

Tutorial FPGA-SoC

Contenido

Instalación de Quartus prime standard edition v 16.0.0.....	3
Requisitos previos.....	3
Instalación.....	4
Crack.....	5
Instalación de Suite de desarrollo embebido.....	5
USB-blaster.....	6
Variables de entorno.....	7
Comunicación Serial con Sistema operativo.....	8
Interfaz y funcionamiento de Quartus II.....	9
New Project Wizard.....	10
Ejemplo de un diseño simple en VHDL.....	13
Compilación y programación del dispositivo.....	16
Configuración previa.....	16
Análisis y Síntesis.....	16
Simulación.....	17
Visor de RTL.....	20
Asignación de Pines.....	21
Compilación.....	23
Establecer modo de configuración de FPGA.....	24
Programador.....	25
Guardar cadena de configuración.....	27
Ejemplo de una máquina de estados.....	28
Configuración de VHDL.....	29
Flip-Flop con Reset.....	29
Lógica de transiciones entre estados.....	31
Integración de módulos.....	32
Archivo SDC.....	34
Simulación de máquina de estado.....	35
Comunicación entre FPGA y ARM.....	36
Qsys.....	36
Sistema de Procesador de núcleo Duro (HPS).....	37
Periféricos paralelos de entrada y salida (PIO).....	39
Implementación de comunicación ARM-FPGA.....	43
1. síntesis de componentes en Qsys.....	43
2. Incorporación de elementos generados.....	48
3. Incorporación de un diseño de hardware personalizado.....	49
4. Asignación de pines y parámetros.....	50
5. Convertir archivos de programación.....	53
6. Crear sistema de precarga.....	54
7. Compilación del sistema de arranque.....	57
8. Generar y compilar el árbol de direcciones.....	59
9. Crear particiones y el sistema de archivos.....	60
10. Probar el sistema de arranque.....	64

11. Compilación del Kernel.....	65
12. Generación del sistema de archivos raiz.....	67
13: Integración del sistema operativo.....	71
14. Aplicaciones de software.....	73
14. Ejecución del código.....	77
Módulo de cómputo dedicado.....	78
Descripción del sistema.....	78
Implementación.....	80
Código de aplicación.....	80

Instalación de Quartus prime standard edition v 16.0.0

Requisitos previos

En sistemas operativos Windows, Quartus Prime no requiere de la instalación de software adicional. Sin embargo algunos el simulador ModelSim requiere la instalación de Microsoft Visual C++ 2013 Redistributable Package (x86) así como de algunas bibliotecas de 32-bits, además de que este programa no tiene soporte para Windows 10. Asimismo, el programa Intel SoC FPGA Embedded Development Suite tampoco tiene soporte para Windows 10.

En sistemas operativos Linux, se requiere de un entorno de escritorio Gnome o KDE, así como de las siguientes dependencias:

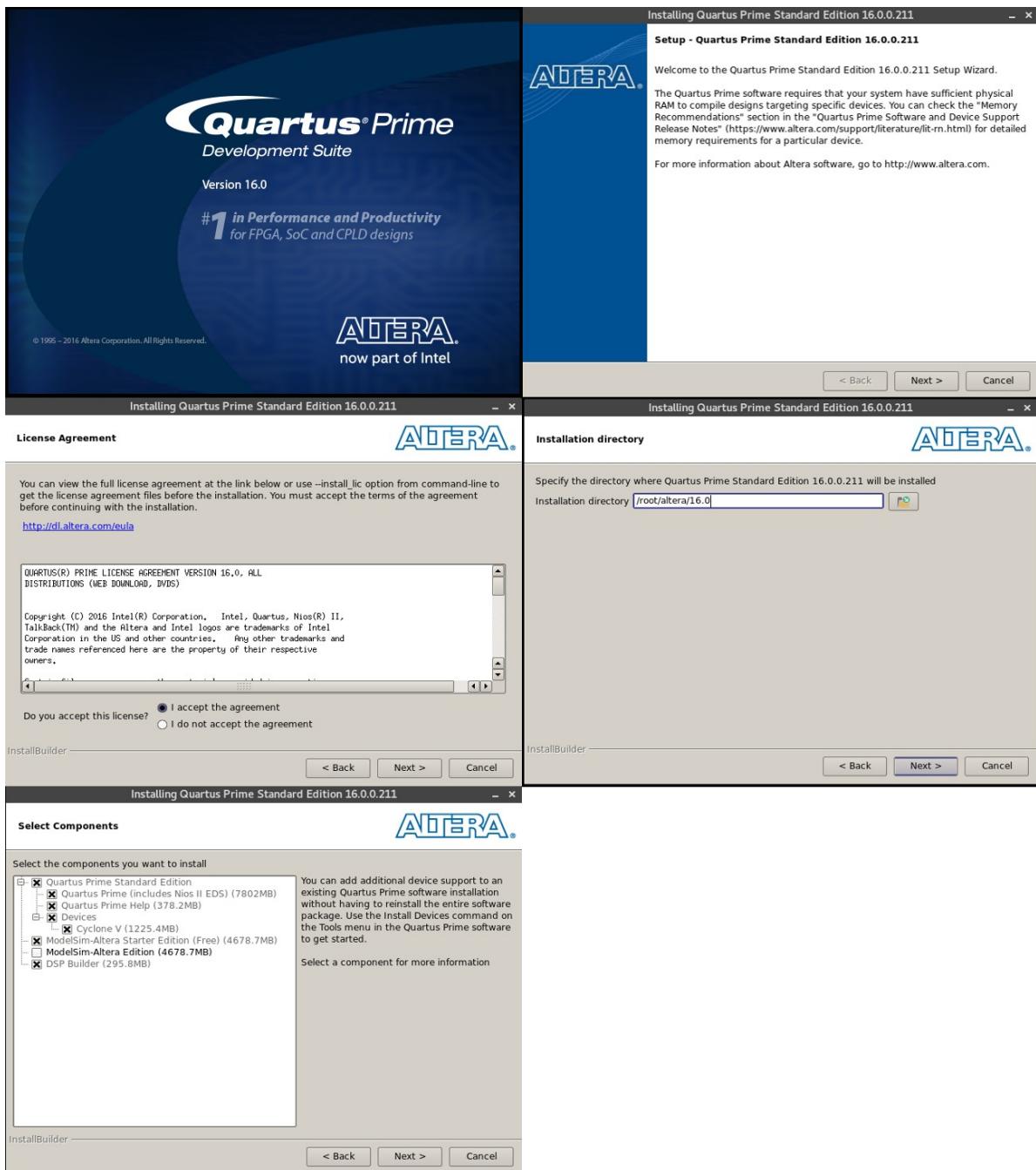
REHL/CentOS: make , libX11.i686 , libXau.i686 , libXdmcp.i686, libXext.i686 , libXft-devel.i686 , libXft.i686 , libXrender.i686 , libXt.i686 , libXtst.i686, unixODBC-libs, unixODBC, ncurses, ncurses-libs, libzmq3, libXext, alsa-lib, libXtst, libXft, libxml2, libedit, libX11, libXi

Ubuntu: libc6:i386 , libncurses5:i386 , libxtst6:i386 , libxft2:i386 , libc6:i386 , libncurses5:i386 , libstdc++6:i386 , libc6-dev-i386 libxft2 , lib32z1 , lib32ncurses5 , lib32bz2-1.0 , libpng12

Para Ubuntu 18 o superior, se requiere adicionalmente de las bibliotecas libqt5xml5 y liblzma-dev. Además, para estas versiones de Ubuntu libpng12 ya no se encuentra disponible mediante apt-get. El instalador de esta biblioteca puede descargarse en la siguiente dirección:
<https://packages.ubuntu.com/xenial/amd64/libpng12-0/download>

Instalación

El primer paso en la instalación es ejecutar el instalador (QuartusSetup-16.0.0.211-linux.run o QuartusSetup-16.0.0.211-windows.exe dependiendo del sistema operativo). Dentro del instalador, acceda al acuerdo de licencia, seleccione el directorio de instalación y seleccione los programas a instalar. En la etapa de selección de programas, asegúrese de tener activada la opción de ModelSim edición gratuita, el dispositivo Cyclone V y el programa DSP builder.



El instalador, pregunta si se tiene una instalación de matlab. Esta no será necesaria para propósitos de este tutorial, por lo que puede omitirse. Para terminar la instalación, asegúrese de que las opciones “Provide your Feedback” y “Launch Quartus Prime Standard edition” **NO estén activas**.

Crack

Para crackear el programa en linux, es necesario que la interfaz ethernet se llame “eth0”. En la mayoría de las distribuciones, esta interfaz tiene otro nombre y por lo tanto hay que cambiarlo. Para hacer esto, siga las instrucciones según su distribución:

Para Ubuntu 16.04 o Ubuntu 18.04

<https://www.itzgeek.com/how-tos/mini-howtos/change-default-network-name-ens33-to-old-eth0-on-ubuntu-16-04.html>

Para RHEL/CentOS7:

<https://www.thegeekdiary.com/centos-rhel-7-how-to-modify-network-interface-names/>

Habiendo cambiado el nombre de la interfaz ethernet, ubique el archivo “license.dat” en la carpeta crack. Con un editor de texto (i.e., vim, nano, gedit, etc.) abra el archivo y reemplace todas las instancias de “XXXXXXXXXXXXXX” por su dirección la dirección MAC de su interfaz ethernet. Para obtener esta dirección, abra una terminal y escriba “ifconfig eth0”. Dentro la salida de este comando, encontrará un número de 12 dígitos separados en pares por “：“, por ejemplo 83:7b:ea:45:55:6f. Su dirección mac es este número en minúscula sin los “：“.

En el mismo directorio crack, ubique los archivos “libgcl_afcq.so” y “libsys_cpt.so”. Copie estos archivos a la carpeta /root/altrea/16.0/quartus/linux64 reemplazando los archivos ya existentes.

Habiendo completado los pasos anteriores, ejecute Quartus. Durante el arranque de la aplicación aparecerá una ventana con opciones de activación de licencia. Seleccione la opción de archivo de licencia local. En la ventana subsecuente, seleccione el archivo “ license.dat”, abra el archivo y de click en OK. Con esto finaliza la activación de la licencia.

Para crackear en Windows, ejecute la aplicación Quartus_Prime_Standard_Pro_16.0_Update2_破解器.exe. Dentro del programa, abra la opción de buscar, ubique el archivo gcl_afcq.dll y de click en abrir. Por defecto, este archivo debe estar en la carpeta C:\altera\16.0\quartus\bin64. Seleccione la opción siguiente, y luego terminar.

Con un editor de texto plano, abra el archivo license.dat y reemplace todas las instancias de “XXXXXXXXXXXXXX” por la ID de su tarjeta de red, en minúsculas, sin “：“ y sin espacios.

Habiendo completado los pasos anteriores, ejecute Quartus. Durante el arranque de la aplicación aparecerá una ventana con opciones de activación de licencia. Seleccione la opción de archivo de licencia local. En la ventana subsecuente, seleccione el archivo “ license.dat”, abra el archivo y de click en OK. Con esto finaliza la activación de la licencia.

Instalación de Suite de desarrollo embebido.

En Linux, ejecute el instalador SoCEDSSetup-16.0.0.211-linux.run, y siga las instrucciones del programa instalador. La misma licencia de Quartus sirve para este programa, por lo que no se requiere de un crack adicional.

Para Windows, se requiere instalar algunos paquetes adicionales para ejecutar parte del software de Linux que se utiliza en el desarrollo de sistemas embebidos. Para esto, entre a <https://cygwin.com/> y descargue la versión de 64-bits de cygwin (setup-x86_64.exe), abra un editor de texto (por ejemplo note pad) y pegue el siguiente comando:

```
<Path to Cygwin Installer>\setup-x86_64.exe --wait --quiet-mode --root C:\Altera\16.0\embedded\host_tools\cygwin --site http://cygwin.mirrors.hoobly.com --packages make,gcc-core,gcc-g+ +,ncurses,inetutils,openssh,mosh,patch,flex,bison,tar,bzip2,zip,unzip,util-linux,git,subversion,vim,xxd,m4,wget,dos2unix,libintl-devel,diffutils,libncurses-devel,iperf,xorg-server,xinit,mingw64-x86_64-gcc-core,mingw64-x86_64-gcc-g++
```

Dentro del editor de texto, reemplace el término <Path to Cygwin Installer> por la ubicación del archivo de instalación de cygwin. Si la instalación de SoCEDS no se hizo en el directorio predeterminado, reemplace C:\Altera\16.0\embedded\host_tools\cygwin por la ubicación en la que se haya instalado. Después de reemplazar estas líneas, asegúrese de que todo el texto esté en una sola linea, copie todo, péguelo en una terminal de windows y presione Enter para ejecutar.

USB-blaster

La programación de los dispositivos FPGA se realiza mediante el protocolo Jtag. Para programar estos dispositivos desde una PC, las tarjetas de Altera/Intel utilizan la interfaz USB-blaster™. Las tarjetas DE1 y DE0, entre otras, incorporan un módulo USB-blaster, de modo que pueden programarse con un cable USB estándar.

Los controladores para esta interfaz en windows se pueden instalar como se muestra en la siguiente liga:

<https://www.intel.com/content/www/us/en/programmable/support/support-resources/download/drivers/usb-blaster/dri-usb-blaster-vista.html>

En Linux no hay necesidad de instalar ningún controlador, pues estos ya vienen incorporados en el kernel de la mayoría de las distribuciones, sin embargo hay que crear un archivo de reglas para utilizar esta interfaz. Este proceso varía según la distribución.

En Ubuntu 18 o superior: Con privilegios de administrador Con privilegios de administrador, se crea el archivo **/etc/udev/rules.d/92-usbblaster.rules** y dentro de éste se añaden las siguientes líneas:

```
# USB-Blaster
SUBSYSTEM=="usb", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6001", MODE=="0666"
SUBSYSTEM=="usb", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6002", MODE=="0666"

SUBSYSTEM=="usb", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6003", MODE=="0666"

# USB-Blaster II
SUBSYSTEM=="usb", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6010", MODE=="0666"
SUBSYSTEM=="usb", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6810", MODE=="0666"
```

En **RHEL/CentOS 5 o superior**, así como en **Ubuntu 12.04/14.04/16.04**: Con privilegios de administrador, se crea el archivo **/etc/udev/rules.d/51-usbblaster.rules** y dentro de éste se añaden las siguientes lineas:

```
# Intel FPGA Download Cable
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6001", MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6002", MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6003", MODE="0666"

# Intel FPGA Download Cable II
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6010", MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6810", MODE="0666"
```

En RHEL/CentOS 4 o inferior: Con privilegios de administrador, se modifica el archivo **/etc/hotplug/usb.usermmap** añadiendo las siguientes lineas:

```
# Intel FPGA Download Cable
#
usbblaster 0x03 0x09fb 0x6001 0x0 0x0
usbblaster 0x03 0x09fb 0x6002 0x0 0x0
usbblaster 0x03 0x09fb 0x6003 0x0 0x0
usbblaster 0x03 0x09fb 0x6010 0x0 0x0
usbblaster 0x03 0x09fb 0x6810 0x0 0x0
```

Luego se crea el archivo **/etc/hotplug/usb/usbblaster** al cual se añaden las siguientes lineas:

```
#!/bin/sh
# Intel FPGA Download Cable hotplug script
# Allow any user to access the cable
chmod 666 $DEVICE
```

Finalmente, haga que este archivo sea ejecutable con el comando:

```
sud chmod +x etc/hotplug/usb/usbblaster
```

Variables de entorno

Para poder ejecutar Quartus desde la terminal sin tener que estar en el directorio de la aplicación es necesario crear variables de entorno para estos programas. En linux, la forma de hacer esto es agregar las siguientes lineas al archivo “.bashrc” que se encuentra dentro del directorio raíz del usuario en el que se instaló Quartus.

```
export QSYS_ROOTDIR="/root/altera/16.0/quartus/sopc_builder/bin"
export ALTERAPATH="/root/altera/16.0/"
export QUARTUS_ROOTDIR=${ALTERAPATH}/quartus
export QUARTUS_ROOTDIR_OVERRIDE="$QUARTUS_ROOTDIR"
export PATH=$PATH:${ALTERAPATH}/quartus/bin
export PATH=$PATH:${ALTERAPATH}/nios2eds/bin
export PATH=$PATH:${ALTERAPATH}/embedded
```

Cierre y vuelva a abrir su sesión para hacer efectivos los cambios.

En windows, para establecer las variables de entorno necesarias, seguimos las instrucciones en los siguientes tutoriales según la versión de Windows.

Windows 7

<https://www.nextofwindows.com/how-to-addedit-environment-variables-in-windows-7>

Windows 8/10

<https://helpdeskgeek.com/windows-10/add-windows-path-environment-variable/>

Comunicación Serial con Sistema operativo

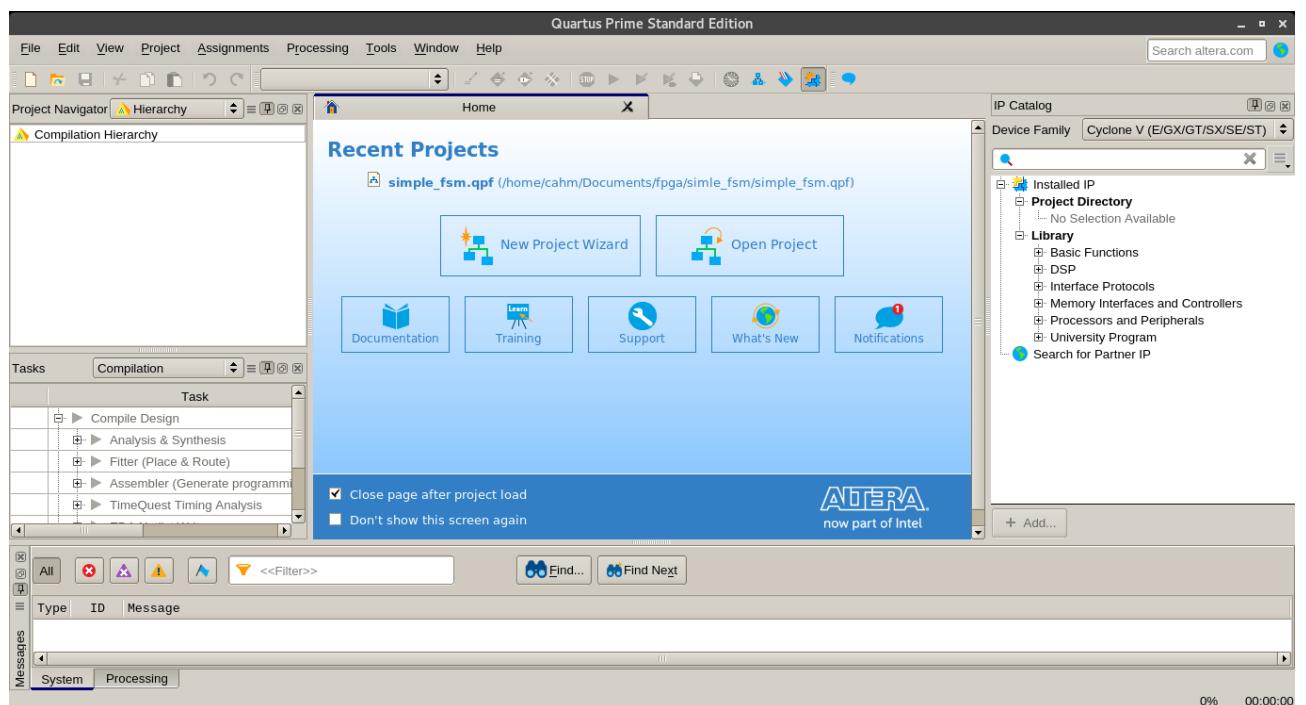
Es posible ejecutar un sistema operativo en el procesador ARM del dispositivo CycloneV de las tarjetas DE1 y DE0. Para esto, podemos usar una serie de programas de comunicación serial.

Tanto para Windows como para Linux, podemos usar el programa Putty.

En Windows, descargue el instalador de la página <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html> y siga las instrucciones del instalador. Para Linux puede usar directamente el manejador de paquetes del sistema. Por ejemplo, “sudo apt-get install putty” en Ubuntu y “sudo yum install putty” en CentOS.

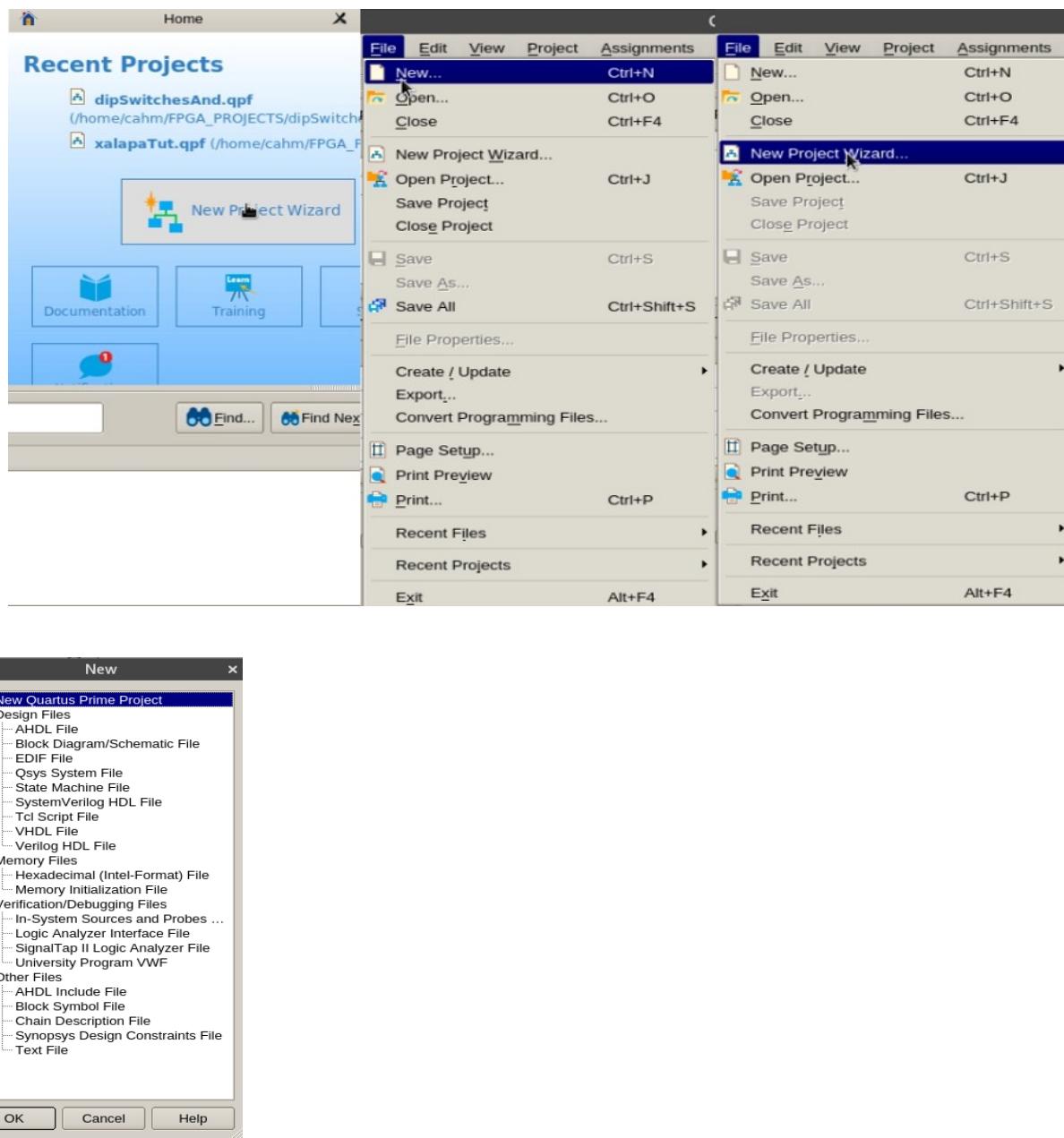
Interfaz y funcionamiento de Quartus II

La interfaz de Quartus tiene diferentes pestañas con las que podemos interactuar. Al centro, tenemos la pestaña de home, en la que se muestran los proyectos recientes, y nos da la opción de crear un proyecto nuevo. A lo largo del fondo de la ventana se encuentra la consola que nos permitirá depurar el proyecto. A la derecha se nos muestra el catálogo de IP's, que son módulos prediseñados a los cuales podemos acceder. Del lado izquierdo en la parte de arriba tenemos el navegador de proyecto, el cual nos muestra los archivos y la jerarquía del diseño, y en la parte inferior el navegador de tareas que nos permite ver el progreso de las diferentes etapas de compilación (Fig. 1).



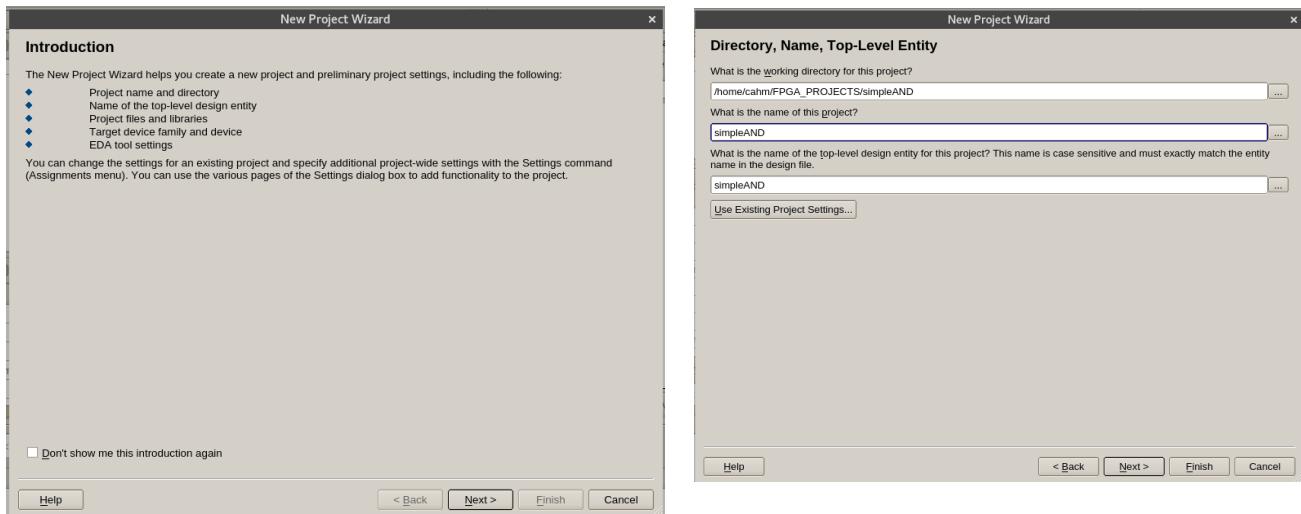
New Project Wizard

Para crear un proyecto, podemos dar click en “New Project Wizard” en la pestaña home, o podemos abrir en el menú File→New Project Wizard, o podemos abrir en el menú File→ New... Las primeras dos opciones nos llevan directamente a la creación de un proyecto nuevo, mientras que la siguiente nos abre la ventana “New”. Ésta ventana nos da la opción de crear un proyecto nuevo, así como los archivos de proyecto que utilizaremos más adelante.



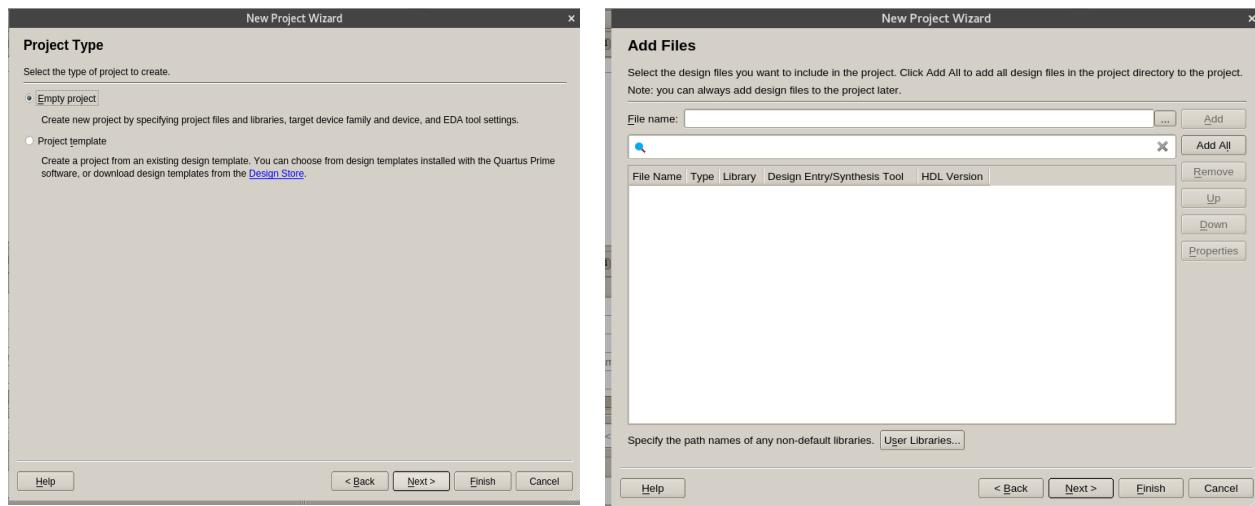
Dentro de la aplicación de proyecto nuevo, lo primero que vemos es una ventana de introducción. Si no queremos que nos la muestre todo el tiempo, seleccione la opción “Don’t show me this introduction again”. En el siguiente paso nos pedirá que designemos un directorio para el proyecto,

el nombre del proyecto y la entidad tope. Esta última se refiere al componente que integra a todos los módulos del proyecto, cómo se explicará más adelante. Es preferible que tanto el proyecto como la entidad top y el directorio del proyecto tengan el mismo nombre para evitar confusión y errores, aunque no es estrictamente necesario.



En el siguiente paso, tenemos la opción de crear un proyecto vacío o usar un template. Para este ejemplo crearemos un proyecto vacío.

Después tenemos la opción de algún archivo al proyecto. Esta opción permite utilizar archivos que hayamos creado previamente. Para este ejemplo no añadiremos ningún archivo .

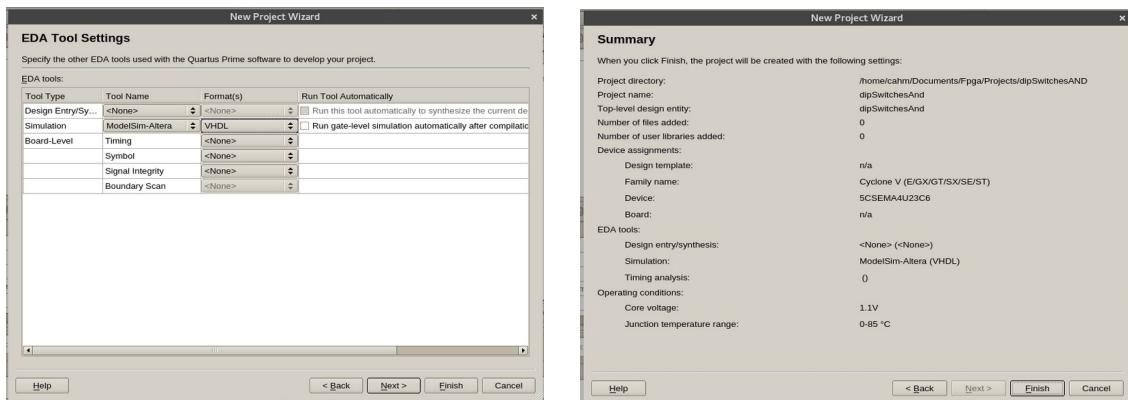


Seguidamente, entraremos al menú de selección de familia, dispositivo y tarjeta. En este menú hay dos opciones que se presentan en las pestañas. En la pestaña “Device” podemos buscar el dispositivo FPGA que contiene nuestra tarjeta, mientras que en la pestaña “Board” podemos buscar como tal la tarjeta con la que estamos trabajando. La diferencia es que si elegimos la tarjeta Quartus añade automáticamente al proyecto una serie de archivos de configuración específicos. Mientras que si elegimos el dispositivo no se añade ningún archivo.



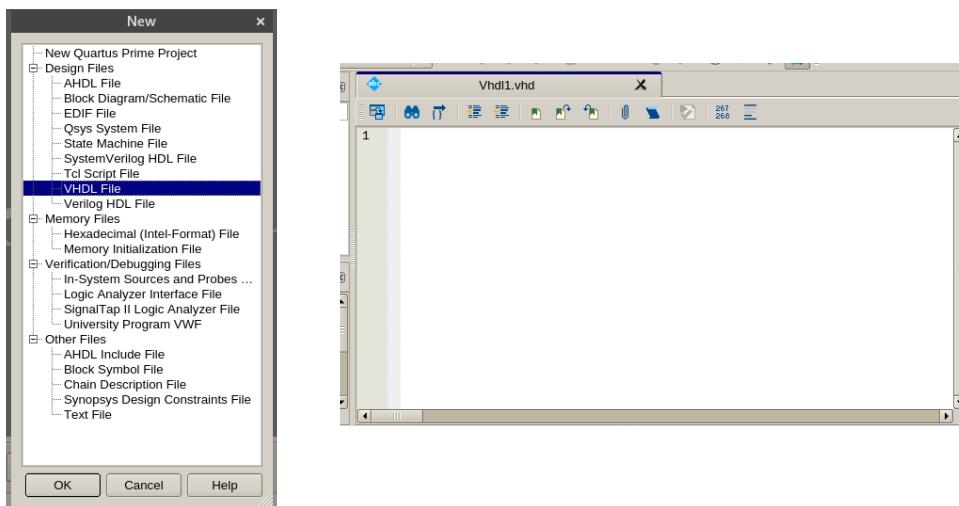
Para este ejemplo, buscaremos el FPGA como dispositivo, con el fin de mostrar cómo se hacen algunas de las configuraciones básicas. En el caso de que estemos usando la tarjeta **DE1-SoC**, el dispositivo correspondiente es el **5CSEMA5F31C6**, u opcionalmente podemos seleccionar la tarjeta con el nombre **DE1-SOC**. Para la tarjeta **DE0-SoC**, el dispositivo es corriente es el **5CSEMA4U23C6N**, o podemos seleccionar la tarjeta aparece con el nombre **Atlas-SoC**.

En el siguiente menú elegiremos la configuración de las herramientas de Automatización de Diseño Electrónico (EDA por sus siglas en inglés). Aquí podemos elegir usar herramientas de software provistas por intel o por terceros. La única herramienta que tenemos instalada y que usaremos será la herramienta de simulación ModelSim, la cual se selecciona en la opción de “Simulation”. En la columna “Properties” indicaremos al simulador el lenguaje que utilizaremos para describir los componentes. En este caso, seleccionaremos el lenguaje VHDL sobre el cual se habla en la siguiente sección. En este punto, podemos finalizar la creación del proyecto o dar siguiente para ver el resumen y cerciorarnos de que todo está en orden.

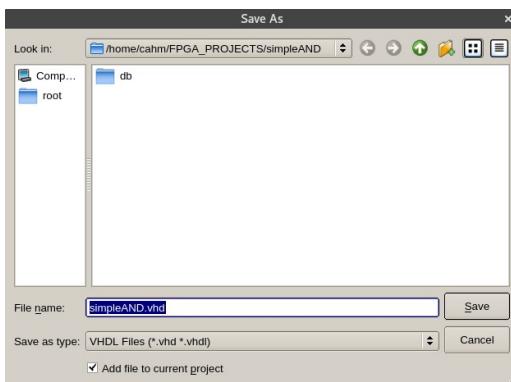


Ejemplo de un diseño simple en VHDL

Despues de completar los pasos anteriores, crearemos un archivo VHDL en blanco. VHDL es un lenguaje de diseño de hardware ampliamente utilizado, que emplearemos a lo largo de este tutorial. Para hacer esto, abrimos el menú “New” (Ctrl+N o File → New...) y en seleccionaremos la opción “**VHDL File**”.



Guardamos con Ctrl+S y automáticamente nos aparece la opción de guardarlo con el nombre de la entidad top.



En nuestro código de VHDL, lo primero que hacemos es importar la biblioteca IEEE y declarar que usaremos las funciones de lógica estándar contenidas en dicha biblioteca. Esto lo hacemos como se muestra en el ejemplo 1.

```
library IEEE;
use IEEE.std_logic_1164.all;
Ejemplo 1. Importación de bibliotecas.
```

El siguiente paso es declarar una entidad, que es como se denomina al módulo de hardware descrito en el archivo y que se declara como se muestra en el ejemplo 2.

```
entity simpleAND is
    port(
        in1,in0: in std_logic;
        ou0 : out std_logic
    );
end simpleAND;
```

Ejemplo 2. Declaración de entidad.

Como se muestra en el ejemplo 2, abrimos esta declaración como “**entity nombreDeLaEntidad is**”, siendo “simpleAnd” el nombre de la entidad en este ejemplo. Nótese que VHDL no distingue entre minúsculas o mayúsculas y que estas se usan sólo a fin de facilitar la lectura. Dentro de la declaración de “entity” se añade el atributo “port” el cual define las conexiones (i.e., puertos) que conforman la interfaz entre la entidad y otros componentes. De manera general, cada puerto se describe de la siguiente manera: “**nombreDelPuerto : direccionalidad tipo;**”. La direccionalidad indica si el puerto es una entrada (in) o una salida (out), mientras que el tipo indica qué clase de datos procesa el puerto, por ejemplo valores lógicos (std_logic) o vectores de valores lógicos (std_logic_vector).

Como se muestra en el ejemplo 2, en caso de que varios puertos tengan el mismo tipo y direccionalidad se pueden escribir en la misma linea de la siguiente manera “**nombreDelPuerto1,nombreDelPuerto2 : direccionalidad tipo;**” Aunque también es posible escribir

“**nombrePuerto1: direccionalidad tipo;**
“**nombrePuerto2 : direccionalidad tipo;**”

Una particularidad de VHDL es que el último puerto listado no debe de tener punto y coma, como se muestra en el ejemplo 2. Agregarle una coma al último puerto es un error muy común, como se muestra en el ejemplo 3.

```
port(  
    in1,in0: in std_logic;  
    ou0 : out std_logic;  
);
```

Ejemplo 3. Error de sintaxis en la definición de un puerto.

En el ejemplo 3, la coma al final de la tercera linea provoca que el diseño compile y el debugger nos arroje el siguiente error.

10500 VHDL synthax error at dipSwitchesAnd.vhd(8) near text “**”;**”; expecting an identifier, or “constant”, or “file”, or “variable”

Finalmente, cerramos la declaración de entidad escribiendo “**end nombreDeLaEntidad;**”.

Después de la declaración de entidad, se define la arquitectura del componente como se muestra en el ejemplo 4. En esta declaración definimos el comportamiento de la entidad.

```
architecture struct of simpleAND is  
begin  
    ou0<=in1 AND in0;  
end struct;
```

Ejemplo 4. Declaración de una arquitectura.

En la primera linea de esta declaración, se define el nombre de la arquitectura y a qué componente pertenece. Como se muestra en el ejemplo 4, se describe como “**architecture**

nombreDeLaArquitectura of nombreDeLaEntidad is". En este ejemplo, el nombre de la arquitectura es "struct" y el nombre de la entidad es "simpleAnd".

Con la linea "**begin**" se divide la sección de arquitectura en dos partes. Antes de ésta linea se declaran conexiones y se definen a otros componentes que pueden ser llamados en el diseño (lo cual no es necesario en esta demostración) y después de la linea mencionada anteriormente se inicia la descripción del comportamiento de la arquitectura en cuestión. En el ejemplo 4 se muestra como la salida del puerto "ou0" es el resultado de la operación lógica "in1 AND in0". Finalmente, la arquitectura se cierra escribiendo "**end nombreDeLaArquitectura;**".

El diseño terminado se muestra en el ejemplo 5.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity simpleAND is
    port(
        in1,in0: in std_logic;
        ou0 : out std_logic
    );
end simpleAND;
architecture struct of simpleAND is
begin
    ou0<=in1 AND in0;
end struct;
```

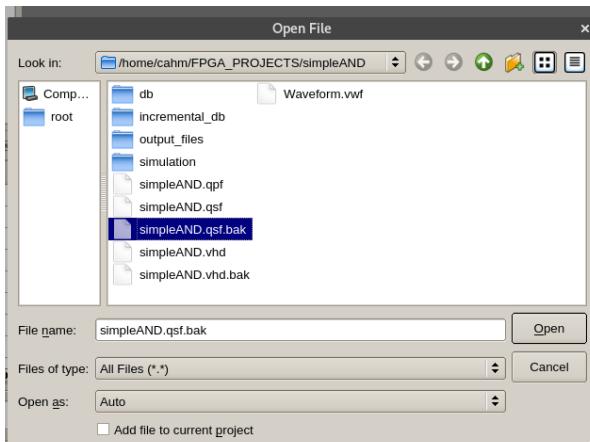
Ejemplo 5. Implementación compuerta AND en VHDL

Compilación y programación del dispositivo

En esta sección, veremos los pasos para compilar el diseño descrito en la sección anterior y cargar el binario generado en el FPGA.

Configuración previa

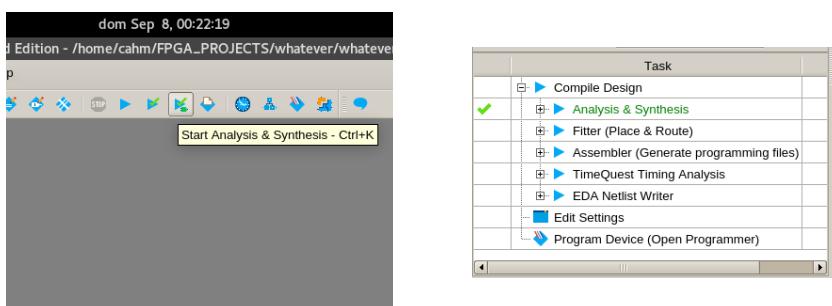
Antes de iniciar la compilación, es recomendable especificar el número de procesadores paralelos que se pueden usar para la compilación. Para hacer esto abrimos el archivo qsf del proyecto como se muestra a continuación.



En este archivo, añadimos la siguiente linea: **set_global_assignment -name NUM_PARALLEL_PROCESSORS 4**. Esto reduce el tiempo de compilación y evita que el debugger nos arroje la siguiente advertencia: **“Warning (18236): Number of processors has not been specified...”**.

Análisis y Síntesis

Quartus maneja un esquema de compilación incremental que permite realizar por separado las diferentes etapas de la compilación. La primera etapa de este proceso es la de análisis y síntesis, en la que se genera una representación en términos de compuertas lógicas y las conexiones entre ellas. Para realizar el análisis y síntesis, haga click en el ícono de “Start Analysis & Synthesis” o presione Ctrl+k.

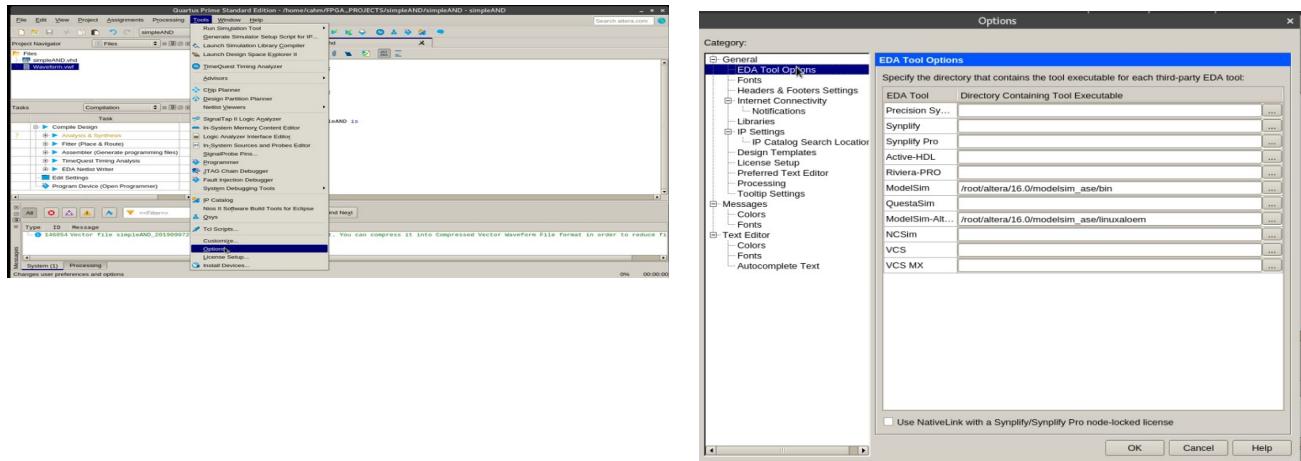


En el navegador de tareas, podemos ver si éste proceso se realizó exitosamente.

Simulación

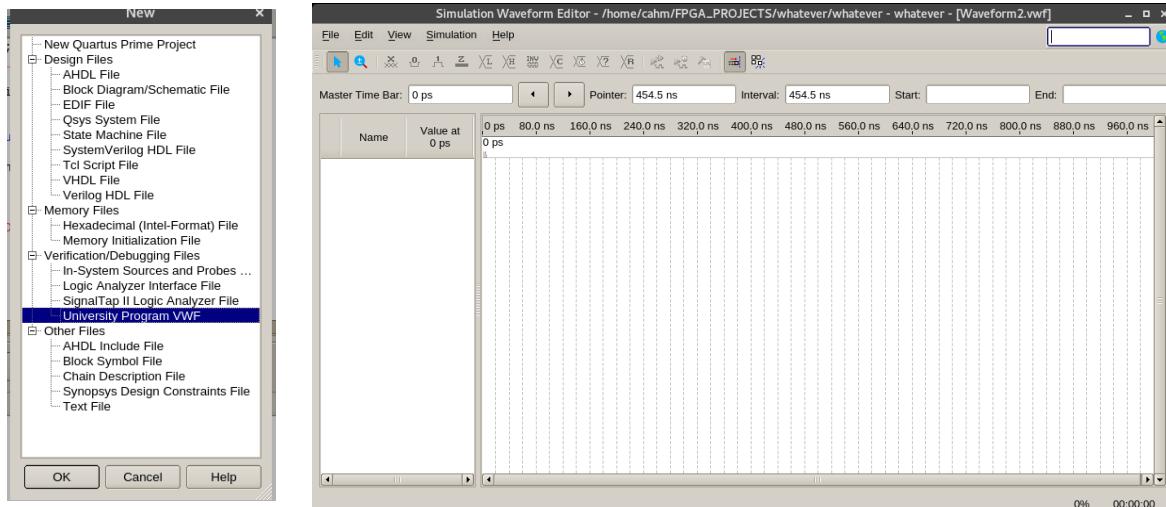
Habiendo completado el proceso de análisis y síntesis, es recomendable simular el comportamiento del diseño antes de completar la compilación.

Antes de simular el diseño, hay que asegurarnos de que Quartus tenga identificada la ubicación de los ejecutables de las herramientas de ModelSim. Damos click en Tools → Options y en la ventana emergente seleccionamos la opción de EDA Tools Options.



El ejecutable de Modelsim debe estar ubicado en **/root/altera/16.0/modelsim_ase/bin**, mientras que el ejecutable de ModelSim-Altera debe estar ubicado en **/root/altera/16.0/modelsim_ase/linuxaloem**. Si este es el caso, de click en OK y continúe.

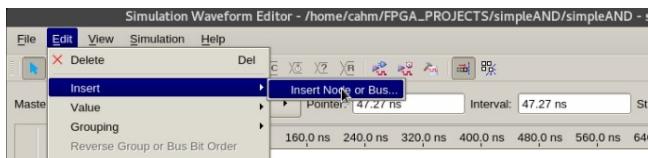
Para iniciar la simulación, abra el menú “New” y cree un archivo tipo “University Program VWF”. Esto abrirá la aplicación “Simulation Waveform Editor” en la cual podemos generar ondas que definen el comportamiento de las entradas en un intervalo de tiempo.



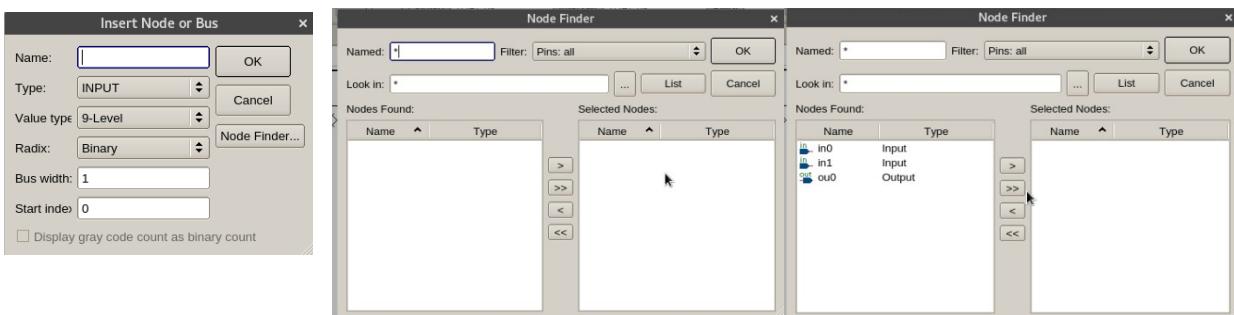
Guarde el archivo de la onda antes de continuar, el nombre por defecto es Waveform.vwf



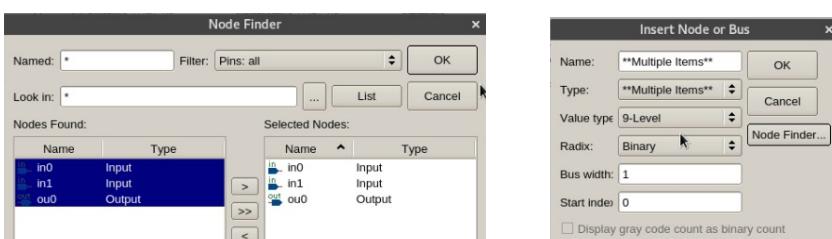
Después de guardar el archivo, haga click en Edit → Insert → Insert Node or Bus...



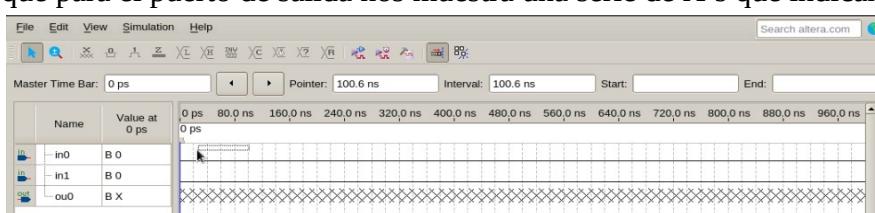
En la ventana emergente, haga click en “Node Finder...” para abrir el buscador de nodos. Dentro de este buscador haga click en “List” para enumerar los puertos.



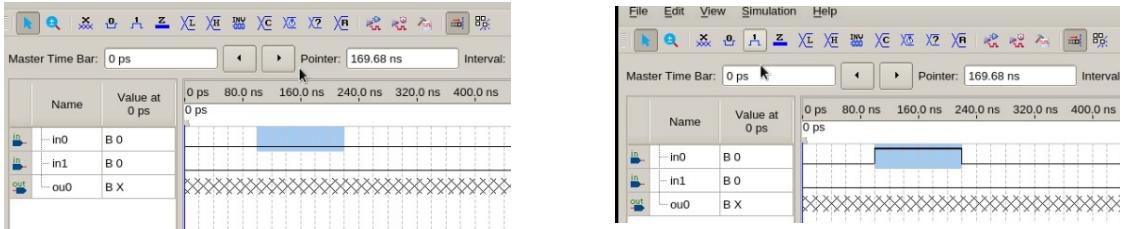
Haga click en el botón >> para seleccionar todos los puertos como nodos, luego haga click en OK para regresar a la ventana de Insert Node or Bus y finalmente haga click en OK para insertar todos los nodos.



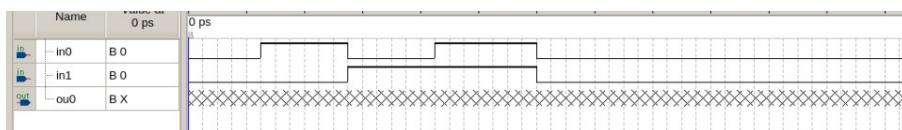
Para los puertos de entrada, el programa genera dos ondas que están continuamente en 0, mientras que para el puerto de salida nos muestra una serie de X's que indican un valor indeterminado.



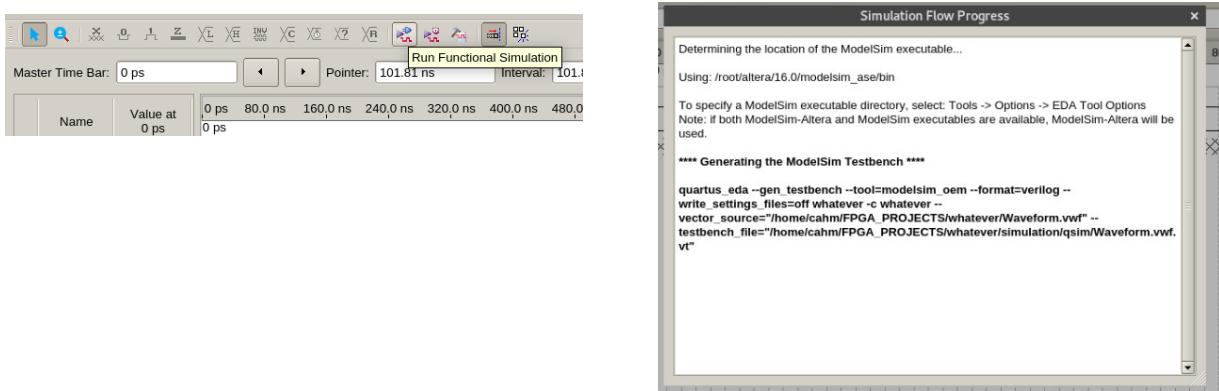
Podemos editar las ondas de los puertos de entrada para emular el comportamiento de la señal de entrada. Seleccionamos un intervalo en el tiempo, y elegimos el valor de ese intervalo como se muestra a continuación.



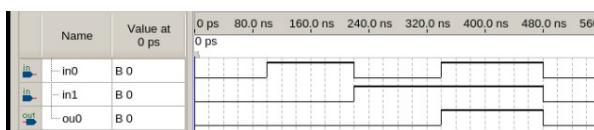
Para esta demostración, Dibujaremos las siguientes ondas, simulando las combinaciones posibles de las señales de entrada.



Damos click en “Run functional level simulation” para iniciar la simulación.



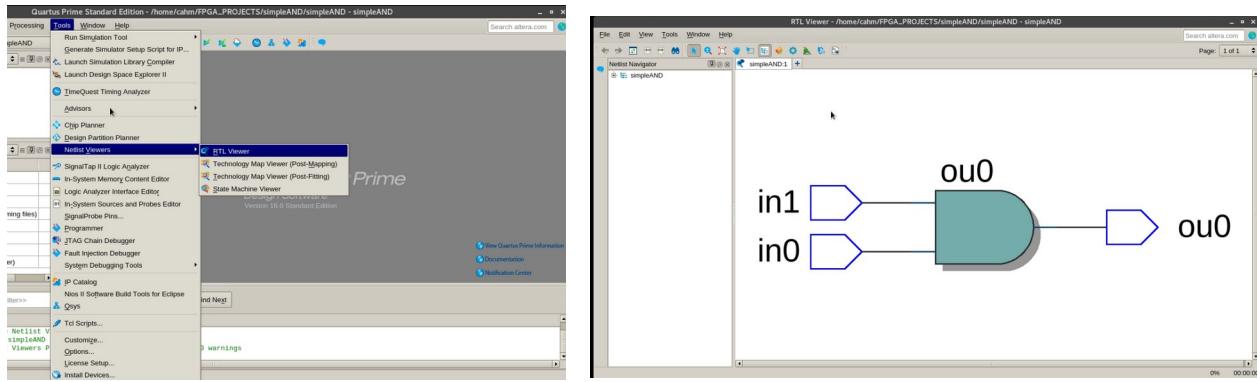
Una vez concluida esta, obtendremos la forma de la onda de la salida.



Como podemos ver, la salida ou1 solamente tiene el valor “1” tanto el puerto in1 como el puerto in0 reciben un valor de uno “1”, que se lo que se espera en una compuerta AND.

Visor de RTL

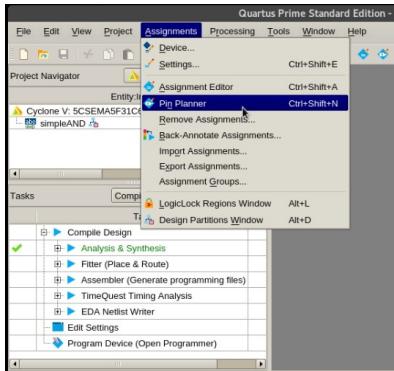
Otra forma de validar el diseño que podemos hacer después de la etapa de Análisis y Síntesis es la visualización del diseño a Nivel de Transferencia de Registros (RTL por sus siglas en inglés). Seleccionamos, Tools → Netlist Viewers → RTL viewer.



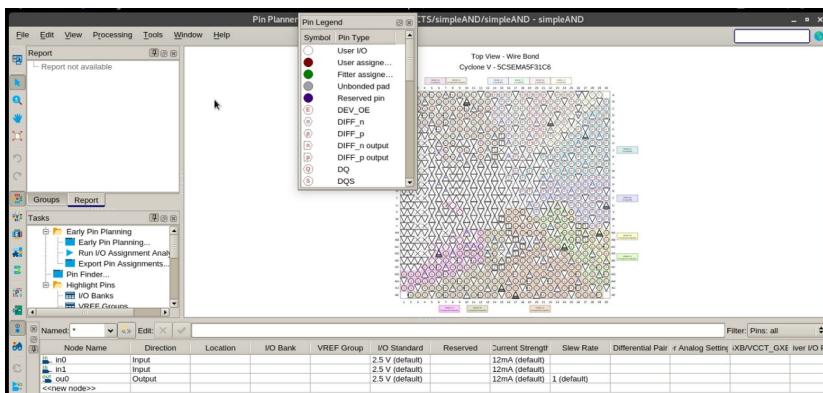
Podemos ver que los puertos de entradas in1 e in0, están conectados con el puerto de salida ou0 está mediante una compuerta AND.

Asignación de Pines

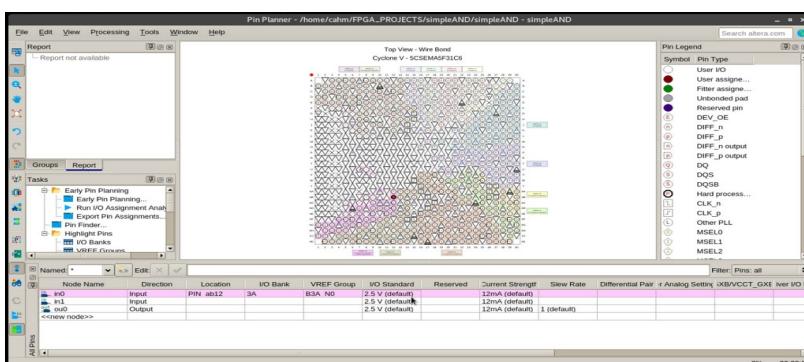
Antes de continuar con la compilación es necesario asignar pines a los puertos del diseño, es decir, hay que vincular cada puerto en nuestro diseño con una interfaz de la capa física del FPGA. Para hacer esto, abramos herramienta “Pin Planer” en Assignments → Pin Planer



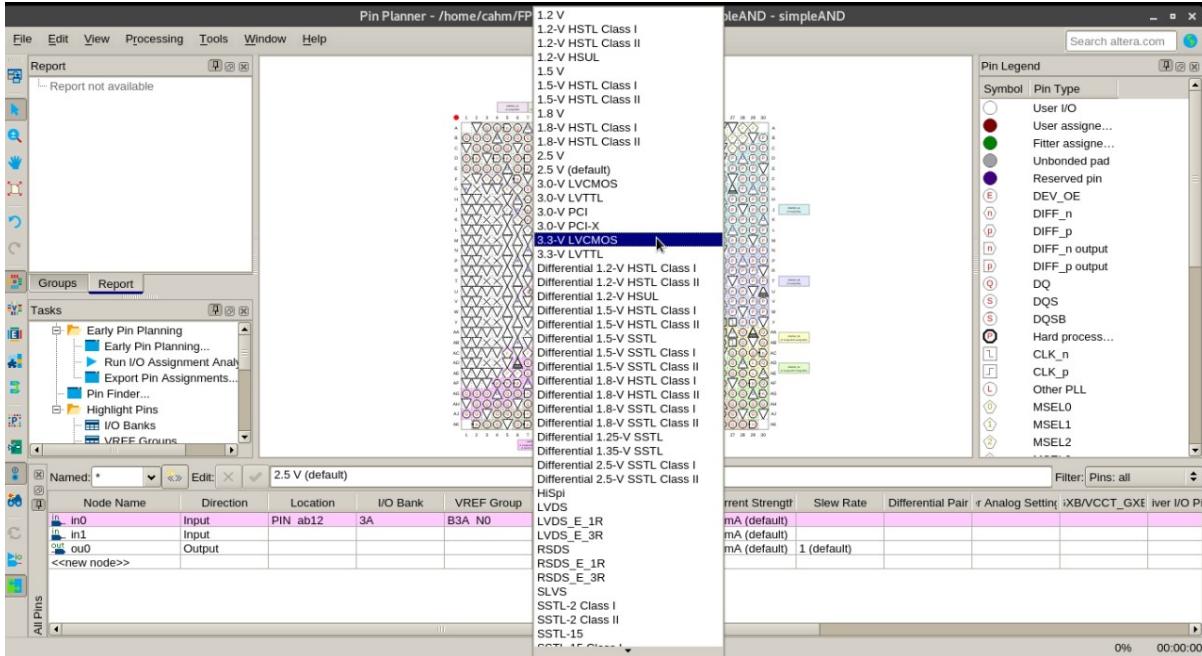
En esta interfaz, vemos al centro un mapa de los pines del FPGA, así como una tabla de puertos en la parte inferior.



En la columna “Location” asignaremos la ubicación de cada pin. Nota: en los manuales de las tarjetas se refieren a este atributo como número de pin. Para esta demostración, conectaremos cada entrada con un switch de la tarjeta y la salida salida con uno de los LEDs. En el manual de la tarjeta que estemos utilizando podemos encontrar la información Por ejemplo, vincularemos el puerto i0 al primer switch de la tarjeta , cuya ubicación es “PIN_AB12” en la **DE1-SoC** y “PIN_L10” en la **DE0-SoC**.



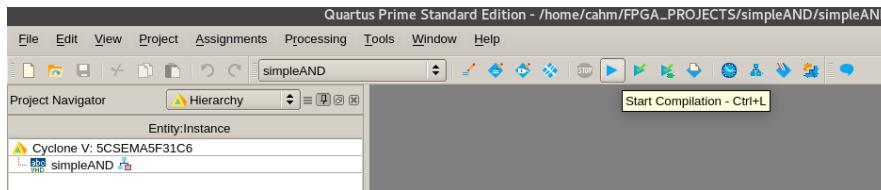
Además de la ubicación, tenemos que especificar el estándar de entrada y salida al que opera el pin que estamos asignando. Esto se refiere al voltaje y semiconductor del pin. Continuando con el puerto in0, el manual nos indica que el pin que le asignamos usa un estándar 3.3V LVCMOS, esto es igual tanto para la tarjeta DE1-SOC como en la DE0-SoC.



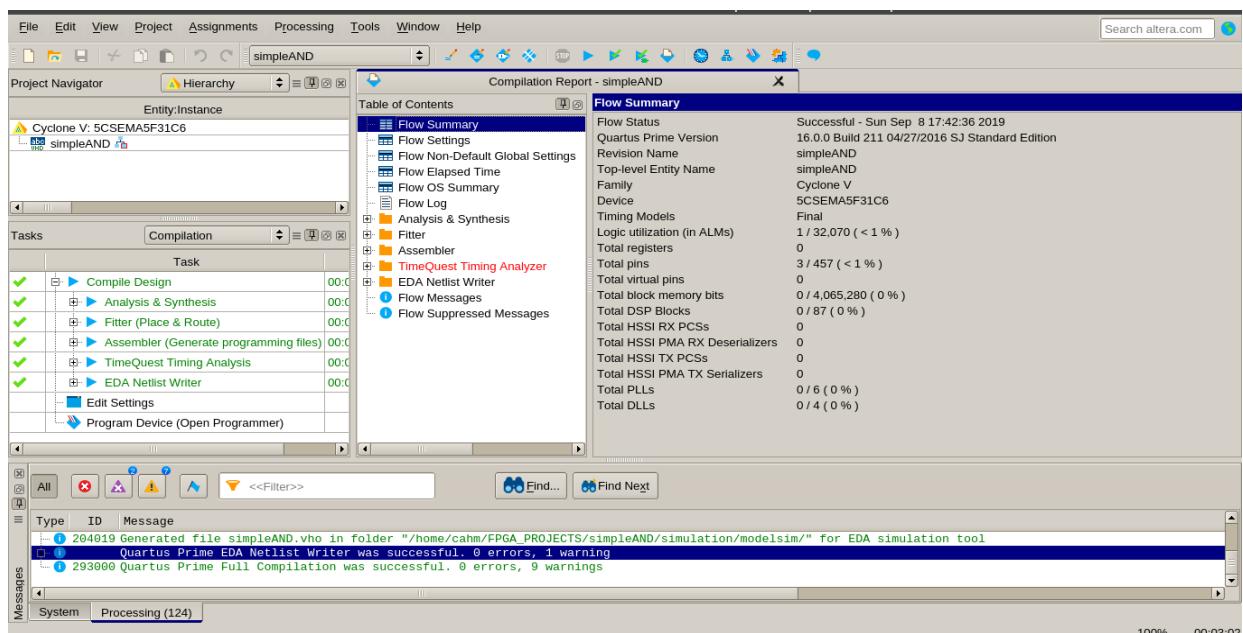
Para finalizar la asignación hacemos lo mismo con los demás puertos. Para la tarjeta **DE1-SoC** asignaremos a los puertos in1 y ou0 las localizaciones “PIN_AC12” y “PIN_V16” respectivamente, ambos con el estándar 3.3-V LVCMOS. Para la tarjeta **DE0-SoC** asignaremos a los puertos in1 y ou0 las localizaciones “PIN_L9” y “PIN_W15” respectivamente, ambos con el estándar 3.3-V LVCMOS. Las asignaciones se guardan automáticamente, por lo que habiendo terminado la asignación simplemente cerramos la ventana.

Compilación

Después de haber asignado los pines, iniciamos la compilación del proyecto. Para esto damos click en el botón de iniciar compilación, o usamos el atajo Ctrl+L



Al final de la compilación, veremos en el navegador de tareas que todas las etapas han sido completadas. Además, veremos que se ha generado un informe de compilación, el cual contiene información relevante al proceso. Este informe se guarda automáticamente y en caso de cerrarse puede abrirse en cualquier momento con el comando Ctrl+R.

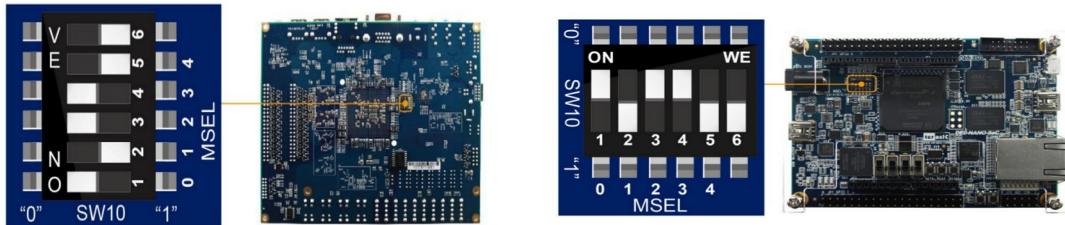


El presente proyecto se compilará con una serie de advertencias y avisos respecto a la ausencia de reloj, de especificaciones de tiempo y de la falta de un archivo ".sdc". Podemos ignorar estas advertencias, dado que la presente demostración es un diseño asíncrono. Más adelante, demostraremos un ejemplo de lógica secuencial en la que sí requiere de estas consideraciones.

Establecer modo de configuración de FPGA

El FPGA puede ser configurado tanto en modo Serial Activo (AS por sus siglas en inglés) como en modo Pasivo Paralelo Rápido (FPP por sus siglas en inglés).

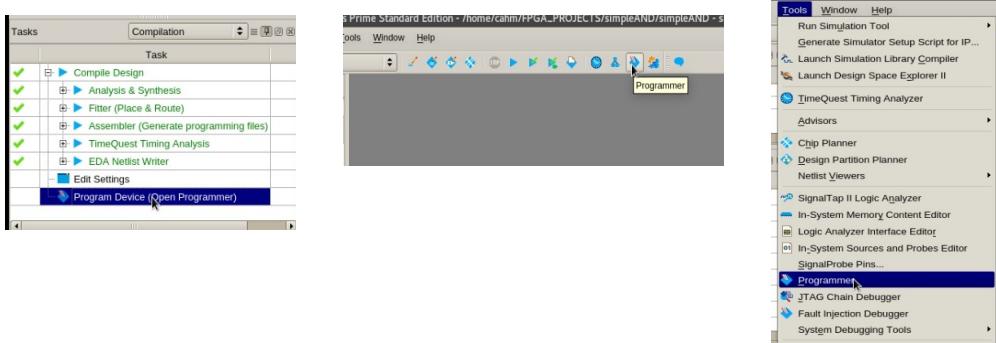
Para seleccionar el modo de configuración, las tarjetas DE1-SoC y DE0-SoC cuentan con un conjunto de switches denominados MSEL. Estos se muestran a continuación para ambas tarjetas en la siguiente ilustración.



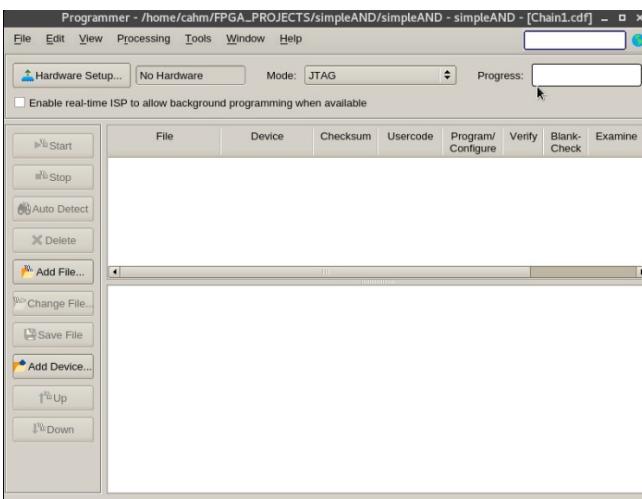
Antes de configurar el FPGA, tenemos que asegurarnos de que estos switches estén en el modo correcto para el tipo de configuración que se requiera. Particularmente, en esta sección mostraremos como configurar el FPGA en modo serial activo. Para seleccionar este modo, tanto en la tarjeta DE1-SoC como en la DE0-SoC, la configuración de los **MSEL, de 4 a 0 es 10010**. Es muy importante que la **tarjeta** debe estar **desconectada** cuando manipulemos los **MSEL**.

Programador

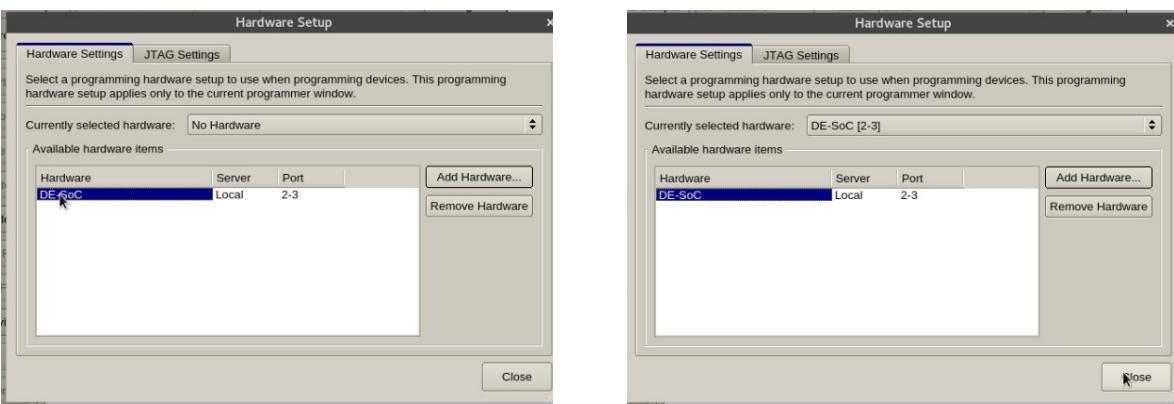
Habiendo configurado los MSEL, conectamos la tarjeta tanto a la corriente como a la PC. Ya con la tarjeta conectada, abrimos la herramienta “Programmer”. Como se muestra a continuación, esta puede ser abierta desde el menú de tareas, desde el ícono de “Programmer”, o en el menú Tools → Programmer.



Cuando abrimos la herramienta, podemos ver que tiene seleccionado el modo de programación JTAG, pero que no se muestra ningún dispositivo detectado ni ha seleccionado ningún archivo para programar.



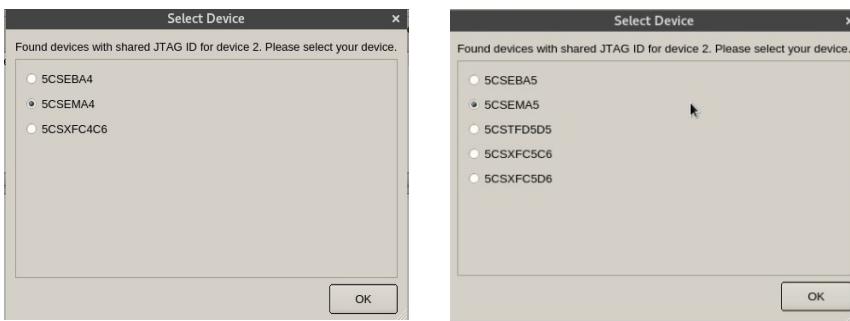
Para empezar, seleccionamos la opción “Hardware setup” para buscar los dispositivos conectados. Damos click en DE-SoC y vemos como el indicador “Currently selected hardware” cambia de “No Hardware” a “DE-SoC [2-3]”



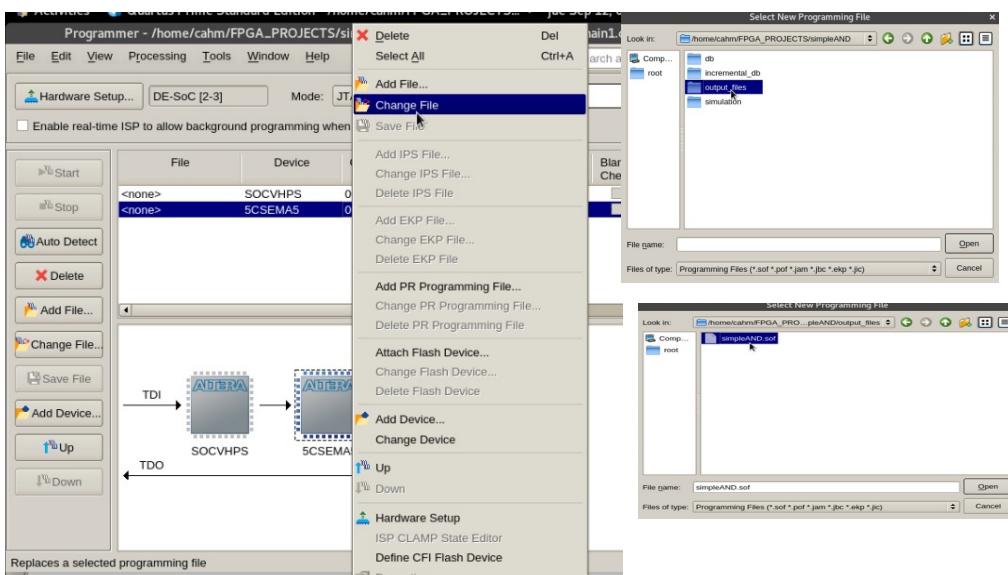
Damos click en “Close” para cerrar la configuración de hardware, y damos click en el botón “Auto Detect”.



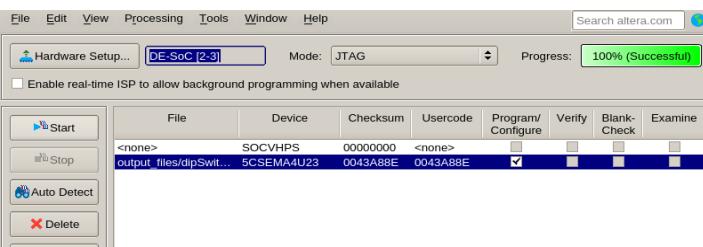
El sistema detectará un dispositivo JTAG, pero como no tendrá la capacidad de identificar exactamente la ID del dispositivo nos pedirá que seleccionemos de entre las opciones posibles. En el caso de la tarjeta DE0-SoC, seleccione la opción 5CSEMA4 o para la tarjeta DE1-SoC seleccione la tarjeta 5CSEMA5.



El programador ahora mostrará el dispositivo FPGA y el procesador ARM. En la interfaz, seleccione el dispositivo FPGA, haga click derecho y seleccione “Change file”. Dentro de la carpeta Output files”, busque el archivo tipo “.sof” con el nombre del proyecto y abra.

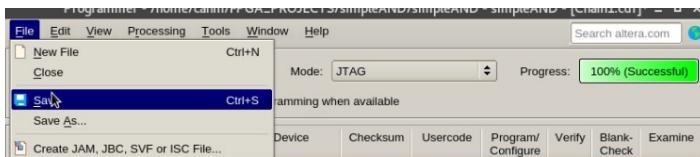


Active la opción de Program/Configure y haga click en “Start” para iniciar la configuración. La barra de progreso marca el avance de la configuración, y marcará en verde cuando esta haya terminado. La tarjeta estará configurada y los dos últimos switches controlarán el led 0 como una compuerta AND.



Guardar cadena de configuración

Para guardar el binario de configuración para nuestro diseño, en el programador abra el menú File → Save. Se generará un archivo “.cdf” el cual podrá ser usado por el programador para configurar el FPGA con el diseño que hemos compilado.



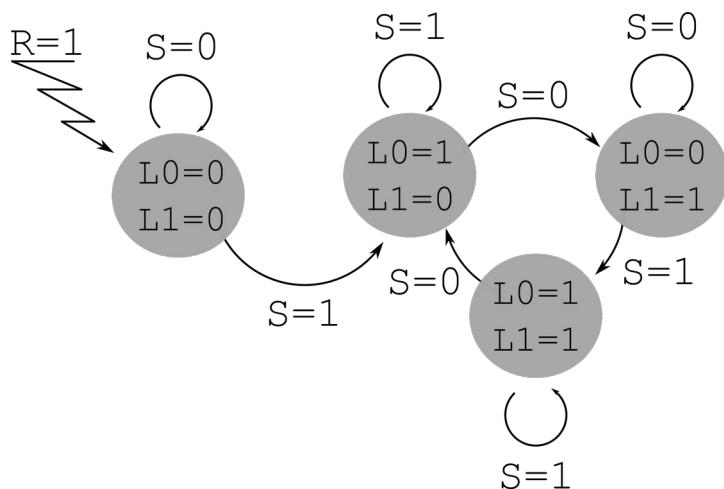
Ejemplo de una máquina de estados

Antes de pasar a una aplicación, mostraremos la implementación de un circuito un poco más complejo que el del ejemplo 5 con el fin de aprender a manejar los conceptos de señales, instancias de componentes, procesos, mapas de puerto, etc.

Las máquinas de estados finitos son circuitos síncronos secuenciales, lo que quiere decir que son circuitos con registros controlados por un solo reloj, cuyas salidas dependen no solo del valor actual de sus entradas, sino además de su propio comportamiento en el pasado. El nombre máquina de estados finitos, surge del hecho de que su comportamiento funcional puede ser representado como un número finito de estados

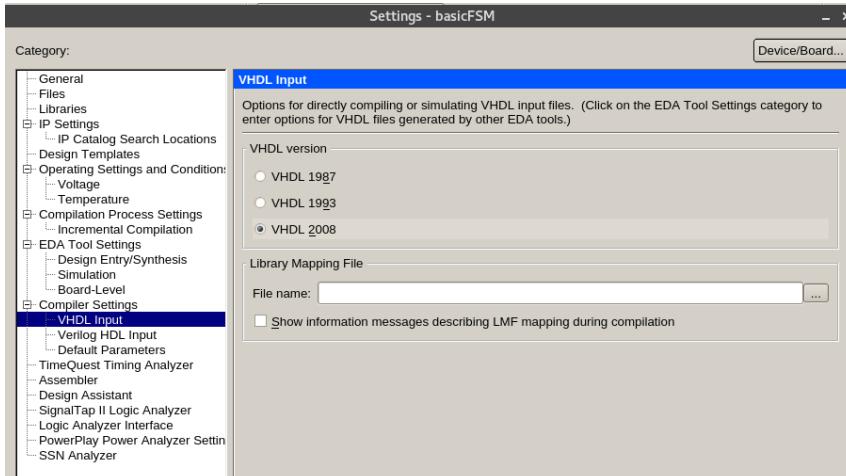
En la siguiente figura, se muestra un diagrama de la máquina de estados que implementaremos en esta demostración. Esta máquina de estados puede transitar entre cuatro estados; en primer lugar un estado de reset en el que las dos salidas del sistema valen 0, seguido por tres estados en los que los valores de la salida van alternando entre “01”, “10” y “11”. Las transiciones entre los estados son controladas por una señal de selección llamada S.

Cuando el sistema recibe la señal de reset, transita al primer estado y permanece ahí mientras que la señal S valga 0. Cuando la señal S valga 1, el sistema transitará a



Configuración de VHDL

Antes de comenzar, configuraremos el compilador para usar el estándar de VHDL 2008. Para esto abrimos Assignments → Settings (o Ctrl+Shift+E) y dentro del menú de settings buscamos Compiler Settings-> VHDL Input. Seleccionamos la versión de VHDL 208 y listo.



Es preciso aclarar que todo lo que estamos haciendo aquí se pudiera hacer en cualquier otra versión del lenguaje, sin embargo esta versión tiene una sintaxis más compacta.

Flip-Flop con Reset

En primer lugar, veremos un ejemplo de un flip-flop. Este es un módulo que tiene una señal de entrada, una señal de salida y una señal de reloj. En el momento en el que la señal de reloj haga un flanco de subida (es decir, cambie su valor de 0 a 1) la señal de salida cambia al valor de la señal de entrada. En cualquier otro momento, la señal de salida mantiene el valor que tuvo en el último flanco de subida sin importar que cambie el valor de la entrada. Cuando hablamos de un flip-flop con reset estamos diciendo que tenemos un flip-flop cuyo valor puede ser forzado a cero con una señal. Cuando la señal de reset vale 1, el valor de la salida se mantiene en 0, sin importar el valor de la señal de entrada o el reloj. El código del ejemplo 6 muestra la implementación de este circuito.

```
library ieee;
use ieee.std_logic_1164.all;
entity flipFlop is
    port(
        clk : in std_logic;
        rst : in std_logic;
        d   : in std_logic_vector(1 downto 0);
        q   : out std_logic_vector(1 downto 0)
    );
end flipFlop;
architecture struct of flipFlop is
begin
    process (clk,rst,q)
    begin
        if (rst='1') then
```

```

        q <= "00";
      elsif rising_edge(clk) then
        q<=d;
      else
        q<=q;
      end if;
    end process;
end;

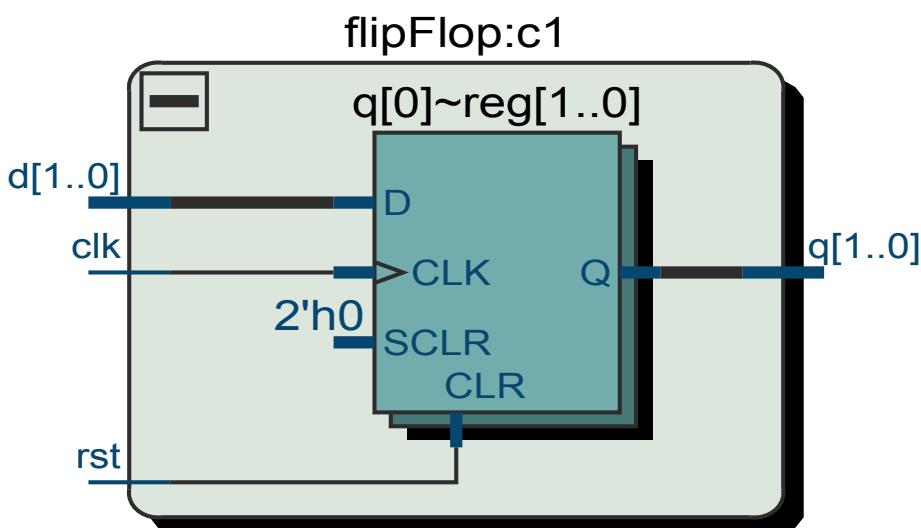
```

Ejemplo 6. Implementación de flip-flop con reset en VHDL 2008

La primera novedad que vemos en comparación con el ejemplo 5 es que hay puertos de tipo vector. En este caso, los puertos d y q son vectores de 2 bits, es decir que van desde el bit 1 hasta el bit0.

Lo principal que hay que resaltar de este módulo, es que en su arquitectura incluye un proceso. Los procesos son descripciones de un circuito desde los detalles de su comportamiento en lugar de su interconexión. Al abrir esta declaración, se pone entre paréntesis la lista de señales a las que este proceso responde. Este proceso en particular es un “if” que escribe el comportamiento de la salida q. Si la señal de reset tiene un valor de lógico de verdadero, representada como ‘1’, entonces los dos bits del vector q valen 0. Esto se representa como “00” indicando el valor bit a bit del vector. Lo siguiente que hay que notar es el enunciado “rising_edge (clk)”. Aquí estamos indicando que la condición del if es un flanco de subida del reloj. Nota, esta declaración es muy diferente en VHDL 2008 que en las versiones anteriores.

Al hacer la compilación, el sistema interpreta el comportamiento descrito y genera la arquitectura que le corresponda. Esto lo podemos ver en la netlist de este componente, en la que se muestra el diagrama de bloque de un flip-flop.



Lógica de transiciones entre estados.

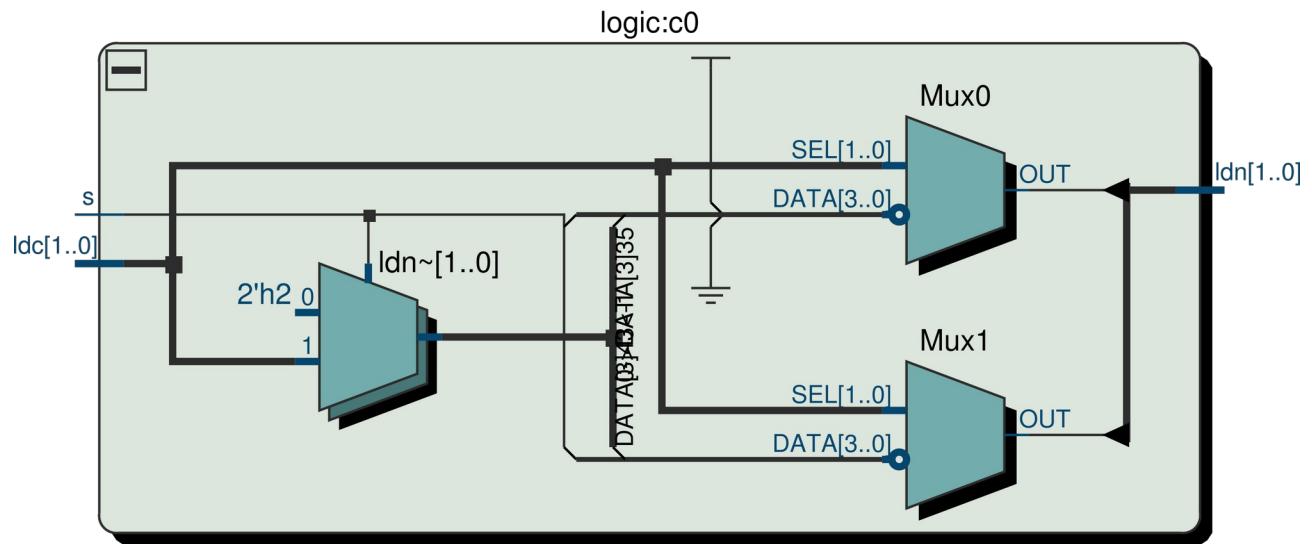
El siguiente módulo del diseño se encarga de procesar las transiciones entre estados. De manera resumida, este módulo detecta el estado del sistema, así como las señales externas, y decide el estado siguiente.

```
library ieee;
use ieee.std_logic_1164.all;
entity logic is
    port(
        s      :      in std_logic;
        ldc   :      in std_logic_vector(1 downto 0);
        ldn   :      out std_logic_vector(1 downto 0)
    );
end logic;
architecture struct of logic is
begin
    process(ldc,s)
    begin
        case (ldc) is
            when "00"=>
                if (s='0') then
                    ldn<=ldc;
                else
                    ldn<="01";
                end if;
            when "01"=>
                if (s='1') then
                    ldn<=ldc;
                else
                    ldn<="10";
                end if;
            when "10"=>
                if (s='1') then
                    ldn<=ldc;
                else
                    ldn<="11";
                end if;
            when others=>
                if (s='1') then
                    ldn<=ldc;
                else
                    ldn<="01";
                end if;
        end case;
    end process;
end;
```

Ejemplo 7. Procesamiento de lógica de máquina de estados

Este ejemplo usa un modo de proceso llamado “case”, en el cual se establece el comportamiento del sistema para cada valor de una señal específica. Dentro de cada caso, se puede hacer una declaración de if.

Como se aprecia en la siguiente imagen, así como en el ejemplo anterior, el compilador genera la lógica necesaria para que el circuito realice el comportamiento.



Integración de módulos

En este punto tenemos el módulo que hace la sincronización con el reloj, y el módulo que selecciona los estados. Ahora lo que sigue es integrarlos. Esto se realiza con el código descrito en el ejemplo 8.

```

library ieee;
use ieee.std_logic_1164.all;
entity basicFSM is
    port(
        clk : in std_logic;
        rst : in std_logic;
        sw : in std_logic;
        ld : out std_logic_vector(1 downto 0)
    );
end basicFSM;

architecture struct of basicFSM is
    signal q,d : std_logic_vector(1 downto 0);

component logic is
    port(
        s : in std_logic;
        ldc : in std_logic_vector(1 downto 0);
        ldn : out std_logic_vector(1 downto 0)
    );
end component;

component flipFlop is

```

```

port(
    clk : in std_logic;
    rst : in std_logic;
    d : in std_logic_vector(1 downto 0);
    q : out std_logic_vector(1 downto 0)
);
end component;

begin
c0 : logic port map (sw,q,d);
c1 : flipFlop port map (clk,rst,d,q);
ld <= q;
end;

```

Ejemplo 8. Procesamiento de lógica de máquina de estados

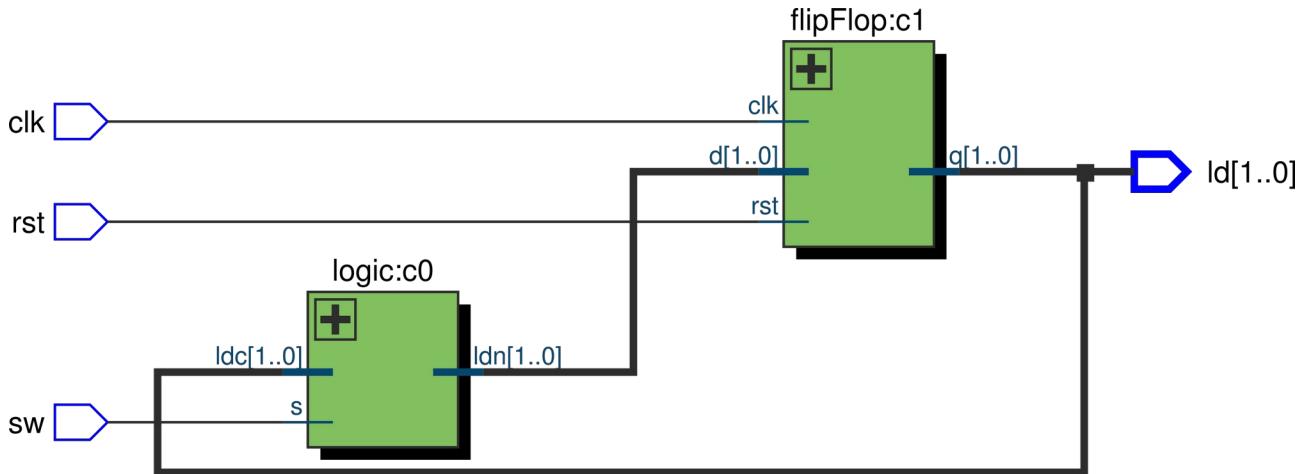
En este código, destaca que en la sección de arquitectura se incluye la definición de conexiones (llamadas señales) y se define a otros componentes. Como se mencionó al explicar el ejemplo 4, estas declaraciones se hacen dentro de la arquitectura antes de la linea begin.

Los componentes se definen con el argumento “Component” el cual es identico a la una declaración de entidad, salvo por las lineas que abren y cierran la declaración. Es importante usar el mismo nombre para el componente, la entidad a la que llama y el archivo que contiene a dicha entidad.

Las conexiones, o señales, se definen mediante el argumento “Signal”, en el cual se definen conexiones que pueden establecerse entre dos o más componentes, o entre un componente y un puerto de la entidad que se define en el proyecto sobre el que se trabaja. Estas son similares a las declaraciones de puerto, a excepción de queno incluyen una direccionalidad. En el caso de la presente demostración, se definen dos señales de dos bits. La primera conecta la salida del módulo de lógica con la entrada del flip-flop, mientras que la segunda conecta la salida del flip-flop con la entrada del módulo de lógica.

Dentro del comportamiento de la arquitectura, es decir después de begin, hacemos las instancias de los componentes. Estas son partes de nuestro diseño en las que se llama a un componente en particular. La sintaxis de una isntancia es “**nombreDeLaInstancia: nombreDelComponente port map (nombreDePuerto1, nombreDePuerto2,...,nombreDePuertoN);**”. Es posible crear múltiples instancias de un mismo componente dentro de un diseño. Por ejemplo, si el diseño necesitara múltiples flip-flops, pudieramos hacer múltiples instancias que llamen al mismo componente. En el argumento “port map” se ponen entre paréntesis los puertos y/o señales que se interfazan con la instancia del componente, en el mismo orden en el que van en la declaración de puertos. Por ejemplo, en el flip-flop los puertos en orden son: clk, rst, d y q. De modo que en el port map ponemos en ese mismo orden las señales y puertos correspondientes; primero los puertos clk y rst, luego las señales d y q. Las señales y puertos en la entidad principal no tienen que coincidir con el mismo nombre que los de la entidad del componente, pero si en el tipo, dimensiones y número.

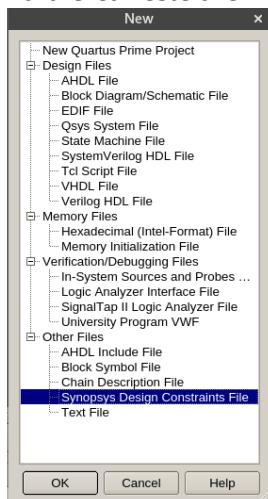
Al hacer el análisis y síntesis, obtenemos la siguiente netlist del diseño. En esta esta, las instancias de los componentes se representan en un diagrama de bloques, se muestran las conexiones que las señales forman entre ellos, se mapean las entradas y salidas de los componentes con los puertos de entrada y salida de la entidad principal y se muestran los anchos de banda de cada conexión.



Archivo SDC

A diferencia del diseño del ejemplo 5, este diseño incluye un reloj y tiene comportamiento síncrono. Por lo tanto, se requiere definir las restricciones a la operación del reloj. Estas se definen en un archivo de tipo “.sdc” cuyas siglas son acrónimo de “Synopsis Design Constraints”.

Para crear este archivo abrimos el menú “New” y buscamos “Synopsis Design Constraints File”.



Este archivo sepecifica en lenguaje Tcl (Tool command language) los parámetros restricciones para el análisis y simulación de los relojes del diseño. En el ejemplo 9 se muestra un archivo sdc para el presente diseño operando en un FPGA CycloneV como el que se encuentra en la tarjeta DE1-SoC o DE0-SoC.

```
# Clock constraints
create_clock -name "clock_50_1" -period 20.000ns [get_ports {clk}]

# Automatically constrain PLL and other generated clocks
derive_pll_clocks -create_base_clocks

# Automatically calculate clock uncertainty to jitter and other effects.

derive_clock_uncertainty
```

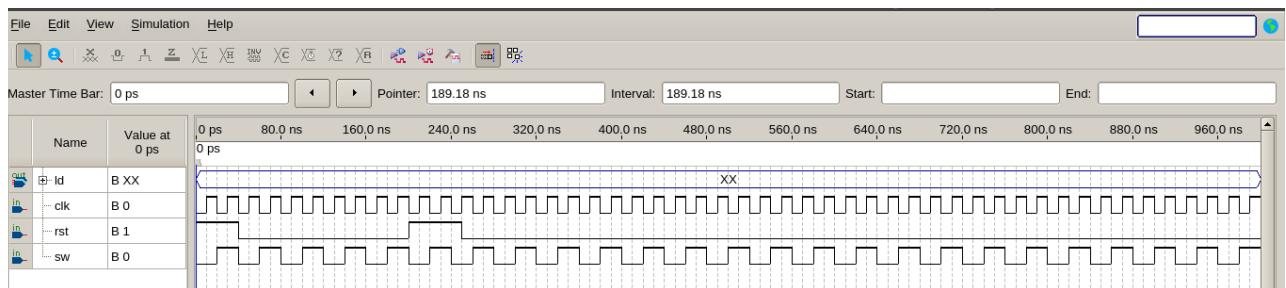
Ejemplo 9. Archivo sdc para un reloj de 50MHz

Lo más importante es la instrucción, “create_clock” pues es la que define el reloj que se utiliza en el diseño. Los argumentos de esta instrucción son el nombre del reloj, el periodo del reloj (20ns, es decir 1/50MHz) y el puerto al que esta asociado el reloj descrito. Guardamos el archivo sdc con el nombre del proyecto y proseguimos al siguiente paso.

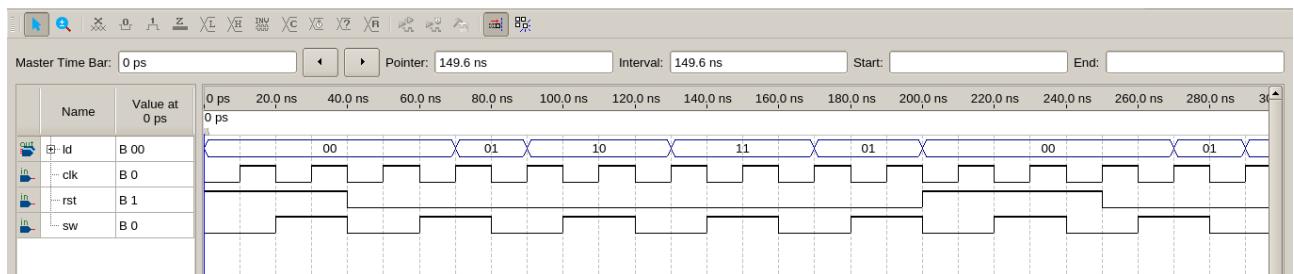
Simulación de máquina de estado

Este diseño se pudiera compilar y programar como la demostración anterior vinculando los puertos con los pines de reloj que aparecen en el manual de cada tarjeta, pero por motivos de brevedad nos limitaremos a la simulación.

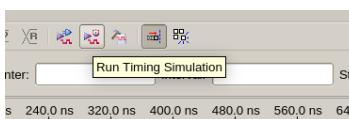
Así como en la primera demostración, creamos un archivo de altera university wave simulator, y llamamos a los puertos como nodos. Seleccionamos el nodo clk y damos click en el ícono de reloj para crear una señal de reloj. En el menú nos pedirá que especifiquemos una frecuencia, elegimos 50MHz y damos enter. El resto de las ondas las dibujamos manualmente como aparece en la siguiente imagen (tip, la señal sw se dibujó creando una onda de reloj a 25MHz).



Corremos la simulación y observamos el resultado en la salida “Id”. Vemos que parte de “00” al recibir la señal de reset, oscila entre los demás valores ante los pulsos de la entrada sw, y regresa a “00” al reactivar el reset.



Hasta ahora, las simulaciones que hemos corrido han sido a nivel de compuertas, pero como en esta demostración estamos agregando el archivo SDC, podemos correr una simulación de timing. Estas simulaciones son particularmente útiles en diseños grandes donde se requiere que el diseño ejecute tareas dentro de marcos de tiempo establecidos.

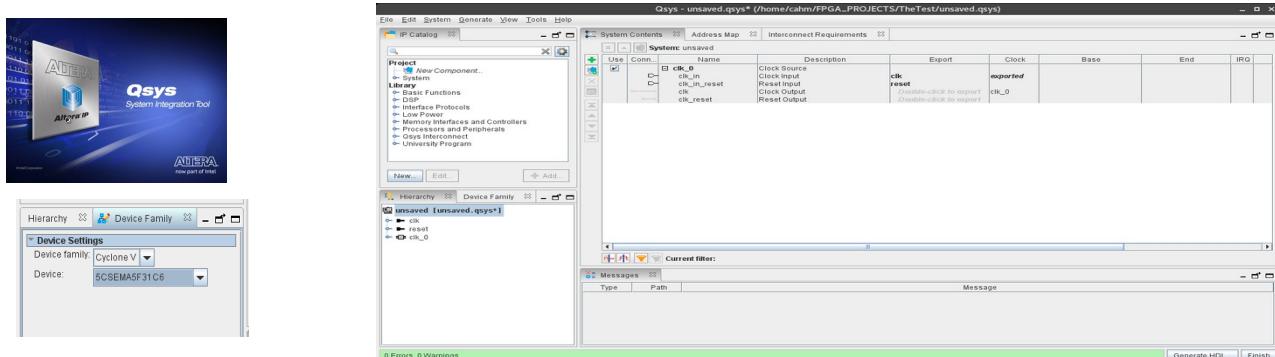


Comunicación entre FPGA y ARM

Varios de los dispositivos FPGA de Altera/Intel son Sistemas en un Chip (SoC por sus siglas en inglés), es decir que integran varios componentes en un solo circuito. En particular, los dispositivos CycloneV contenidos en las tarjetas DE1 y DE0 son SoC's que incorporan un módulo FPGA, una memoria RAM y un procesador ARM cortex A9. Este es un procesador de 32-bits con una arquitectura tipo RISC. En esta sección aprenderemos a realizar un diseño que aprovecha esta capacidad del dispositivo para programar un diseño hardware sobre el módulo FPGA e interactuar con este diseño desde una capa de software. Para este fin, emplearemos la interfaz de memoria mapeada Avalon, la cual permite al desarrollador describir interfaces de lectura y escritura basadas en direcciones de memoria.

Qsys

Para generar el módulo de comunicación entre el FPGA y el ARM, utilizaremos la herramienta Qsys, la cual permite diseñar dentro de una interfaz gráfica sistemas digitales que integren componentes como, procesadores, controladores de memoria, etc. Tales componentes pueden ser prediseñados o pueden ser diseñados por el usuario. Los sistemas diseñados en la interfaz gráfica, se usan para generar de manera automática el código en lenguaje de descripción de hardware para su implementación en el FPGA. A lo largo de esta guía, generaremos un sistema de interconexión FPGA-ARM basado en módulos prediseñados.



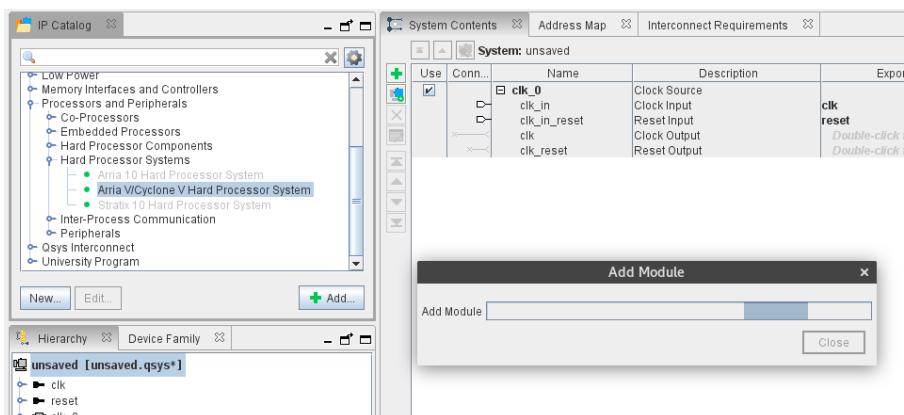
Para ejecutar esta herramienta, damos clic en el menú Tools → Qsys. Se abre una ventana con la aplicación, en la que vemos cuatro paneles principales de los que hablaremos a continuación. En la esquina superior tenemos el panel de catalogo de IP, el cual nos muestra una biblioteca de módulos que podemos agregar a nuestro sistema. Justo debajo del catálogo de IP, tenemos un panel con dos pestañas; la pestaña de jerarquía que nos resume las relaciones entre los módulos del sistema y la pestaña de familia de dispositivo que nos indica cuál es el modelo del FPGA que el programa está tomando en cuenta para generar un sistema de interconexiones. A lo largo del fondo de la ventana tenemos un panel de mensajes, que nos indica errores, advertencias y eventos del sistema. Finalmente, tenemos un panel con pestañas que nos presenta el contenido del sistema, el mapa de direcciones y los requerimientos de interconexión. Es en este panel donde realizaremos el diseño del sistema. Nota: en caso de cerrar cualquiera de los paneles, con el atajo de teclado Ctrl+1 se restablece la vista por defecto.

Cuando se abre la ventana de Qsys empezamos con un proyecto en blanco, el cual por defecto siempre incluye un módulo de reloj. En la pestaña de System contents veremos una tabla en la que cada fila corresponde a un conector de un módulo y las columnas indican las propiedades del mismo. Dentro de un proyecto de Qsys, podemos agregar módulos llamados componentes y manejar las conexiones entre estos.

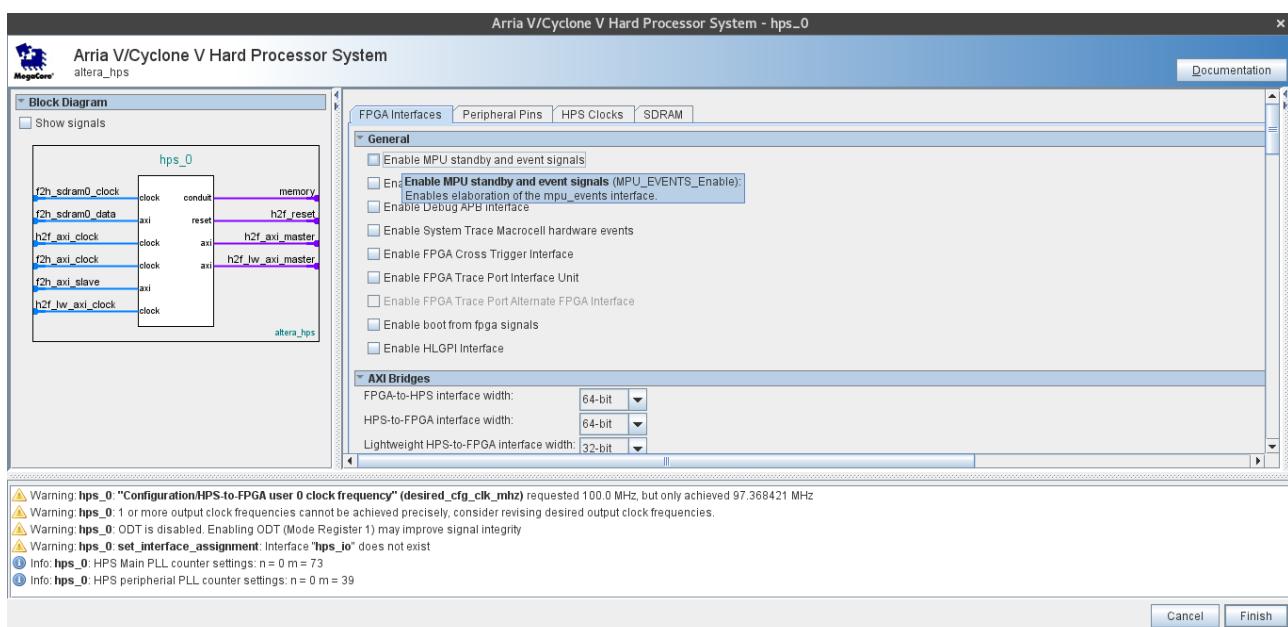
Sistema de Procesador de núcleo Duro (HPS)

A modo de exemplificar el proceso de añadir un componente a un sistema de Qsys, integraremos un módulo de procesador de núcleo duro duro (Siglas de Hardcore Processor System siglas en inglés) dentro de un proyecto en blanco. Cuando hablamos de un procesador de núcleo duro, nos referimos a un procesador que está implementado directamente como un circuito integrado, en lugar de como una implementación sobre el FPGA. Un ejemplo de un HPS es el procesador ARM que viene en los dispositivos CycloneV-SoC de las tarjetas DE1-SoC y DE0-SoC.

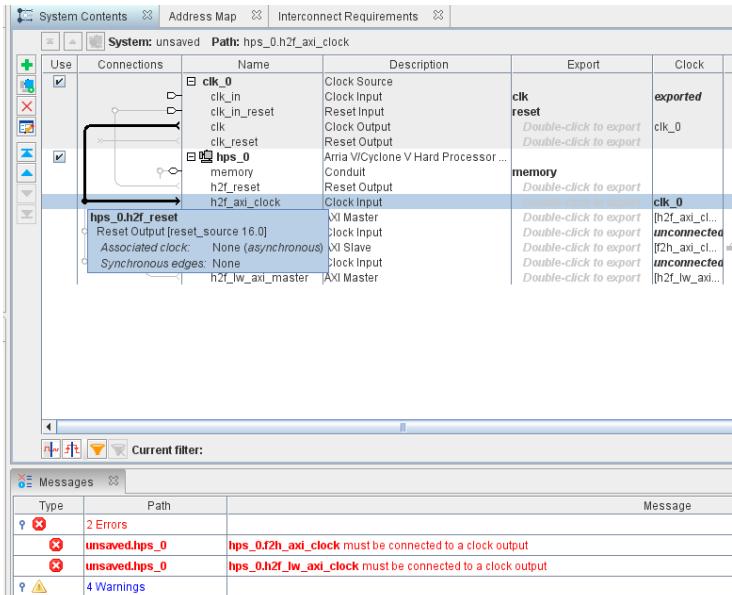
Para añadir este componente, buscamos en el panel de IP la categoría de **procesadores y periféricos** y dentro de ésta buscamos la categoría de **sistemas de procesadores duros**. Como el dispositivo FPGA-SoC que tiene nuestra tarjeta es un CycloneV, seleccionamos el componente “ArriaV/CycloneV Hard Processor System” y damos clic en añadir.



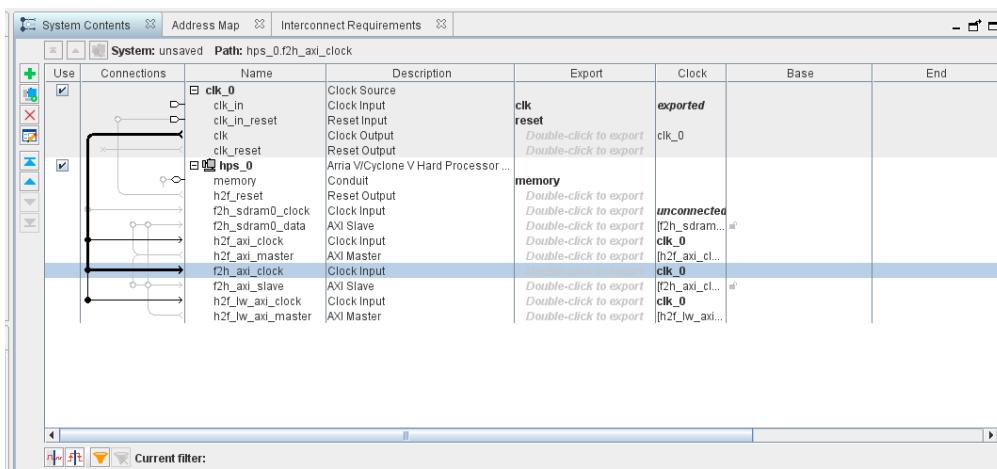
El programa abrirá automáticamente una ventana en la cual podemos configurar los parámetros del componente. Por ejemplo, en la pestaña “FPGA interfaces”, en la sección “General” tenemos la opción de deshabilitar o habilitar distintas interfaces y/o tipos de señales. Por ejemplo, podemos deshabilitar la interfaz mpu_events.



Habiendo configurado los parámetros del componente, damos clic en Finish y vemos que se añade al proyecto el componente, con todos sus conectores asociados. Como vemos en la siguiente figura, el componente HPS viene con una serie de mensajes de error desde el momento que lo añadimos.



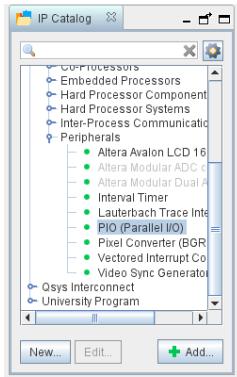
Estos errores se deben a que las interfaces del HPS que corresponden a las conexiones externas requieren recibir una interfaz de reloj. Para resolver estos errores, fijémonos en que dentro de la pestaña System contents, en la columna de Conectores hay una serie de flechas de color tenue que representan las posibles conexiones entre los puertos de cada módulo. En las intersecciones entre flechas flechas, vemos unos círculos del mismo color tenue. Presionamos estos círculos para establecer una conexión entre ambos módulos. Como se aprecia en la siguiente imagen, al establecer una conexión, las flechas se ven de color negro. Conectamos la salida clk del componente de reloj a cada puente del HPS para eliminar estos errores.



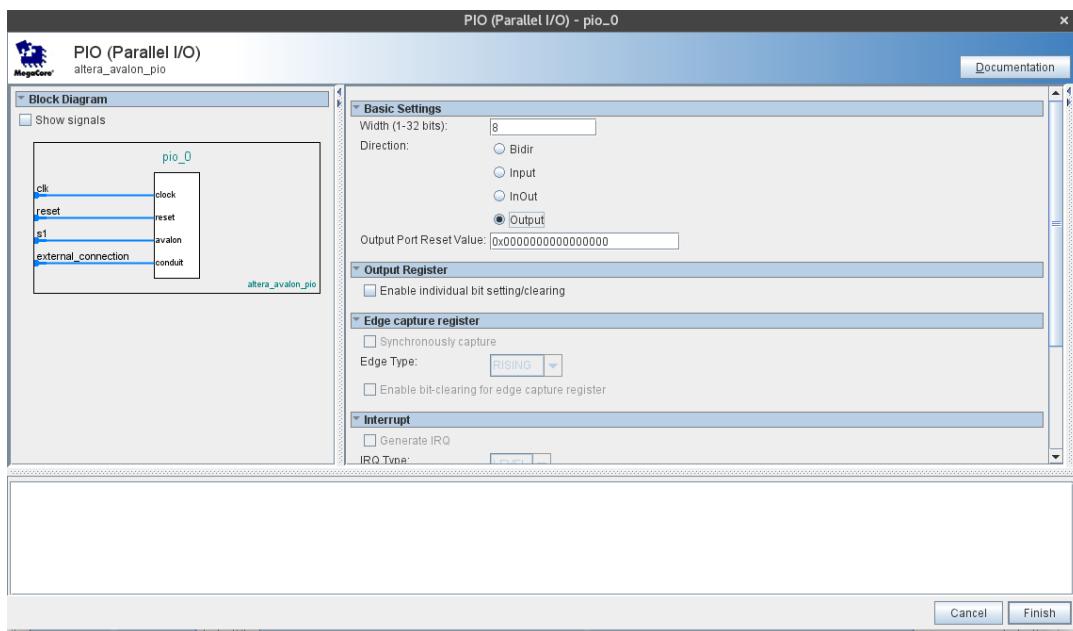
En la siguiente sección, agregaremos más componentes que se conectan a las interfaces de memoria del HPS que hemos agregado en esta sección. Por lo que se recomienda seguir trabajando sobre este mismo proyecto para continuar sobre esta misma sección.

Periféricos paralelos de entrada y salida (PIO).

La manera más simple de establecer comunicación entre el FPGA y el HPS es mediante un tipo de componente llamado periférico paralelo de entrada y salida (PIO). Para añadir un componente de este tipo al diseño, en el panel de IP catalog abrimos Processors and Peripherals → Peripherals → PIO(Parallel I/O) y damos click en Add.



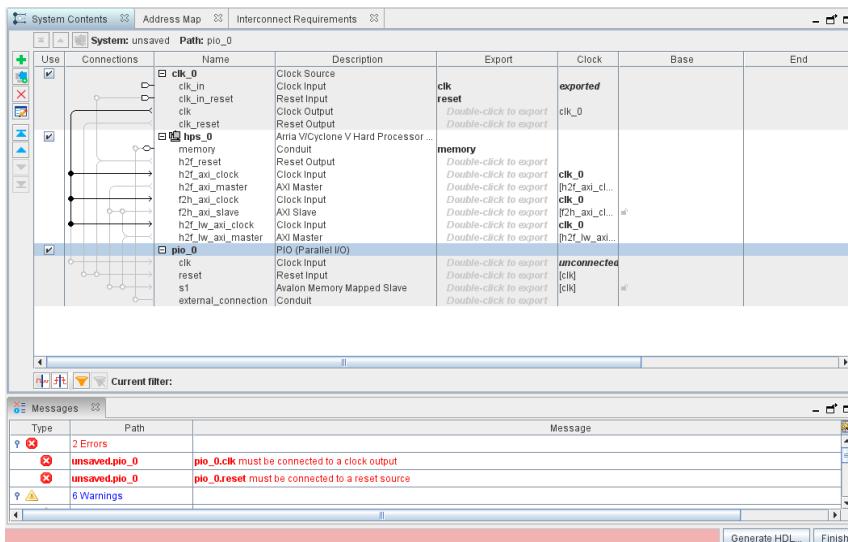
Así como en el componente HPS, se abre una ventana de configuración. En la parte izquierda de esta ventana hay un diagrama de bloques, donde se observa que el componente tiene cuatro conexiones. Primero hay una entrada de reloj y una entrada de reset, esto es porque el componente es básicamente un registro que guarda bits en una serie de flip-flops. Después hay una entrada "s1" de tipo avalon. Esta interfaz recibe una señal del procesador mediante la interfaz Avalon que habilita las operaciones de lectura y escritura sobre este registro. Finalmente, hay una interfaz de conexión externa de tipo "conduit". Éste tipo de interfaces son las que interactúan con el FPGA. La mayor parte de los parámetros que podemos configurar corresponden a las propiedades de esta interfaz.



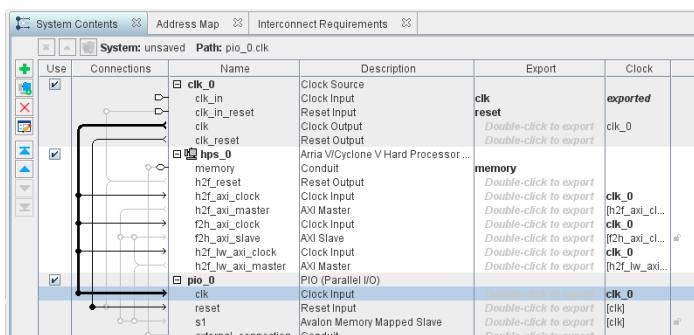
A la derecha del diagrama de bloques tenemos la lista de los parámetros que podemos configurar. Nos enfocaremos en las configuraciones básicas. El primer parámetro que podemos configurar es el ancho del registro, el cual va de 1 a 32 bits. Éste límite superior corresponde al ancho del puente Lightweight HPS-to-FPGA, que es usado por el procesador para transferir datos entre FPGA y ARM. Después tenemos la direccionalidad del puerto, que puede ser entrada, salida, bidireccional etc. Finalmente, tenemos el valor de salida del puerto cuando recibe una señal de reset. Si configuramos la dirección del componente como una entrada, notaremos que se bloquea la edición de este último parámetro.



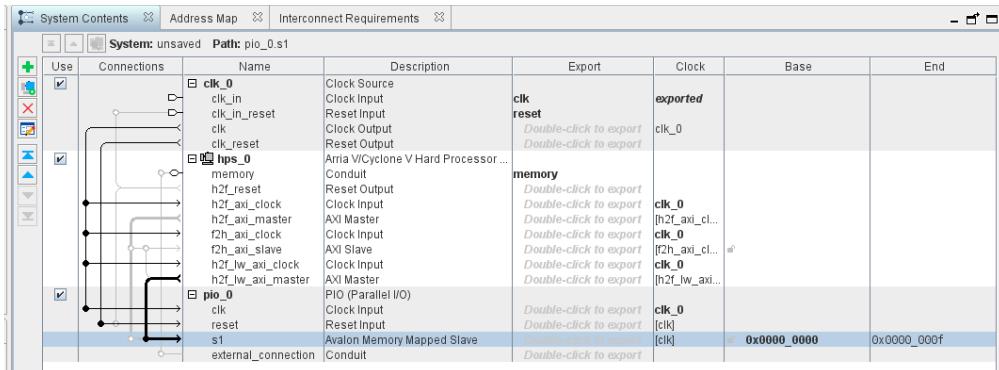
Configuramos este componente como un input de 16 bits, damos click en “Finish” y se agregará el puerto al sistema. Así como ocurrió con el HPS, podemos ver que cuando añadimos el componete PIO se generan dos mensajes de error .



El primer mensaje de error nos indica que requerimos una fuente de reloj, y lo resolvemos conectando la salida clk del componente de reloj a la entrada clk de nuestro módulo PIO, así como lo hicimos para el HPS. El segundo mensaje de error nos indica que el componente requiere de una señal de reset. Dicha señal, la obtendremos de la salida clk_reset del componente clk. El sistema debe quedar conectado como se muestra en la siguiente figura.

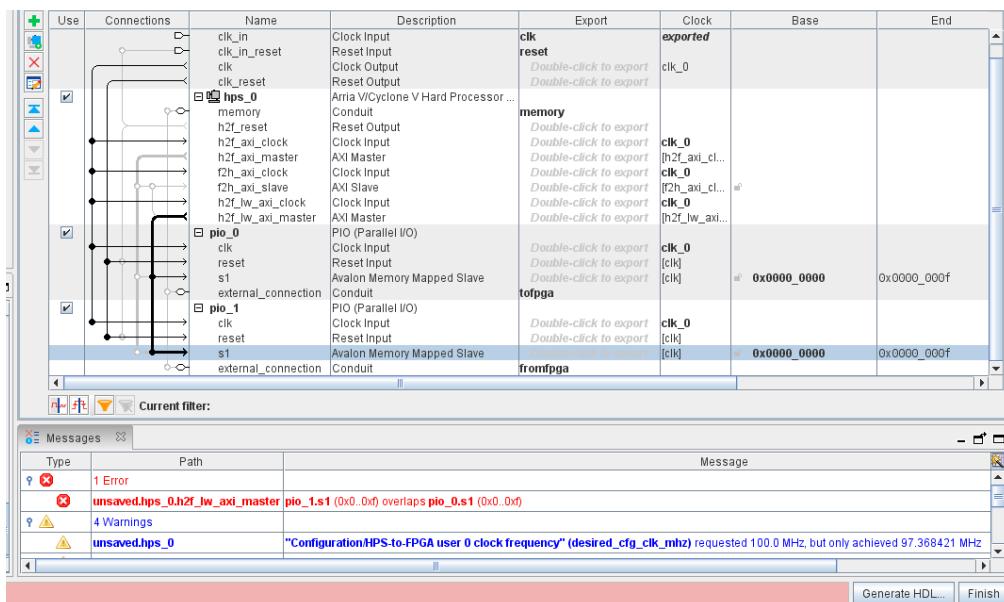


Como podemos ver en la figura anterior, la conexión s1 del componente PIO puede establecerse con los puertos “h2f_axi_master” o “h2f_lw_axi_master” que corresponden a las dos interfaces avalon con mapas de memoria. Estos son los puertos pueden ser usados como masters, que controlan las operaciones de lectura/escritura en los registros correspondientes a los PIO. Para esta demostración, conectaremos s1 con el puerto “h2f_lw_axi_master”.

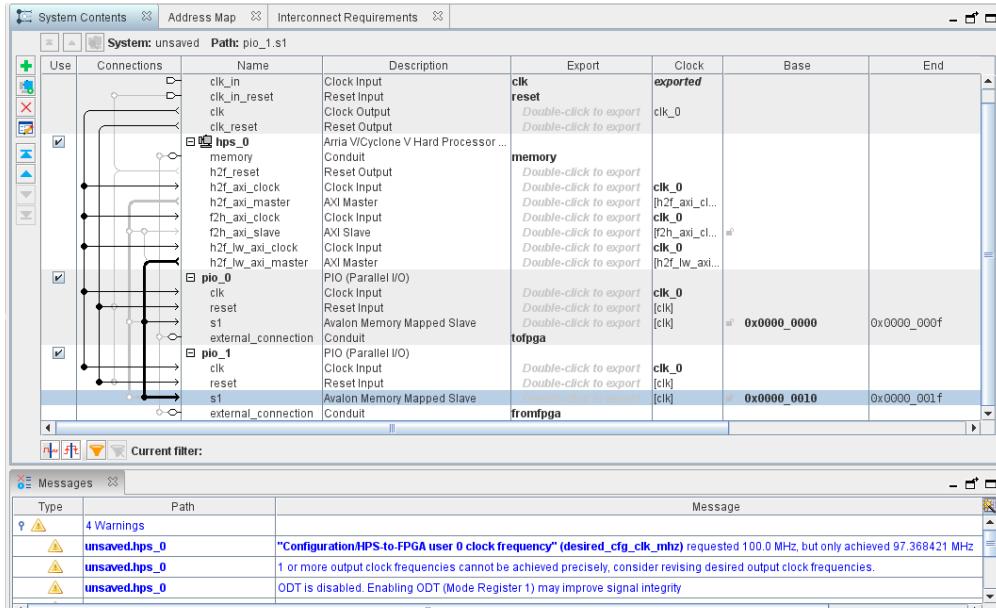


Podemos ver que al momento de conectar estos puertos, en las columnas de Base y End aparecen dos números en hexadecimal. Estos representan el inicio y fin del rango de direcciones que el sistema asignó a nuestro componente. Podemos ver que este rango tiene 16 bits de ancho, que corresponden a los 16 bits del registro del componente que creamos.

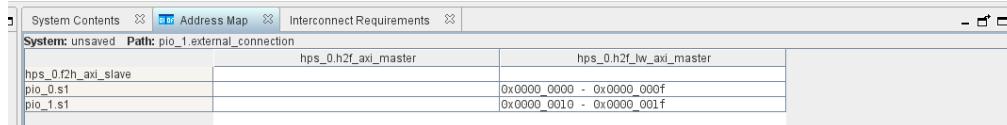
En la figura anterior, vemos que en la columna “Export” dice “Double-click-to export”. En este caso, exportar una conexión significa que esta va a ser un puerto del diseño global que estamos diseñando en el proyecto de Qsys. Damos doble clic en la columna export, justo en la celda que corresponde a la conexión conduit del componente PIO. Cuando hacemos esto, nos pide que le demos un nombre a la conexión, en este caso, le llamaremos “toFPGA”. Habiendo hecho esto, añadimos un segundo PIO de 16 bits, solo que este será un input. Conectamos el componente con el reloj y el hps como lo hicimos para el primer PIO y nombramos el conduit “fromFPGA”. El proyecto debe verse como en la siguiente figura.



Cuando hagamos la conexión de la entrada s1 con la salida del master h2f_lw_axi_master, aparecerá un error en el que nos indica que las memorias de ambos componentes se traslanan. Para resolver este error, damos click en la base de uno de los PIOs y le damos un valor fuera del rango del otro. Por ejemplo, seleccionamos a base del componente pio_1 y le damos el valor 0x0000010, que es justo el que sigue después del final de pio_0. El valor final del rango de direcciones, se corrige de manera automática, y el mensaje de error desaparece.



Si entramos a la pestaña de “Address Map” podemos ver el mapa memoria del sistema que hemos diseñado. En la siguiente figura, vemos que hay tres interfaces mapeadas en memoria. La primera corresponde a un esclavo axi que viene con el hps al que no se le ha asignado ninguna dirección de memoria, mientras que las otras dos corresponden a las interfaces que hemos creado en esta sección. Vemos que dentro de la interfaz de memoria h2f_lw_axi_master aparecen las direcciones de memoria que se asignaron anteriormente.



El presente diseño de Qsys ya puede ser usado para generar el código, sin embargo, el propósito de esta sección ha sido meramente aprender a manejar los componentes dentro de Qsys. En la siguiente sección utilizaremos un diseño hecho por el autor para generar código en VHDL que se conectará con un diseño en el FPGA.

Implementación de comunicación ARM-FPGA

En esta sección mostraremos cómo la implementación de un sistema que permite enviar datos desde una aplicación de software que se ejecute en el procesador ARM hacia la capa física del FPGA, realizar una función en hardware y recuperar los resultados en la aplicación de software. Esta comunicación se basa en los periféricos PIO descritos en la sección anterior. Utilizaremos un repositorio que se encuentra en el github del autor de esta guía (<https://github.com/iluan/SimpleDE0>). Este repositorio puede descargarse manualmente en la página web o directamente desde la terminal de linux con el comando:

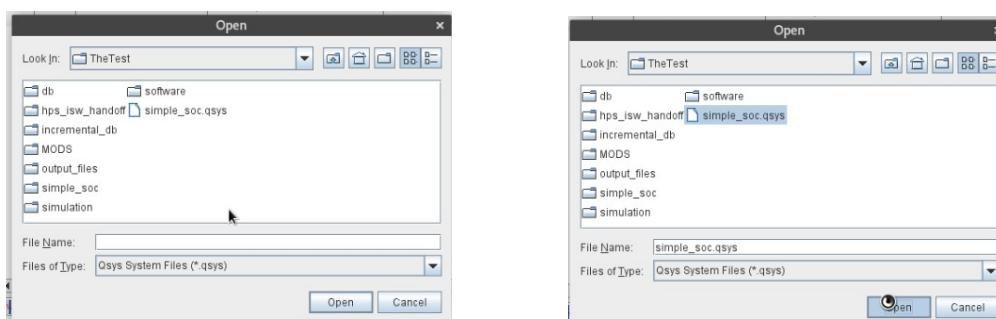
```
git clone https://github.com/iluan/SimpleDE0
```

Dentro de este repositorio, se encuentra el archivo de proyecto soc_system.qpf sobre el cual trabajaremos en esta demostración. La entidad principal de este proyecto se llama Overwrap, que consiste en una implementación en VHDL del “Diseño de Oro de Sistema de Hardware de Referencia” (En inglés GHRD: Golden Hardware Reference Design) el cual es un diseño de referencia provisto por terasic para cada una de sus tarjetas, que incluye todos los uertos del FPGA. Desafortunadamente, este diseño está en el lenguaje de descripción de hardware Verilog, por lo que el autor decidió hacer su propia implementación a fin de trabajar en un solo lenguaje. El diseño está pensado como una plantilla base, sobre la cual el usuario puede implementar sus propios sistemas. Además de la entidad principal, el proyecto incluye un archivo de Qsys para generar la conexión FPGA-ARM y un archivo llamado WRAP0.vhd en el cual el usuario puede incorporar cualquier diseño de hardware que quiera.

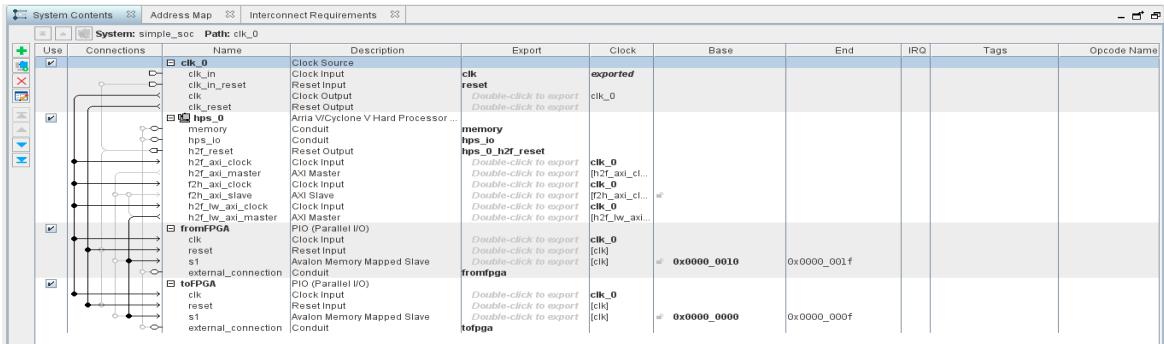
En las secciones subsecuentes, se usará este diseño como una base para mostrar paso a paso cómo implementar la comunicación entre el ARM y el FPGA.

1. Síntesis de componentes en Qsys

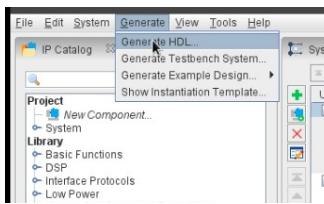
Abrimos en Quartus el diseño soc_system.qpf y luego ejecutamos la herramienta Qsys. Dentro de este programa, abrimos el archivo de diseño simple_soc.qsys.



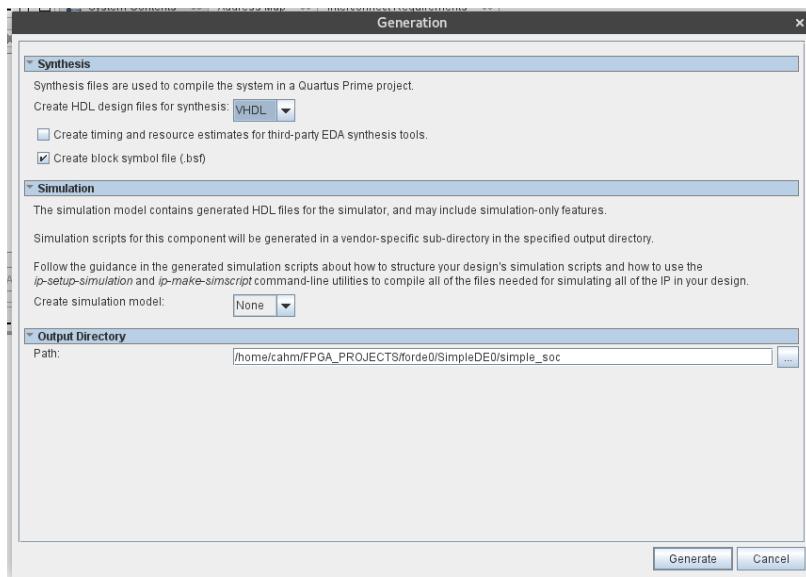
Este diseño, incluye un módulo HPS ya configurado para las especificaciones de la tarjeta DE0-SoC, así como dos componentes PIO de 16-bits ya con sus direcciones correspondientes, y sus interfaces exportadas. Las interfaces están marcadas claramente como “fromFPGA” y “toFPGA” indicando el flujo de datos.



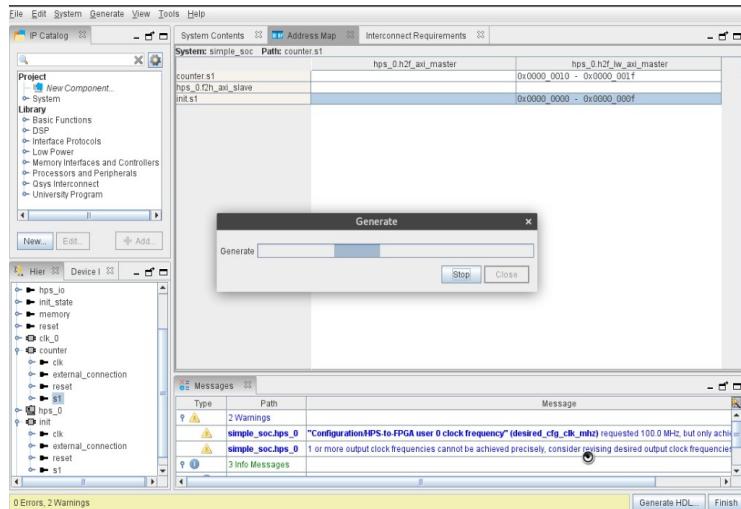
Habiendo cargado el sistema abrimos el menú Generate → Generate HDL... Aparecerá una ventana titulada “Generation” en la que se nos darán opciones para generar el código en lenguaje de descripción de hardware que corresponde al diseño.



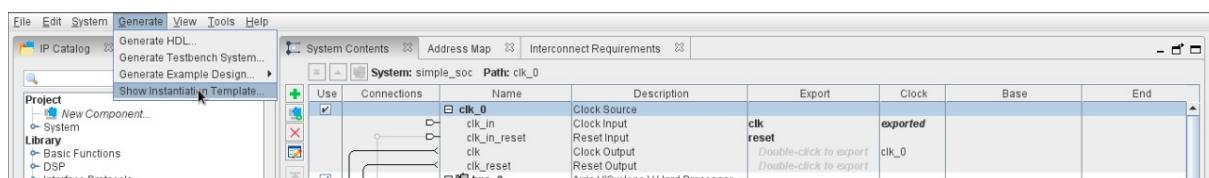
En la categoría síntesis, seleccionaremos el lenguaje de descripción de hardware en el que queremos que se genere el código de nuestro diseño, así como si queremos generar archivos para su simulación con herramientas externas. En la categoría Simulation le indicamos al programa qué lenguaje de descripción debe de tomar en cuenta para hacer una simulación del diseño generado, y finalmente en Output directory se nos pide que indiquemos en qué carpeta se guardarán los archivos generados. La simulación del sistema generado es un paso opcional que omitiremos para agilizar el proceso de síntesis, por lo que seleccionaremos “NONE” en la opción de lenguaje para simulación. En cuanto a las demás opciones, usaremos VHDL como lenguaje de síntesis y dejaremos el directorio de salida en la carpeta que venga por defecto, como se muestra en la siguiente figura.



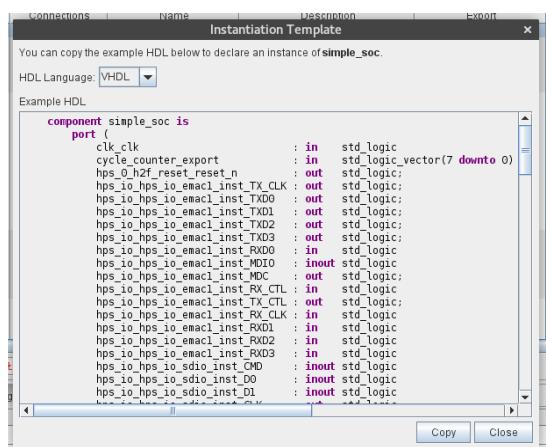
Damos click en generate y se da inicio al proceso de síntesis de los archivos de diseño de hardware.



Cuando se termine la síntesis, antes de cerrar Qsys abrimos el menú Generate -> Show Instantiation Template... y emergerá una ventana en la que se nos da la opción de generar una plantilla de instancia para el componente diseñado.



En la ventana emergente, seleccionamos el lenguaje VHDL y vemos como en el cuadro inferior aparece un código en dicho lenguaje. Este código contiene la declaración de componente que habría que pegar en la entidad principal Overwrap dentro de la arquitectura antes de iniciar la descripción de comportamiento (antes de BEGIN), así como el mapa de puertos que habría que pegar en esta misma arquitectura en la sección en donde se describe el comportamiento del diseño (después de BEGIN). Copiamos las declaraciones y damos click en cerrar. **NOTA: No se necesita hacer esto para esta demostración porque el autor ya las puso en el archivo Overwrap.vhd.** Sin embargo, se muestra el procedimiento porque si el lector quisiera modificar el diseño de Qsys, abrirían cambios en los puertos del sistema generado y por lo tanto habría que modificar las declaraciones correspondientes en la entidad principal.



Si se realiara algún cambio en el diseño, el procedimiento de modificar las declaraciones es el siguiente. Para la declaración de componente, basta con reemplazar la declaración que aparece en el archivo Overwrap por la declaración que aparece en el archivo de instancia. Por otro lado, la declaración de mapa de puertos requiere de una serie de modificaciones que se tienen que hacer manualmente. El problema es que cada puerto del mapa debe de ser conectado con un puerto de la entidad principal y la plantilla generada no incluye estas conexiones. Afortunadamente, el usuario puede tomar como guía el mapa de puertos que viene en el archivo Overwrap.vhd, ya que la mayoría de los puertos siempre van a ser los mismos. Veamos parte del código de esta declaración para entender mejor el problema.

```
u0 : component simple_soc
  port map (
    clk_clk      => FPGA_CLK1_50,          --clk.clk
    fromfpga_export  => F2H,   -fromfpga.export
    hps_0_h2f_reset_reset_n  => HPS_H2F_RST,      --
    hps_0_h2f_reset.reset_n
```

Ejemplo 10. Fragmento de la declaración del componente simple_soc.

En la primera linea del ejemplo tenemos el nombre del componente. Este nombre se genera de manera automática según el nombre del archivo qsys, de modo que será siempre el mismo a menos que el usuario modifique el nombre del archivo. Dentro del mapa de puertos, observamos que el componente simple_soc tiene un puerto llamado fromfpga_export, el cual está conectado con la señal F2H de la entidad principal. Como se mencionó anteriormente, “fromFPGA” es el nombre de uno de los PIOs del diseño de Qsys. Si el usuario decide borrar esta entidad del diseño, tendría que eliminar esta linea del mapa de puertos. Si decidiera cambiarle de nombre, tendría que borrar la linea, buscar el nombre nuevo del puerto en la plantilla generada por Qsys y añadir al mapa de puertos la linea correspondiente. Por ejemplo, supongamos que el usuario modifica el proyecto de qsys cambiando únicamente el nombre de “fromFPGA” a “AAAA”. Adicionalmente, supongamos que el nombre asignado por Qsys para el puerto sea “aaaa_export”. En la declaración de componente en el archivo Overwrap.vhd el usuario primero tendría que borrar la linea “fromfpga_export => F2H, –fromfpga.export”. Luego, tendría que añadir la linea “aaaa_export => F2H, –aaaa_export”, sin embargo, esta linea NO podría ir en el mismo lugar que la linea borrada. El motivo de esto, es que Qsys genera los puertos en orden alfabético, de modo que la linea con el puerto nuevo, tendría que ir en una posición diferente según su nombre. En este caso hipotético, el nombre AAAA sería colocado como el primer elemento de la lista de puertos y, por lo tanto, se tendría que declarar como se muestra en el siguiente ejemplo. Nótese que como sólo se cambió el nombre de la interfaz, pero no su tipo ni su ancho, esta iría conectada a la misma señal. Asimismo, como no se modificaron las demás interfaces, sus puertos irían en el mismo orden y conectados a los mismos elementos de la entidad principal.

```
u0 : component simple_soc
  port map (
    aaaa_export      => F2H, -- aaaa_export
    clk_clk          => FPGA_CLK1_50,          --clk.clk
    hps_0_h2f_reset_reset_n  => HPS_H2F_RST,      --
    hps_0_h2f_reset.reset_n
```

Ejemplo 11. Fragmento modificado de la declaración del componente simple_soc, cambiando el nombre de una interfaz PIO.

En otro caso hipotético, en el que el usuario añadiera una interfaz nueva sin hacer ningún otro cambio al diseño. Usemos el mismo nombre “AAAA” que usamos en el ejemplo anterior y supongamos que esta interfaz tiene un ancho de 8 bits y que Qsys le vuelve a dar el mismo nombre aaaa_export al puerto generado. El usuario tendría que agregar el puerto nuevo a la lista en la posición correcta según el orden alfabético y además tendría que mapear el puerto a un elemento de la entidad principal. En este caso, como no podemos conectar simultáneamente aaaa_export y fromfpga_export al mismo elemento, vincularemos la señal nueva el puerto al puerto LED de la entidad principal (este está vinculado a los pines LED de la tarjeta DE0-SoC), como se muestra en el siguiente ejemplo.

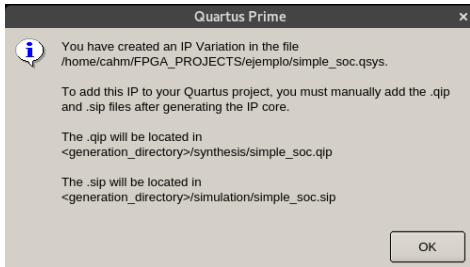
```
u0 : component simple_soc
  port map (
    aaaa_export      => LED, --
    clk_clk          => FPGA_CLK1_50,           --clk.clk
    fromfpga_export  => F2H,      -fromfpga.export
    hps_0_h2f_reset_reset_n => HPS_H2F_RST,      --
    hps_0_h2f_reset.reset_n  => HPS_H2F_RST,      --
```

Ejemplo 12. Fragmento modificado de la declaración del componente simple_soc, agregando una segunda interfaz PIO.

Este mismo procedimiento, aplica a cualquier otra modificación que el usuario desee hacer al diseño en Qsys.

2. Incorporación de elementos generados

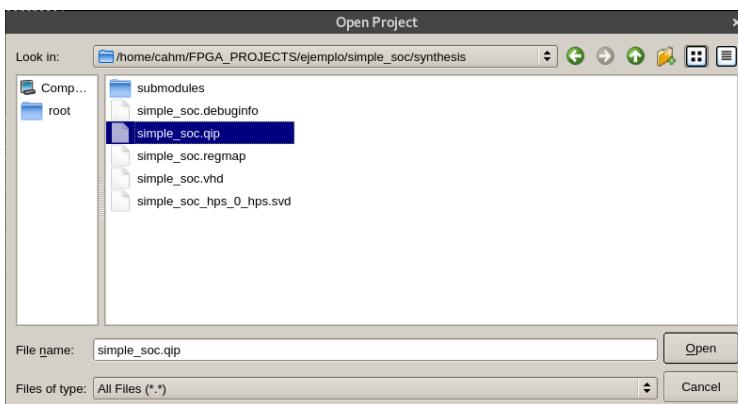
Después de generar el sistema de Qsys, veremos una ventana emergente en la que se nos indica que tenemos que añadir los archivos de síntesis y simulación generados en la sección anterior. A continuación, veremos justamente como hacer eso.



Abrimos “Settings” entramos al menú files, damos click en el botón “...” que aparece junto al campo “File name” y buscamos los archivos que queremos añadir.



Durante el proceso de síntesis, Qsys generó una carpeta llamada simple_soc (u otro nombre en caso de que el usuario haya cambiado el nombre del archivo simple_soc.qsys), dentro de la cual hay una carpeta llamada “synthesis”. En esta carpeta hay un archivo llamado simple_soc.qip, el cual es un archivo de variación de IP. Tal archivo contiene todas las instrucciones que Quartus necesita para ubicar y cargar los diseños de hardware creados por Qsys en la sección anterior.



Abrimos el archivo, damos click en add, aplicamos cambios, damos click en OK y cerramos settings. Nota: en la sección anterior omitimos el paso de simulación, pero si lo hubiéramos realizado, tendríamos que agregar el archivo de simulación al proyecto.

El proceso es virtualmente idéntico que para agregar el archivo de variación de IP, solo que ahora, dentro de la carpeta simple_soc buscamos la carpeta que se llama “simulation” y en esa carpeta abrimos el archivo de simulación simple_soc.sif.

Habiendo añadido estos archivos al proyecto, hay que remover el archivo de Qsys. Esto es importante porque de lo contrario Quartus considerará que las entidades del sistema tienen declaraciones duplicadas, y eso será un error que no nos permitirá compilar el sistema.

3. Incorporación de un diseño de hardware personalizado

Como se mencionó al inicio de este capítulo, dentro del componente WRAP0 el usuario puede poner cualquier diseño de hardware que le plazca. Si abrimos el código, vemos que sólo contiene una declaración de librerías, una declaración de entidad sin puertos y una declaración de arquitectura sin componentes ni descripción de comportamiento.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
entity WRAP0 is
    port (
        --inserta aquí tus puertos
    );
end;
architecture synth of WRAP0 is
--inserta aquí tus señales y componentes
begin
--inserta aquí tus mapas de puerto, procesos y conexiones
end;
```

Ejemplo 13. Plantilla de componente para comunicación

En principio, el usuario puede llenar esta declaración con cualquier cosa, desde un simple buffer hasta un módulo de cómputo dedicado que llame a una larga serie de componentes. En esta demostración, incorporaremos un diseño de lo más simple, el cual simplemente recibe un valor de 16 bits y los pasa todos por una compuerta lógica NOT. Para hacer esto, reemplazamos el contenido de WRAP0.vhd por el siguiente código.

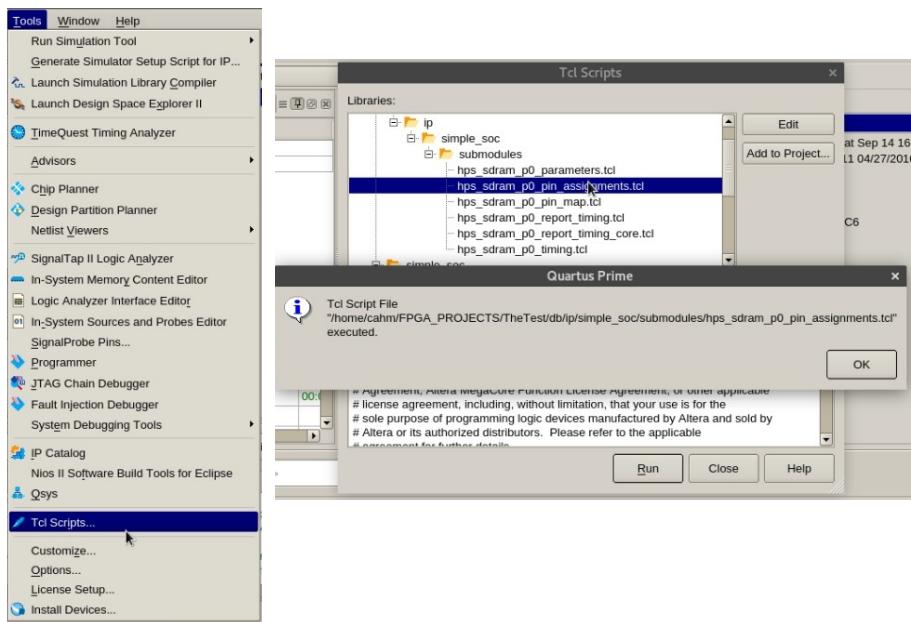
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
entity WRAP0 is
    port (
        fromHPS: in std_logic_vector(15 downto 0);
        toHPS: out std_logic_vector(15 downto 0)
    );
end;
architecture synth of WRAP0 is
--inserta aquí tus señales y componentes
begin
    toHPS<=NOT fromHPS;
end;
```

Ejemplo 14. Circuito simple que recibe una entrada y la pasa por una compuerta NOT.

Guardamos el archivo y nos movemos a la entidad principal -Overwrap.vhd. Vemos que en el diseño del repositorio ya están incluidas estas dos declaraciones y que ya están conectadas al HPS mediante F2H y H2F, cuyos nombres nos indican que son los datos que van del FPGA al HPS y de regreso. Sin otra cosa que cambiar, empezamos análisis y síntesis.

4. Asignación de pines y parámetros

Como en cualquier otra compilación, después de terminar el análisis y síntesis, hay que realizar la asignación de pines y parámetros. En el repositorio de esta plantilla, el autor ha incluido con las asignaciones de pines y parámetros para **casi** todos los pines del sistema. Casi, porque los pines de la RAM del procesador HPS deben de ser configurados de manera específica para cada diseño de Qsys. Estos parámetros específicos, se encuentran en dos archivos Tcl (Tool Comand Language) que son generados por Qsys durante el paso de síntesis. Para ejecutar estos archivos, abrimos el menú Tools → Tcl Scripts.

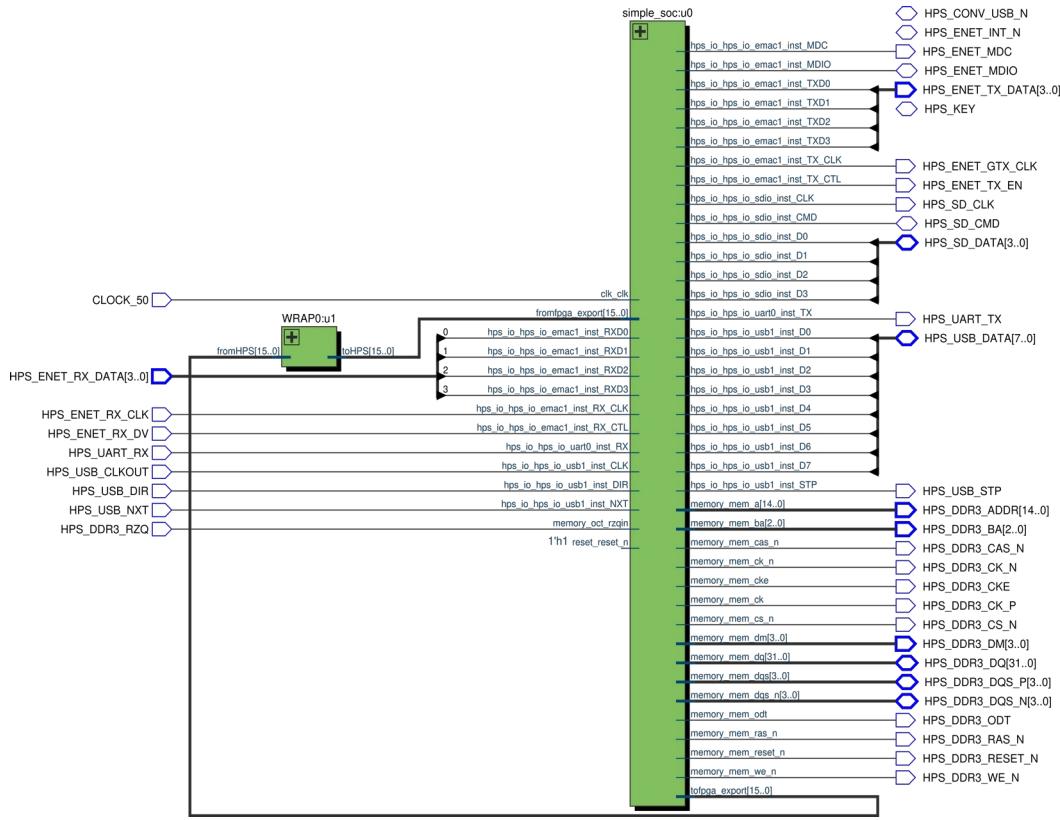


Buscamos y ejecutamos los archivos `hps_sdram_p0_parameters.tcl` y `hps_sdram_p0_pin_assignments.tcl`. Después de la ejecución de cada uno, aparecerá una notificación que nos avisa que no ha habido problema, la cal cerramos dando click en OK. En la lista de archivos Tcl veremos que los archivos aparecen dos veces. No importa cual ejecute, ambos objetos en la lista son iguales.

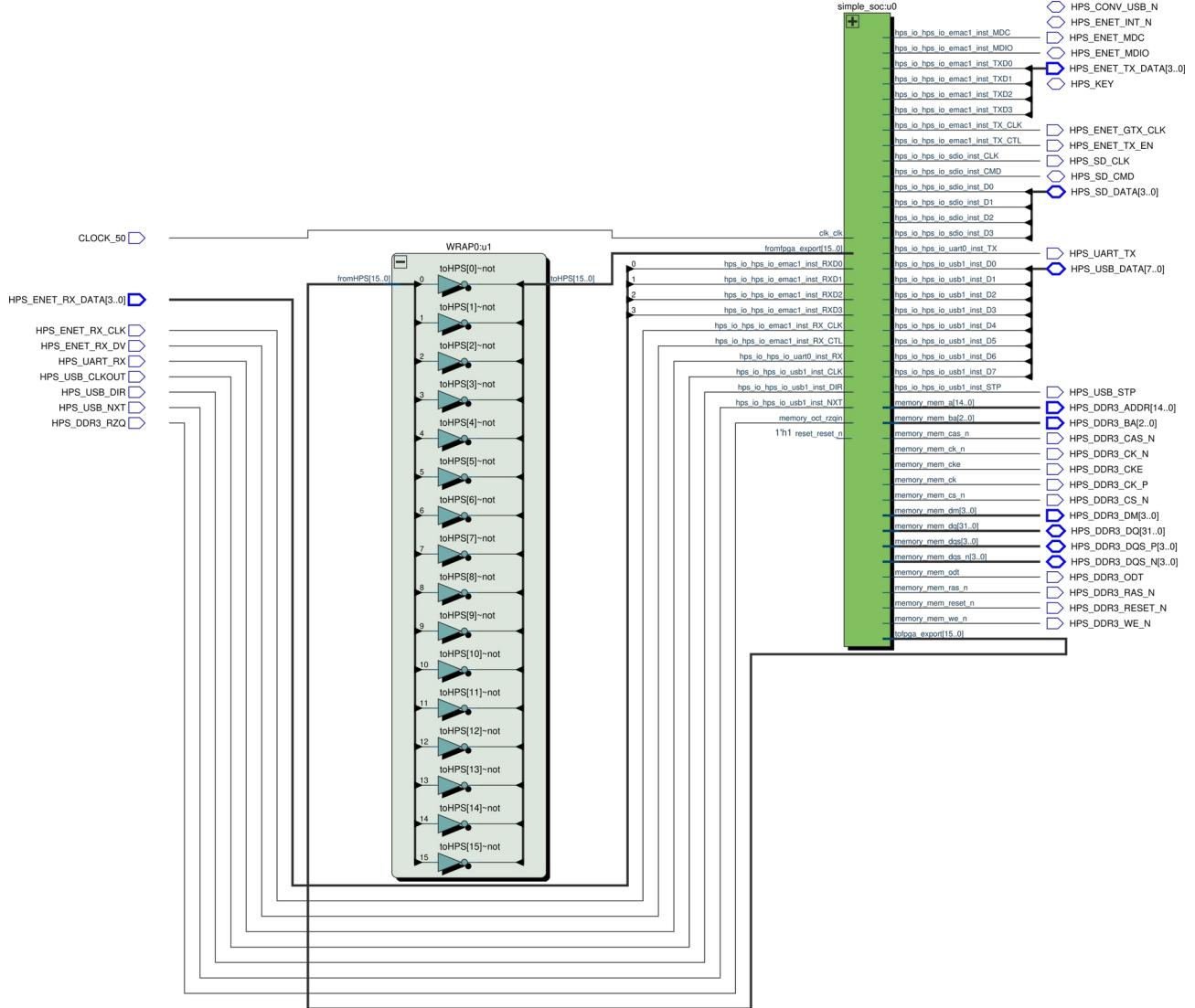
Nota: Si no se hizo el análisis y síntesis del proyecto antes de ejecutar los scripts de Tcl, la ejecución de estos marcará error.

Después de completar la ejecución de los scripts Tcl, iniciamos el resto del proceso de compilación. Si no se ejecutan estos archivos después de continuar, la compilación será abortada por una serie de errores de asignación de pines.

Si vemos el viso de RTL podemos ver que el diseño esta compuesto por los dos módulos que hemos incluido en el diseño. Hay una conexión que sale del componente HPS y entra al componente Wrap0, y a su vez hay una señal que sale de Wrap0 y regresa al HPS.

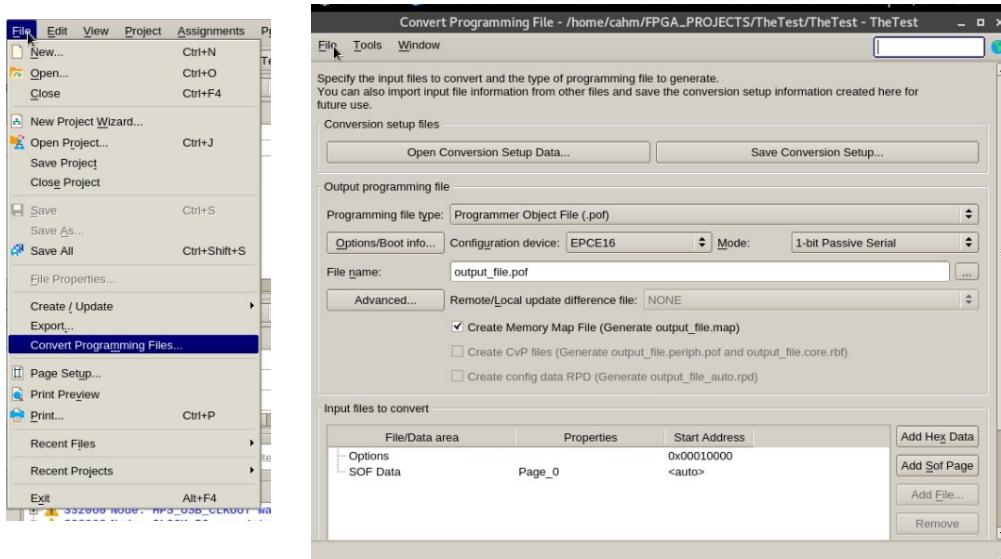


Si abrimos los detalles del componente wrap0, vemos qubit por bit, todas las entradas que recibe el circuito son negadas.

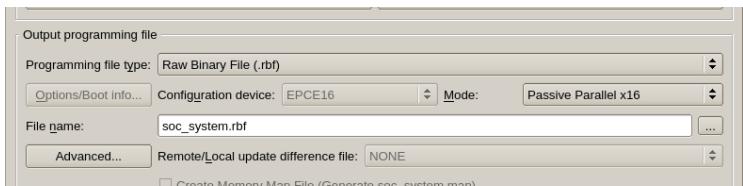


5. Convertir archivos de programación

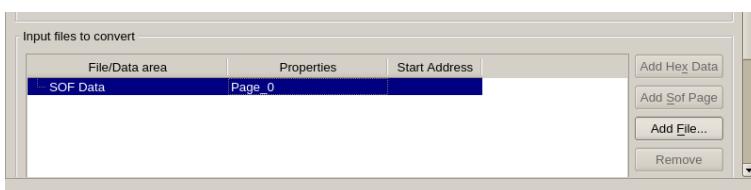
Al principio de este capítulo, mencionamos que el objetivo es enviar datos al FPA desde una aplicación de software en el ARM, por lo que requerimos generar un archivo de programación que pueda ser cargado al FPGA por el sistema operativo. En particular, crearemos un archivo para la interfaz de programación Pasivo Paralelo Rápido de 16 bits (FPPx16 por su abreviatura en inglés). Si hemos concretado exitosamente la compilación del diseño, abrimos el menú File → Convert Programming Files. Emerge una ventana con la aplicación de convertidor, la cual trae varias opciones que tenemos que modificar.



Para empezar, cambiamos el tipo de archivo de programación de Programmer Object File (.pof) a Raw Binary File (.rbf), seleccionamos el modo Pasivo Paralelo x 16 y designamos el nombre del archivo como soc_system.rbf.

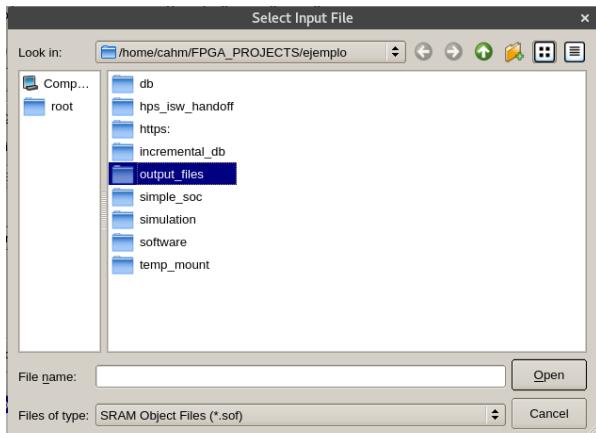


Despues, vamos a la sección de Input File to convert, seleccionamos la entrada “SOF Data” y damos click en Add File.

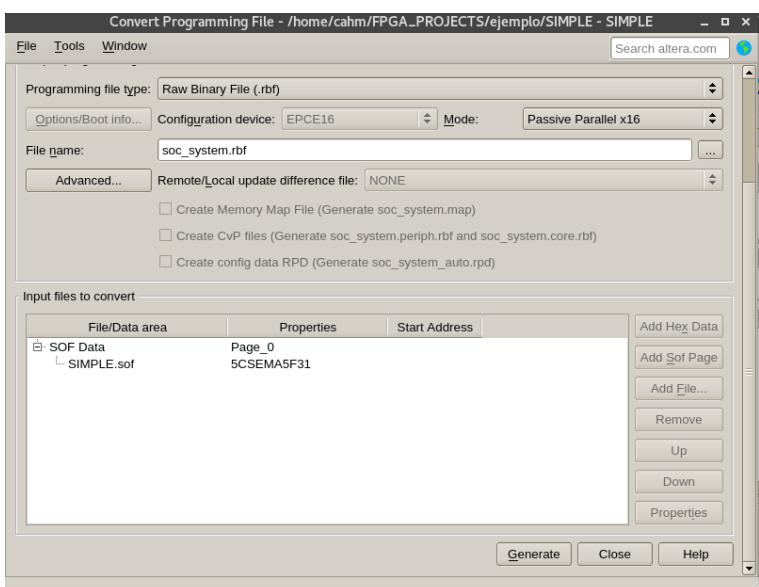


Vamos a buscar el archivo de programación generado por Quartus durante la compilación del diseño, el cual debe de estar dentro de la carpeta output_files y debe de ser de tipo .sof. El lector

quizás note que en el repositorio tal cual se descargó de la página de github no hay ninguna carpeta llamada output_files. Esta carpeta, entre otras, es generada a lo largo del proceso de quartus y Qsys.



Después de abrir el archivo, nos cercioramos de que todos los parámetros estén correctos (incluido el nombre) y damos click en generar.



Esto concluye la parte del proceso que se realiza dentro de la suite de Quartus, los pasos posteriores se realizarán con la suite de diseño embebido.

6. Crear sistema de precarga

La capa de software que ejecutaremos en el ARM estará montada sobre un sistema operativo Linux. Por lo tanto, antes de seguir avanzando en la implementación de este sistema, hay que analizar algunos conceptos básicos del sistema de arranque de un sistema operativo. De manera muy simple, el arranque de un sistema operativo comienza con la ejecución de una secuencia de funciones de bajo nivel específicas al microprocesador, en las que se habilitan los componentes de hardware de manera ordenada. Primero se ejecutan una serie de directivas que vienen programadas de fábrica en la ROM, las cuales llaman a un programa de precarga (“preLoader”) que se encuentra en memoria externa. A este programa se le llama también Programa Cargador Secundario (SPL por sus siglas en inglés Secondary Program Loader). El programa de precarga se encarga de configurar el microprocesador a bajo nivel, para asegurarse de que el cargador de arranque del sistema operativo

pueda ser ejecutado. Dado que se estamos modificando las configuraciones de hardware en la capa de de FPGA del dispositivo, es necesario generar un SPL específico para nuestro diseño.

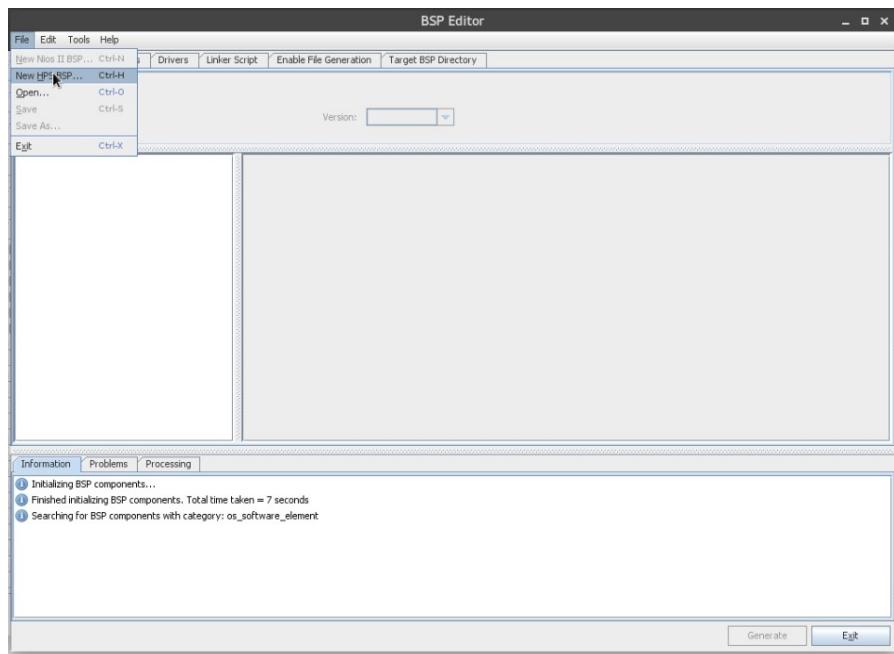
En esta demostración usaremos el cargador de arranque U-Boot, ofrece la facilidad de que el código fuente de su programa de precarga puede ser modificado para generar programas de precarga específicos. Para generar de manera automática la porción del código fuente necesario para obtener un SPL basado en U-Boot que se acople a nuestro diseño de FPGA usaremos el programa BSP Editor de Altera/Intel.

Para llamar a esta herramienta, primer ejecutamos el programa llamado terminal de comandos embebidos (embedded_command_shell) que forma parte de la suite embebida de Altera. En windows esta suite se inicia con el comando Embedded_Command_Shell.bat, mientras que en linux es inicia con el comando embedded_command_shell.sh. En linux, si se siguió el paso de establecer las variables de entorno como se indicó en el capítulo de instalación, este comando puede ser ejecutado directamente en el directorio del proyecto.

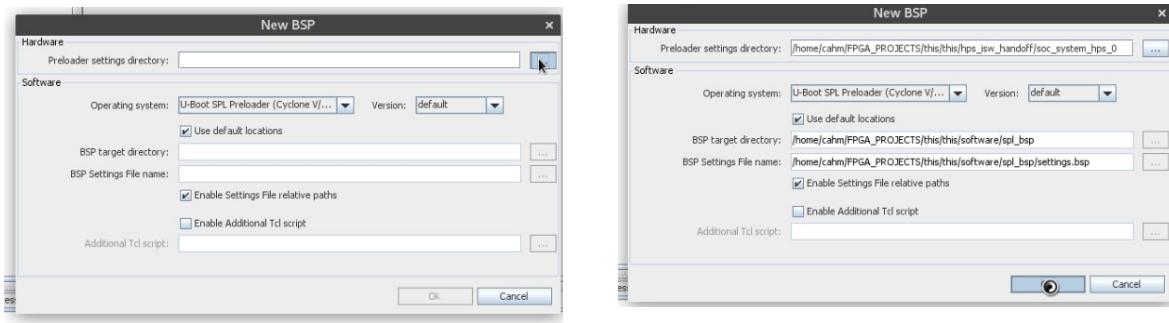
En la terminal de comandos embebidos, navegamos al directorio del proyecto (si no iniciamos ahí) y ejecutamos el comando:

```
bsp-editor &.
```

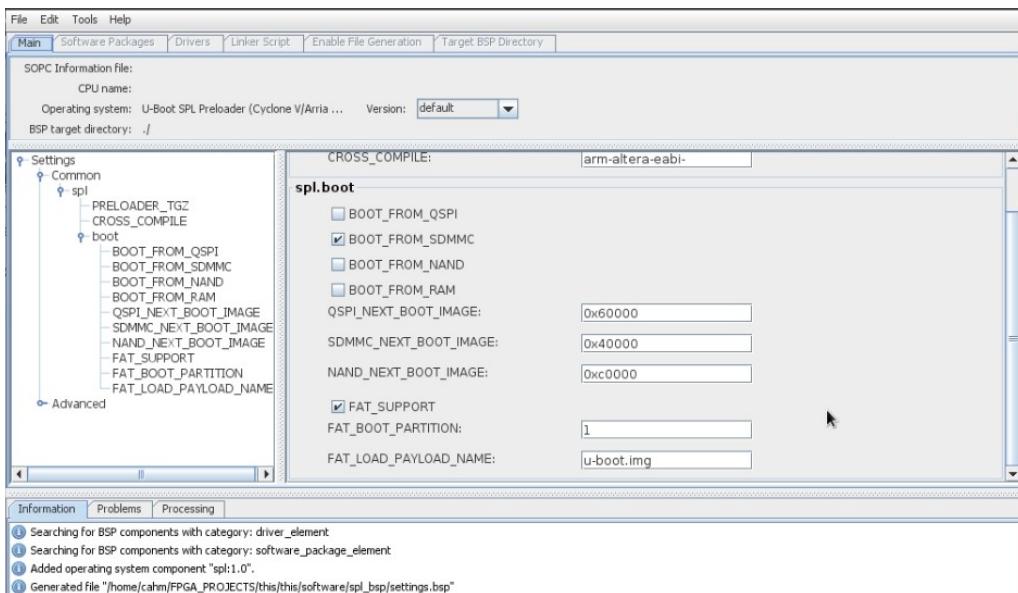
Este comando abre la GUI del editor de Paquetes de Soporte De Tarjeta (Board Supoprt Package). Este programa, generará los archivos de configuración para compilar el sistema de precarga. Nótese que ese “&” al final de la linea es opcional y solo sirve para indicarle al sistema que queremos seguir usando la terminal mientras trabajamos en la GUI. Ya en el programa BSP Editor, creamos un archivo nuevo mediante el menú File → New HPS BSP.



En la ventana emergente nos preguntará por el directorio de las configuraciones del sistema de precarga (Preloader settings directory), damos click en “...”, en el navegador entramos la carpeta “hps_isw_handoff/soc_system_hps_0” y damos click en “open”.



De regreso en la ventana de nuevo BSP damos click en OK y esperamos un momento mientras se crea el nuevo archivo de configuración. Ya creado el archivo de configuración, en la sección de “common” buscamos y habilitamos la opción FAT_SUPPORT, para indicar que necesitamos usar un sistema de archivos tipo FAT.



En esta misma sección, nos aseguramos de que la partición de arranque FAT (FAT_BOOT_PARTITION) esté puesta en “1” y que la opción BOOT_FROM_SDMMC esté activa. Esto es para asegurarnos de que el sistema de arranque cargará de la tarjeta SD, y que el sistema de precarga buscará el sistema de arranque en la primera partición. En las demás opciones no ha que cambiar nada, pero si es recomendable asegurarnos que en “Advanced → SPL → Reset Assert no haya ninguna opción activa, y que en SAdvanced→spl→performance” la opción “SERIAL_SUPPORT” si lo esté. Estas opciones nos permiten asegurarnos de que el sistema de arranque no esté atorado en Reset y que éste va a enviar mensajes de sistema a través de la interfaz UART durante todo el arranque. Damos click en generar, esperamos un par de segundos en lo que se generan los archivos y cerramos el editor de BSP.

Desde la carpeta del proyecto, navegamos a la carpeta del sistema de precarga mediante el comando

```
cd software/spl_bsp.
```

Si dentro de este directorio enlistamos todos los archivos mediante el comando **ls**, veremos que hay dentro de este. Sin cambiar de directorio, ejecutamos el comando;

```
make
```

Con el cual se inicia la compilación del sistema de precarga. Cuando la compilación haya terminado, veremos que se generó un nuevo directorio llamado “**uboot-socfpga**”.

7. Compilación del sistema de arranque

Ya habiendo creado los archivos del sistema de precarga, compilaremos el sistema de arranque. La compilación de este programa, así como la de todos los demás componentes de software, usaremos la cadena de herramientas de compilación cruzada `gcc-linaro-arm-linux-gnueabihf-4.9-2`. Necesitamos una herramienta de compilación cruzada, porque el microprocesador en el que se ejecutará el sistema operativo tiene una arquitectura tipo ARM, y el procesador de nuestra PC muy posiblemente tenga una arquitectura de tipo x86_64 o i686. Elegimos esta versión particular de la cadena de herramientas, porque incluye el set de instrucciones ARMv7 que utiliza el procesador ARM cortex A9 embebido en dispositivo CycloneV.

Si seguimos dentro de la carpeta `software/spl_bsp.`, hay que movernos a la carpeta `software`. Eso lo hacemos con el comando

```
cd ..
```

Ya en la carpeta `software`, vamos a descargar la cadena de herramientas que mencionamos anteriormente. Esto lo podemos hacer directamente con el comando

```
wget  
https://releases.linaro.org/archive/14.09/components/toolchain/binaries/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux.tar.xz
```

Cuando se acabe la descarga, descomprimimos el archivo con el comando:

```
tar -xvf gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux.tar.xz
```

Cuando se descomprima, los archivos ejecutables de la cadena de herramientas estarán en la carpeta **gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux**. Para no tener que hacer referencia a esta ubicación en cada paso subsecuente, exportaremos la ubicación del compilador como una variable de entorno mediante el comando.

```
export CROSS_COMPILE=$PWD/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux/bin/arm-linux-gnueabihf-
```

NOTA: esta exportación no es permanente, si uno sale de la terminal de comandos embebidos, tendría que volver a exportar este directorio. Opcionalmente, podemos mover esta carpeta a un directorio que no esté asociado a un proyecto específico y exportar su ubicación como una variable de entorno.

Ahora, descargaremos el código fuente de **u-boot** modificado por con el que compilaremos por fin el sistema de arranque. Esto lo hacemos mediante el comando:

```
git clone https://github.com/altera-opensource/u-boot-socfpga.git
```

Los archivos del repositorio serán descargado a la **carpeta u-boot-socfpga**, a la que entramos con el comando

```
cd u-boot-socfpga
```

Dentro de la carpeta ejecutamos el comando:

```
git checkout rel_socfpga_v2013.01.01_15.09.01_pr
```

Para establecer que vamos a usar la rama del proyecto correspondiente a septiembre de 2015. Existen versiones más recientes, pero el autor de esta guía no puede garantizar que la compilación será exitosa con las versiones nuevas.

Ya habiendo establecido la rama del proyecto, limpiamos el directorio mediante el comando

```
make mrproper
```

Dentro del direcorio de u-boot-socfpga, configuramos de acuerdo a las especificaciones del dispositivo Cycloe V, para esto, ejecutamos el comando:

```
make socfpga_cyclone5_config
```

Despues de que se ejecute la configuración, compilamos u-boot con el comando

```
make
```

Ya habiendo compiado u-boot, ahora generaremos un script de arranque. Este script es usado pr el propio u-boot para ejecutar instrucciones particulares en el proceso de arranque.

En la carpeta de software, abrimos nuestro editor de texto de preferencia (desde la terminal, podemos usar vim) y creamos un archivo que se llame “boot.script”.

Dentro de este archivo, pegamos el siguiente código

```
echo -- Programming FPGA --
fatload mmc 0:1 $fpgadata soc_system.rbf;
fpga load 0 $fpgadata $filesize;
run bridge_enable_handoff;

echo -- Setting Env Variables --
setenv fdtimage soc_system.dtb;
setenv mmcroot /dev/mmcblk0p2;
setenv mmcload 'mmc rescan;${mmcloadcmd} mmc 0:${mmcloadpart} ${loadaddr} ${bootimage};${mmcloadcmd} mmc 0:${mmcloadpart} ${fdtaddr} ${fdtimage};';
setenv mmcboot 'setenv bootargs console=ttyS0,115200 root=${mmcroot} rw
rootwait; bootz ${loadaddr} - ${fdtaddr}';

run mmcload;
run mmcboot;
```

Este es un script en una versión simplificada de bash, que le indica a u-boot que debe de cargar el programa al FPGA y que debe de establecer las variables de entorno que le permitirán al sistema operativo acceder a este dispositivo. El script debe de ser compilado en un binario para poder ser cargado en el arranque. Esto lo hacemos con el comando

```
mkimage -A arm -O linux -T script -C none -a 0 -e 0 -n "Boot Script Name" -d boot.script u-boot.scr
```

8. Generar y compilar el árbol de direcciones

Dado que nuestro diseño de FPGA se conecta físicamente al procesador mediante ciertas direcciones de memoria en la interfaz Avalon, necesitamos generar un árbol de dispositivos para nuestro diseño. Los árboles de dispositivos son estructuras de datos que determina las direcciones de los componentes de hardware en un dispositivo, de modo que puedan ser accedidos por el Kernel, además, especifican los anchos de memoria física para cada dispositivo y los drivers que deben de ser cargados para controlarlo. Para generar el árbol de dispositivos, hay que regresar a la carpeta del proyecto de quartus. En el paso anterior terminamos en la carpeta software, por lo que podemos regresar a la carpeta del proyecto con el comando

```
cd ..
```

Ya en la dirección del proyecto, generaremos el código fuente para nuestro árbol de direcciones utilizando la herramienta de la terminal de sistemas embebidas “sopc2dts”. Esta herramienta usa los archivos de configuración generados en Qsys para generar el código fuente del árbol de dispositivos. Así pues, ejecutamos el comando:

```
sopc2dts --input simple_soc.sopcinfo --output soc_system.dts --type dts --board soc_system_board_info.xml --board hps_common_board_info.xml --bridge-removal all --clocks
```

En este comando, el primer argumento dice que tomamos como entrada al archivo simple_soc.sopcinfo; este archivo contiene la información generada por Qsys para la interfaz HPS-FPGA. El segundo argumento, dice que la salida va a ser el archivo soc_system.dts. El tercer argumento indica que el tipo de esta salida efectivamente es un archivo de código fuente de árbol de direcciones. Los argumentos cuarto y quinto son argumentos específicos a la tarjeta, y corresponden a archivos xml creados por terasic con información de la misma. Los últimos dos argumentos tratan de las estrategias que debe de seguir DTS para manejar los relojes y puentes entre el FPG y el HPS.

Finalmente, el código fuente del árbol de dispositivos se compila en un binario llamado device tree blob mediante el programa DTC (Device Tree Compiler). Para hacer esto, utilizamos el siguiente comando:

```
dtc -I dts -O dtb -o soc_system.dtb soc_system.dts
```

Los primeros dos argumentos le indican al programa que estamos usando un .dts como entrada y un .dtb como salida. NOTA: En principio esto puede hacerse al revés, descompilando un archivo dtb en su código fuente correspondiente.

9. Crear particiones y el sistema de archivos

El sistema operativo linux que estamos desarrollando, requiere que el disco duro o memoria en que esté instalado tenga tres particiones: una para el sistema de arranque y sus archivos asociados, otra para el sistema de archivos del sistema operativo, y una para el sistema de precarga.

Para crear un esquema de particiones, primero que nade crearemos una imagen de disco sobre la cual luego particionaremos. En el directorio del proyecto de quartus, ejecutaremos el siguiente comando:

```
sudo dd if=/dev/zero of=sdcard.img bs=512M count=1
```

En este caso, dd es el programa disk dump, el cual permite hacer toda una serie de operaciones relacionadas al manejo de discos duros e imágenes de discos. El argumento if, indica cuál es el archivo que vamos a leer. En este caso, leemos dev/zero, que es un archivo especial de los sistemas tipo UNIX que provee de tantos caracteres NULL como se lean de éste. El argumento of, es el archivo que vamos a escribir. En este caso, sdcard.img es un archivo de imagen de disco que no existe, pero que vamos a crear mediante dd. El argumento bs, le dice a dd cuantos bytes van a ser leídos en un bloque, y cout indica cuantos bloques van a ser leídos. Entonces, el comando anterior genera una imagen de un bloque de 512Mbytes formada por caracteres NULL.

Para crear un esquema de particiones sobre esta imagen, la vamos a montar como si fuera un disco, en un dispositivo de tipo loopback, que es un archivo de dispositivo (un dispositivo virtual mapeado en la RAM). Para hacer esto, usamos el comando

```
sudo losetup --show -f sdcard.img
```

La salida de este comando, nos indica cuál es dispositivo sobre el cual se montó la imagen. Si no hay otro dispositivo virtual montado, lo más probable es que el resultado sea:

```
/dev/loop0
```

El lector debe fijarse bien en este resultado, porque si fuera diferente habría que reemplazar el término dev/loop0 de todos los comandos que siguen en esta sección por el resultado que se obtenga.

Para generar particiones sobre la imagen de disco montada, utilizaremos el programa fdisk. Éste programa nos permite diseñar un esquema de particiones que solo se escribe en el dispositivo hasta que indiquemos. Para indicarle a fdisk que va a crear particiones para el dispositivo virtual que tenemos montado, ejecutamos el comando

```
sudo fdisk /dev/loop0
```

Este comando abre una interfaz de usuario de texto que dice lo siguiente:

```
Welcome to fdisk (util-linux 2.30.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.
```

```
Device does not contain a recognized partition table.
Created a new DOS disklabel with disk identifier 0xf3236d5b.
```

```
Command (m for help) :
```

Aquí, la interfaz nos dice que por el momento el dispositivo sobre el que estamos trabajando no tiene ninguna tabla de particiones. Para ver las opciones que tenemos en la interfaz de texto, escribimos la letra “m” y presionamos enter.

```
Command (m for help): m
Command action
  a  toggle a bootable flag
  b  edit bsd disklabel
  c  toggle the dos compatibility flag
  d  delete a partition
  g  create a new empty GPT partition table
  G  create an IRIX (SGI) partition table
  l  list known partition types
  m  print this menu
  n  add a new partition
  o  create a new empty DOS partition table
  p  print the partition table
  q  quit without saving changes
  s  create a new empty Sun disklabel
  t  change a partition's system id
  u  change display/entry units
  v  verify the partition table
  w  write table to disk and exit
  x  extra functionality (experts only)
```

Lo primero que haremos, será crear la partición donde estará la imagen del sistema de precarga. Para esto, primero ingresamos el comando “n” para crear una partición nueva. El programa nos pregunta qué tipo de partición vamos a crear. Presionamos “p” para indicar que esta partición va a ser una partición primaria.

```
Command (m for help): n
Partition type:
  p  primary (0 primary, 0 extended, 4 free)
  e  extended
Select (default p): p
Partition number (1-4, default 1): 3
First sector (2048-1048575, default 2048):
Using default value 2048
Last sector, +sectors or +size{K,M,G} (2048-1048575, default 1048575): +1M
Partition 3 of type Linux and of size 1 MiB is set
Command (m for help):
```

En el número de partición indicamos que esta es la tercera partición. Luego damos pregunta cual es el sector inicial de la partición. Presionamos Enter sin poner ningún valor para usar el valor por defecto. Cuando nos pida el sector final, escribimos +1M, para indicar que este valor está un Megabyte después del inicio. Vemos que por defecto esta partición generara es de tipo Linux. Sin embargo, este tipo de partición no puede ser leído por las directivas programadas en ROM que llaman al sistema de precarga. Tenemos que cambiar esta partición a una de tipo A2 (Altera custom), la cual es una partición de datos crudos sin ningún sistema de archivos.

```
Command (m for help): t
Selected partition 3
Hex code (type L to list all codes): a2
Changed type of partition 'Linux' to 'unknown'
Command (m for help):
```

Como se muestra en la imagen anterior, ingresamos el comando “t” para cambiar el tipo de una partición. Como solo hay una partición creada, el programa automáticamente elige la partición 3. Cuando nos pide el código de la partición, ingresamos a2, y el sistema nos indicará que cambió la partición de “Linux” a “unknown”. Ahora vamos a crear la partición de tipo linux en la que montaremos el sistema de archivos del sistema operativo. Para esto, ingresamos nuevamente el comando “n” para crear una partición nueva. Volvemos a indicar que es una partición primaria con el comando “p”, declaramos que esta es la partición número 2, presionamos Enter sin poner ningún valor para usar el valor por defecto del primer sector, escribimos +254M, para indicar que este valor del último sector está a 254 Megabytes después del inicio. Vemos que por defecto esta partición generara es de tipo Linux, lo cual es correcto y no hay que modificarla.

```
Command (m for help): n
Partition type:
  p  primary (1 primary, 0 extended, 3 free)
  e  extended
Select (default p): p
Partition number (1,2,4, default 1): 2
First sector (4096-1048575, default 4096):
Using default value 4096
Last sector, +sectors or +size{K,M,G} (4096-1048575, default 1048575): +254M
Partition 2 of type Linux and of size 254 MiB is set

Command (m for help):
```

Finalmente, crearemos la partición del sistema de arranque, la cual será de tipo FAT 95. Ingresamos una vez más el comando “n” para crear una partición nueva, indicamos que es una partición primaria con el comando “p”, declaramos que esta es la partición número 1, presionamos Enter sin poner ningún valor para usar el valor por defecto del primer sector y otra vez enter para usar el valor por defecto del último sector.

```
Command (m for help): n
Partition type:
  p  primary (2 primary, 0 extended, 2 free)
  e  extended
Select (default p): p
Partition number (1,4, default 1): 1
First sector (524288-1048575, default 524288):
Using default value 524288
Last sector, +sectors or +size{K,M,G} (524288-1048575, default 1048575):
Using default value 1048575
Partition 1 of type Linux and of size 256 MiB is set

Command (m for help):
```

La partición tendrá un tamaño de 256 Megabytes y será de tipo Linux. Hay que cambiar el tipo de la partición a FAT. Para esto volvemos a ingresar el comando t, seleccionamos la partición número 1, e ingresamos el código “b” para indicar que es una partición de tipo FAT 95.

```
Command (m for help): t
Partition number (1-3, default 3): 1
Hex code (type L to list all codes): b

WARNING: If you have created or modified any DOS 6.xpartitions, please see the fdisk manual page for additionalinformation.

Changed type of partition 'Linux' to 'W95 FAT32'

Command (m for help):
```

Usamos el comando p para ver la tabla de particiones, la cual se debería de ver como en la siguiente figura.

```
Command (m for help): p
Disk /dev/loop0: 536 MB, 536870912 bytes, 1048576 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x7c9ale40

      Device Boot      Start        End     Blocks   Id  System
/dev/loop0p1          524288    1048575    262144    b  W95 FAT32
/dev/loop0p2           4096     524287    260096    83  Linux
/dev/loop0p3          2048     4095       1024    a2 Unknown

Partition table entries are not in disk order

Command (m for help):
```

Vemos que en la tabla de particiones, tenemos las tres particiones generadas, las cuales se asocian a los dispositivos de fichero /dev/loop0p1, /dev/loop0p2 y /dev/loop0p3.

Ingresamos el comando w, para guardar los cambios, escribir las particiones en el dispositivo /dev/loop0 y cerrar fdisk.

Las modificaciones que hicimos a la tabla de particiones no son detectadas automáticamente por el sistema operativo. Para corregir este problema, utilizaremos el comando partprobe como se muestra a continuación:

```
sudo partprobe /dev/loop0
```

Aquí estamos indicando que queremos detectar las particiones del dispositivo /dev/loop0. Verificamos que las particiones hayan sido detectadas con el comando

```
ls /dev/loop0*
```

O bien con el comando:

```
lsblk
```

Cualquiera de los dos nos deben de indicar la existencia de las particiones /dev/loop0p1, /dev/loop0p2 y /dev/loop0p3 dentro del dispositivo /dev/loop0.

Ahora vamos a configurar las particiones que hemos creado. Primero, vamos a escribir la imagen del sistema de precarga en la partición de tipo A2. Esto lo haremos con dd usando el siguiente comando:

```
sudo dd if=software/spl_bsp/preloader-mkpimage.bin of=/dev/loop0p3 bs=64k seek=0
```

Aquí estamos indicandole a dd que lea el archivo de la imagen del sistema de precarga, que la escriba sobre la partición /dev/loop0p3 y que el tamaño del bloque son 64Kbytes. Con el argumento seek=0, estamos indicándole al programa que no se salte ningún bloque al inicio de la partición en la que va a escribir.

La partición sobre la que acabamos de escribir no tenía ningún sistema de archivos y no lo necesita porque la imagen que escribimos encima de ella es un binario crudo. Sin embargo, las demás particiones sí necesitan un sistema de archivos. Usaremos el progama mkfs para generar los sistemas de archivos correspondientes.

Primero, crearemos un sistema de arvhivos vfat en la partición /dev/loop0p1 mediante el comando:

```
sudo mkfs -t vfat /dev/loop0p1
```

Si a la salida de este comando vemos un mensaje que dice “unable to get drive geometry, using default 255/63” no hay nada de que preocuparse, eso es lo que esperamos.

Luego crearemos el sistema de archivos tipo Ext4 para la partición del sistema de archivos de Linux. Usaremos el comando:

```
sudo mkfs.ext4 /dev/loop0p2
```

Ya habiendo creado los formatos de archivos en cada partición, lo que sigue es pegar los archivos generados en los pasos anteriores en la partición del programa de arranque del sistema operativo.

Para hacer esto, primero creamos un directorio temporal, luego montamos la partición /dev/loop0p1 en este directorio y finalmente pegamos todos los archivos. Usamos los comandos:

```
mkdir temp_mount  
sudo mount /dev/loop0p1 ./temp_mount  
sudo cp software/u-boot-socfpga/u-boot.img software/u-boot.scr soc_system.dtb  
soc_system.rbf temp_mount
```

Después de copiar todo, nos aseguramos de que todo los archivos que estemos guardando hayan pasado de la RAM al disco/memoria-no-volatil. Usamos el comando:

```
sync
```

Desmontamos la partición del directorio con el comando

```
sudo umount temp_mount
```

Para continuar con este proceso, necesitamos insertar la tarjeta microSD. Después de insertar la tarjeta, identificamos el nombre que el sistema operativo le asignó a la tarjeta con el comando **lsblk**. Un nombre usual para este tipo de dispositivos es **mmcblk0**, así que para los pasos subsecuentes, esta guía manejará el nombre /dev/mmcblk0. Ahora, quemaremos la imagen de disco en la tarjeta microSD usando los comandos:

```
sudo dd if=sdcard.img of=/dev/mmcblk0 bs=2048  
sync
```

Hasta que sync no se haya completado, no remueva la tarjeta de la computadora.

10. Probar el sistema de arranque

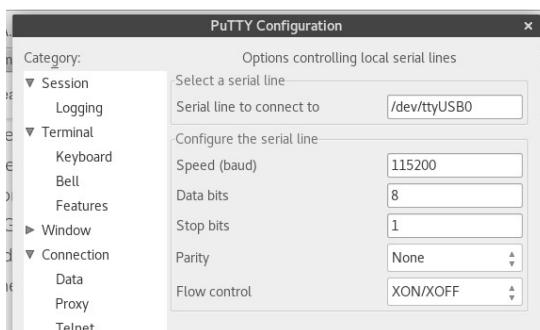
Inserte la memoria microSD en la ranura de la tarjeta DE0 y conecte el cable serial entre la tarjeta y la PC. Para identificar el dispositivo que corresponde a este cable, en linux usaremos el comando:

```
ls /dev/ttyUSB*
```

Por ejemplo, puede ser /dev/ttyUSB0

En windows, identificamos el puerto siguiendo las instrucciones de la siguiente página <https://uplogix.com/docs/local-manager-user-guide/introduction/connecting-to-usb-console-windows-10>

Ejecutamos PuttY en modo administrador, y configuramos la conexión.



En la configuración, ponemos el nombre del puerto que hayamos identificado, un baud rate de 115200, 8 bits de datos, un bit de stop, sin paridad y sin flow control. Damos open para iniciar la comunicación. En la consola veremos los mensajes de arranque del sistema, Hasta que eventualmente veremos el siguiente mensaje.

```
-- Setting Env Variables --
reading zImage
** Unable to read file zImage ***
reading soc_system.dtb
24275 bytes read in 8 ms (2.9 MiB/s)
Bad Linux ARM zImage magic!
```

Este mensaje nos indica que no hay un kernel del sistema operativo, como es de esperarse en este punto de la implementación del sistema. Cerramos Putty, apagamos la tarjeta y volvemos a insertar la memoria microSD a la computadora.

11. Compilación del Kernel

En la carpeta de software, vamos a descargar un repositorio correspondiente a un fork del Kernel de linux manejado por Altera/Intel.

```
cd software
git clone https://github.com/altera-opensource/linux-socfpga.git
```

Esta descarga puede tardar varios minutos. Cuando termine, entramos al directorio y elegimos la rama del proyecto:

```
cd linux-socfpga
git checkout rel_socfpga-4.1_15.09.01_pr
```

Habiendo seleccionado la rama del proyecto, iniciamos la configuración previa a la compilación. Primero que nada, establecemos que vamos a usar arquitectura ARM y que vamos a cargar las configuraciones default para un sistema FPGA-SoC. Esto lo hacemos con el siguiente comando:

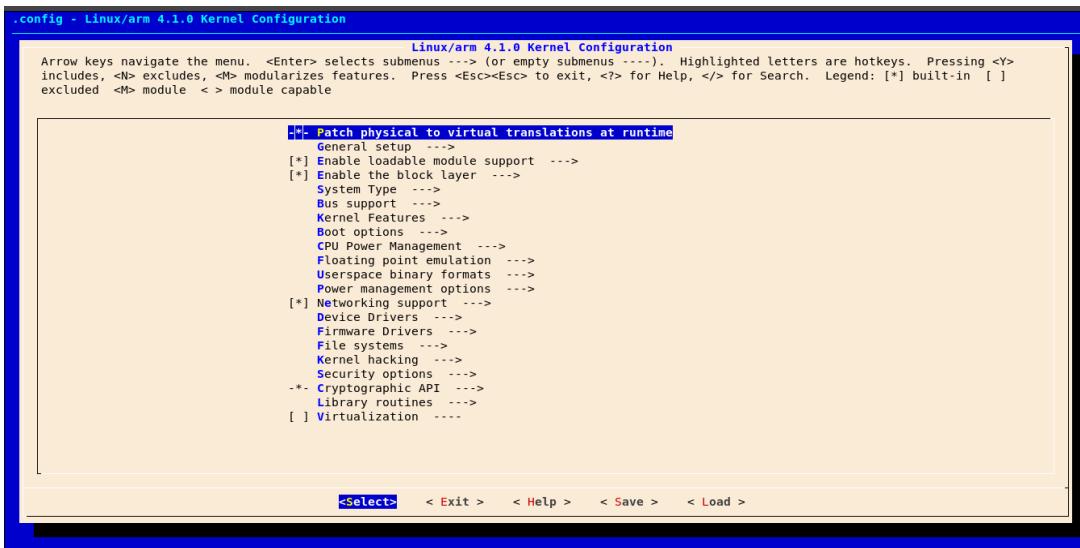
```
make ARCH=arm socfpga_defconfig
```

Ahora, vamos a ejecutar un comando que nos permite abrir una interfaz de texto para configurar de manera más fina las opciones del kernel.

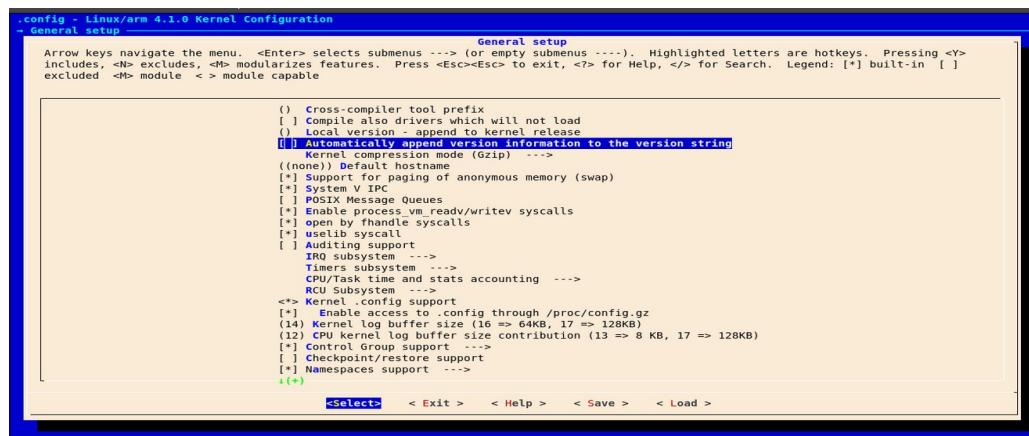
```
make ARCH=arm menuconfig
```

Dependiendo de la versión de ncurses que tenga su sistema operativo, debería de ver una interfaz como la que se ve en la siguiente imagen

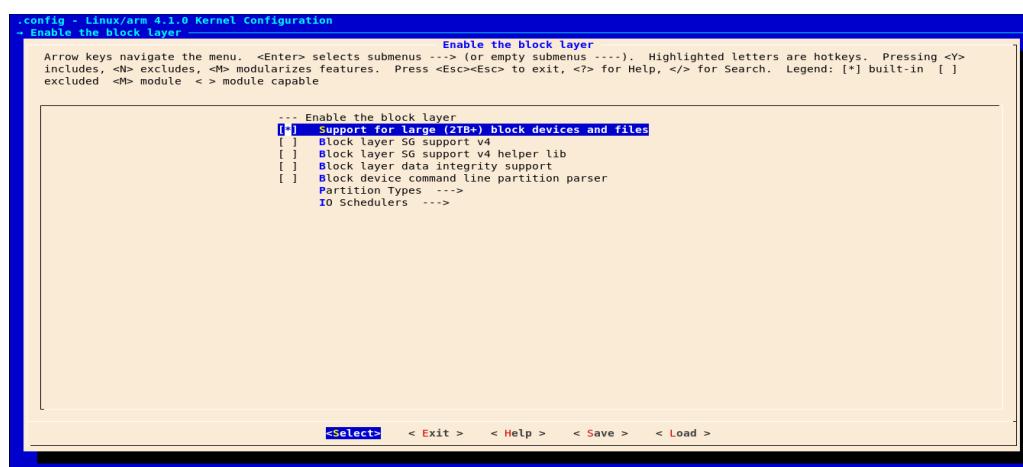
Entramos al menú “General Setup”, y deshabilitamos la opción “Automatically append version information to the version string”.

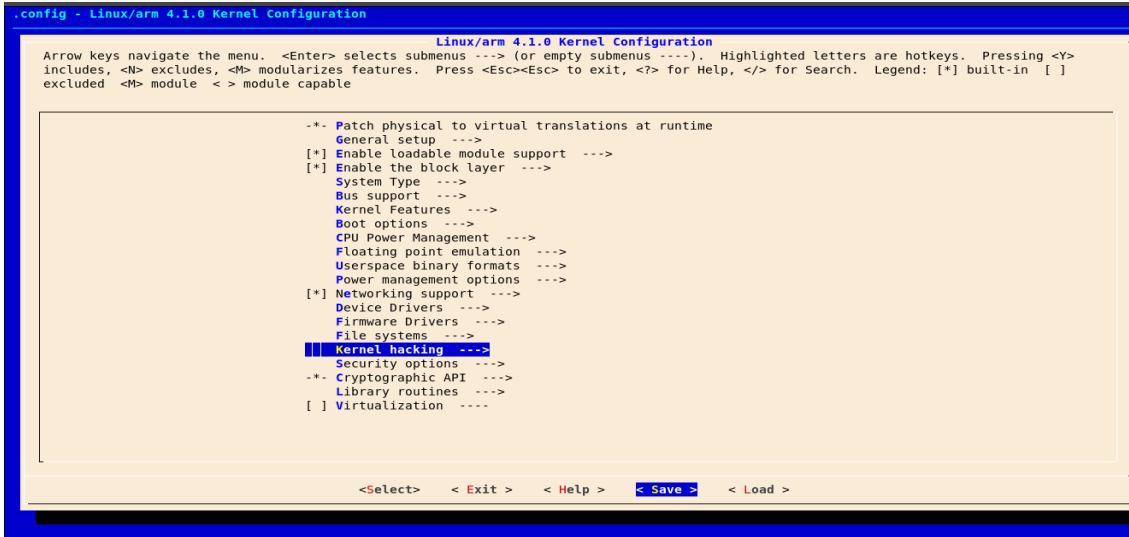


Después de deshabilitar esta opción, regresamos al menú principal y entramos al menú “Enable the block layer” En este menú, habilitamos la opción “Support for large (2TB+) block devices and files”.

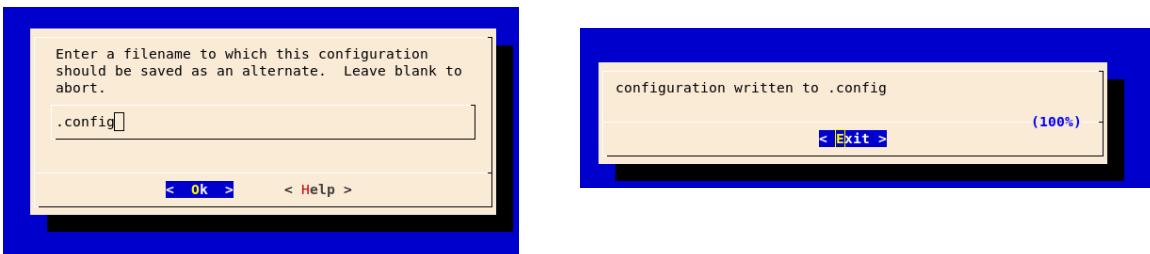


Las demás configuraciones as podemos dejar en los valores por defecto. Ahora, seleccionamos la opción Save para guardar las configuraciones que hemos designado





En el mensaje emergente nos da el nombre por defecto del archivo a guardar, damos OK, y luego otra vez OK cuando se acabe de escribir el archivo .config.



En el directorio ejecutamos el siguiente comando para compilar el kernel

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- LOCALVERSION= zImage
```

Después de varios minutos, la compilación concluye. Dentro de la carpeta arch/arm/boot/zImage encontraremos la imagen del kernel.

```
SHIPPED arch/arm/boot/compressed/bswapsdi2.5
AS      arch/arm/boot/compressed/bswapsdi2.0
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
[root@centos linux-socfpga]#
```

12. Generación del sistema de archivos raíz

Además del Kernel, necesitamos un sistema de archivos para el sistema operativo. Para generar este sistema utilizaremos el programa “buildroot” que genera las carpetas del sistema operativo, archivos de configuración y binarios de los programas básicos que se necesitan para utilizar el sistema. Regresamos a la carpeta software, con el comando cd .. y descargamos el repositorio de buildroot mediante el comando:

```
git clone http://git.buildroot.net/git/buildroot.git
```

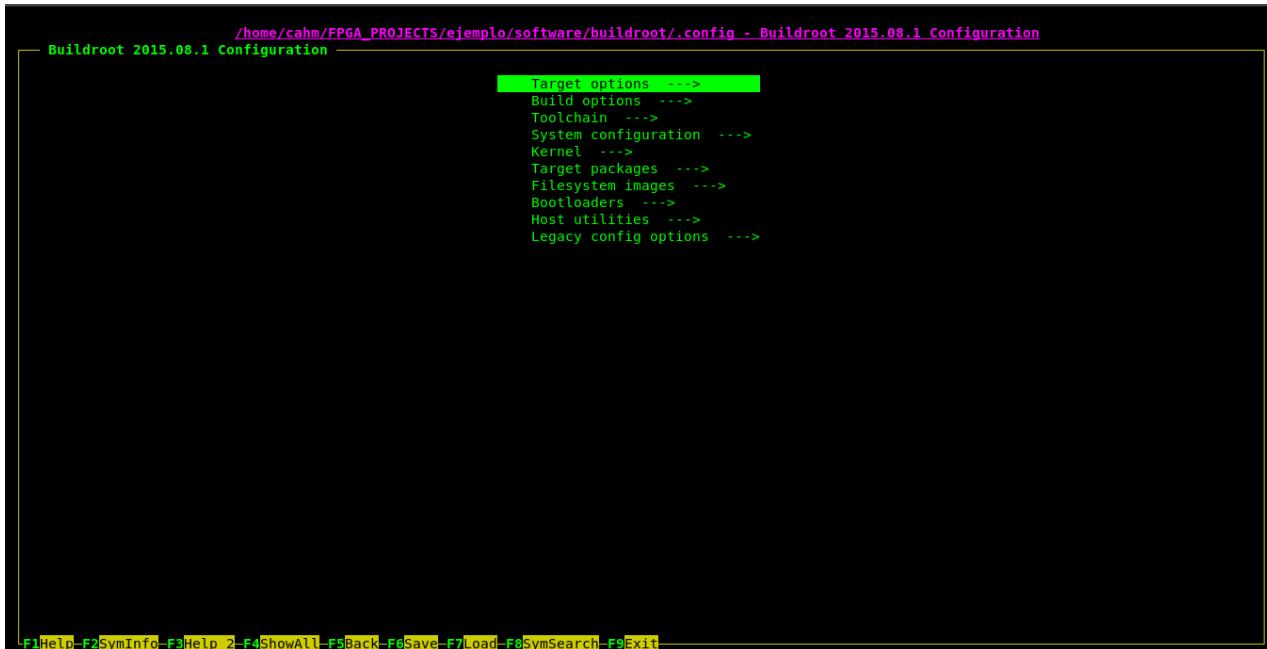
Después de la descarga, entramos al directorio del repositorio, establecemos la rama del proyecto, y regresamos a la carpeta software con los siguientes comandos.

```
cd buildroot  
git checkout 2015.08.x  
cd ..
```

Despues, desde el directorio de software vamos a configurar el compilador para generar el sistema de archivos raíz, de manera similar a como hicimos con el kernell.

```
make -C buildroot ARCH=ARM BR2_TOOLCHAIN_EXTERNAL_PATH=$(pwd)/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux nconfig
```

La interfaz de usuario basada en texto debe de verse como en la siguiente imagen



En el menú Target Options vamos a hacer varios cambios. Target Architecture lo vamos a configurar como “ARM (little endian)”, porque esta es la variante de arquitectura del procesador ARM con la que vamos a trabajar. Para la opción Target Architecture Variant” seleccione la opción “cortex-A9”, porque esa es precisamente la versión del ARM con la que estamos trabajando. En “Target ABI” seleccionamos “EABIhf” para habilitar el procesamiento de punto flotante. Habilitamos la opción “NEON SIMD extension support” para permitir usar los múltiples núcleos del ARM de manera paralela, y por último configuramos la opción “Floating point strategy” a la opción “NEON”. Asegurese de que en la configuración “ARM instruction set” esté seleccionada la opción ARM. La configuración debe de verse como se muestra en la siguiente imagen.

```
/home/cahm/FPGA_PROJECTS/ejemplo/software/buildroot/.config - Buildroot 2015.08.1 Configuration

Target options
    Target Architecture (ARM (little endian)) --->
        Target Binary Format (ELF) --->
        Target Architecture Variant (cortex-A9) --->
        Target ABI (EABIhf) --->
        [*] Enable NEON SIMD extension support
            Floating point strategy (NEON) --->
            ARM instruction set (ARM) --->

F1 Help-F2 SymInfo-F3 Help 2-F4 ShowAll-F5 Back-F6 Save-F7 Load-F8 SymSearch-F9 Exit
```

Regresamos al menú principal y entramos a “Toolchain” para configurar las opciones de la cadena de herramientas de compilación. En “Toolchain type”, seleccionamos “External Toolchain”. La opción “Toolchain” debería de cambiar a ““Linaro ARM 2014.09”, si no lo hace haga el cambio manualmente. En la opción “Toolchain origin” seleccione “Pre-installed toolchain”. Finalmente, asegurese de que la opción “copy gdb server to the Target” esté habilitada. Esta opción lo que nos permite es debuggear los programas que se ejecuten en el ARM. La configuración debe de verse como la de la siguiente imagen.

```
/home/cahm/FPGA_PROJECTS/ejemplo/software/buildroot/.config - Buildroot 2015.08.1 Configuration

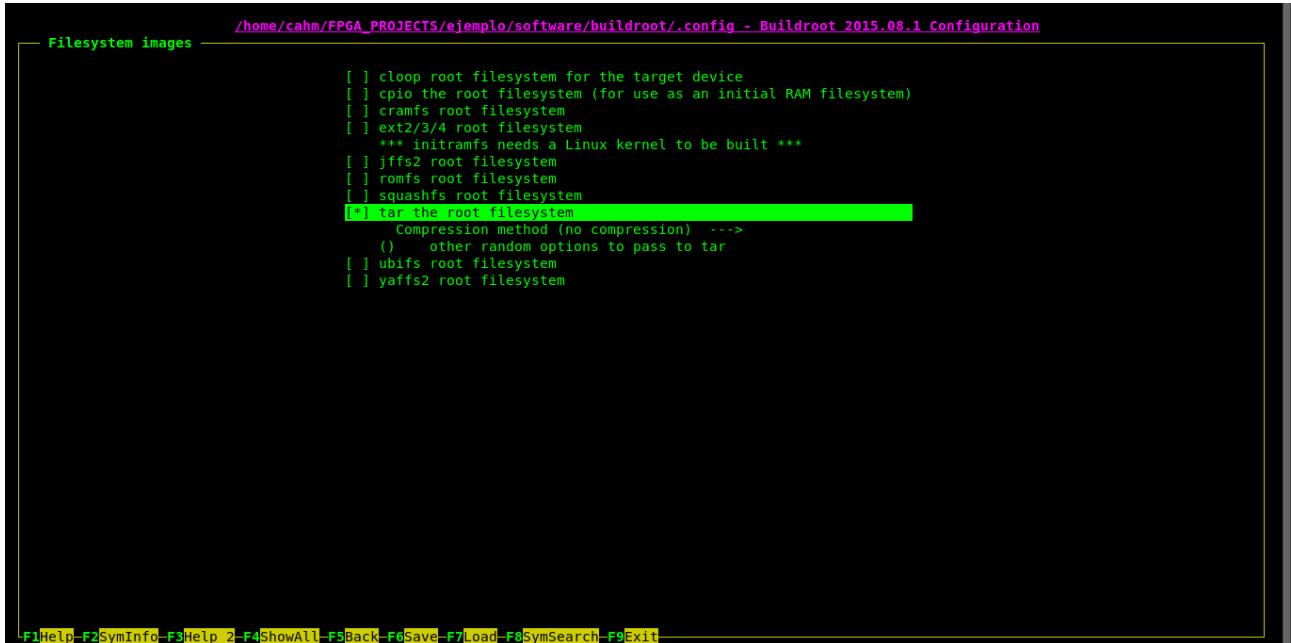
Toolchain
    Toolchain type (External toolchain) --->
        Toolchain (Linaro ARM 2014.09) --->
        Toolchain origin (Pre-installed toolchain) --->
        (/path/to/toolchain/usr) Toolchain path
        [*] Copy gdb server to the Target
        [ ] Purge unwanted locales
            () Generate locale data
        [ ] Copy gconv libraries
        [*] Enable MMU support
            () Target Optimizations
            () Target linker options
        [ ] Register toolchain within Eclipse Buildroot plug-in

F1 Help-F2 SymInfo-F3 Help 2-F4 ShowAll-F5 Back-F6 Save-F7 Load-F8 SymSearch-F9 Exit
```

Regresamos al menú principal y entramos a la opción “System configuration”. En realidad, aquí no es necesario cambiar nada, pero es importante saber que en esta sección podemos designar el nombre del equipo, crear un mensaje de inicio para linux, crear un usuario y establecer un password.

Esto será útil para identificar a qué proyecto pertenece cada memoria microSD. Otro menú que es útil conocer es Target packages. En este menú, se pueden elegir programas para instalar en el Linux que estamos generando. Estos van desde compiladores y herramientas básicas de red, hasta videojuegos y entornos gráficos. En esta demostración estamos usando un espacio muy pequeño para el sistema de archivos, no es conveniente instalar nada, aunque recomendamos al usuario explorar estas opciones a fin de conocer las posibilidades que tiene para otros proyectos.

Regresamos al menú principal y entramos al menú “Filesystem images”. En este menú, es importante que esté seleccionada la opción “tar the root filesystem” como se muestra en la siguiente imagen.



```
/home/cahm/FPGA_PROJECTS/ejemplo/software/buildroot/.config - Buildroot 2015.08.1 Configuration
```

Filesystem images

- [] cloop root filesystem for the target device
- [] cpio the root filesystem (for use as an initial RAM filesystem)
- [] cramfs root filesystem
- [] ext2/3/4 root filesystem
 - *** initramfs needs a Linux kernel to be built ***
- [] jffs2 root filesystem
- [] romfs root filesystem
- [] squashfs root filesystem
- [*] tar the root filesystem**
 - Compression method (no compression) --->
 - () other random options to pass to tar
- [] ubifs root filesystem
- [] yaffs2 root filesystem

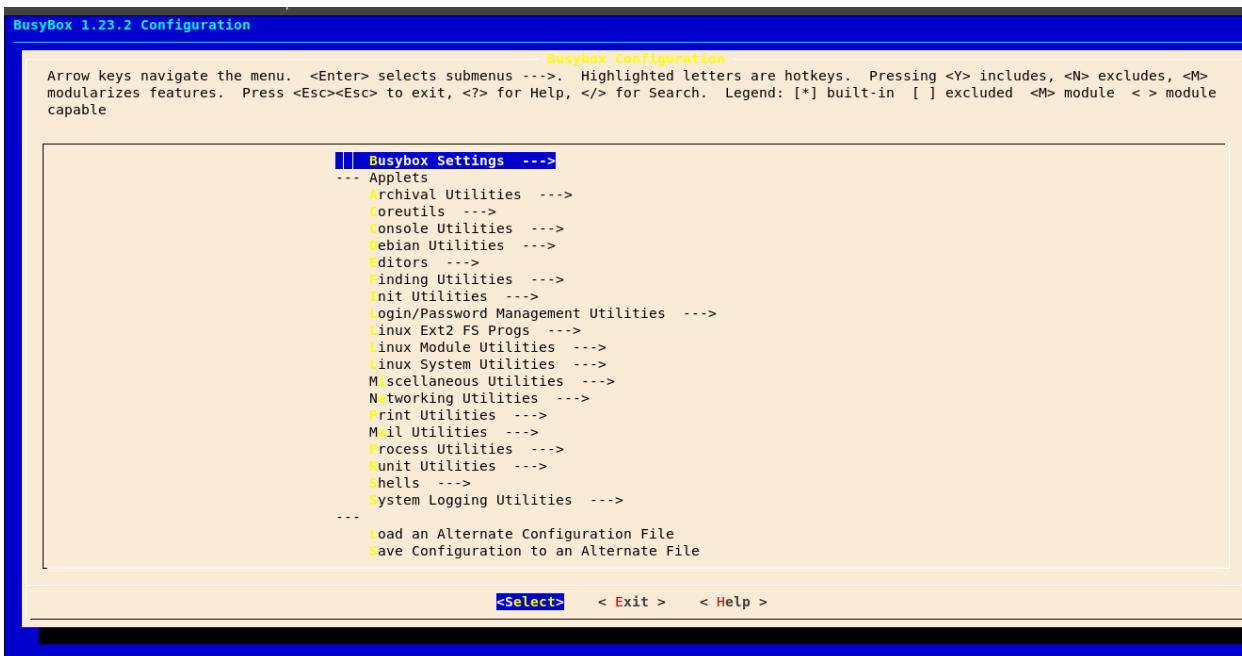
F1 Help F2 SymInfo F3 Help 2 F4 ShowAll F5 Back F6 Save F7 Load F8 SymSearch F9 Exit

Presione F6 para guardar las configuraciones y salga de la interfaz.

Ahora, ejecutamos la interfaz Busybox, la cual viene incluida como parte del repositorio de buildroot. Busybox es básicamente un programa que genera las configuraciones para instalar las herramientas de software del sistema operativo, como algunos drivers, soporte para múltiples sistemas de archivos, etc. Para configurar esta herramienta ejecutamos el comando:

```
make -C buildroot busybox-menuconfig
```

Se abre otra interfaz de usuario basada en texto como la que se muestra en la siguiente imagen. En este caso no vamos a hacer ningún cambio, solo damos Exit y guardamos la configuración.



Finalmente, iniciamos la compilación del sistema de archivo raíz basada en todos los archivos de configuración que hemos creado hasta ahora. Iniciamos la compilación con el comando.

```
make -C buildroot BR2_TOOLCHAIN_EXTERNAL_PATH=$(pwd)/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux all
```

Esto generó un archivo comprimido .tar que contiene todas las carpetas, binarios y archivos de configuración del sistema operativo.

13: Integración del sistema operativo

Volvemos a insertar la memoria microSD en la PC y ejecutamos el comando lsblk para identificar en qué carpeta se han montado las particiones de nuestro sistema operativo, como vemos en el ejemplo de la siguiente imagen,

```
[root@centos software]# lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda     8:0    0 931,5G  0 disk
└─sda1   8:1    0   16  0 part /boot
└─sda2   8:2    0 930,5G  0 part
  └─centos-root 253:0  0   526  0 lvm /
  └─centos-swap 253:1  0   3,7G  0 lvm [SWAP]
  └─centos-home 253:2  0 874,8G  0 lvm /home
sr0    11:0    1 1024M  0 rom
loop0   7:0    0   512M  0 loop
└─loop0p1 259:0  0   256M  0 loop
└─loop0p2 259:1  0   254M  0 loop
└─loop0p3 259:2  0     1M  0 loop
loop1   7:1    0   512M  0 loop
└─loop1p1 259:3  0   256M  0 loop
└─loop1p2 259:4  0   254M  0 loop
└─loop1p3 259:5  0     1M  0 loop
mmcblk0 179:0   0   7,4G  0 disk
└─mmcblk0p1 179:1  0   256M  0 part /run/media/cahm/4718-7705
└─mmcblk0p2 179:2  0   254M  0 part /run/media/cahm/54ed9d97-c340-4aa1-8881-40277b3870a3
└─mmcblk0p3 179:3  0     1M  0 part
```

Vamos a pegar la imagen del kernel en la partición de 256MB del sistema de arranque, y vamos a descomprimir el sistema de archivos raíz en la partición de 254MB.

```

sudo cp linux-socfpga/arch/arm/boot/zImage <Reemplaza este texto por la
ubicación dela partición de 256MB>
sudo tar -xvf buildroot/output/images/rootfs.tar -C <Reemplaza este texto por la
ubicación dela partición de 254MB>
sync

```

En la siguiente imagen se muestra un ejemplo de como ejecutar estos comandos reemplazando los placeholders por las ubicaciones que determina lsblk.

```

[loopip3 259:5 0 1M 0 loop
mmcblk0 179:0 0 7.4G 0 disk
└mmcblk0p1 179:1 0 256M 0 part /run/media/cahm/4718-7705
└mmcblk0p2 179:2 0 254M 0 part /run/media/cahm/54ed9d97-c340-4aa1-8881-40277b3870a3
└mmcblk0p3 179:3 0 1M 0 part
[root@centos software]# cp linux-socfpga/arch/arm/boot/zImage /run/media/cahm/4718-7705/
[root@centos software]# tar -xvf buildroot/output/images/rootfs.tar -C /run/media/cahm/54ed9d97-c340-4aa1-8881-40277b3870a3/

```

Desmontamos la memoria, la conectamos en la tarjeta, conectamos el cable serial, establecemos la comunicación con PuttY y encendemos la tarjeta. Después de un momento, veremos la opción de iniciar sesión en linux como el que se muestra en la siguiente imagen.



Hay que recalcar que esta imagen muestra un caso en el que se creó un mensaje de entrada personalizado. Si se dejara todo por default, el mensaje diría

Welcome to Buildroot
buildroot login:

Por defecto habría que iniciar sesión con el usuario root, el cual no tiene password para iniciar sesión.

En este punto, ya tenemos un sistema operativo Linux, que programa al FPGA como parte de su proceso de arranque. A continuación, aprenderemos a crear aplicaciones que se ejecuten en dicho sistema operativo y que puedan acceder al diseño montado sobre el FPGA.

14. Aplicaciones de software.

Dentro de la carpeta de software, crearemos una carpeta llamada userspace en la cual se trabajaremos sobre nuestro propio código en C. Nos movemos adentro de esta carpeta, e iniciamos el programa sopc-create-header-files para crear los encabezados para nuestro código en C. Así pues, ejecutamos.

```
mkdir userspace
cd userspace
sopc-create-header-files ../../simple_soc.sopcinfo --single hps_0.h --module hps_0
```

El último comando, creó un encabezado llamado hps_0.h a partir del archivo simple_soc.sopc.info, el cual fue generado por Qsys. Este archivo, contiene la información de los esclavos mapeados a memoria que se requiere para crear el encabezados de C. Vamos a ver que hay dentro del encabezado con el comando:

En las primeras dos líneas, veremos las siguientes declaraciones.

```
#ifndef _ALTERA_HPS_0_H_
#define _ALTERA_HPS_0_H_
```

Tales líneas le dicen al compilador que si la macro `_ALTERA_HPS_0_H_` no está definida debe de definiría junto con una serie de macros y parámetros, entre los que se encuentran las definiciones asociadas a los esclavos avalon.

Por ejemplo, para el esclavo que escribe al FPGA se define su nombre, su base, su ancho, su final etc.

```
#define TOFPGA_COMPONENT_TYPE altera_avalon_pio
#define TOFPGA_COMPONENT_NAME toFPGA
#define TOFPGA_BASE 0x0
#define TOFPGA_SPAN 16
#define TOFPGA_END 0xf
```

Asimismo, para tenemos una serie de definiciones al esclavo que recibe datos del FPGA. Por ejemplo:

```
#define FROMFPGA_COMPONENT_TYPE altera_avalon_pio
#define FROMFPGA_COMPONENT_NAME fromFPGA
#define FROMFPGA_BASE 0x10
#define FROMFPGA_SPAN 16
#define FROMFPGA_END 0x1f
```

Finalmente, se cierran la condicional asociada a las definiciones y termina el archivo con la siguiente linea.

```
#endif /* _ALTERA_HPS_0_H_ */
```

En el repositorio del proyecto, se encuentran los archivos Makefile y write_wait_read.c. Estos contienen un ejemplo básico de compilación cruzada de una aplicación y de como leer y escribir datos al FPGA.

Analizemos primero el código en C. Como es lo normal, al principio de este código se incluyen una serie de bibliotecas incluyendo el encabezado que generamos anteriormente en esta sección.

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

```
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <error.h>
#include <stdint.h>
#include <sys/mman.h>
#include "hps_0.h"
```

Después, se define el inicio y la longitud del puente Lightweight bridge que estamos usando en nuestro diseño.

```
// The start address and length of the Lightweight bridge
#define HPS_TO_FPGA_LW_BASE 0xFF200000
#define HPS_TO_FPGA_LW_SPAN 0x0020000
```

Después, iniciamos una función “main”, la cual recibe argumento del usuario al momento de ser ejecutada.

```
int main(int argc, char ** argv)
{
```

Dentro de esta función, se declaran las siguientes variables

```
void * lw_bridge_map = 0;
short* h2f = 0;
short * f2h = 0;
int devmem_fd = 0;
uint16_t result = 0;
uint16_t data = 0;
```

la primera variable es un apuntador genérico a datos que no definiremos más adelante en la función. Las variables h2f y f2h son apuntadores a variables de 16 bits. La variable devmem_fd es un entero, y las variables result y data son enteros de 16 bits sin signo.

Después de definir las variables, verificaremos que el usuario haya ingresado un número válido de entradas. En este caso, queremos una sola entrada. Si el usuario ingresa otra cantidad de entradas el programa se cierra con un fallo.

```
// Check to make sure they entered a valid input value
if(argc != 2)
{
    printf("Please, enter a valid number of values\n");
    exit(EXIT_FAILURE);
}
```

Después, el valor del argumento del usuario se convierte a un valor numérico, y este valor se asigna a la variable data.

```
data = atoi(argv[1]);
```

Después, vamos a abrir una función que nos permite acceder a los datos guardados en la RAM, de modo tal que el programa cierre en caso de no poder acceder a esta.

```
// Open up the /dev/mem device (aka, RAM)
devmem_fd = open("/dev/mem", O_RDWR | O_SYNC);
if(devmem_fd < 0) {
    perror("devmem open");
    exit(EXIT_FAILURE);
}
```

Tras haber abierto la ram haremos un mapeo de memoria que nos permita acceder específicamente a nuestros componentes personalizados.

```
// mmap() the entire address space of the Lightweight bridge so we can access  
our custom module  
    lw_bridge_map = (uint32_t*)mmap(NULL, HPS_TO_FPGA_LW_SPAN, PROT_READ|  
PROT_WRITE, MAP_SHARED, devmem_fd, HPS_TO_FPGA_LW_BASE);  
  
    if(lw_bridge_map == MAP_FAILED) {  
        perror("devmem mmap");  
        close(devmem_fd);  
        exit(EXIT_FAILURE);  
    }
```

Después del mapeo de memoria podemos asignar a los apuntadores h2f y f2h las direcciones de memoria correspondientes a los esclavos avalon.

```
// Set the init to the correct offsets within the RAM (TOFPGA_BASE and  
FROMFPGA_BASE are from "hps_0.h")  
    h2f = (short*)(lw_bridge_map + TOFPGA_BASE );  
    f2h = (short*)(lw_bridge_map + FROMFPGA_BASE);
```

Enviamos los datos, e imprimimos qué valor se está enviando.

```
// WRITE DATA IN  
*h2f = data;  
  
//print inputs  
printf(" SENDING THIS VALUE %d ", data );  
  
// PRINT THE DATA  
printf("DATA IS SET TO %x \n", (short)*h2f);
```

Esperamos un microsegundo para dar tiempo a que todo se estabilice.

```
// Wait a bit  
    usleep(1);
```

Imprimimos el valor que hemos recibido del FPGA

```
// PRINT THE result  
printf("THE RESULT IS %x \n", (short)*f2h);
```

Y finalmente cerramos el mapa de memoria, cerramos la RAM y termina la función main

```
// Unmap everything and close the /dev/mem file descriptor  
result = munmap(lw_bridge_map, HPS_TO_FPGA_LW_SPAN);  
if(result < 0) {  
    perror("devmem munmap");  
    close(devmem_fd);  
    exit(EXIT_FAILURE);  
}  
  
close(devmem_fd);  
exit(EXIT_SUCCESS);
```

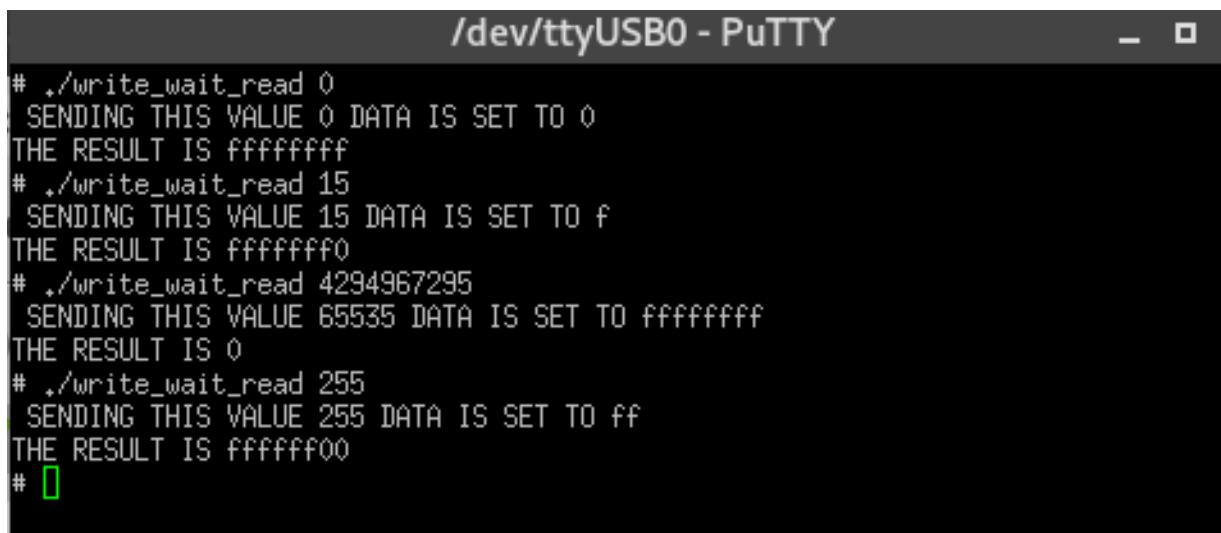
Compilamos el programa con el comando **make**, obteniendo un ejecutable que se llama `write_wait_read`. Tal ejecutable lo copiamos a la carpeta `root` del sistema de archivos del linux que hemos creado en esta guía. Para hacer esto usamos **lsblk** para ubicar la partición de 254MB, y copiamos el archivo a la carpeta `root` que está dentro de esta partición. Por ejemplo, si este comando me dice que la partición de 254MB está montada en “`/run/media/user/54a`”, tendría que ejecutar.

```
cp write_wait_read /run/media/user/54a/root/
sync
```

Desmontamos la memoria, la conectamos a la tarjeta, conectamos el cable usb, iniciamos Putty, prendemos la tarjeta, iniciamos sesión, y ahora por fin podremos ejecutar el programa.

14. Ejecución del código.

En la sección “Incorporación de un diseño de hardware personalizado” se mencionó que el FPGA recibirá una serie de bits y regresará la negación de estos. Entonces, esperamos que lo si le enviamos valor de 16-bits al FPGA, recibiremos equivalente este escribirá la negación bit por bit en el registro de lectura. Para el ejemplo del programa anterior, si le damos al programa un valor de “0” este lo escribirá como un entero sin signo en el registro “toFPGA”, es decir escribirá “0000000000000000” en el registro conectado al puerto de entrada del FPGA. Con esa entrada, el FPGA debería de entregar a la salida un valor de “1111111111111111”. Si el programa en C lee ese registro y lo interpreta como un entero sin signo, nos debería de entregar el valor hexadecimal ffffffff. En el caso opuesto, si le entregamos al programa un valor decimal de “4294967295” este lo interpretará como un entero sin signo y escribirá “1111111111111111” en el registro que entra al FPGA, y por lo tanto deberíamos de recibir un valor de “0”. De la misma forma, si mandamos un valor de 15, el programa escribirá “0000000000001111” en el registro toFPGA, el FPGA escribirá “111111111110000” en el registro fromFPGA y el programa nos dará un valor de ffffff0, para 255 le enviaríamos un valor de “000000011111111” y el valor negado de estos bits se interpreta en forma de entero como “fffffff00” y así con cualquier otro valor entero de 16-bits.



```
# ./write_wait_read 0
SENDING THIS VALUE 0 DATA IS SET TO 0
THE RESULT IS ffffffff
# ./write_wait_read 15
SENDING THIS VALUE 15 DATA IS SET TO f
THE RESULT IS ffffff0
# ./write_wait_read 4294967295
SENDING THIS VALUE 65535 DATA IS SET TO ffffffff
THE RESULT IS 0
# ./write_wait_read 255
SENDING THIS VALUE 255 DATA IS SET TO ff
THE RESULT IS ffffff00
#
```

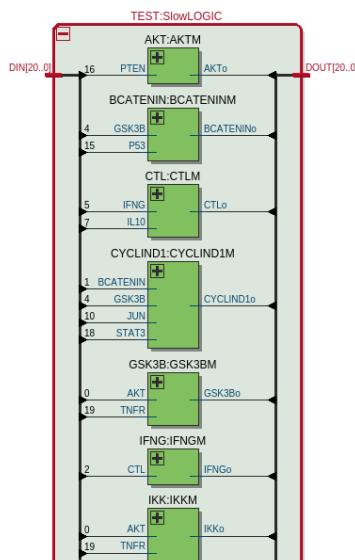
De esta forma, hemos concluido la implementación de la comunicación entre el FPGA y el ARM dentro del SoC desde una interfaz de software.

Módulo de cómputo dedicado

En el capítulo anterior, hablamos de cómo implementar la comunicación entre una capa de software que se ejecuta en el ARM con un diseño de hardware montado en el FPGA. En este capítulo, mostraremos como esta misma comunicación puede utilizarse para interactuar con un módulo de cómputo dedicado que implementa un algoritmo de simulación para una aplicación biológica. Los detalles biológicos de esta simulación no son relevantes, pero se pueden consultar en la tesis del autor la cual está disponible en <https://github.com/iluan/tesisCAHM>. En este mismo repositorio encontraremos los archivos para su implementación en el FPGA.

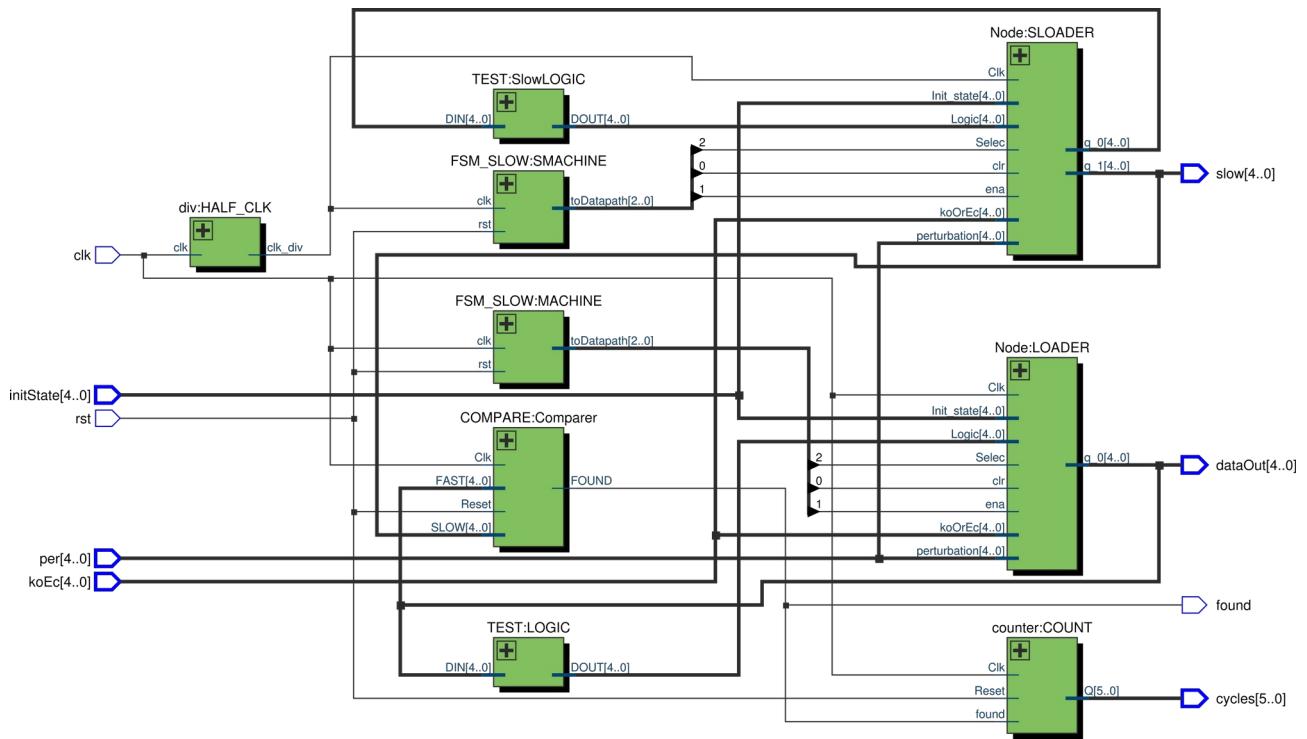
Descripción del sistema

A groso modo, el componente que se ha diseñado resuelve un sistema de ecuaciones lógicas que describen a un sistema biológico que varía en el tiempo y busca un tipo de patrón de comportamiento particular. Esto lo hace para todos los estados iniciales posibles. Como podemos ver en la siguiente imagen, cada una de las ecuaciones lógicas se implementó como componentes de hardware individuales que se incorporan dentro de un solo módulo.



Esta implementación permite que todas las ecuaciones en el sistema se ejecuten de manera paralela, lo que es una de las grandes ventajas de usar FPGAs en lugar de un procesador de uso general. El módulo es incorporado como parte de un diseño más complejo que realiza el proceso de cargar los estados iniciales y detectar los patrones. Los detalles de este algoritmo se pueden consultar en la tesis del autor. Por el momento nos limitaremos a describir los detalles del sistema que nos permiten interactuar con el mismo.

Como podemos ver en el siguiente diagrama, el sistema recibe cuatro entradas. Por el momento solo nos interesan las primeras tres, que son una señal de reloj, una señal de reset y un valor llamado initState, que corresponde al estado inicial que envía el HPS.



A la salida, vemos cuatro señales, de las cuales, solo las últimas dos serán leídas por el ARM. La señal “found” indica que el circuito ha encontrado el patrón que buscamos, y el valor cycles corresponde a un detalle particular del parón encontrado. Si vemos el diagrama del circuito completo, notamos que es esencialmente el mismo diseño que la implementación ejemplo del capítulo anterior.



Implementación

Para implementar este, descargamos los archivos del repositorio del autor abrimos el proyecto de quartus y repetiremos el proceso de generar el código con Qsys como se mostró en el capítulo anterior. Después de esto, haremos el análisis y síntesis, asignación de pines, compilación y generación de archivos de programación como se ha explicado antes. El proceso de crear la distribución de linux es igual el mismo. Sin embargo no se pueden simplemente usar los mismos archivos, porque al haber modificado el diseño, se modifican los archivos .sopcinfo, rbf, y toda la información del BSP. Asimismo, el encabezado para compilar aplicaciones personalizadas en C también será diferente. Lo único que si se puede reutilizar, son los archivos zImage del Kernal de Linux y el archivo tar del sistema de archivos raíz.

Habiendo concretado la creación de nuestra memoria micro-SD con un la distribución de Linux embebida que corresponde a nuestro proyecto, generamos los encabezados y procedemos a diseñar un código para la aplicación.

Código de aplicación

En el repositorio del presente proyecto, el autor ha incluido algunos ejemplos de código que puede ser compilado para este diseño. Primero que nada, revisemos el encabezado para entender un poco mejor los programas en C que vamos a compilar. Vemos que es bastante similar al encabezado generado en el ejemplo del capítulo pasado, pero hay ciertos cambios notorios.

En primer lugar, vemos que se han cambiado los nombres de los componentes, así como los nombres de sus bases sus anchos respectivos. Hemos renombrado “fromfpga” y “tofpga” como “init” y “counter” respectivamente porque estos nombres son más descriptivos desde el punto de vista de la ejecución del algoritmo.

```
#define INIT_COMPONENT_TYPE altera_avalon_pio
#define INIT_COMPONENT_NAME init
#define INIT_BASE 0x0
#define INIT_SPAN 32
...
#define COUNTER_COMPONENT_TYPE altera_avalon_pio
#define COUNTER_COMPONENT_NAME counter
#define COUNTER_BASE 0x20
#define COUNTER_SPAN 16
```

NOTA: el puerto “count” tiene un ancho de 32 bits para estar acorde a los tipos de datos de C, pero en realidad los 10 bits más significativos son ignorados por el FPGA. En cuanto a los siguientes 22 caracteres. El bit más significativo corresponde a la señal de reset, mientras que los siguientes 21 bits son los estados iniciales. Se está enviando tanto la señal inicial como la señal de reset en la misma dirección para no tener que crear múltiples registros y así minimizar el uso del espacio de memoria asignado para el puente avalon. Esto no es necesario y el lector debe saber que usar diferentes direcciones para las direcciones y los datos es una buena práctica de diseño. En cuanto a init. Este registro tiene 16 bits, los cuales se usan en su totalidad, siendo el bit más significativo correspondiente a la señal de que el sistema ha encontrado el patrón esperado a partir del último estado inicial. Entendiendo el uso de estas variables, pasemos a la implementación de código propio.

Entre los programas ejemplo que se encuentran en la carpeta de software del autor, nos fijamos en el archivo counter.c. Este programa realiza dos funciones, por un lado envía al FPGA todos los estados iniciales posibles del sistema de ecuaciones, y por otro lado recibe todos los valores del parámetro cycles, los suma e imprime el total de ciclos. Estas funciones se describen en el siguiente fragmento de código.

```
// Set the init to the correct offset within the RAM
(CUSTOM_SLAVE_0_BASE is from "hps_0.h")
    init = (uint32_t*)(lw_bridge_map + INIT_BASE );
    cycles = (uint32_t*)(lw_bridge_map + COUNTER_BASE);
    count = 0;
    for (i = 0; i < 2097151; i++) {
        *init = 2097152;
        *init = i;
        while((uint32_t)*cycles < 32768) {
            usleep(1);
        }
        current = (uint32_t)*cycles & 32767;
        count = count + current;
    }
    printf("IT TOOK %x CYCLES", count);
```

Primero establece las direcciones de memoria a las que tiene que leer y se asegura que la variable “cycles” valga cero. Después, inicia un ciclo for, en el que envía todos los valores posibles a los estados iniciales. En este caso, los estados iniciales posibles son todas las combinaciones de valores lógicos entre “00000000000000000000000000” y “11111111111111111111111111”. Para generar esta combinatoria, el programa enviará como estados iniciales todos los números entre 0 y 2097151. En cada iteración del ciclo, el programa primero escribe el número 2097152 en el registro init. Este número es equivalente a “10000000000000000000000000” en binario. Recordemos que justamente el bit más significativo de “init” es el que envía la señal de reset. Después de enviar al fpga la señal de reset, el programa escribe sobre init el valor actual del ciclo for, y abre un ciclo while. Este ciclo permite que el programa espere mientras que el valor de cycles sea menor a 32768. Este valor corresponde al valor binario “10000000000000000000000000”. Recordemos que éste último bit significativo es el que nos avisa que el FPGA ha encontrado un patrón. Para obtener el valor real del parámetro cycles, tenemos que eliminar el “1” de bit más significativo de éste registro. El programa hace esta operación con un simple AND con el valor 32767. Esta operación lógica garantiza que éste bit tomará un calor de “0” y todos los demás bits mantendrán su valor. Conforme avanza el ciclo for, el programa va sumando cada valor que recibe en la variable “count”. Finalmente, imprime la variable y le da al usuario la suma total del parámetro cycles para todas las iteraciones. Compilamos, copiamos a la memoria micro-SD y ejecutamos el programa. Después de aproximadamente 35 segundos, obtendremos el número de ciclos totales.

De esta manera hemos logrado implementar la comunicación y control de un módulo de cómputo dedicado. Completando el objetivo de la presente guía. El lector encontrará en el repositorio del proyecto otros ejemplos en C que se enlazan con este mismo diseño y que puede usar como una guía para diseñar sus propios diseños.