya que la herramienta no permite tener opciones de síntesis diferentes para cada módulo.

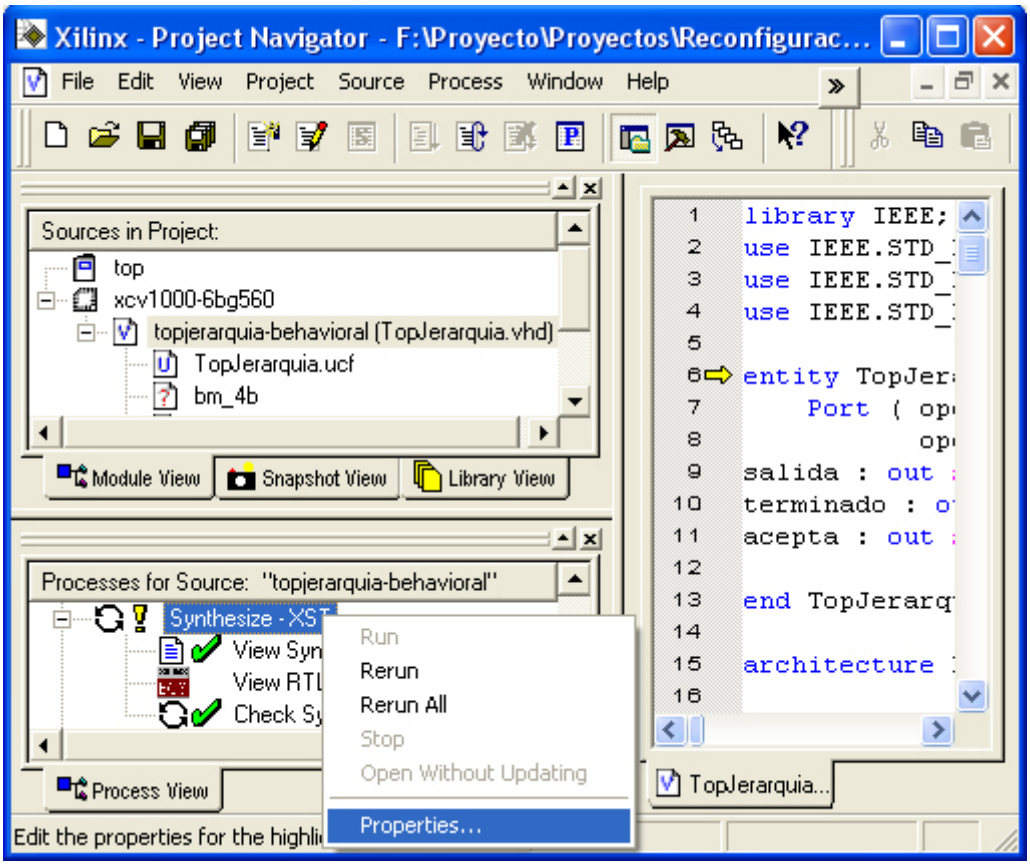
A continuación hay que seguir configurando la herramienta. Lo primero que haremos es poner en "*avanzado*" los procesos de la herramienta, esto nos permite tener todas las posibilidades necesarias para configurar la herramienta. Se hace de la siguiente manera:

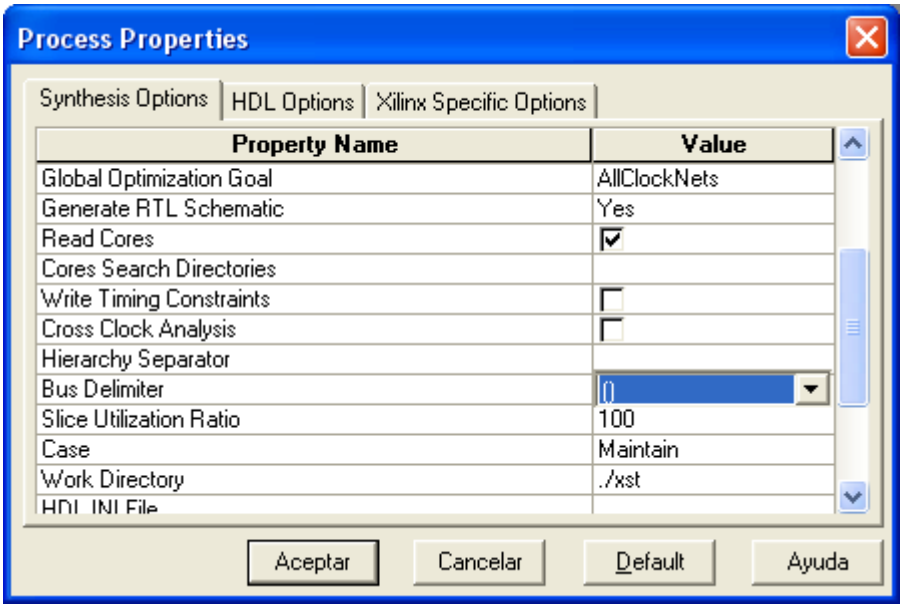




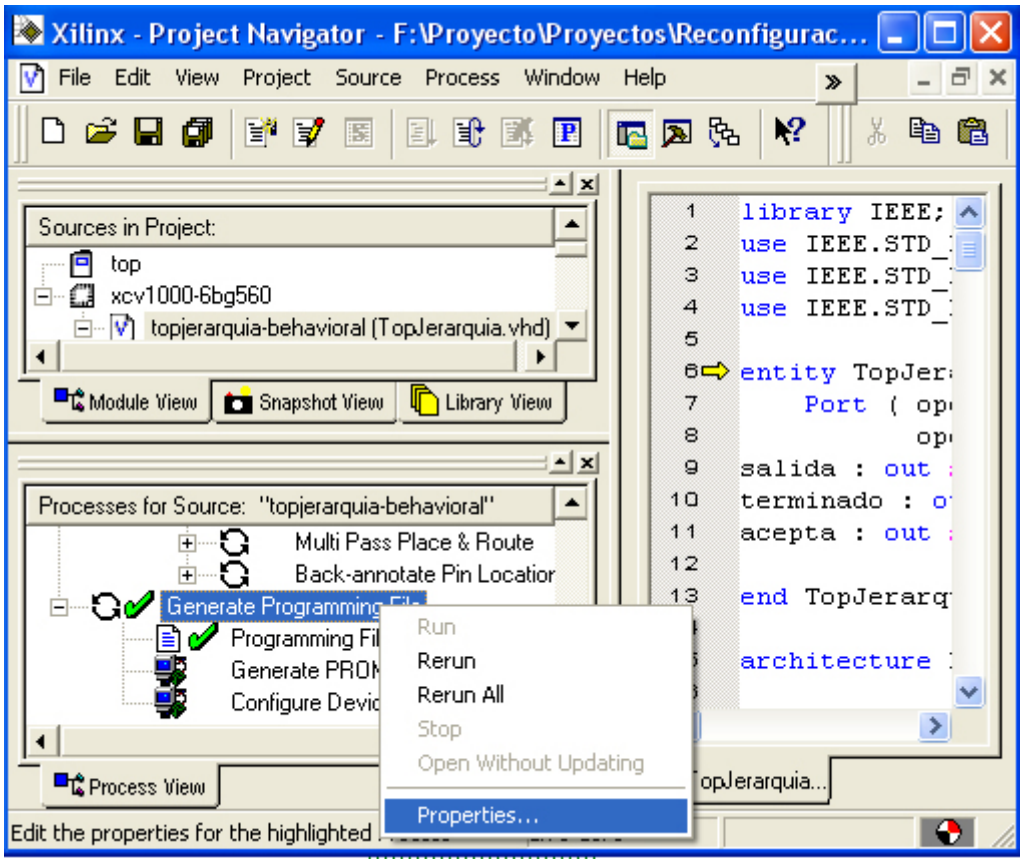
El siguiente paso en la configuración de la herramienta es poner el *"Bus Delimiter"* con la opción "()" dentro de las propiedades

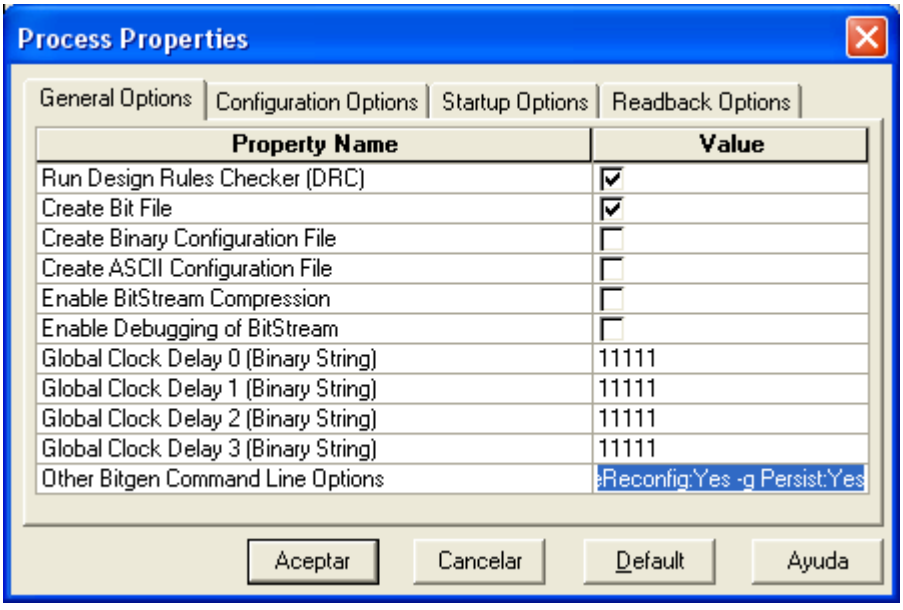
de *"Synthesize"* ya que es así como nombramos a los buses en el archivo *"TopJerarquia.ucf"*. Esto se hace como se muestra en las imágenes:





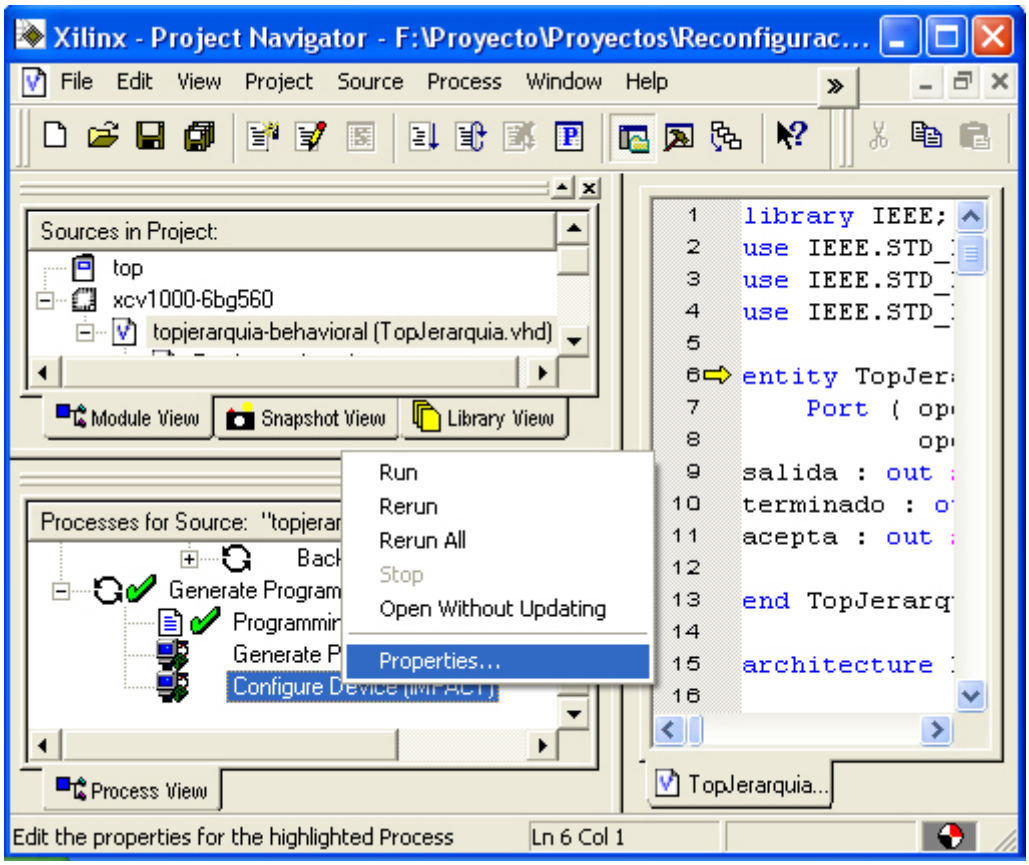
El siguiente paso es poner la línea de comando *"-g ActiveReconfig:Yes -g Persist:Yes"* en las propiedades de *"Generate Programming File"* como se muestra a continuación:

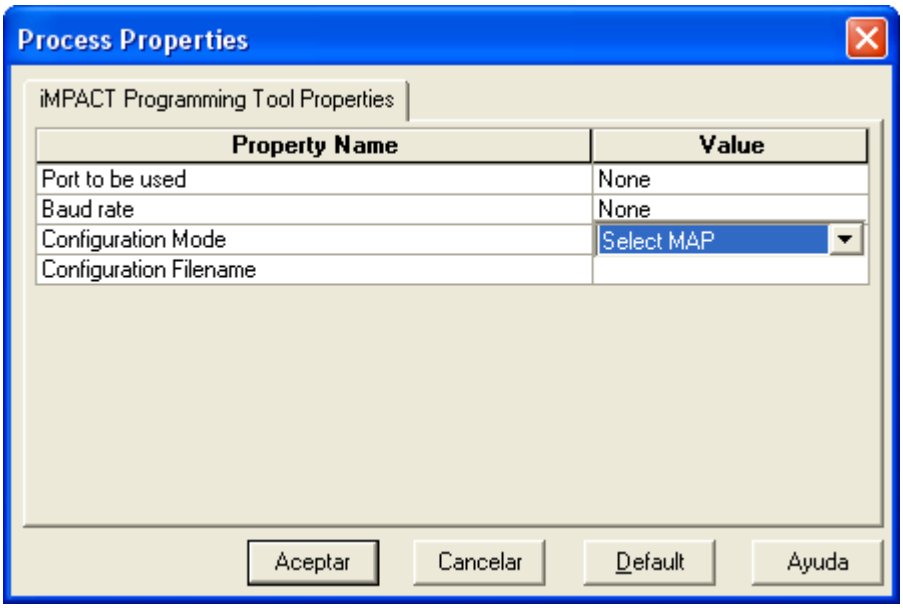




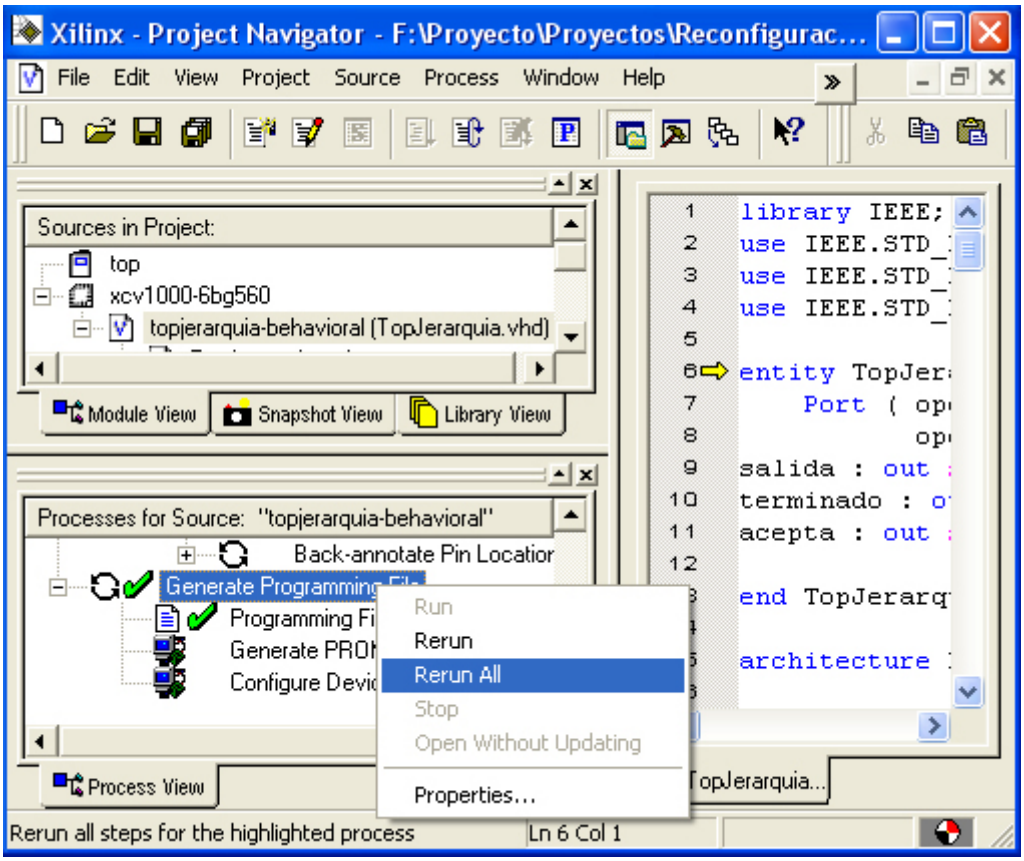
La última de las configuraciones adicionales es poner *"Select Map"* en el modo de configuración de las propiedades de *"Generate*

Programming File/Configure Device". Esto se hace como se muestra a continuación:





Ahora llega el momento de crear el archivo "*.bit*" pinchando con el botón derecho del ratón sobre *"Generate Programming File"* y pulsando en *"Rerun All"* como se muestra a continuación:



Para comprobar que cada módulo ha sido ubicado en la FPGA dónde y cómo queremos, revisamos el Map report y utilizamos el floorplaner o el FPGA_Editor.

En nuestro caso hemos elegido dos métodos para introducir las imágenes "*.bit*" en la fpga, la imagen del "*top*" se añade mediante el procedimiento *"directo desde un archivo"* explicado en el apartado 4.2.2.2.1 y la imagen del "*top2*" se añade mediante un "*Archivo .h"* explicado en el apartado 4.2.2.2.3.

Debido a esto la imagen *"topjerarquia.bit"* que pertenece al "*top*" es necesario que esté junto al archivo ejecutable del programa Host, por ello nosotros la copiamos en la carpeta "*EJECUTABLE*" de la

estructura de directorios donde más tarde copiaremos también el ejecutable del Host.

A la imagen *"topjerarquia2.bit"* hay que pasarla por el programa "*gencfg*" explicado en el apartado 4.3.4.3 para crear un archivo "*.h*", ya que el método elegido para añadir la imagen ha sido el de *"Mediante archivo .h"* explicado en el apartado 4.2.2.2.3. Para realizar esto se copia el archivo "*topjerarquia2.bit"* de la carpeta "*top2*" a la carpeta "*Imagen*" donde se encuentra la herramienta "*gencfg*" y un archivo ".*bat*" que crea automáticamente el archivo *"parcial.h"*, es decir sólo tenemos que copiar el archivo *"topjerarquia2.bit"* a esta carpeta y ejecutar el archivo ".*bat*" para que se cree el archivo *"parcial.h".*

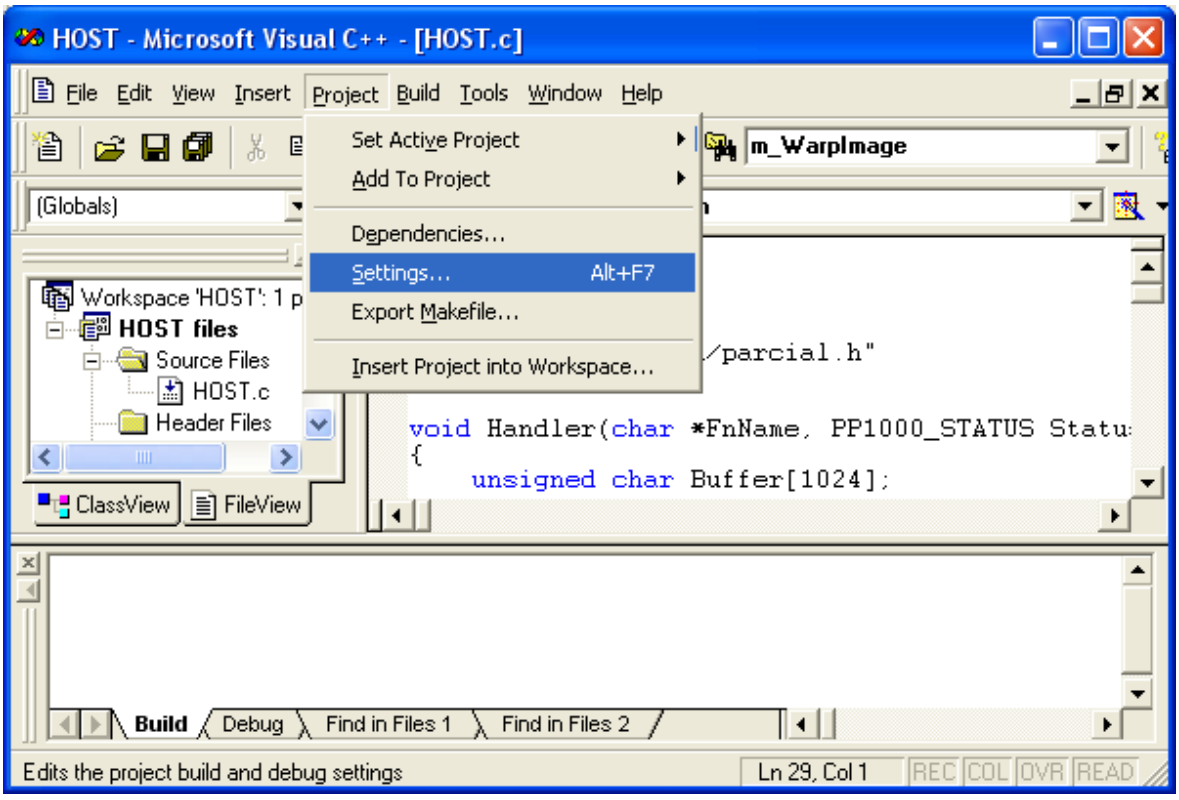
Una vez copiado el archivo *"topjerarquia.bit"* a la carpeta "EJECUTABLE" y creado el archivo *"parcial.h"* hay que pasar a la configuración del Host.

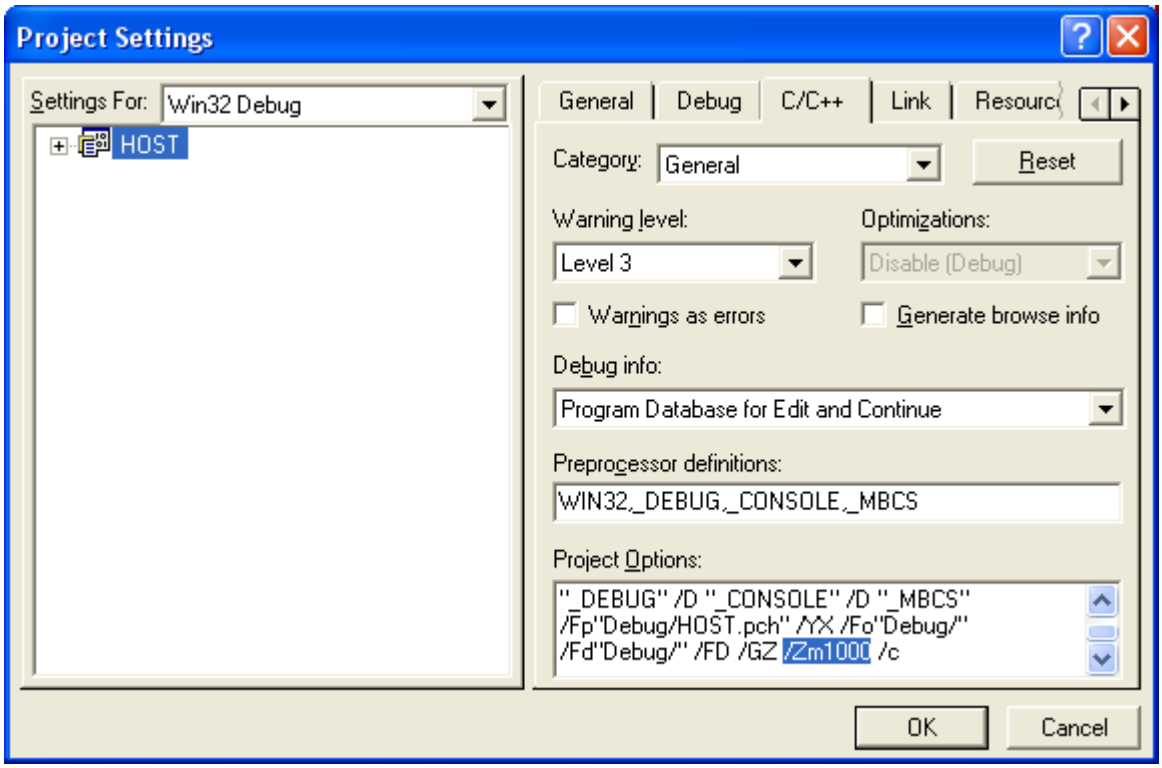
Nótese que el archivo *"parcial.h"* creado a partir del archivo *"topjerarqui2.bit"* con la herramienta "*gencfg*" no hay que moverle de sitio ya que el programa Host está configurado para que lo obtenga de esta ubicación.

9.4.5. CONFIGURAR EL HOST

El proyecto del Host le ubicamos en la carpeta "*HOST*" de la estructura de directorios, hay que crear un proyecto y configurarlo como se muestra en el apartado 4.2.1 añadiendo como archivo fuente "*HOST.c"* creado anteriormente.

Una vez configurado como se muestra en el apartado 4.2.1 hay que hacerle otra pequeña configuración para aumentar el límite del heap y poder cargar la imagen *"parcial.h"* ya que ésta es muy grande. Hay que poner *"/Zm1000"* en la siguiente ubicación:





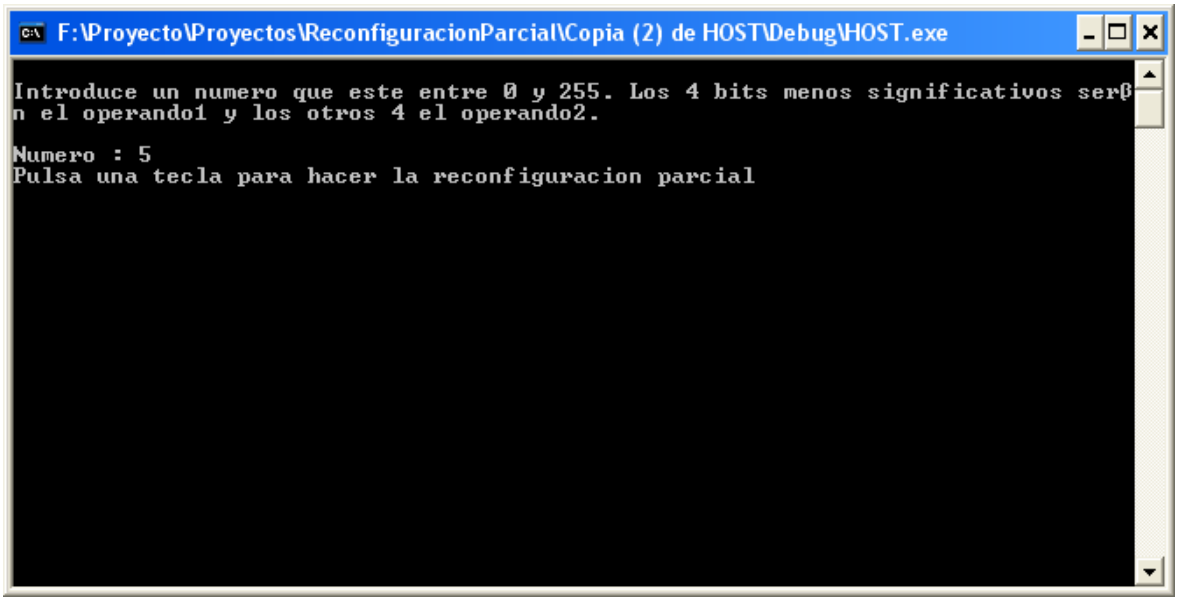
Una vez hechos todos estos pasos la herramienta ya está lista para crear el archivo ejecutable.

Una vez creado el ejecutable, éste se encuentra en la carpeta "*Debug*" del proyecto, copiamos el archivo ejecutable *"Host.exe"* y lo ponemos en la carpeta "*EJECUTABLE*" junto al archivo *"topjerarquia.bit"* para tener ambos archivos juntos y bien localizados.

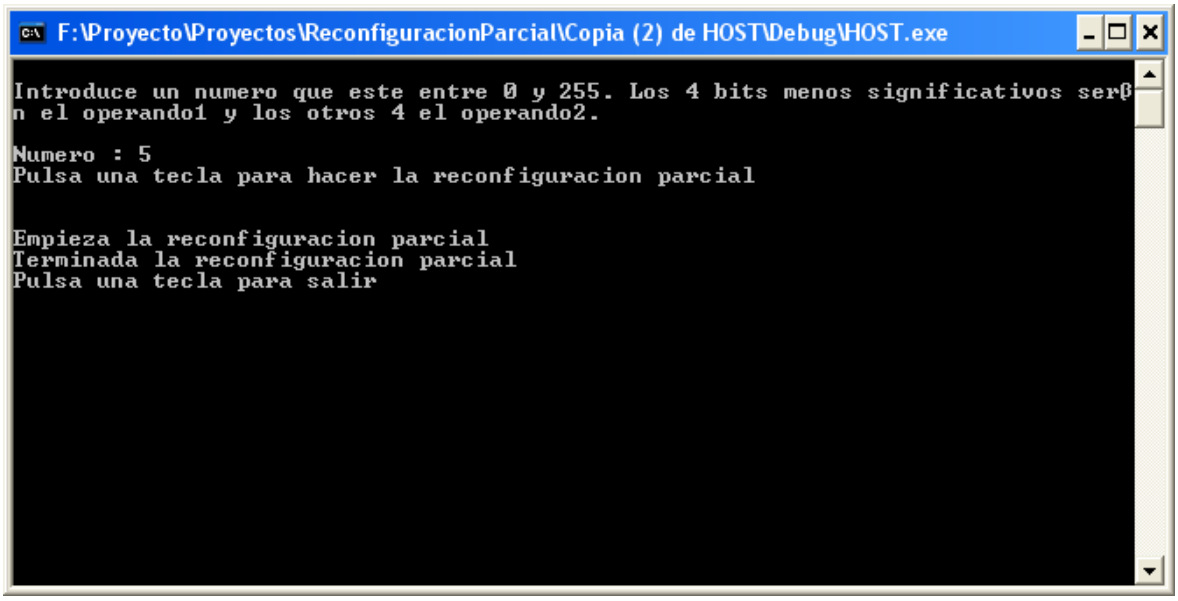
9.4.6. EJECUTAR Y VER LOS RESULTADOS

Ahora sólo queda ejecutar el archivo *"Host.exe"* que está en la carpeta "*EJECUTABLE*" asegurándonos de que también se encuentra en la misma ubicación el archivo *"topjerarquia.bit".*

Una vez ejecutado el programa te pide un número por teclado, introduces un número y pulsas "*enter*" para ver el resultado en los led's de la placa.



Ahora si pulsamos una tecla hacemos la reconfiguración parcial y cambiamos el módulo reconfigurable por el módulo reconfigurable2, se puede observar el cambio en los led's de la placa.



El resultado de los led's según el número introducido se puede seguir con esta tabla:

Teclado	Entradas		Salida sin la Reconfiguración		Salida con la Reconfiguración	
Número	Operando2	Operando1	(Op1+5)+(Op2+2)+3	Led's	(Op1+5)-(Op2+2)+3	Led's
0	0	0	10	1010	6	0100
1	0	1	11	1011	7	0111
2	0	2	12	1100	8	1000
3	0	3	13	1101	9	1001
4	0	4	14	1110	10	1010
5	0	5	15	1111	11	1011
6	0	6	16	0000	12	1100
7	0	7	17	0001	13	1101
8	0	8	18	0010	14	1110
9	0	9	19	0011	15	1111
10	0	10	20	0100	16	0000

9.5. PASOS PARA HACER UNA RECONFIGURACIÓN PARCIAL (MODO SCRIPTS)

9.5.1. ESTRUCTURA DE DIRECTORIOS RECOMENDADA POR XILINX

- Hdl: directorio en el que ubicaremos los archivos fuente VHDL.

- Modules: dentro suyo tendremos un subdirectorio por módulo (fijo o reconfigurable) que tenga nuestro diseño. Ahí tendremos el script active.bat, el archivo de la bus macro (bm_4b.nmc), el UCF, el EDIF (o NGC) resultado de sintetizar el módulo (paso 1 de los pasos generales utilizando scripts) y el NGD del top correspondiente (resultado de la fase Initial Budgeting). Para los módulos fijos el NGD será el del top de la configuración inicial. Para los módulos reconfigurables, será el del top al que pertenezcan.

- Pims: durante la fase Active Module Implementation se crearán aquí subdirectorios para cada módulo.

- Src: contiene los EDIF (o NGC) resultado de sintetizar todos los módulos y los tops en el paso 1 de los pasos generales utilizando scripts. En nuestro caso también contiene un subdirectorio Modulos, que contiene a su vez un subdirectorio para cada módulo y top donde tenemos los proyectos con los que los sintetizamos en dicho paso 1. Sin embargo, puede resultar más claro tener otro directorio al mismo nivel que Src, por ejemplo llamado síntesis, que contenga dichos proyectos.

- Top: habrá tantos directorios “Top<n>” como posibles configuraciones hay. Cada uno de estos directorios tiene a su vez dos subdirectorios

    - Initial: aquí tendremos el script initial.bat, el archivo de la bus macro (bm_4b.nmc) y el EDIF (o NGC) resultado de sintetizar el módulo (paso 1 de los pasos generales utilizando scripts).

- Assemble: tendremos el script assemble.bat, el archivo de la bus macro (bm_4b.nmc), el UCF, el EDIF (o NGC) resultado de sintetizar el módulo (paso 1 de los pasos generales utilizando scripts) y el NGD resultado de la fase Initial Budgeting.

9.5.2. INITIAL BUDGETING

Las operaciones de esta fase deben ser realizadas en el directorio Initial del top de la estructura de directorios recomendada por Xilinx.

Lo primero de todo es crear el UCF. Debemos pensar la ubicación de las áreas de los módulos en la FPGA, teniendo en cuenta para dichas áreas lo siguiente:

- Tienen una anchura mínima de 4 slices (CLBs).
- Tiene una anchura que es siempre múltiplo de 4 slices.
- Tiene la altura completa del dispositivo.
- El límite izquierdo del área siempre está ubicado en un CLB múltiplo de 4.

Hay que fijarse en que no se superpongan áreas y que todo lo contenido en el VHDL de cada módulo esté contenido en su área correspondiente. La lógica que esté en el top-level y no pertenezca a ningún módulo, debe ser restringida a lugares específicos del dispositivo mediante restricciones LOC. No debe haber lógica propia del top-level que no tenga restricciones.

Las restricciones LOC de cada bus macro deben ser insertadas manualmente en el UCF. Cada bus macro ocupará una sección de TBUF de 1 fila por 8 columnas. Hay que ubicar los bus macro exactamente entre los módulos formando un puente de

comunicación. Por tanto, deberá tener 4 columnas en el área de uno de los módulos y otras 4 en el área del otro módulo. Para más información referente a las bus macro, consultar el apartado 6.2.1.Bus Macro.

A continuación, hay que generar un NGO y un NGD para cada diseño top-level. Para ello, se ejecuta NGDBuild para crear dichos archivos para cada top con todos los módulos instanciados representados como bloques no expandidos. Para esto es necesario un archivo netlist, ya sea EDIF o NGC, generado por XST (Xilins Synthesis Technology). Si se utiliza NGC hay que asegurarse de especificar la extensión .ngc como parte del nombre de tu diseño. La opción de NGDBuild que se corresponde con todo lo explicado es –*modular initial*.

No podemos utilizar el NGD producido para la fase del MAP.

9.5.3. ACTIVE MODULE IMPEMENTATION

Llegados a este punto el diseño ha sido sintetizado, planeada su ubicación en la FPGA y creadas sus restricciones. Ahora hay que empezar con la implementación (“place and route”) de todos los módulos (tanto fijos como reconfigurables). Cada módulo será implementado por separado, pero siempre en el contexto de la lógica y las restricciones del top-level. De esta forma, se generarán los .bit parciales para todos los módulos reconfigurables. En esta sección describiremos cómo implementar independientemente cada módulo.

En primer lugar, hay que copiar el UCF creado durante la fase Initial Budgeting (directorio Initial del top) a los subdirectorios correspondientes a cada módulo que hay en el directorio Modules. Con esto, en el directorio en el que vamos a trabajar en esta fase (el de cada módulo en cuestión, dentro de Modules) tendremos el UCF y el netlist del módulo.

A continuación, ejecutamos NGDBuild, Map, Par, BitGen y PimCreate para cada módulo. Esto dará como resultado un archivo NGD para cada módulo y el .bit parcial correspondiente a cada módulo. Comentar que como opción de NGDBuild hay que poner ***–modular module –active*** *nombre_modulo*.

El proceso PimCreate publica los archivos asociados a cada módulo y producto de esta fase en el directorio Pims (NGO, NGD y NCD), que será utilizado en la fase Final Assembly.

Finalmente, habrá que comprobar con el FPGA_Editor que todo se ha ubicado en el lugar que le correspondía.

9.5.4. FINAL ASSEMBLY

En esta fase se ensambla el diseño top-level y los módulos en un archivo NGD y luego se implementa el archivo. Los pasos de esta fase serán ejecutados en el subdirectorio Assemble del top correspondiente. El resultado será el .bit del diseño completo.

En primer lugar habrá que copiar el archivo UCF creado durante la fase de Initial Budgeting (directorio Initial) al directorio assemble.

Se ejecutará NGDBuild para crear un archivo NGD completo expandido que contiene la lógica para el diseño top-level y para cada uno de los módulos físicamente implementados (PIMs). A continuación habrá que implementar este NGD ejecutando MAP y PAR, creando así un NCD completamente expandido. Las opciones que hay que darle a NGDBuild son:

-modular assemble –pimpath ruta_del_directorio_PIM **–use_pim** nombre_modulo1 **–use_pim** nombre_modulo2 ...

Con la opción –pimpath vamos a especificar la ruta del directorio que contiene los PIMs. Con la opción –use_pim identificamos todos los módulos del directorio PIM que han sido publicados. Hay que asegurarse de utilizar exactamente los nombres de los PIMs, incluyendo mayúsculas y minúsculas.

9.5.5. CONFIGURAR EL HOST

Igual que el apartado análogo del punto 9.4.5, ya que la única diferencia entre este punto (9.5) y el 9.4 es que el 9.4 es el paso a paso utilizando Project Navigator y el 6.5 es utilizando scripts, y eso de cara al ejecutable que se ejecuta en el PC y con el que interactúa el usuario no influye.

9.5.6. EJECUTAR Y VER LOS RESULTADOS

Al igual que en el párrafo anterior, ir al apartado análogo del paso a paso con Project Navigator, ya que este paso funciona igual en ambos casos.

10. CÓDIGOS FUENTE

10.1. SOFTWARE ADICIONAL

10.1.1. PACATOHEADERHEX

10.1.1.1. CÓDIGO C DESARROLLADO. PACATOHEADERHEX.C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
 * Main program
 */
  int main( int argc, char *argv[] )

{

    FILE *origen, *destino;
    char letra;
          char byte[4];
          char cabecera[29]="unsigned short datosShort[]={";
          int cont, cont1;

    origen=fopen(argv[1],"r");
    destino=fopen(argv[2],"w");

    if (origen==NULL || destino==NULL)
       {
       printf( "Problemas con los ficheros.\n" );
       exit( 1 );
       }

          //Pongo la cabecera
          for(cont=0; cont<29; cont++){
                    putc(cabecera[cont], destino);
          }

          //Quito la dirección del principio "0:" del principio
    letra=getc(origen);
    letra=getc(origen);

          //pongo la primera cabecera
          putc('\n', destino);
          putc('0', destino);
```

```
        putc('x', destino);
        letra=getc(origen);
        letra=getc(origen);

        //Pongo todos los datos
        cont=0; cont1=0;
        while (feof(origen)==0)
   {
                byte[cont]=letra;
                if(cont==3){
                        cont=-1;
                        if((byte[0]=='0') && (byte[1]=='0') && (byte[2]=='0') &&
(byte[3]=='0')) putc('0', destino);
                        else if((byte[0]=='0') && (byte[1]=='0') && (byte[2]=='0') &&
(byte[3]=='1')) putc('1', destino);
                        else if((byte[0]=='0') && (byte[1]=='0') && (byte[2]=='1') &&
(byte[3]=='0')) putc('2', destino);
                        else if((byte[0]=='0') && (byte[1]=='0') && (byte[2]=='1') &&
(byte[3]=='1')) putc('3', destino);
                        else if((byte[0]=='0') && (byte[1]=='1') && (byte[2]=='0') &&
(byte[3]=='0')) putc('4', destino);
                        else if((byte[0]=='0') && (byte[1]=='1') && (byte[2]=='0') &&
(byte[3]=='1')) putc('5', destino);
                        else if((byte[0]=='0') && (byte[1]=='1') && (byte[2]=='1') &&
(byte[3]=='0')) putc('6', destino);
                        else if((byte[0]=='0') && (byte[1]=='1') && (byte[2]=='1') &&
(byte[3]=='1')) putc('7', destino);
                        else if((byte[0]=='1') && (byte[1]=='0') && (byte[2]=='0') &&
(byte[3]=='0')) putc('8', destino);
                        else if((byte[0]=='1') && (byte[1]=='0') && (byte[2]=='0') &&
(byte[3]=='1')) putc('9', destino);
                        else if((byte[0]=='1') && (byte[1]=='0') && (byte[2]=='1') &&
(byte[3]=='0')) putc('a', destino);
                        else if((byte[0]=='1') && (byte[1]=='0') && (byte[2]=='1') &&
(byte[3]=='1')) putc('b', destino);
                        else if((byte[0]=='1') && (byte[1]=='1') && (byte[2]=='0') &&
(byte[3]=='0')) putc('c', destino);
                        else if((byte[0]=='1') && (byte[1]=='1') && (byte[2]=='0') &&
(byte[3]=='1')) putc('d', destino);
                        else if((byte[0]=='1') && (byte[1]=='1') && (byte[2]=='1') &&
(byte[3]=='0')) putc('e', destino);
                        else if((byte[0]=='1') && (byte[1]=='1') && (byte[2]=='1') &&
(byte[3]=='1')) putc('f', destino);
                }
                if(cont1==15){
                        cont1=-1;
                   letra=getc(origen);
                   letra=getc(origen);
                        if(feof(origen)==0){
                                putc(',',destino);
```

```
                                putc('\n', destino);
                                putc('0', destino);
                                putc('x', destino);
                        }
                }
            if(cont1!=-1) letra=getc(origen);
                cont++;
                cont1++;
        }

        //Pongo el final "};"
          putc('\n',destino);
          putc('}',destino);
          putc(';',destino);

   if (fclose(origen)!=0)
      printf( "Problemas al cerrar el fichero origen.txt\n" );
   if (fclose(destino)!=0)
      printf( "Problemas al cerrar el fichero destino.txt\n" );

   return 0;
}
```

10.2. COMUNICACIÓN CON LA FPGA

10.2.1. CONTADORLEDS (COMUNICACIÓN SINGLE BIT)

10.2.1.1. CONTADORLEDS.VHDL

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

entity ContadorLeds is
   Port ( activo : in std_logic;
                clk : in std_logic;
        leds : out std_logic_vector(4 downto 1));
end ContadorLeds;

architecture Behavioral of ContadorLeds is
signal auxs: std_logic_vector(4 downto 1);
begin
        leds <=auxs;
        process(activo, clk, auxs)
        variable cont : integer :=0;
        begin
                if clk'event and clk='1' and activo ='1' then
```

```
                    if cont=1000000 then
                        cont:=0;
                        auxs<=auxs+1;
                    else cont:=cont+1;
                end if;
                end if;
        end process;

end Behavioral;
```

10.2.1.2. CONTADORLEDS.UCF

```
NET "activo" LOC = "AL9"  ;
NET "clk" LOC = "D17"  ;
NET "leds<1>" LOC = "AN28";
NET "leds<2>" LOC = "AL26"  ;
NET "leds<3>" LOC = "AM27"  ;
NET "leds<4>" LOC = "AK24"  ;
```

10.2.2. MULTIPLICADORX2 (COMUNICACIÓN SINGLE BYTE)

10.2.2.1. CÓDIGO C DESARROLLADO PARA EL HOST. MULTIPLICADORX2.C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pp1000.h"

void Handler(char *FnName, PP1000_STATUS Status)
{
   unsigned char Buffer[1024];

   PP1000StatusToString(Status, Buffer, sizeof(Buffer));
   printf("\naddone - %s\n", Buffer);

   exit(1);
}

/*
 * Programa Principal
 */
int main(void)
{
   PP1000_HANDLE Handle;
   char Buffer[256];
```

```
unsigned char Number;
unsigned char ReturnVal;

/*
 * Se Instala el error handler
 */
PP1000InstallErrorHandler(Handler);

/*
 * Se abre la primera tarjeta del sistema
 */
PP1000OpenFirstCard(&Handle);

/*
 * Se pone la frecuencia del reloj de la tarjeta a 20MHz
 */
PP1000SetClockRate(Handle, PP1000_MCLK, 20e6);

/*
 * Se carga el .bit en la FPGA desde un archivo
 */
PP1000ConfigureFromFile(Handle, "multiplicadorx2.bit");

/*
 * Ahora se envían los bytes a la FPGA y se reciben los resultados
 */
printf("\nIntroduce un numero para multiplicarlo por 2: Para salir pulsar 0.\n\n");
do
{
    /*
     * Se captura el número
     */
    printf("Numero : ");

    gets(Buffer);
    Number = (unsigned char)(atol(Buffer) & 0xff);

    /*
     * Se envía el número a la FPGA
     */
    PP1000WriteControl(Handle, Number);

    /*
     * Se captura el resultado devuelto por la FPGA
     */
    PP1000ReadStatus(Handle, &ReturnVal);

    /*
     * Se imprime el resultado
     */
```

```
        printf("El numero multiplicado por 2 es: %d\n\n", ReturnVal);
    } while (Number!=0);

    /*
     * Se cierra la tarjeta
     */
    PP1000CloseCard(Handle);

    return 0;
}
```

10.2.2.2. CÓDIGO C DESARROLLADO PARA LA FPGA. MULTIPLICADORX2.HCC

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pp1000.h"

void Handler(char *FnName, PP1000_STATUS Status)
{
    unsigned char Buffer[1024];

    PP1000StatusToString(Status, Buffer, sizeof(Buffer));
    printf("\naddone - %s\n", Buffer);

    exit(1);
}


/*
 * Programa Principal
 */
int main(void)
{
    PP1000_HANDLE Handle;
    char Buffer[256];
    unsigned char Number;
    unsigned char ReturnVal;

    /*
     * Se Instala el error handler
     */
    PP1000InstallErrorHandler(Handler);

    /*
     * Se abre la primera tarjeta del sistema
     */
    PP1000OpenFirstCard(&Handle);
```

```
    /*
     * Se pone la frecuencia del reloj de la tarjeta a 20MHz
     */
    PP1000SetClockRate(Handle, PP1000_MCLK, 20e6);

    /*
     * Se carga el .bit en la FPGA desde un archivo
     */
    PP1000ConfigureFromFile(Handle, "multiplicadorx2.bit");

    /*
     * Ahora se envían los bytes a la FPGA y se reciben los resultados
     */
    printf("\nIntroduce un numero para multiplicarlo por 2: Para salir pulsar 0.\n\n");
    do
    {
        /*
         * Se captura el número
         */
        printf("Numero : ");

        gets(Buffer);
        Number = (unsigned char)(atol(Buffer) & 0xff);

        /*
         * Se envía el número a la FPGA
         */
        PP1000WriteControl(Handle, Number);

        /*
         * Se captura el resultado devuelto por la FPGA
         */
        PP1000ReadStatus(Handle, &ReturnVal);

        /*
         * Se imprime el resultado
         */
        printf("El numero multiplicado por 2 es: %d\n\n", ReturnVal);
    } while (Number!=0);

    /*
     * Se cierra la tarjeta
     */
    PP1000CloseCard(Handle);

    return 0;
}
```

10.2.3. SUMADORMAS1 (TRANSFERENCIA DMA)

10.2.3.1. CÓDIGO C DESARROLLADO PARA EL HOST. SUMADORMAS1.C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pp1000.h"

void Handler(char *FnName, PP1000_STATUS Status)
{
    unsigned char Buffer[1024];

    PP1000StatusToString(Status, Buffer, sizeof(Buffer));
    printf("\naddone - %s\n", Buffer);

    exit(1);
}

/*
 * Programa Principal
 */
int main(void)
{
    PP1000_HANDLE Handle;
          PP1000_CHANNEL Channel;
          unsigned long Imagen[]={0x0,0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15};
          unsigned long Imagen2[]={0x0,0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0,  0x0,  0x0,  0x0,  0x0,  0x0};
          char tecla[256];
          unsigned char status;


    /*
     * Se Instala el error handler
     */
    PP1000InstallErrorHandler(Handler);

    /*
     * Se abre la primera tarjeta
     */
    PP1000OpenFirstCard(&Handle);

    /*
     * Se pone la frencuencia del reloj de la tajeta a 20 MHz
     */
    PP1000SetClockRate(Handle, PP1000_MCLK, 20e6);
```

```
/*
 * Se lee el .bit desde un archivo
 */
PP1000ConfigureFromFile(Handle, "sumadormas1.bit");

        printf("Valores que vamos a volcar a la memoria.....\n\n");

        printf("Valor del array: %x\n",Imagen[0]);
        printf("Valor del array: %x\n",Imagen[1]);
        printf("Valor del array: %x\n",Imagen[2]);
        printf("Valor del array: %x\n",Imagen[3]);
        printf("Valor del array: %x\n",Imagen[4]);
        printf("Valor del array: %x\n",Imagen[5]);
        printf("Valor del array: %x\n",Imagen[6]);
        printf("Valor del array: %x\n",Imagen[7]);
        printf("Valor del array: %x\n",Imagen[8]);
        printf("Valor del array: %x\n",Imagen[9]);
        printf("Valor del array: %x\n",Imagen[10]);
        printf("Valor del array: %x\n",Imagen[11]);
        printf("Valor del array: %x\n",Imagen[12]);
        printf("Valor del array: %x\n",Imagen[13]);
        printf("Valor del array: %x\n",Imagen[14]);
        printf("Valor del array: %x\n\n",Imagen[15]);

        printf("Valores que tiene el array antes volcarle la memoria.....\n\n");

        printf("Valor del array: %x\n",Imagen2[0]);
        printf("Valor del array: %x\n",Imagen2[1]);
        printf("Valor del array: %x\n",Imagen2[2]);
        printf("Valor del array: %x\n",Imagen2[3]);
        printf("Valor del array: %x\n",Imagen2[4]);
        printf("Valor del array: %x\n",Imagen2[5]);
        printf("Valor del array: %x\n",Imagen2[6]);
        printf("Valor del array: %x\n",Imagen2[7]);
        printf("Valor del array: %x\n",Imagen2[8]);
        printf("Valor del array: %x\n",Imagen2[9]);
        printf("Valor del array: %x\n",Imagen2[10]);
        printf("Valor del array: %x\n",Imagen2[11]);
        printf("Valor del array: %x\n",Imagen2[12]);
        printf("Valor del array: %x\n",Imagen2[13]);
        printf("Valor del array: %x\n",Imagen2[14]);
        printf("Valor del array: %x\n\n",Imagen2[15]);


        //Escribo en el banco0 de memoria
        printf("Escribiendo la memoria.....\n\n");

        //Se configura el canal DMA
        PP1000SetupDMAChannel(Handle, Imagen, 0x0, 0x40,
PP1000_PCI2LOCAL, &Channel);
```

```
        //Se solicita el banco 0
        PP1000RequestMemoryBank(Handle, 0x1);

        //Se ejecuta la tansmisión
        PP1000DoDMA(Channel);

        //Se libera el banco 0
        PP1000ReleaseMemoryBank(Handle, 0x1);

        //Se cierra el canal DMA
        PP1000CloseDMAChannel(Channel);

        //Se la dice a la fpga que ya puede modificar los datos
        PP1000WriteControl(Handle, status);


        //Se espera a que la fpga haya terminado de modificar los datos
        PP1000ReadStatus(Handle, &status);

        //Se lee del Banco1 de memoria
        printf("Leyendo del banco1 de la memoria.....\n\n");

        //Se configura el canal DMA
        PP1000SetupDMAChannel(Handle, Imagen2, 0x200000, 0x40,
PP1000_LOCAL2PCI, &Channel);

        //Se solicita el banco 1
        PP1000RequestMemoryBank(Handle, 0x2);

        //Se realiza la transmisión
        PP1000DoDMA(Channel);

        //Se libera el banco 1
        PP1000ReleaseMemoryBank(Handle, 0x2);

        //Se cierra el canal DMA
        PP1000CloseDMAChannel(Channel);


        printf("Valores del array después de volcarle la memoria.......\n\n");

        printf("Valor del array: %x\n",Imagen2[0]);
        printf("Valor del array: %x\n",Imagen2[1]);
        printf("Valor del array: %x\n",Imagen2[2]);
        printf("Valor del array: %x\n",Imagen2[3]);
        printf("Valor del array: %x\n",Imagen2[4]);
        printf("Valor del array: %x\n",Imagen2[5]);
        printf("Valor del array: %x\n",Imagen2[6]);
        printf("Valor del array: %x\n",Imagen2[7]);
```

```
        printf("Valor del array: %x\n",Imagen2[8]);
        printf("Valor del array: %x\n",Imagen2[9]);
        printf("Valor del array: %x\n",Imagen2[10]);
        printf("Valor del array: %x\n",Imagen2[11]);
        printf("Valor del array: %x\n",Imagen2[12]);
        printf("Valor del array: %x\n",Imagen2[13]);
        printf("Valor del array: %x\n",Imagen2[14]);
        printf("Valor del array: %x\n",Imagen2[15]);


    /*
     * Se cierra la tarjeta
     */
    PP1000CloseCard(Handle);

        printf("Pulsa una tecla para salir....");
        gets(tecla);
    return 0;
}
```

10.2.3.2. CÓDIGO C DESARROLLADO PARA LA FPGA. SUMADORMAS1.HCC

```
#define PP1000_32BIT_RAMS
#define PP1000_DIVIDE1
#define PP1000_CLOCK PP1000_MCLK
#define PP1000_BOARD_TYPE PP1000_V2_VIRTEX
#include "pp1000.h"

/*
 * Programa Principal
 */
void main(void)
{
    unsigned 8 status;
        unsigned 32 Dato;
        unsigned 21 i;

    //Se espera a que el Host escriba en la memoria
    PP1000ReadControl(status);

        //Se solicitan los bancos 0 y 1
        PP1000RequestMemoryBank(0x3);

        for(i=0; i<16; i++){
                //Se lee el dato del banco 0 de memoria y se mete en Dato
                PP1000ReadBank0(Dato, i);
```

```
            //Se suma 1 al dato y se le envía la banco 1 de la memoria
            PP1000WriteBank1(i, Dato+1);
        }

        //Se liberan los bancos de memoria 0 y 1
        PP1000ReleaseMemoryBank(0x3);

    //Se le dice al Host que ya he modificado la memoria
        PP1000WriteStatus(status);

}
```

10.3. UNIÓN DE VHDL CON HANDELC

10.3.1. FILTRO BLANCO NEGRO

10.3.1.1. CÓDIGOS C DESARROLLADOS PARA EL HOST

10.3.1.1.1. MAINFRM.CPP

```
// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "video.h"

#include "MainFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
        //{{AFX_MSG_MAP(CMainFrame)
                // NOTE - the ClassWizard will add and remove mapping macros
here.
                //    DO NOT EDIT what you see in these blocks of generated code !
```

```
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
	// TODO: add member initialization code here

}

CMainFrame::~CMainFrame()
{
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	if( !CFrameWnd::PreCreateWindow(cs) )
		return FALSE;

   cs.cx = VISIBLE_WIDTH + GetSystemMetrics(SM_CXFIXEDFRAME) * 2;
   cs.cy = VISIBLE_HEIGHT + GetSystemMetrics(SM_CYFIXEDFRAME) * 2 +
           GetSystemMetrics(SM_CYMENU) +
GetSystemMetrics(SM_CYCAPTION);

   cs.style = WS_BORDER | WS_CAPTION | WS_MINIMIZEBOX |
              WS_MAXIMIZEBOX | WS_VISIBLE | WS_SYSMENU;

	return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
	CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
	CFrameWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMainFrame message handlers
```

10.3.1.1.2. STDAFX.CPP

```
// stdafx.cpp : source file that includes just the standard includes
//	video.pch will be the pre-compiled header
//	stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```

10.3.1.1.3. VIDEO.CPP

```
// video.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "video.h"

#include "MainFrm.h"
#include "videoDoc.h"
#include "videoView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CVideoApp

BEGIN_MESSAGE_MAP(CVideoApp, CWinApp)
	//{{AFX_MSG_MAP(CVideoApp)
	ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
		// NOTE - the ClassWizard will add and remove mapping macros
here.
		//    DO NOT EDIT what you see in these blocks of generated code!
	//}}AFX_MSG_MAP
	// Standard file based document commands
	ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
	ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CVideoApp construction

CVideoApp::CVideoApp()
{
	// TODO: add construction code here,
```

```
	// Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CVideoApp object

CVideoApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CVideoApp initialization

BOOL CVideoApp::InitInstance()
{
	// Standard initialization
	// If you are not using these features and wish to reduce the size
	//  of your final executable, you should remove from the following
	//  the specific initialization routines you do not need.

#ifdef _AFXDLL
	Enable3dControls();			// Call this when using MFC in a
shared DLL
#else
	Enable3dControlsStatic(); // Call this when linking to MFC statically
#endif

	// Change the registry key under which our settings are stored.
	// TODO: You should modify this string to be something appropriate
	// such as the name of your company or organization.
	SetRegistryKey(_T("Local AppWizard-Generated Applications"));

	LoadStdProfileSettings(0);  // Load standard INI file options (including MRU)

	// Register the application's document templates.  Document templates
	//  serve as the connection between documents, frame windows and views.

	CSingleDocTemplate* pDocTemplate;
	pDocTemplate = new CSingleDocTemplate(
		IDR_MAINFRAME,
		RUNTIME_CLASS(CVideoDoc),
		RUNTIME_CLASS(CMainFrame),	// main SDI frame window
		RUNTIME_CLASS(CVideoView));
	AddDocTemplate(pDocTemplate);

	// Parse command line for standard shell commands, DDE, file open
	CCommandLineInfo cmdInfo;
	ParseCommandLine(cmdInfo);

	// Dispatch commands specified on the command line
	if (!ProcessShellCommand(cmdInfo))
		return FALSE;
```

```
	// The one and only window has been initialized, so show and update it.
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	return TRUE;
}


/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
	CAboutDlg();

// Dialog Data
	//{{AFX_DATA(CAboutDlg)
	enum { IDD = IDD_ABOUTBOX };
	//}}AFX_DATA

	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CAboutDlg)
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support
	//}}AFX_VIRTUAL

// Implementation
protected:
	//{{AFX_MSG(CAboutDlg)
		// No message handlers
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
	//{{AFX_DATA_INIT(CAboutDlg)
	//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CAboutDlg)
	//}}AFX_DATA_MAP
}
```

```
BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
	//{{AFX_MSG_MAP(CAboutDlg)
		// No message handlers
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CVideoApp::OnAppAbout()
{
	CAboutDlg aboutDlg;
	aboutDlg.DoModal();
}

/////////////////////////////////////////////////////////////////////////////
// CVideoApp message handlers
```

10.3.1.1.4. VIDEODOC.CPP

```
// videoDoc.cpp : implementation of the CVideoDoc class
//

#include "stdafx.h"
#include "video.h"

#include "videoDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CVideoDoc

IMPLEMENT_DYNCREATE(CVideoDoc, CDocument)

BEGIN_MESSAGE_MAP(CVideoDoc, CDocument)
	//{{AFX_MSG_MAP(CVideoDoc)
		// NOTE - the ClassWizard will add and remove mapping macros
here.
		//    DO NOT EDIT what you see in these blocks of generated code!
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CVideoDoc construction/destruction

CVideoDoc::CVideoDoc()
```

```
{
	// TODO: add one-time construction code here

}

CVideoDoc::~CVideoDoc()
{
}

BOOL CVideoDoc::OnNewDocument()
{
	if (!CDocument::OnNewDocument())
		return FALSE;

	// TODO: add reinitialization code here
	// (SDI documents will reuse this document)

	return TRUE;
}



/////////////////////////////////////////////////////////////////////////////
// CVideoDoc serialization

void CVideoDoc::Serialize(CArchive& ar)
{
	if (ar.IsStoring())
	{
		// TODO: add storing code here
	}
	else
	{
		// TODO: add loading code here
	}
}

/////////////////////////////////////////////////////////////////////////////
// CVideoDoc diagnostics

#ifdef _DEBUG
void CVideoDoc::AssertValid() const
{
	CDocument::AssertValid();
}

void CVideoDoc::Dump(CDumpContext& dc) const
{
	CDocument::Dump(dc);
}
```

```
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CVideoDoc commands
```

10.3.1.1.5. VIDEOVIEW.CPP

```
// videoView.cpp : implementation of the CVideoView class
//

#include "stdafx.h"
#include "video.h"

#include "videoDoc.h"
#include "pp1000.h"
#include "../Imagen/warp.h"
#include "lena.h"
#include "grid.h"
#include "videoView.h"
#include <sys/timeb.h>
#include <stdlib.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

UINT ThreadHelper(LPVOID Arg);

/////////////////////////////////////////////////////////////////////////////
// CVideoView

IMPLEMENT_DYNCREATE(CVideoView, CView)

BEGIN_MESSAGE_MAP(CVideoView, CView)
        //{{AFX_MSG_MAP(CVideoView)
        ON_WM_ERASEBKGND()
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CVideoView construction/destruction

CVideoView::CVideoView() : m_Complete(0)
{
   PP1000_STATUS Status;
   unsigned long *Buffer;
   PP1000_CHANNEL DMAHandle;
   unsigned long i, x, y;
```

```
    /*
     * Initialise the DIB
     */
    m_ProcDIBHeader.bmiHeader.biSize=sizeof(BITMAPINFOHEADER);
    m_ProcDIBHeader.bmiHeader.biWidth=BITMAP_WIDTH;
    m_ProcDIBHeader.bmiHeader.biHeight=-BITMAP_HEIGHT;
    m_ProcDIBHeader.bmiHeader.biPlanes=1;
    m_ProcDIBHeader.bmiHeader.biBitCount=(unsigned
short)(BYTES_PER_PIXEL*8);
    m_ProcDIBHeader.bmiHeader.biCompression=BI_RGB;
    m_ProcDIBHeader.bmiHeader.biSizeImage=0;
    m_ProcDIBHeader.bmiHeader.biXPelsPerMeter=0;
    m_ProcDIBHeader.bmiHeader.biYPelsPerMeter=0;
    m_ProcDIBHeader.bmiHeader.biClrUsed=0;
    m_ProcDIBHeader.bmiHeader.biClrImportant=0;
    m_ProcDIBHeader.bmiHeader.biClrImportant=0;
    m_ProcBits = new
char[BITMAP_WIDTH*BITMAP_HEIGHT*BYTES_PER_PIXEL];

    /*
     * Register the FPGA configuration image
     */
    Status = PP1000RegisterImage(warpBuffer, warpLength,
                                 &m_WarpImage);

    /*
     * Open the RC1000-PP card
     */
    Status = PP1000OpenFirstCard(&m_Handle);

    /*
     * Set the programmable clock
     */
    Status = PP1000SetClockRate(m_Handle, PP1000_MCLK, CLOCK_RATE);

    /*
     * Send grid data
     */
    Buffer = (unsigned long *)malloc(32768*sizeof(unsigned long));
    for (i=0; i<32768; i++)
    {
        x = GridData[i*2];
        y = GridData[i*2+1];
        Buffer[i] = (y<<8) | x;
    }
    PP1000SetupDMAChannel(m_Handle, Buffer, 0x100000,
32768*sizeof(unsigned long),
                          PP1000_PCI2LOCAL, &DMAHandle);
    PP1000DoDMA(DMAHandle);
```

```
   PP1000CloseDMAChannel(DMAHandle);
   free(Buffer);

   /*
    * Send image data (shorts need to be swapped for endianness)
    */
   for (i=0; i<BITMAP_WIDTH*BITMAP_HEIGHT; i+=2)
   {
      unsigned short Temp;

      Temp = ImageData[i];
      ImageData[i] = ImageData[i+1];
      ImageData[i+1] = Temp;
   }
   PP1000SetupDMAChannel(m_Handle, ImageData, 0,
BITMAP_WIDTH*BITMAP_HEIGHT*BYTES_PER_PIXEL,
                         PP1000_PCI2LOCAL, &DMAHandle);
   PP1000DoDMA(DMAHandle);
   PP1000CloseDMAChannel(DMAHandle);

   /*
    * Configure the FPGA
    */
   Status = PP1000ConfigureFPGA(m_Handle, m_WarpImage);
   if (Status!=PP1000_SUCCESS)
   {
      AfxMessageBox("Could not configure the FPGA");
   }

         }

         CVideoView::~CVideoView()
         {

   /*
    * Kill helper
    */
   m_Cleanup=1;
   //m_Complete.Lock();
   delete m_ProcBits;

   /*
    * Free configuration image
    */
   PP1000FreeImage(m_WarpImage);

   /*
    * Close the RC1000-PP card
    */
   PP1000CloseCard(m_Handle);
```

```
}

BOOL CVideoView::PreCreateWindow(CREATESTRUCT& cs)
{
	return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CVideoView drawing

void CVideoView::OnDraw(CDC* pDC)
{
	UpdateWindow(pDC);
}

void CVideoView::UpdateWindow(CDC *pDC)
{
	RECT Rect;

	if (m_hWnd!=NULL)
	{
      GetClientRect(&Rect);

	    // Display the processed image
      if (m_ProcBits!=NULL)
	    {
		    StretchDIBits(pDC->GetSafeHdc(),
		0, 0,
		VISIBLE_WIDTH, VISIBLE_HEIGHT,
		VISIBLE_LEFT, VISIBLE_TOP,
		VISIBLE_WIDTH, VISIBLE_HEIGHT,
		m_ProcBits,
		&m_ProcDIBHeader,
		0, SRCCOPY);

		    }
	}
}

/////////////////////////////////////////////////////////////////////////////
// CVideoView diagnostics

#ifdef _DEBUG
void CVideoView::AssertValid() const
{
	CView::AssertValid();
}

void CVideoView::Dump(CDumpContext& dc) const
{
```

```
		CView::Dump(dc);
	}

	CVideoDoc* CVideoView::GetDocument() // non-debug version is inline
	{
		ASSERT(m_pDocument-
>IsKindOf(RUNTIME_CLASS(CVideoDoc)));
		return (CVideoDoc*)m_pDocument;
	}
	#endif //_DEBUG

	/////////////////////////////////////////////////////////////////////////////
	// Image processing thread

	// Helper function to start thread off
	UINT ThreadHelper(LPVOID Arg)
	{
	  ((CVideoView *)Arg)->ProcessImage();

		return 0;
	}

	void CVideoView::ProcessImage(void)
	{

	//variables de los archivos
   FILE *destino;

	//variables bin->hex
	unsigned long datosShort[200000]; //*******cambiar tamaño aqui
	const long N_PIXELES=200000;  //*******cambiar tamaño aqui

	//variables hex->bin
	unsigned long dividendo,cociente;
	long cont3;
	int cont2, cont4, cont5;
	char datosBin[32];


   CDC *WindowDC=GetDC();
   PP1000_CHANNEL DMAHandle=NULL;
   PP1000_STATUS Status;
   struct _timeb StartTime;
   struct _timeb EndTime;
   int Frame;
   PP1000_CARD_INFO CardInfo;

   /*
    * Lock down image buffer
    */
```

```
   Status = PP1000GetCardInfo(m_Handle, &CardInfo);
   if (Status!=PP1000_SUCCESS)
   {
      AfxMessageBox("PP1000GetCardInfo.-Could not communicate with card");
   }

        AfxMessageBox("Ten paciencia, el proceso puede durar algunos minutos");


   /*
    * Continue processing until told to stop
    */
   Frame=0;
   _ftime(&StartTime);
   while (m_Cleanup==0)
         {


   Status = PP1000SetupDMAChannel(m_Handle, m_ProcBits,
CardInfo.RAMBankSpace[0],
                           BYTES_PER_PIXEL*BITMAP_WIDTH*BITMAP_HEIGHT,
                           PP1000_LOCAL2PCI, &DMAHandle);
   if (Status!=PP1000_SUCCESS)
   {
      AfxMessageBox("PP1000SetupDMAChannel.-Could not communicate with
card");
   }


      /*
       * Start FPGA processing
       */
      Status = PP1000WriteControl(m_Handle, NULL);



      /*
       * Wait for FPGA to signal completion
       */
      Status = PP1000ReadStatus(m_Handle, NULL);

      /*
       * Get image from RC1000
       */
      Status = PP1000RequestMemoryBank(m_Handle, 3);

      Status = PP1000DoDMA(DMAHandle);

      Status = PP1000ReleaseMemoryBank(m_Handle, 3);
```

```
                PP1000CloseDMAChannel(DMAHandle);

                PP1000SetupDMAChannel(m_Handle, datosShort, 0x200000,
400000, PP1000_LOCAL2PCI,&DMAHandle);


                PP1000RequestMemoryBank(m_Handle, 0x2);


                PP1000DoDMA(DMAHandle);


                PP1000ReleaseMemoryBank(m_Handle, 0x2);


                PP1000CloseDMAChannel(DMAHandle);

//**********************************************************

        //PASO DEL ARRAY imagenFinal EN HEXADECIMAL A UN FICHERO
.PACA

//**********************************************************

        //abro el fichero destino

   destino=fopen("ImagenPaca.paca","w");
   if (destino==NULL)
   {
      printf( "Problemas con los ficheros.\n" );
      exit( 1 );
         }

        //Paso de hexadécimal a formato .PACA

        putc('0',destino);
        putc(':',destino);
        putc('\n',destino);
        for(cont3=0; cont3<N_PIXELES; cont3++)
        {
                dividendo=datosShort[cont3];
                cont2=31;
                for(cont4=0; cont4<32; cont4++)
```

```
                {
                        cociente=dividendo/2;
                        if(dividendo-(2*cociente)==1) datosBin[cont2]='1';
                        else datosBin[cont2]='0';
                        dividendo=cociente;
                        cont2--;
                }
                for(cont5=0; cont5<16; cont5++)
                {
                  putc(datosBin[cont5], destino);
                }
                putc('\n', destino);

                for(cont5=16; cont5<32; cont5++)
                {
                  putc(datosBin[cont5], destino);
                }
                putc('\n', destino);


        }



        // cierro el fichero destino

    if (fclose(destino)!=0)
      printf( "Problemas al cerrar el fichero destino\n" );

//*****************************************************************************

        /*
         * Draw screen
         */
                UpdateWindow(WindowDC);


      Frame++;
        if (Frame==100)
        {
           char Buffer[1024];
           float Time;

           _ftime(&EndTime);
           Time = ((float)EndTime.time + ((float)EndTime.millitm/1000)) -
                    ((float)StartTime.time + ((float)StartTime.millitm/1000));
           sprintf(Buffer, "Frames per second = %.2f\n", ((float)Frame)/Time);
           AfxMessageBox(Buffer);
        }
```

```
}


/*
 * Unlock image buffer
 */
if (DMAHandle!=NULL)
{
    PP1000CloseDMAChannel(DMAHandle);
}



/*
 * Release context
 */
if (m_hWnd!=NULL)
{
    ReleaseDC(WindowDC);
}

/*
 * Signal completion
 */
m_Complete.Unlock();



      }

      ////////////////////////////////////////////////////////////////////////////
      // CVideoView message handlers

      void CVideoView::OnInitialUpdate()
      {
            CView::OnInitialUpdate();

            /*
             * Spawn off helper thread
             */
            m_Cleanup=0;
            AfxBeginThread(ThreadHelper, this);
      }

      BOOL CVideoView::OnEraseBkgnd(CDC* pDC)
      {
            return 0;
      }
```

10.3.1.1.6. MAINFRM.H

```
// MainFrm.h : interface of the CMainFrame class
//
/////////////////////////////////////////////////////////////////////////////

#if
!defined(AFX_MAINFRM_H__596DC07B_9672_11D2_A49B_00105A42108C__I
NCLUDED_)
#define
AFX_MAINFRM_H__596DC07B_9672_11D2_A49B_00105A42108C__INCLUDE
D_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CMainFrame : public CFrameWnd
{

protected: // create from serialization only
	CMainFrame();
	DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:

// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CMainFrame)
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
	//}}AFX_VIRTUAL

// Implementation
public:
	virtual ~CMainFrame();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

// Generated message map functions
protected:
	//{{AFX_MSG(CMainFrame)
		// NOTE - the ClassWizard will add and remove member functions
here.
		//    DO NOT EDIT what you see in these blocks of generated code!
```

```
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif //
!defined(AFX_MAINFRM_H__596DC07B_9672_11D2_A49B_00105A42108C__I
NCLUDED_)
```

10.3.1.1.7. RESOURCE.H

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by VIDEO.RC
//
#define IDD_ABOUTBOX                    100
#define IDR_MAINFRAME                       128
#define IDR_VIDEOTYPE                   129

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_3D_CONTROLS                1
#define _APS_NEXT_RESOURCE_VALUE    130
#define _APS_NEXT_CONTROL_VALUE             1000
#define _APS_NEXT_SYMED_VALUE       101
#define _APS_NEXT_COMMAND_VALUE             32771
#endif
#endif
```

10.3.1.1.8. STDAFX.H

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

#if
!defined(AFX_STDAFX_H__596DC079_9672_11D2_A49B_00105A42108C__INC
LUDED_)
#define
AFX_STDAFX_H__596DC079_9672_11D2_A49B_00105A42108C__INCLUDED
_
```

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define VC_EXTRALEAN            // Exclude rarely-used stuff from Windows
headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxdtctl.h>               // MFC support for Internet Explorer 4 Common
Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>                     // MFC support for Windows Common
Controls
#endif // _AFX_NO_AFXCMN_SUPPORT


//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif //
!defined(AFX_STDAFX_H__596DC079_9672_11D2_A49B_00105A42108C__INC
LUDED_)
```

10.3.1.1.9. VIDEO.H

```
// video.h : main header file for the VIDEO application
//

#if
!defined(AFX_VIDEO_H__596DC077_9672_11D2_A49B_00105A42108C__INCL
UDED_)
#define
AFX_VIDEO_H__596DC077_9672_11D2_A49B_00105A42108C__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#ifndef __AFXWIN_H__
        #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols

#define VISIBLE_WIDTH 480
#define VISIBLE_HEIGHT 448
```

```
/////////////////////////////////////////////////////////////////////////////
// CVideoApp:
// See video.cpp for the implementation of this class
//

class CVideoApp : public CWinApp
{
public:
	CVideoApp();

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CVideoApp)
	public:
	virtual BOOL InitInstance();
	//}}AFX_VIRTUAL

// Implementation
	//{{AFX_MSG(CVideoApp)
	afx_msg void OnAppAbout();
		// NOTE - the ClassWizard will add and remove member functions
here.
		//    DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif //
!defined(AFX_VIDEO_H__596DC077_9672_11D2_A49B_00105A42108C__INCL
UDED_)
```

10.3.1.1.10. VIDEODOC.H

```
// videoDoc.h : interface of the CVideoDoc class
//
/////////////////////////////////////////////////////////////////////////////

#if
!defined(AFX_VIDEODOC_H__596DC07D_9672_11D2_A49B_00105A42108C__
INCLUDED_)
#define
AFX_VIDEODOC_H__596DC07D_9672_11D2_A49B_00105A42108C__INCLUD
ED_
```

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


class CVideoDoc : public CDocument
{
protected: // create from serialization only
	CVideoDoc();
	DECLARE_DYNCREATE(CVideoDoc)

// Attributes
public:

// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CVideoDoc)
	public:
	virtual BOOL OnNewDocument();
	virtual void Serialize(CArchive& ar);
	//}}AFX_VIRTUAL

// Implementation
public:
	virtual ~CVideoDoc();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
	//{{AFX_MSG(CVideoDoc)
		// NOTE - the ClassWizard will add and remove member functions
here.
		//    DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
```

```
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif //
!defined(AFX_VIDEODOC_H__596DC07D_9672_11D2_A49B_00105A42108C__
INCLUDED_)
```

10.3.1.1.11. VIDEOVIEW.H

```
// videoView.h : interface of the CVideoView class
//
/////////////////////////////////////////////////////////////////////////////

#if
!defined(AFX_VIDEOVIEW_H__596DC07F_9672_11D2_A49B_00105A42108C_
_INCLUDED_)
#define
AFX_VIDEOVIEW_H__596DC07F_9672_11D2_A49B_00105A42108C__INCLUD
ED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "afxmt.h"
#include "pp1000.h"

#define BITMAP_WIDTH 512
#define BITMAP_HEIGHT 512
#define VISIBLE_TOP 32
#define VISIBLE_LEFT 0
#define BYTES_PER_PIXEL 2
#define CLOCK_RATE 60e6
#define WARP_FILE "warp.dat"

class CVideoView : public CView
{
protected: // create from serialization only
        CVideoView();
        DECLARE_DYNCREATE(CVideoView)

// Attributes
public:
        CVideoDoc *GetDocument();
    CSemaphore m_Complete;

    BITMAPINFO m_ProcDIBHeader;
    char *m_ProcBits;
    int m_Cleanup;
    PP1000_IMAGE m_WarpImage;
```

```
    PP1000_HANDLE m_Handle,m_Handle2;

// Operations
public:
    void UpdateWindow(CDC *pDC);
    void ProcessImage(void);

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CVideoView)
        public:
        virtual void OnDraw(CDC* pDC);  // overridden to draw this view
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        virtual void OnInitialUpdate();
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CVideoView();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
        //{{AFX_MSG(CVideoView)
        afx_msg BOOL OnEraseBkgnd(CDC* pDC);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in videoView.cpp
inline CVideoDoc* CVideoView::GetDocument()
   { return (CVideoDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif //
!defined(AFX_VIDEOVIEW_H__596DC07F_9672_11D2_A49B_00105A42108C_
_INCLUDED_)
```

10.3.1.2. CÓDIGO C DESARROLLADO PARA LA FPGA. BLANCONEGRO.HCC

```
#define PP1000_32BIT_RAMS
#define PP1000_DIVIDE4
#define PP1000_BOARD_TYPE PP1000_V2_VIRTEX
#include "pp1000.h"


#define IMAGE_WIDTH 512         // Ancho y alto de la imagen en pixeles
#define IMAGE_HEIGHT 512
#define GRID_BASE 0x40000       // Base del grid en palabras


unsigned 1 auxComienzo=0;

/*
 * Warper
 */
macro proc WarpData()
{
   unsigned 8 status;
        unsigned 21 i;    //tiene que ser 21 siempre
        unsigned 32 dato;
        unsigned 32 salidaFiltro;
        unsigned 1 inutil;
        unsigned 16 Dato0;
        unsigned 16 Dato1;

        //Interface para el filtro
        interface filtroBlancoNegro
                (unsigned 32 datosSalida, unsigned 1 terminado)
                filtroBN
        (unsigned 1 comienzo = auxComienzo, unsigned 1 reloj = __clock,
unsigned 32 datosEntrada = dato)with {busformat= "B(I)"};

        //Espero a que el Host escriba en la memoria
   PP1000ReadControl(status);

        //Se solicita los bancos 0 y 1
        PP1000RequestMemoryBank(0x3);

        i=0;

        do
        {
                auxComienzo=0;
                auxComienzo=1;
```

```
                dato = PP1000Bank0[i];
                while (!(filtroBN.terminado))
                        inutil=0;
                salidaFiltro = filtroBN.datosSalida;
                Dato0=salidaFiltro[15:0];
                Dato1=salidaFiltro[31:16];
                PP1000WriteBank1(i, Dato0@Dato1);
                i++;
        }while(i!=524288);//2 a la 19 son 524288 que es el número de palabras de
32 bit de un baco de la RAM

        //Se liberan los bancos 0 y 1
        PP1000ReleaseMemoryBank(0x3);


   //Le digo al Host que ya he modificado la memoria
        PP1000WriteStatus(status);
}


/*
 * Main program
 */
void main(void)
{
   unsigned 8 Reg with {warn = 0};  /* Nunca se lee pero está bien */

   while(1)
   {
      /*
       * Espero a que el Host termine
       */
      PP1000ReadControl(Reg);

      /*
       * Se solicita el banco 0 y 1
       */
      PP1000RequestMemoryBank(0x3);

      /*
       * Proceso
       */
      WarpData();

      /*
       * Se liberan los bancos 0 y 1
       */
      PP1000ReleaseMemoryBank(0x3);

      /*
```

```
        * Se le dice al Host que ya ha terminado
        */
       PP1000WriteStatus(0);
   }
}
```

10.3.1.3. CÓDIGO VHDL. FILTROBLANCONEGRO.VHD

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-- Este filtro lo que va a tener una entrada para indicar el comienzo del proceso
-- una entrada/salida de datos que va a transformar y una señal de salida que
indica
-- cuando ha transformado los datos

entity filtroBlancoNegro is
    port (
        comienzo: in STD_LOGIC;
                    reloj : in STD_LOGIC;
        datosEntrada: in STD_LOGIC_VECTOR(31 downto 0);
        datosSalida: out STD_LOGIC_VECTOR(31 downto 0);
        terminado: out STD_LOGIC
        );
end filtroBlancoNegro;


architecture filtroBlancoNegro_arch of filtroBlancoNegro is

signal Raux, Gaux, Baux, RGB: std_logic_vector (4 downto 0);
signal Raux2, Gaux2, Baux2, RGB2: std_logic_vector (4 downto 0);
signal estado: std_logic_vector(1 downto 0);

begin

-- Salida = 0,256 * Rojo + 0,512 *  Verde + 0,128 * Azul
-- 512->1000000000 -> desplazar 1 a la dch
-- 256->0100000000 -> desplazar 2 a la dch
-- 128->0010000000 -> desplazar 3 a la dch

   -- Cuando los datos de entrada cambian, se ejecuta el proceso
   process (comienzo,estado,reloj)
   begin

    if(comienzo = '0')then
          estado <= "00";
--        terminado <= '0';
    elsif (reloj'event and reloj = '1') then
                            case estado is
```

```
                        when "00" =>
                                -- Se lee la información y se carga cada
componente del pixel leído
                                -- en la señal auxiliar correspondiente.
                                -- Los bits se cargan ya desplazadas las
posiciones necesarias.

                                if(comienzo = '1') then

                                        Baux(0) <= datosEntrada(3);
                                Baux(1) <= datosEntrada(4);
                                Baux(2) <= '0';
                                Baux(3) <= '0';
                        Baux(4) <= '0';
                        Gaux(0) <= datosEntrada(6);
                        Gaux(1) <= datosEntrada(7);
                        Gaux(2) <= datosEntrada(8);
                        Gaux(3) <= datosEntrada(9);
                        Gaux(4) <= '0';
                        Raux(0) <= datosEntrada(12);
                        Raux(1) <= datosEntrada(13);
                        Raux(2) <= datosEntrada(14);
                        Raux(3) <= '0';
                        Raux(4) <= '0';

                                        Baux2(0) <= datosEntrada(19);
                                Baux2(1) <= datosEntrada(20);
                                Baux2(2) <= '0';
                                Baux2(3) <= '0';
                        Baux2(4) <= '0';
                        Gaux2(0) <= datosEntrada(22);
                        Gaux2(1) <= datosEntrada(23);
                        Gaux2(2) <= datosEntrada(24);
                        Gaux2(3) <= datosEntrada(25);
                        Gaux2(4) <= '0';
                        Raux2(0) <= datosEntrada(28);
                        Raux2(1) <= datosEntrada(29);
                        Raux2(2) <= datosEntrada(30);
                        Raux2(3) <= '0';
                        Raux2(4) <= '0';

                        estado <= "01";

                        --  En otro caso se queda en el estado 00
                                end if;

                when "01" =>
                        -- Se calcula la suma para hallar el gris que corresponde
```

```
            RGB <= Raux + Gaux + Baux;
            RGB2 <= Raux2 + Gaux2 + Baux2;
            estado <= "10";

    when "10" =>

            -- Se almacenan los datos
            datosSalida(0) <= RGB(0);
            datosSalida(1) <= RGB(1);
            datosSalida(2) <= RGB(2);
            datosSalida(3) <= RGB(3);
            datosSalida(4) <= RGB(4);
            datosSalida(5) <= RGB(0);
            datosSalida(6) <= RGB(1);
            datosSalida(7) <= RGB(2);
            datosSalida(8) <= RGB(3);
            datosSalida(9) <= RGB(4);
            datosSalida(10) <= RGB(0);
            datosSalida(11) <= RGB(1);
            datosSalida(12) <= RGB(2);
            datosSalida(13) <= RGB(3);
            datosSalida(14) <= RGB(4);
            datosSalida(15) <= '0';

            datosSalida(16) <= RGB2(0);
            datosSalida(17) <= RGB2(1);
            datosSalida(18) <= RGB2(2);
            datosSalida(19) <= RGB2(3);
            datosSalida(20) <= RGB2(4);
            datosSalida(21) <= RGB2(0);
            datosSalida(22) <= RGB2(1);
            datosSalida(23) <= RGB2(2);
            datosSalida(24) <= RGB2(3);
            datosSalida(25) <= RGB2(4);
            datosSalida(26) <= RGB2(0);
            datosSalida(27) <= RGB2(1);
            datosSalida(28) <= RGB2(2);
            datosSalida(29) <= RGB2(3);
            datosSalida(30) <= RGB2(4);
            datosSalida(31) <= '0';

          -- Se indica que ha terminado

            terminado <= '1';

    when others =>
            estado <= "00";

    end case;
  end if;
```

```
   end process;

end filtroBlancoNegro_arch;
```

10.3.2. FILTRO NEGATIVIZADO

10.3.2.1. CÓDIGO C DESARROLLADO PARA EL HOST. VIDEOVIEW.CPP

Debido a la similitud con el código del filtro Blanco Negro sólo vamos a poner el único archivo que es diferente.

```
// videoView.cpp : implementation of the CVideoView class
//

#include "stdafx.h"
#include "video.h"

#include "videoDoc.h"
#include "pp1000.h"
#include "../Imagen/warp.h"
#include "lena.h"
#include "grid.h"
#include "videoView.h"
#include <sys/timeb.h>
#include <stdlib.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

UINT ThreadHelper(LPVOID Arg);

/////////////////////////////////////////////////////////////////////////////
// CVideoView

IMPLEMENT_DYNCREATE(CVideoView, CView)

BEGIN_MESSAGE_MAP(CVideoView, CView)
        //{{AFX_MSG_MAP(CVideoView)
        ON_WM_ERASEBKGND()
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
```

```
// CVideoView construction/destruction

CVideoView::CVideoView() : m_Complete(0)
{
    PP1000_STATUS Status;
    unsigned long *Buffer;
    PP1000_CHANNEL DMAHandle;
    unsigned long i, x, y;

    /*
     * Initialise the DIB
     */
    m_ProcDIBHeader.bmiHeader.biSize=sizeof(BITMAPINFOHEADER);
    m_ProcDIBHeader.bmiHeader.biWidth=BITMAP_WIDTH;
    m_ProcDIBHeader.bmiHeader.biHeight=-BITMAP_HEIGHT;
    m_ProcDIBHeader.bmiHeader.biPlanes=1;
    m_ProcDIBHeader.bmiHeader.biBitCount=(unsigned
short)(BYTES_PER_PIXEL*8);
    m_ProcDIBHeader.bmiHeader.biCompression=BI_RGB;
    m_ProcDIBHeader.bmiHeader.biSizeImage=0;
    m_ProcDIBHeader.bmiHeader.biXPelsPerMeter=0;
    m_ProcDIBHeader.bmiHeader.biYPelsPerMeter=0;
    m_ProcDIBHeader.bmiHeader.biClrUsed=0;
    m_ProcDIBHeader.bmiHeader.biClrImportant=0;
    m_ProcDIBHeader.bmiHeader.biClrImportant=0;
    m_ProcBits = new
char[BITMAP_WIDTH*BITMAP_HEIGHT*BYTES_PER_PIXEL];

    /*
     * Register the FPGA configuration image
     */
    Status = PP1000RegisterImage(warpBuffer, warpLength,
                                 &m_WarpImage);

    /*
     * Open the RC1000-PP card
     */
    Status = PP1000OpenFirstCard(&m_Handle);

    /*
     * Set the programmable clock
     */
    Status = PP1000SetClockRate(m_Handle, PP1000_MCLK, CLOCK_RATE);

    /*
     * Send grid data
     */
    Buffer = (unsigned long *)malloc(32768*sizeof(unsigned long));
    for (i=0; i<32768; i++)
    {
```

```
        x = GridData[i*2];
        y = GridData[i*2+1];
        Buffer[i] = (y<<8) | x;
    }
    PP1000SetupDMAChannel(m_Handle, Buffer, 0x100000,
32768*sizeof(unsigned long),
                        PP1000_PCI2LOCAL, &DMAHandle);
    PP1000DoDMA(DMAHandle);
    PP1000CloseDMAChannel(DMAHandle);
    free(Buffer);

    /*
     * Send image data
     */
    PP1000SetupDMAChannel(m_Handle, ImageData, 0,
BITMAP_WIDTH*BITMAP_HEIGHT*BYTES_PER_PIXEL,
                        PP1000_PCI2LOCAL, &DMAHandle);
    PP1000DoDMA(DMAHandle);
    PP1000CloseDMAChannel(DMAHandle);

    /*
     * Configure the FPGA
     */
    Status = PP1000ConfigureFPGA(m_Handle, m_WarpImage);
    if (Status!=PP1000_SUCCESS)
    {
        AfxMessageBox("Could not configure the FPGA");
    }

          }

          CVideoView::~CVideoView()
          {

    /*
     * Kill helper
     */
    m_Cleanup=1;
    //m_Complete.Lock();
    delete m_ProcBits;

    /*
     * Free configuration image
     */
    PP1000FreeImage(m_WarpImage);

    /*
     * Close the RC1000-PP card
     */
    PP1000CloseCard(m_Handle);
```

```
}

BOOL CVideoView::PreCreateWindow(CREATESTRUCT& cs)
{
	return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CVideoView drawing

void CVideoView::OnDraw(CDC* pDC)
{
	UpdateWindow(pDC);
}

void CVideoView::UpdateWindow(CDC *pDC)
{
	RECT Rect;

	if (m_hWnd!=NULL)
	{
       GetClientRect(&Rect);

	    // Display the processed image
       if (m_ProcBits!=NULL)
	    {
		    StretchDIBits(pDC->GetSafeHdc(),
		0, 0,
		VISIBLE_WIDTH, VISIBLE_HEIGHT,
		VISIBLE_LEFT, VISIBLE_TOP,
		VISIBLE_WIDTH, VISIBLE_HEIGHT,
		m_ProcBits,
		&m_ProcDIBHeader,
		0, SRCCOPY);

		    }
	}
}

/////////////////////////////////////////////////////////////////////////////
// CVideoView diagnostics

#ifdef _DEBUG
void CVideoView::AssertValid() const
{
	CView::AssertValid();
}

void CVideoView::Dump(CDumpContext& dc) const
{
```

```
		CView::Dump(dc);
	}

	CVideoDoc* CVideoView::GetDocument() // non-debug version is inline
	{
		ASSERT(m_pDocument-
>IsKindOf(RUNTIME_CLASS(CVideoDoc)));
		return (CVideoDoc*)m_pDocument;
	}
	#endif //_DEBUG

	/////////////////////////////////////////////////////////////////////////////
	// Image processing thread

	// Helper function to start thread off
	UINT ThreadHelper(LPVOID Arg)
	{
	  ((CVideoView *)Arg)->ProcessImage();

		return 0;
	}

	void CVideoView::ProcessImage(void)
	{

	//variables de los archivos
   FILE *destino;

	//variables bin->hex
	unsigned long datosShort[200000]; //*******cambiar tamaño aqui
	const long N_PIXELES=200000;  //*******cambiar tamaño aqui

	//variables hex->bin
	unsigned long dividendo,cociente;
	long cont3;
	int cont2, cont4, cont5;
	char datosBin[32];


   CDC *WindowDC=GetDC();
   PP1000_CHANNEL DMAHandle=NULL;
   PP1000_STATUS Status;
   struct _timeb StartTime;
   struct _timeb EndTime;
   int Frame;
   PP1000_CARD_INFO CardInfo;

   /*
    * Lock down image buffer
    */
```

```
   Status = PP1000GetCardInfo(m_Handle, &CardInfo);
   if (Status!=PP1000_SUCCESS)
   {
      AfxMessageBox("PP1000GetCardInfo.-Could not communicate with card");
   }

        AfxMessageBox("Ten paciencia, el proceso puede durar algunos minutos");


   /*
    * Continue processing until told to stop
    */
   Frame=0;
   _ftime(&StartTime);
   while (m_Cleanup==0)
         {


   Status = PP1000SetupDMAChannel(m_Handle, m_ProcBits,
CardInfo.RAMBankSpace[0],
                           BYTES_PER_PIXEL*BITMAP_WIDTH*BITMAP_HEIGHT,
                           PP1000_LOCAL2PCI, &DMAHandle);
   if (Status!=PP1000_SUCCESS)
   {
      AfxMessageBox("PP1000SetupDMAChannel.-Could not communicate with
card");
   }


      /*
       * Start FPGA processing
       */
      Status = PP1000WriteControl(m_Handle, NULL);



      /*
       * Wait for FPGA to signal completion
       */
      Status = PP1000ReadStatus(m_Handle, NULL);

      /*
       * Get image from RC1000
       */
      Status = PP1000RequestMemoryBank(m_Handle, 3);

      Status = PP1000DoDMA(DMAHandle);

      Status = PP1000ReleaseMemoryBank(m_Handle, 3);
```

```
                PP1000CloseDMAChannel(DMAHandle);

                PP1000SetupDMAChannel(m_Handle, datosShort, 0x200000,
400000, PP1000_LOCAL2PCI,&DMAHandle);


                PP1000RequestMemoryBank(m_Handle, 0x2);


                PP1000DoDMA(DMAHandle);


                PP1000ReleaseMemoryBank(m_Handle, 0x2);


                PP1000CloseDMAChannel(DMAHandle);

//**********************************************************

        //PASO DEL ARRAY imagenFinal EN HEXADECIMAL A UN FICHERO
.PACA

//**********************************************************

        //abro el fichero destino

   destino=fopen("ImagenPaca.paca","w");
   if (destino==NULL)
   {
      printf( "Problemas con los ficheros.\n" );
      exit( 1 );
         }

        //Paso de hexadécimal a formato .PACA

        putc('0',destino);
        putc(':',destino);
        putc('\n',destino);
        for(cont3=0; cont3<N_PIXELES; cont3++)
        {
                dividendo=datosShort[cont3];
                cont2=31;
                for(cont4=0; cont4<32; cont4++)
```

```
                {
                        cociente=dividendo/2;
                        if(dividendo-(2*cociente)==1) datosBin[cont2]='1';
                        else datosBin[cont2]='0';
                        dividendo=cociente;
                        cont2--;
                }
                for(cont5=0; cont5<16; cont5++)
                {
                  putc(datosBin[cont5], destino);
                }
                putc('\n', destino);

                for(cont5=16; cont5<32; cont5++)
                {
                  putc(datosBin[cont5], destino);
                }
                putc('\n', destino);


        }



        // cierro el fichero destino

    if (fclose(destino)!=0)
      printf( "Problemas al cerrar el fichero destino\n" );

//*****************************************************************************

        /*
         * Draw screen
         */
                UpdateWindow(WindowDC);


      Frame++;
        if (Frame==100)
        {
           char Buffer[1024];
           float Time;

           _ftime(&EndTime);
           Time = ((float)EndTime.time + ((float)EndTime.millitm/1000)) -
                    ((float)StartTime.time + ((float)StartTime.millitm/1000));
           sprintf(Buffer, "Frames per second = %.2f\n", ((float)Frame)/Time);
           AfxMessageBox(Buffer);
        }
```

```
}


/*
 * Unlock image buffer
 */
if (DMAHandle!=NULL)
{
    PP1000CloseDMAChannel(DMAHandle);
}



/*
 * Release context
 */
if (m_hWnd!=NULL)
{
    ReleaseDC(WindowDC);
}

/*
 * Signal completion
 */
m_Complete.Unlock();



      }

      ////////////////////////////////////////////////////////////////////////////
      // CVideoView message handlers

      void CVideoView::OnInitialUpdate()
      {
            CView::OnInitialUpdate();

            /*
             * Spawn off helper thread
             */
            m_Cleanup=0;
            AfxBeginThread(ThreadHelper, this);
      }

      BOOL CVideoView::OnEraseBkgnd(CDC* pDC)
      {
            return 0;
      }
```

10.3.2.2. CÓDIGO C DESARROLLADO PARA LA FPGA. NEGATIVIZADO.HCC

```
#define PP1000_32BIT_RAMS
#define PP1000_DIVIDE4
#define PP1000_BOARD_TYPE PP1000_V2_VIRTEX
#include "pp1000.h"


#define IMAGE_WIDTH 512         // Ancho y alto de la imagen en pixeles
#define IMAGE_HEIGHT 512
#define GRID_BASE 0x40000       // Base del grid de datos en palabras


unsigned 1 auxComienzo=0;

/*
 * Warper
 */
macro proc WarpData()
{
   unsigned 8 status;
        unsigned 21 i;    //tiene que ser 21 siempre
        unsigned 32 auxDato;
        unsigned 32 dato;
        unsigned 32 salidaFiltro;
        unsigned 1 inutil;
        unsigned 16 Dato0;
        unsigned 16 Dato1;

//Interface para el filtro
        interface filtroNegativizado
                (unsigned 32 datosSalida, unsigned 1 terminado)
        filtroNeg
        (unsigned 1 comienzo = auxComienzo, unsigned 1 reloj = __clock,
unsigned 32 datosEntrada = dato)with {busformat= "B(I)"};

        //Espero a que el Host escriba en la memoria
   PP1000ReadControl(status);

        //Se solicita los bancos de memoria 0 y 1
        PP1000RequestMemoryBank(0x3);

        i=0;

        //Sacamos esta vuelta fuera para que en la primera del bucle el
PP1000Write tenga algo del filtro para escribir
        auxComienzo=0;
```

```
        auxComienzo=1;
        dato = PP1000Bank0[i];
        i++;
        do
        {
                auxComienzo=0;
                auxComienzo=1;
                par
                {
                        dato = PP1000Bank0[i];
                        while (!(filtroNeg.terminado))
                                inutil=0;
                        PP1000WriteBank1(i, filtroNeg.datosSalida);
                        i++;
                }
        }while(i!=524288);//2 a la 19 son 524288 que es el número de palabras de
32 bit de un baco de la RAM

        //Sacamos la escritura de la última vuelta
        PP1000WriteBank1(i, filtroNeg.datosSalida);

        //Se liberan los bancos de memoria 0 y 1
        PP1000ReleaseMemoryBank(0x3);


    //Le digo al Host que ya he modificado la memoria
        PP1000WriteStatus(status);
}


/*
 * Programa principal
 */
void main(void)
{
    unsigned 8 Reg with {warn = 0};  /* Nunca lee pero está bien */

    while(1)
    {
        /*
         * Espera a que el Host termine
         */
        PP1000ReadControl(Reg);

        /*
         * Solicita los bancos 0 y 1
         */
        PP1000RequestMemoryBank(0x3);

        /*
```

```
         * proceso
         */
        WarpData();

        /*
         * Libera los bancos de meoria 0 y 1
         */
        PP1000ReleaseMemoryBank(0x3);

        /*
         * Le dice al servidor que ya ha terminado
         */
        PP1000WriteStatus(0);
    }
}
```

10.3.2.3. CÓDIGO VHDL. FILTRONEGATIVIZADO.VHD

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-- Este filtro lo que va a tener una entrada para indicar el comienzo del proceso
-- una entrada/salida de datos que va a transformar y una señal de salida que
indica
-- cuando ha transformado los datos

entity filtroNegativizado is
    port (
        comienzo: in STD_LOGIC;
                    reloj : in STD_LOGIC;
        datosEntrada: in STD_LOGIC_VECTOR(31 downto 0);
        datosSalida: out STD_LOGIC_VECTOR(31 downto 0);
        terminado: out STD_LOGIC
        );
end filtroNegativizado;


architecture filtroNegativizado_arch of filtroNegativizado is

signal estado: std_logic_vector(1 downto 0);

begin

-- Salida = 0,256 * Rojo + 0,512 *  Verde + 0,128 * Azul
-- 512->1000000000 -> desplazar 1 a la dch
-- 256->0100000000 -> desplazar 2 a la dch
-- 128->0010000000 -> desplazar 3 a la dch

   -- Cuando los datos de entrada cambian, se ejecuta el proceso
```

```
   process (comienzo,estado,reloj)
   begin

    if(comienzo = '0')then
          estado <= "00";
--        terminado <= '0';
    elsif (reloj'event and reloj = '1') then
                          case estado is

                                    when "00" =>
                                          -- Se lee la información y se carga cada
componente del pixel leído
                                          -- en la señal auxiliar correspondiente.
                                          -- Los bits se cargan ya desplazadas las
posiciones necesarias.

                                          if(comienzo = '1') then

                                                datosSalida(0) <= not datosEntrada(0);
                                                datosSalida(1) <= not datosEntrada(1);
                                                datosSalida(2) <= not datosEntrada(2);
                                                datosSalida(3) <= not datosEntrada(3);
                                                datosSalida(4) <= not datosEntrada(4);
                                                datosSalida(5) <= not datosEntrada(5);
                                                datosSalida(6) <= not datosEntrada(6);
                                                datosSalida(7) <= not datosEntrada(7);
                                                datosSalida(8) <= not datosEntrada(8);
                                                datosSalida(9) <= not datosEntrada(9);
                                                datosSalida(10) <= not datosEntrada(10);
                                                datosSalida(11) <= not datosEntrada(11);
                                                datosSalida(12) <= not datosEntrada(12);
                                                datosSalida(13) <= not datosEntrada(13);
                                                datosSalida(14) <= not datosEntrada(14);
                                                datosSalida(15) <= not datosEntrada(15);
                                                datosSalida(16) <= not datosEntrada(16);
                                                datosSalida(17) <= not datosEntrada(17);
                                                datosSalida(18) <= not datosEntrada(18);
                                                datosSalida(19) <= not datosEntrada(19);
                                                datosSalida(20) <= not datosEntrada(20);
                                                datosSalida(21) <= not datosEntrada(21);
                                                datosSalida(22) <= not datosEntrada(22);
                                                datosSalida(23) <= not datosEntrada(23);
                                                datosSalida(24) <= not datosEntrada(24);
                                                datosSalida(25) <= not datosEntrada(25);
                                                datosSalida(26) <= not datosEntrada(26);
                                                datosSalida(27) <= not datosEntrada(27);
                                                datosSalida(28) <= not datosEntrada(28);
                                                datosSalida(29) <= not datosEntrada(29);
                                                datosSalida(30) <= not datosEntrada(30);
                                                datosSalida(31) <= '0';
```

```
                                estado <= "01";

                                --  En otro caso se queda en el estado 00
                                    end if;

                    when "01" =>
                                -- Se indica que ha terminado
                                terminado <= '1';

                    when others =>
                            estado <= "00";

                    end case;
                end if;
    end process;

end filtroNegativizado_arch;
```

10.4. RECONFIGURACIÓN PARCIAL

10.4.1. CÓDIGO C DESARROLLADO PARA EL HOST. HOST.C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pp1000.h"
#include "../Imagen/parcial.h"

void Handler(char *FnName, PP1000_STATUS Status)
{
    unsigned char Buffer[1024];

    PP1000StatusToString(Status, Buffer, sizeof(Buffer));
    printf("\naddone - %s\n", Buffer);

    exit(1);
}

/*
 * Programa Principal
 */
int main(void)
{
    PP1000_HANDLE Handle;
        PP1000_CHANNEL Channel;
        char Buffer[256];
    unsigned char Number;
```

```
    /*
     * Se Instala el error handler
     */
    PP1000InstallErrorHandler(Handler);

    /*
     * Se abre la primera tarjeta del sistema
     */
    PP1000OpenFirstCard(&Handle);

    /*
     * Se pone la frecuencia del reloj de la tarjeta a 20MHz
     */
    PP1000SetClockRate(Handle, PP1000_MCLK, 20e6);

    /*
     * Se carga el .bit en la FPGA desde un archivo
     */
    PP1000ConfigureFromFile(Handle, "topjerarquia.bit");

    /*
     * Ahora se envían los bytes a la FPGA y se reciben los resultados
     */
    printf("\nIntroduce un número que esté entre 0 y 255. Los 4 bits menos
significativos serán el operando1 y los otros 4 el operando2.\n\n");

        /*
         * Se captura el número
         */
        printf("Numero : ");

        gets(Buffer);

        Number = (unsigned char)(atol(Buffer) & 0xff);
        /*
         * Se envía el número a la FPGA
         */
        PP1000WriteControl(Handle, Number);

        printf("Pulsa una tecla para hacer la reconfiguracion parcial\n\n");
        gets(Buffer);

        //Hacemos la reconfiguración parcial
        PP1000SetupSelectMapChannel(Handle,parcialBuffer,parcialLength,PP100
0_PCI2LOCAL, &Channel);    /* Returned DMA channel handle */

    /*
     * Do DMA to card
     */
```

```
    printf("Empieza la reconfiguracion parcial\n");
    PP1000DoDMA(Channel);
    printf("Terminada la reconfiguracion parcial\n");

        printf("Pulsa una tecla para salir\n\n");
        gets(Buffer);

    /*
     * Se cierra la tarjeta
     */
    PP1000CloseCard(Handle);

    return 0;
}
```

10.4.2. CÓDIGOS VHDL

10.4.2.1. TOPJERARQUIA.VHD

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

entity TopJerarquia is
    Port ( operando1 : in std_logic_vector(3 downto 0);
           operando2 : in std_logic_vector(3 downto 0);
salida : out std_logic_vector(7 downto 0);
terminado : out std_logic;
acepta : out std_logic
                      );
end TopJerarquia;

architecture Behavioral of TopJerarquia is

        COMPONENT bm_4b
        PORT(
                    LI : in  std_logic_vector (3 downto 0);
                    LT : in  std_logic_vector (3 downto 0);
                    RI : in  std_logic_vector (3 downto 0);
                    RT : in  std_logic_vector (3 downto 0);
                    O  : out std_logic_vector (3 downto 0)
                    );
        END COMPONENT;

        COMPONENT ModIzquierda
        PORT( operandoIZQ1 : in std_logic_vector(3 downto 0);
       operandoIZQ2 : in std_logic_vector(3 downto 0);
```

```
                    salidaIZQ1 : out std_logic_vector(3 downto 0);
                    salidaIZQ2 : out std_logic_vector(3 downto 0)
                    );
    END COMPONENT;


    COMPONENT ModReconfigurable
    PORT( operandoR1 : in std_logic_vector(3 downto 0);
   operandoR2 : in std_logic_vector(3 downto 0);
                    salidaR : out std_logic_vector(3 downto 0)
                    );
    END COMPONENT;

    COMPONENT ModDerecha
    PORT( entradaDCH : in std_logic_vector(3 downto 0);
                    salidaDCH : out std_logic_vector(3 downto 0)
                    );
    END COMPONENT;

signal eIZQ1,eIZQ2,sIZQ1,sIZQ2,sReconf : std_logic_vector(3 downto 0);

signal auxSIZQ1,auxSIZQ2,auxsalida,auxSReconf : std_logic_vector(3 downto 0);

signal GND, Vcc: std_logic;

begin
terminado <= '0';
acepta <= '0';
GND <= '0';
Vcc <= '1';
            eIZQ1 <= operando1;
            eIZQ2 <= operando2;

            --------------------------------
            --Instancio todas las bus macro
            --------------------------------
            BM_operandoIZQ1: bm_4b PORT MAP(
              --GND activa y Vcc desactiva
    LI => sIZQ1,
              --
    LT(3) => GND,
    LT(2) => GND,
    LT(1) => GND,
    LT(0) => GND,
              -- ----------------------
              RI(3) => GND,  -- dato desechado
              RI(2) => GND,  -- dato desechado
              RI(1) => GND,  -- dato desechado
              RI(0) => GND,  -- dato desechado
              --
```

```
RT(3) => Vcc,
RT(2) => Vcc,
RT(1) => Vcc,
RT(0) => Vcc,
            -- ----------------------
            O => auxSIZQ1
          );

          BM_operandoIZQ2: bm_4b PORT MAP(
            --GND activa y Vcc desactiva
LI => sIZQ2,
            --
LT(3) => GND,
LT(2) => GND,
LT(1) => GND,
LT(0) => GND,
            -- ----------------------
            RI(3) => GND,  -- dato desechado
            RI(2) => GND,  -- dato desechado
            RI(1) => GND,  -- dato desechado
            RI(0) => GND,  -- dato desechado
            --
RT(3) => Vcc,
RT(2) => Vcc,
RT(1) => Vcc,
RT(0) => Vcc,
            -- ----------------------
            O => auxSIZQ2
          );

          BM_salidaR: bm_4b PORT MAP(
            --GND activa y Vcc desactiva
LI => sReconf,
            --
LT(3) => GND,
LT(2) => GND,
LT(1) => GND,
LT(0) => GND,
            -- ----------------------
            RI(3) => GND,  -- dato desechado
            RI(2) => GND,  -- dato desechado
            RI(1) => GND,  -- dato desechado
            RI(0) => GND,  -- dato desechado
            --
RT(3) => Vcc,
RT(2) => Vcc,
RT(1) => Vcc,
RT(0) => Vcc,
            -- ----------------------
            O => auxSReconf
```

```
);

-------------------------------------------------
--Instancio el módulo de la izquierda: ModIzquierda
-------------------------------------------------
ModuloIzquierda: ModIzquierda PORT MAP(

operandoIZQ1 => eIZQ1,

operandoIZQ2 => eIZQ2,

salidaIZQ1 => sIZQ1,

salidaIZQ2 => sIZQ2

);

-----------------------------------------------------------------
--Instancio el módulo del centro (el reconfigurable): ModReconfigurable
-----------------------------------------------------------------
ModuloReconfigurable: ModReconfigurable PORT MAP(

operandoR1 => auxSIZQ1,

operandoR2 => auxSIZQ2,

salidaR => sReconf

);

-----------------------------------------------------------------
--Instancio el módulo de la derecha: ModIzquierda
-----------------------------------------------------------------
ModuloDerecha: ModDerecha PORT MAP(

entradaDCH => auxSReconf,

salidaDCH => auxsalida

);
salida(0) <= auxsalida(0);
salida(1) <= auxsalida(1);
salida(2) <= auxsalida(2);
salida(3) <= auxsalida(3);
end Behavioral;
```

10.4.2.2. TOPJERARQUIA2.VHD

```
library IEEE;
```

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

entity TopJerarquia2 is
    Port ( operando1 : in std_logic_vector(3 downto 0);
           operando2 : in std_logic_vector(3 downto 0);
salida : out std_logic_vector(7 downto 0);
terminado : out std_logic;
acepta : out std_logic
                      );
end TopJerarquia2;

architecture Behavioral of TopJerarquia2 is

        COMPONENT bm_4b
        PORT(
                        LI : in  std_logic_vector (3 downto 0);
                        LT : in  std_logic_vector (3 downto 0);
                        RI : in  std_logic_vector (3 downto 0);
                        RT : in  std_logic_vector (3 downto 0);
                        O  : out std_logic_vector (3 downto 0)
                        );
        END COMPONENT;

        COMPONENT ModIzquierda
        PORT( operandoIZQ1 : in std_logic_vector(3 downto 0);
       operandoIZQ2 : in std_logic_vector(3 downto 0);
                        salidaIZQ1 : out std_logic_vector(3 downto 0);
                        salidaIZQ2 : out std_logic_vector(3 downto 0)
                        );
        END COMPONENT;

        COMPONENT ModReconfigurable2
        PORT( operandoR1 : in std_logic_vector(3 downto 0);
       operandoR2 : in std_logic_vector(3 downto 0);
                        salidaR : out std_logic_vector(3 downto 0)
                        );
        END COMPONENT;

        COMPONENT ModDerecha
        PORT( entradaDCH : in std_logic_vector(3 downto 0);
                        salidaDCH : out std_logic_vector(3 downto 0)
                        );
        END COMPONENT;

signal eIZQ1,eIZQ2,sIZQ1,sIZQ2,sReconf : std_logic_vector(3 downto 0);

signal auxSIZQ1,auxSIZQ2,auxsalida,auxSReconf : std_logic_vector(3 downto 0);
```

```
signal GND, Vcc: std_logic;

begin
terminado <= '0';
acepta <= '0';
GND <= '0';
Vcc <= '1';
                eIZQ1 <= operando1;
                eIZQ2 <= operando2;

                -------------------------------
                --Instancio todas las bus macro
                -------------------------------
                BM_operandoIZQ1: bm_4b PORT MAP(
                  --GND activa y Vcc desactiva
        LI => sIZQ1,
                  --
        LT(3) => GND,
        LT(2) => GND,
        LT(1) => GND,
        LT(0) => GND,
                  -- ----------------------
                  RI(3) => GND,  -- dato desechado
                  RI(2) => GND,  -- dato desechado
                  RI(1) => GND,  -- dato desechado
                  RI(0) => GND,  -- dato desechado
                  --
        RT(3) => Vcc,
        RT(2) => Vcc,
        RT(1) => Vcc,
        RT(0) => Vcc,
                  -- ----------------------
                  O => auxSIZQ1
                );

                BM_operandoIZQ2: bm_4b PORT MAP(
                  --GND activa y Vcc desactiva
        LI => sIZQ2,
                  --
        LT(3) => GND,
        LT(2) => GND,
        LT(1) => GND,
        LT(0) => GND,
                  -- ----------------------
                  RI(3) => GND,  -- dato desechado
                  RI(2) => GND,  -- dato desechado
                  RI(1) => GND,  -- dato desechado
                  RI(0) => GND,  -- dato desechado
                  --
        RT(3) => Vcc,
```

```
RT(2) => Vcc,
RT(1) => Vcc,
RT(0) => Vcc,
            -- ----------------------
            O => auxSIZQ2
           );

           BM_salidaR: bm_4b PORT MAP(
            --GND activa y Vcc desactiva
LI => sReconf,
            --
LT(3) => GND,
LT(2) => GND,
LT(1) => GND,
LT(0) => GND,
            -- ----------------------
            RI(3) => GND,  -- dato desechado
            RI(2) => GND,  -- dato desechado
            RI(1) => GND,  -- dato desechado
            RI(0) => GND,  -- dato desechado
            --
RT(3) => Vcc,
RT(2) => Vcc,
RT(1) => Vcc,
RT(0) => Vcc,
            -- ----------------------
            O => auxSReconf
           );

           ------------------------------------------------------
           --Instancio el módulo de la izquierda: ModIzquierda
           ------------------------------------------------------
           ModuloIzquierda: ModIzquierda PORT MAP(

                           operandoIZQ1 => eIZQ1,

                           operandoIZQ2 => eIZQ2,

                           salidaIZQ1 => sIZQ1,

                           salidaIZQ2 => sIZQ2

                           );

           ---------------------------------------------------------------
           --Instancio el módulo del centro (el reconfigurable): Reconfigurable
           ---------------------------------------------------------------
           ModuloReconfigurable: ModReconfigurable2 PORT MAP(

                           operandoR1 => auxSIZQ1,
```

```
                operandoR2 => auxSIZQ2,

                salidaR => sReconf

                );

        ----------------------------------------------------------------
        --Instancio el módulo de la derecha: ModIzquierda
        ----------------------------------------------------------------
        ModuloDerecha: ModDerecha PORT MAP(

                entradaDCH => auxSReconf,

                salidaDCH => auxsalida

                );
salida(0) <= auxsalida(0);
salida(1) <= auxsalida(1);
salida(2) <= auxsalida(2);
salida(3) <= auxsalida(3);

end Behavioral;
```

10.4.2.3. MODDERECHA.VHD

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

entity ModDerecha is
    Port ( entradaDCH : in std_logic_vector(3 downto 0);
           salidaDCH : out std_logic_vector(3 downto 0));
end ModDerecha;

architecture Behavioral of ModDerecha is

begin
        salidaDCH <= not (entradaDCH + 3);
end Behavioral;
```

10.4.2.4. MODRECONFIGURABLE.VHD

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

```
entity ModReconfigurable is
    Port ( operandoR1 : in std_logic_vector(3 downto 0);
           operandoR2 : in std_logic_vector(3 downto 0);
           salidaR : out std_logic_vector(3 downto 0));
end ModReconfigurable;

architecture Behavioral of ModReconfigurable is

begin
          salidaR <= operandoR1 + operandoR2;
end Behavioral;
```

10.4.2.5. MODRECONFIGURABLE2.VHD

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

entity ModReconfigurable2 is
    Port ( operandoR1 : in std_logic_vector(3 downto 0);
           operandoR2 : in std_logic_vector(3 downto 0);
           salidaR : out std_logic_vector(3 downto 0));
end ModReconfigurable2;

architecture Behavioral of ModReconfigurable2 is

begin
          salidaR <= operandoR1 - operandoR2;
end Behavioral;
```

10.4.2.6. MODIZQUIERDA.VHD

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

entity ModIzquierda is
    Port ( operandoIZQ1 : in std_logic_vector(3 downto 0);
           operandoIZQ2 : in std_logic_vector(3 downto 0);
           salidaIZQ1 : out std_logic_vector(3 downto 0);
           salidaIZQ2 : out std_logic_vector(3 downto 0));
end ModIzquierda;

architecture Behavioral of ModIzquierda is

begin
          salidaIZQ1 <= operandoIZQ1 + 5;
```

```
        salidaIZQ2 <= operandoIZQ2 + 2;
end Behavioral;
```

10.4.3. CÓDIGOS DE RESTRICCIONES ".UCF"

10.4.3.1. TOPJERARQUIA.UCF (PRUEBA CON PROJECT NAVIGATOR)

```
#Se definen las entradas y las salidas

NET "operando2(3)" LOC = "AL18" ;
NET "operando2(2)" LOC = "AK18" ;
NET "operando2(1)" LOC = "AJ18" ;
NET "operando2(0)" LOC = "AN19" ;
NET "operando1(3)" LOC = "AL19" ;
NET "operando1(2)" LOC = "AK19" ;
NET "operando1(1)" LOC = "AM20" ;
NET "operando1(0)" LOC = "AJ19" ;
NET "acepta" LOC = "AL20" ;

NET "salida(0)" LOC = "AN28";      #leds
NET "salida(1)" LOC = "AL26"  ;  #leds
NET "salida(2)" LOC = "AM27"  ; #leds
NET "salida(3)" LOC = "AK24"  ;  #leds


# area para el ModuloIzquierda

AREA_GROUP "areaIzq" RANGE = CLB_R20C10:CLB_R40C30 ;
INST "ModuloIzqu*" AREA_GROUP = "areaIzq" ;


# area para el ModuloReconfigurable

AREA_GROUP "areaRec" RANGE = CLB_R20C40:CLB_R40C60 ;
AREA_GROUP "areaRec" MODE = RECONFIG;
INST "ModuloReco*" AREA_GROUP = "areaRec" ;


# area para el ModuloDerecha

AREA_GROUP "areaDech" RANGE = CLB_R20C70:CLB_R40C90 ;
INST "ModuloDe*" AREA_GROUP = "areaDech" ;

#Definimos la localización específica de las bus macros:

INST "BM_operandoIZQ1" LOC = "TBUF_R20C12.0" ;
INST "BM_operandoIZQ2" LOC = "TBUF_R22C12.0" ;
INST "BM_salidaR" LOC = "TBUF_R24C44.0" ;
```

10.4.3.2. TOPJERARQUIA.UCF (PRUEBA CON SCRIPTS)

```
# area para el ModuloIzquierda
AREA_GROUP "areaIzq" RANGE = CLB_R20C12:CLB_R35C35 ;
AREA_GROUP "areaIzq" RANGE = TBUF_R20C12:TBUF_R35C34 ;
AREA_GROUP "areaIzq" MODE = RECONFIG;
INST "ModuloIzqu*" AREA_GROUP = "areaIzq" ;


# area para el ModuloReconfigurable
AREA_GROUP "areaRec" RANGE = CLB_R20C36:CLB_R35C59 ;
AREA_GROUP "areaRec" RANGE = TBUF_R20C36:TBUF_R35C58 ;
AREA_GROUP "areaRec" MODE = RECONFIG;
INST "ModuloReco*" AREA_GROUP = "areaRec" ;


# area para el ModuloDerecha
AREA_GROUP "areaDech" RANGE = CLB_R20C60:CLB_R35C83 ;
AREA_GROUP "areaDech" RANGE = TBUF_R20C60:TBUF_R35C82 ;
AREA_GROUP "areaDech" MODE = RECONFIG;
INST "ModuloDe*" AREA_GROUP = "areaDech" ;

#Definimos la localización específica de las bus macros:
INST "BM_operandoIZQ1" LOC = "TBUF_R20C32.0" ;
INST "BM_operandoIZQ2" LOC = "TBUF_R22C32.0" ;
INST "BM_salidaR" LOC = "TBUF_R24C56.0" ;

#Hay que cambiar los "<>" por "()" después de cambiar la propiedad "Bus
delimiter"
# del paso Synthesis para que funcionara lo del busmacro
#NET "terminado" LOC = "AN15" ;
#NET "salida(7)" LOC = "AJ15" ;
#NET "salida(6)" LOC = "AK15" ;
#NET "salida(5)" LOC = "AL15" ;
#NET "salida(4)" LOC = "AM16" ;
#NET "salida(3)" LOC = "AL16" ;
#NET "salida(2)" LOC = "AJ16" ;
#NET "salida(1)" LOC = "AK16" ;
#NET "salida(0)" LOC = "AN17" ;

NET "salida(0)" LOC = "AN28";      #leds
NET "salida(1)" LOC = "AL26"  ;  #leds
NET "salida(2)" LOC = "AM27"  ; #leds
NET "salida(3)" LOC = "AK24"  ;  #leds
```

```
NET "operando2(3)" LOC = "AL18" ;
NET "operando2(2)" LOC = "AK18" ;
NET "operando2(1)" LOC = "AJ18" ;
NET "operando2(0)" LOC = "AN19" ;
NET "operando1(3)" LOC = "AL19" ;
NET "operando1(2)" LOC = "AK19" ;
NET "operando1(1)" LOC = "AM20" ;
NET "operando1(0)" LOC = "AJ19" ;
NET "acepta" LOC = "AL20" ;
```

10.4.4. SCRIPTS "*.BAT*"

10.4.4.1. INITIAL.BAT (TOP)

```
ngdbuild  -p xcv1000-6bg560 -modular initial topjerarquia.edf
pause
```

10.4.4.2. INITIAL.BAT (TOP2)

```
ngdbuild  -p xcv1000-6bg560 -modular initial topjerarquia.edf
pause
```

10.4.4.3. ACTIVE.BAT (MÓDULO DERECHA)

```
ngdbuild  -p xcv1000-6bg560 -modular module -active modderecha
../../top/Initial/topjerarquia.ngo
map  -pr b topjerarquia.ngd -o topjerarquia_map.ncd  topjerarquia.pcf
par  -w -ol 5 -n 3 -s 3 topjerarquia_map.ncd mppr.dir topjerarquia.pcf
copy mppr.dir\4_4_3.ncd topjerarquia.ncd
bitgen -d -g ActiveReconfig:yes topjerarquia.ncd
trce topjerarquia.ncd topjerarquia.pcf
pimcreate -ncd topjerarquia.ncd -ngm topjerarquia_map.ngm ../../Pims
pause
```

10.4.4.4. ACTIVE.BAT (MÓDULO IZQUIERDA)

```
ngdbuild  -p xcv1000-6bg560 -modular module -active modizquierda
../../top/Initial/topjerarquia.ngo
map  -pr b topjerarquia.ngd -o topjerarquia_map.ncd  topjerarquia.pcf
par  -w -ol 5 -n 3 -s 3 topjerarquia_map.ncd mppr.dir topjerarquia.pcf
copy mppr.dir\4_4_3.ncd topjerarquia.ncd
bitgen -d -g ActiveReconfig:yes topjerarquia.ncd
trce topjerarquia.ncd topjerarquia.pcf
```

```
pimcreate -ncd topjerarquia.ncd -ngm topjerarquia_map.ngm ../../Pims
pause
```

10.4.4.5. ACTIVE.BAT (MÓDULO RECONFIGURABLE)

```
ngdbuild  -p xcv1000-6bg560 -modular module -active modreconfigurable
../../top/Initial/topjerarquia.ngo
map  -pr b topjerarquia.ngd -o topjerarquia_map.ncd  topjerarquia.pcf
par  -w -ol 5 -n 3 -s 3 topjerarquia_map.ncd mppr.dir topjerarquia.pcf
copy mppr.dir\4_4_3.ncd topjerarquia.ncd
bitgen -d -g ActiveReconfig:yes topjerarquia.ncd
trce topjerarquia.ncd topjerarquia.pcf
pimcreate -ncd topjerarquia.ncd -ngm topjerarquia_map.ngm ../../Pims
pause
```

10.4.4.6. ACTIVE.BAT (MÓDULO RECONFIGURABLE2)

```
ngdbuild  -p xcv1000-6bg560 -modular module -active modreconfigurable2
../../top2/Initial/topjerarquia.ngo
map  -pr b topjerarquia.ngd -o topjerarquia_map.ncd  topjerarquia.pcf
par  -w -ol 5 -n 3 -s 3 topjerarquia_map.ncd mppr.dir topjerarquia.pcf
copy mppr.dir\4_4_3.ncd topjerarquia.ncd
bitgen -d -g ActiveReconfig:yes topjerarquia.ncd
trce topjerarquia.ncd topjerarquia.pcf
pimcreate -ncd topjerarquia.ncd -ngm topjerarquia_map.ngm ../../Pims
pause
```

10.4.4.7. ASSEMBLE.BAT (TOP)

```
ngdbuild  -p xcv1000-6bg560 -modular assemble -pimpath ../../Pims
topjerarquia.edf
map -pr b  topjerarquia.ngd -o topjerarquia_map.ncd  topjerarquia.pcf
par  -w  topjerarquia_map.ncd topjerarquia.ncd  topjerarquia.pcf
bitgen topjerarquia.ncd
trce topjerarquia.ncd topjerarquia.pcf
pause
```

10.4.4.8. ASSEMBLE.BAT (TOP2)

```
ngdbuild  -p xcv1000-6bg560 -modular assemble -pimpath ../../Pims
topjerarquia.edf
map -pr b  topjerarquia.ngd -o topjerarquia_map.ncd  topjerarquia.pcf
par  -w  topjerarquia_map.ncd topjerarquia.ncd  topjerarquia.pcf
bitgen topjerarquia.ncd
trce topjerarquia.ncd topjerarquia.pcf
pause
```

11. BIBLIOGRAFÍA

- RC1000 Hardware Reference Manual. Versión 2.3

  Celoxica

  http://www.celoxica.com

- RC1000 Software Reference Manual. Versión 1.3

  Celoxica

  http://www.celoxica.com

- RC1000 Functional Reference Manual. Versión 1.3

  Celoxica

  http://www.celoxica.com

- Integrating Handel-C with VHDL, Verilog and EDIF.

  Celoxica

  http://www.celoxica.com

- Handel-C language reference.

  Celoxica

  http://www.celoxica.com

- Two Flows for Partial Reconfiguration: Module Based or Difference Based. XAPP290 (v1.1) November 25, 2003

  Xilinx

  http://www.xilinx.com

- Advanced Architecture Configuration Techniques. UG001 (v1.0) October 9, 2000

  Xilinx

  http://www.xilinx.com

- BitGen and PROMGen Program Information. UG001 (v1.0) October 9, 2000

  Xilinx

  http://www.xilinx.com

- Development System Reference Guide

  Xilinx

  http://toolbox.xilinx.com/docsan/xilinx6/books/data/docs/dev/dev0001_1.html

  Índice:
  http://toolbox.xilinx.com/docsan/xilinx6/books/data/docs/dev/dev0006_3.html

- Examples of the partial reconfiguration flow in VHDL

  Xilinx

  http://www.xilinx.com/bvdocs/appnotes/xapp290.zip

  Éste es el que hemos utilizado:

  ftp://ftp.xilinx.com/pub/applications/xapp/xapp290.zip

- Procesado de imágenes digitales sobre un sistema hardware dinámicamente reconfigurable.

Isaac Álvarez, Sergio Villalobos, Gerardo Robledillo, Mª Pilar Rodríguez

Proyecto de Sistemas Informáticos. Año 2002-2003

- Foro en internet

  Imperial College London. Faculty of Engineering, Department of Computing

  http://www.doc.ic.ac.uk/~akf99/handel-c/cgi-bin/forum.cgi