PYNQ-Z1

Introducción

La tarjeta FPGA PYNQ es una emocionante plataforma de desarrollo que combina la flexibilidad de las FPGAs (Field-Programmable Gate Arrays) con la facilidad de programación en Python. Esta tarjeta está diseñada para permitir a ingenieros, desarrolladores y entusiastas de la electrónica y la informática explorar y desarrollar aplicaciones de hardware personalizadas de manera accesible y efectiva.

Antes de sumergirnos en el mundo de la tarjeta PYNQ, es importante comprender qué es una FPGA. Las FPGAs son dispositivos de hardware programables que permiten a los usuarios diseñar y personalizar circuitos digitales a nivel de hardware. A diferencia de los microcontroladores o microprocesadores tradicionales, que ejecutan instrucciones de software, las FPGAs pueden ser configuradas para realizar tareas específicas de manera altamente eficiente y paralela. Esto las hace ideales para una amplia gama de aplicaciones, desde procesamiento de señales hasta control de hardware y aceleración de algoritmos.

Características Clave de la Tarjeta FPGA PYNQ:

1. **Sistema en Chip (SoC):** La tarjeta FPGA PYNQ generalmente incorpora un sistema en chip (SoC) que combina un procesador ARM con una FPGA en un solo dispositivo. Esto permite una estrecha integración entre el hardware y el software, lo que simplifica el desarrollo de aplicaciones.
2. **Programación en Python:** Una de las características más distintivas de la tarjeta PYNQ es su capacidad de programación en Python. Esto significa que puedes utilizar el lenguaje de programación Python, ampliamente conocido y utilizado, para controlar y personalizar el hardware en la FPGA.
3. **Acceso a Hardware Personalizado:** Con la tarjeta PYNQ, puedes diseñar y cargar circuitos personalizados en la FPGA para adaptarla a tus

necesidades específicas. Esto es especialmente útil para aplicaciones que requieren alta velocidad y paralelismo.

4. **Entorno de Desarrollo Integrado:** La tarjeta FPGA PYNQ se suministra con un entorno de desarrollo integrado que facilita la programación y el desarrollo de aplicaciones, lo que la hace adecuada tanto para principiantes como para expertos.

Componentes Claves de la Tarjeta FPGA PYNQ:

1. Un SoC Zynq-7000 con un núcleo ARM Cortex-A9 de doble núcleo y 650 MHz y una lógica programable equivalente a una FPGA Artix-7.
2. 512 MB de memoria DDR3 con un ancho de banda de 16 bits y 1050 Mbps.
3. Un conector micro SD para almacenar el sistema operativo Linux basado en Ubuntu y los archivos del proyecto.
4. Un puerto Ethernet Gigabit para la comunicación de red y la transferencia de datos.
5. Un puerto USB OTG para la alimentación, la depuración y la conexión de dispositivos periféricos.
6. Un puerto USB UART para la comunicación serie con el procesador ARM.
7. Dos conectores PMOD para la conexión de sensores, actuadores y otros módulos.
8. Un conector Arduino compatible para la conexión de placas y escudos Arduino.
9. Un conector Raspberry Pi compatible para la conexión de placas y cámaras Raspberry Pi.
10. Un conector HDMI de entrada y salida para la captura y reproducción de vídeo.
11. Un conector de audio estéreo para la entrada y salida de audio.
12. Un interruptor, un botón y cuatro LED para el control y la indicación del estado.

La tarjeta PYNQ se puede programar utilizando el marco PYNQ, que es un proyecto de código abierto desarrollado por Xilinx. El marco PYNQ proporciona una capa de abstracción entre el hardware y el software, permitiendo al usuario

escribir código Python que se ejecuta en el procesador ARM y que puede interactuar con el hardware mediante llamadas a funciones especiales llamadas superposiciones.

Las superposiciones son bloques de diseño que se cargan en la lógica programable del Zynq y que exponen una interfaz Python al usuario. Las superposiciones pueden implementar funciones específicas del dominio, como procesamiento de imágenes, aprendizaje automático, visión por computadora, etc. El usuario puede crear sus propias superposiciones o utilizar las existentes proporcionadas por el marco PYNQ o por la comunidad.

El marco PYNQ también proporciona un entorno interactivo basado en el navegador web llamado Jupyter Notebook, que permite al usuario escribir, ejecutar y documentar código Python, así como visualizar los resultados mediante gráficos, imágenes, vídeos, etc. El Jupyter Notebook se puede acceder desde cualquier dispositivo conectado a la misma red que la tarjeta PYNQ.

La tarjeta PYNQ es una herramienta ideal para aprender sobre los sistemas embebidos, el diseño FPGA, el procesamiento de señales, el aprendizaje automático y otras áreas relacionadas. La tarjeta PYNQ también es una plataforma potente y flexible para desarrollar prototipos y desplegar aplicaciones embebidas reales que requieren un alto rendimiento, una baja latencia y una personalización del hardware. La tarjeta PYNQ está disponible en diferentes versiones y kits, adaptados a diferentes necesidades y presupuestos.

Ya sea que estés interesado en la electrónica, la informática, la robótica o cualquier otro campo relacionado, la tarjeta PYNQ es una herramienta poderosa que puede impulsar tus proyectos de hardware y software a un nuevo nivel. ¡Comencemos este emocionante viaje de descubrimiento y desarrollo!.

Conectar la PYNQ al computador

Necesitarás:

- ➢ Una tarjeta PYNQ.
- ➢ Una Micro SD mínima de 8 GB.
- ➢ Un cable Ethernet.
- ➢ Un cable microUSB.
- ➢ Un computador.

1. **Ahora vamos a preparar la tarjeta SD**

La tarjeta SD se utiliza para alojar el sistema operativo Linux que se ejecuta en la tarjeta, Esto permite que la tarjeta funcione como una computadora embebida, lo que facilita la ejecución de aplicaciones y la interacción con la tarjeta, además del almacenamiento de los programas y proyectos que tengamos. Así que comencemos:

- Descargar el archivo de Imagen (http://www.pynq.io/board.html) y descomprimalo.

Development Boards

PYNQ supports Zynq based boards (Zynq, Zynq Ultrascale+, Zynq RFSoC), **Kria SOMs**, **Xilinx Alveo** accelerator boards and **AWS-F1** instances.
See the **PYNQ Alveo Getting Started guide** for details on installing PYNQ for use with Alveo and AWS-F1.

Downloadable PYNQ images

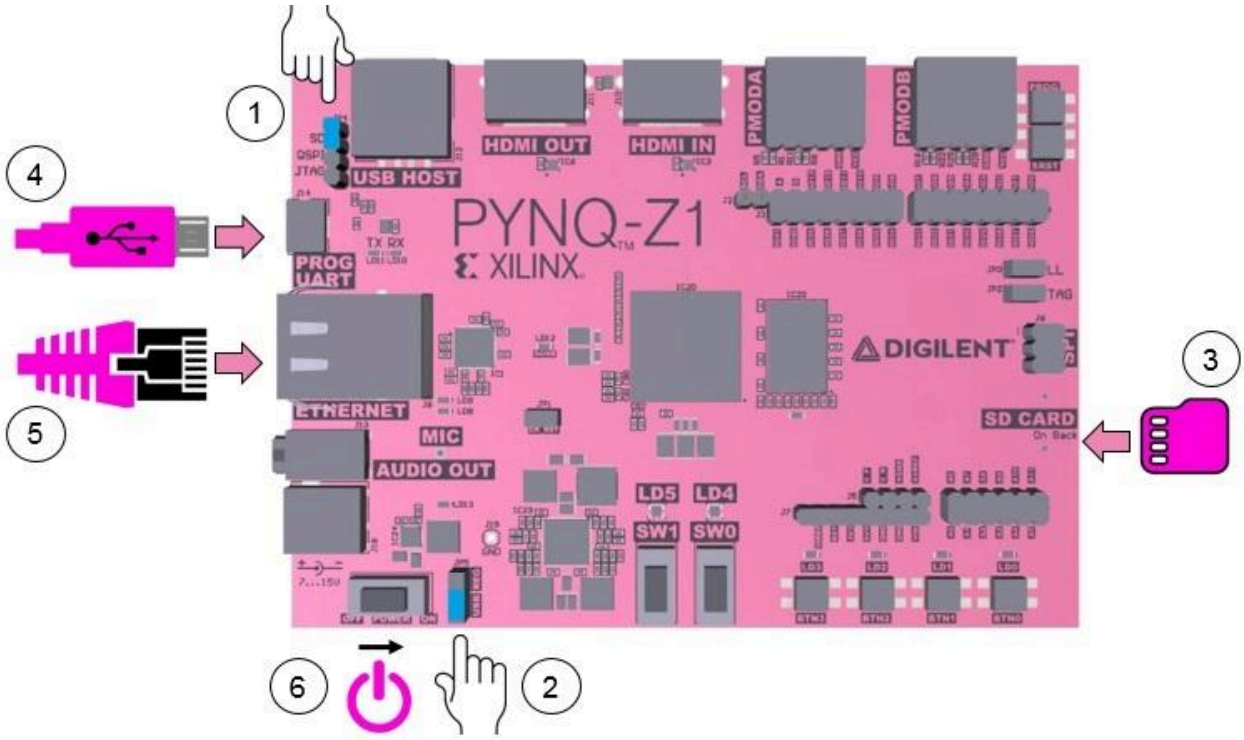
If you have a Zynq board, you need a PYNQ SD card image to get started. You can download a pre-compiled PYNQ image from the table below. If an image is not available for your board, you can build your own SD card image (see details below).

Board	SD card image	Previous versions	Documentation	Board webpage
PYNQ-Z2	v3.0.1	v2.7 v2.6	PYNQ setup guide	TUL Pynq-Z2
PYNQ-Z1	v3.0.1	v2.7 v2.6	PYNQ setup guide	Digilent Pynq-Z1
PYNQ-ZU	v3.0.1	v2.7 v2.6	GitHub project page	TUL PYNQ-ZU
Kria KV260*	Ubuntu 22.04		Kria PYNQ setup	Xilinx Kria KV260
Kria KR260*	Ubuntu 22.04		Kria PYNQ setup	Xilinx Kria KR260
ZCU104	v3.0.1	v2.7 v2.6	PYNQ setup guide	Xilinx ZCU104
RFSoC 2x2	v3.0.1	v2.7 v2.6	RFSoC-PYNQ	XUP RFSoC 2x2
RFSoC 4x2	v3.0.1	v2.7	RFSoC-PYNQ	XUP RFSoC 4x2
ZCU111	v3.0.1	v2.7 v2.6	RFSoC-PYNQ	Xilinx ZCU111
ZCU208	v3.0.1		RFSoC-PYNQ	Xilinx ZCU208
Ultra96V2	v3.0.1	v2.7 v2.6	Avnet PYNQ webpage	Avnet Ultra96V2
Ultra96 (legacy)	v3.0.1	v2.7 v2.6	See Ultra96V2	See Ultra96V2
ZUBoard 1CG	v3.0.1		GitHub project page	Avnet ZUBoard 1CG
TySOM-3-ZU7EV	v2.7		GitHub project page	Aldec TySOM-3-ZU7EV
TySOM-3A-ZU19EG	v2.7		GitHub project page	Aldec TySOM-3A-ZU19EG

- Grabar la Imagen en la tarjeta SD (Puede usar Win32 Disk Imager).

Con eso ya todo estaría listo para comenzar con la tarjeta.

2. Iniciar la Tarjeta



1. Configure el puente JP4/Boot en la posición SD colocando el puente sobre los dos pines superiores de JP4 como se muestra en la imagen. (Esto configura la placa para que arranque desde la tarjeta Micro-SD).
2. Para alimentar la PYNQ-Z1 desde el cable micro USB, configure el puente JP5/Power en la posición USB. (También puede alimentar la placa desde un regulador de alimentación externo de 12V configurando el puente en REG).
3. Inserte la tarjeta Micro SD cargada con la imagen PYNQ-Z1 en la ranura de la placa.
4. Conecte el cable USB a su PC y al puerto Micro USB de la placa.
5. Conecte el cable Ethernet.
    a. Si tiene un router cerca, realice una conexión directa de su kit de desarrollo Ethernet con el puerto Ethernet del router.
    b. También se puede conectarlo (si tiene la opción) al puerto Ethernet de su computadora. Si se tiene conexión a internet por Wifi se configurará la dirección IP en la FPGA automáticamente usando el servidor DHCP. Pero si por algún motivo no tiene conectividad a Internet se deberá configurar la dirección IP de su PC/Laptop. Debe

configurar su dirección IP como 192.168.2.1 y su máscara de subred como 255.255.255.0. Deje los demás campos en blanco. Esto se llama asignar una dirección IP estática. La placa de desarrollo tiene una dirección IP estática predeterminada como 192.168.2.99.

6. Prenda la PYNQ-Z1 y verifique la secuencia de inicio.
    a. El LED rojo LD13 se encenderá inmediatamente para confirmar que la placa tiene energía. Después de unos segundos, el LED amarillo/verde LD12 se iluminará para mostrar que el dispositivo Zynq® está operativo. Después de un minuto, debería ver dos LED azules LD4 y LD5 y cuatro LED amarillos/verdes LD0-LD3 parpadear simultáneamente. Los LED azules LD4-LD5 se encenderán y apagarán mientras los LED amarillos/verdes LD0-LD3 permanecen encendidos. El sistema ahora está iniciado y listo para usar.

3. Trabajar con la Tarjeta

Para conectarse a la PYNQ abra un programa Serial Terminal como TeraTerm, PuTTY o cualquier otro. Configure el programa serial en Comunicación serial con puerto COM (que se muestra en el Administrador de dispositivos de su sistema operativo), y el setup a una velocidad de 115200, 8 data bits, 1 stop bit, no parity y no flow control.

Ahora podemos conectarnos a Jupyter Notebook:

- En su navegador de confianza escriba la dirección IP de la PYNQ y presione Enter.
    - El nombre de host predeterminado es pynq y la dirección IP estática predeterminada es 192.168.2.99. En caso de que esta dirección no funcione en el terminal del programa serial use **ipconfig** para conocer la dirección que se le ha asignado a la PYNQ.
- Si todo va bien debería verse esto:

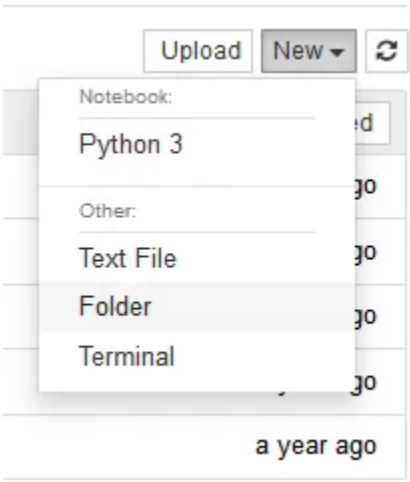
- El nombre de usuario es xilinx y la contraseña también es xilinx. Después de ingresar deberia verse asi:

    - La primera vez que se conecte, es posible que su computadora tarde un poco.

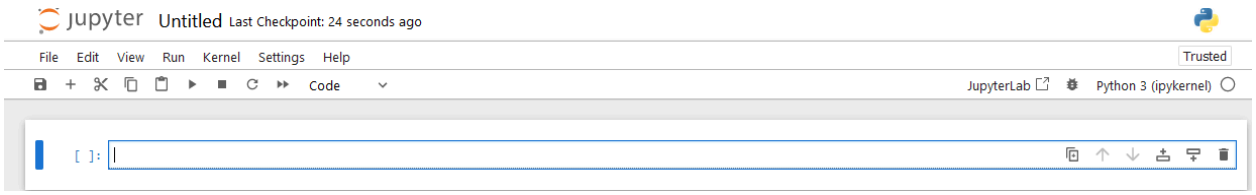
Jupyter Notebooks

Los Jupyter Notebooks son una aplicación web de código abierto que permite crear documentos que contienen código, texto, ecuaciones, visualizaciones y otros elementos multimedia, que se pueden compartir fácilmente con otros usuarios. Fueron desarrollados inicialmente como parte del proyecto IPython y luego se separaron como una herramienta independiente. Los Jupyter Notebooks son ampliamente utilizados en la programación científica y de datos debido a su capacidad para combinar la narrativa textual con fragmentos de código ejecutable en tiempo real. (Recomendamos tener algún tipo de experiencia previa usando Python)

Para iniciar le damos a nuevo y tenemos varias opciones:



Podemos crear carpetas para organizar nuestros notebooks, también poseemos un terminal donde podemos instalar los paquetes de Python que queramos pero por ahora solo crearemos un nuevo cuaderno, se vera asi:



Y desde aquí puedes ejecutar desde cosas sencillas como:

```
[1]  a=2

[2]  b=6

[3]  c=2*6

[4]  e=2**3

     print("c:",c)
     print("e:",e)

     c: 12
     e: 8
```

A las más complicadas que te imagines, aqui les dejo un ejemplo para controlar LED's, Switch y botones de la placa desde el Jupyter Notebook:

Example: Controlling all the LEDs, switches and buttons

The example below creates 3 separate lists, called *leds*, *switches* and *buttons*.

```
# Set the number of LED, Switches and Buttons
MAX_LEDS = 4
MAX_SWITCHES = 2
MAX_BUTTONS = 4

# Create lists for each of the IO component groups
leds = [base.leds[index] for index in range(MAX_LEDS)]
switches = [base.switches[index] for index in range(MAX_SWITCHES)]
buttons = [base.buttons[index] for index in range(MAX_BUTTONS)]
```

First, all LEDs are set to off. Then each switch is read, and if a switch is in the on position, the corresponding led is turned on. You can execute this cell a few times, changing the position of the switches on the board.

```
# LEDs start in the off state
for i in range(MAX_LEDS):
    leds[i].off()

# if a slide-switch is on, light the corresponding LED
for i in range(MAX_LEDS):
    if switches[i%2].read():
        leds[i].on()
    else:
        leds[i].off()
```

The last part toggles the corresponding led (on or off) if a pushbutton is pressed. You can execute this cell a few times pressing different pushbuttons each time.

```
# if a button is depressed, toggle the state of the corresponding LED
for i in range(MAX_LEDS):
    if buttons[i].read():
        leds[i].toggle()
```

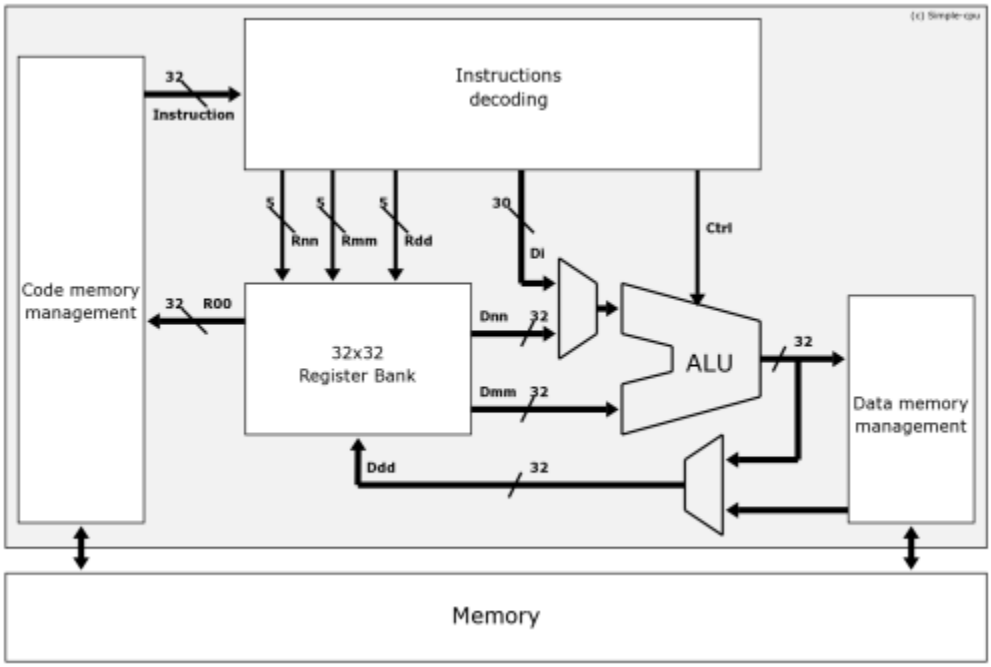
https://github.com/Xilinx/PYNQ_Workshop/blob/master/Session_1/4_Programming_onboard_peripherals.ipynb

Nota: En teoría como en cualquier otro entorno python eres capaz de instalar las librerías que consideres necesarias aunque en la documentación se recomienda usar: `pip3 install seaborn` *a nosotros solo nos funcionó:* `pip3 install -U seaborn --extra-index-url https://www.piwheels.org/simple`

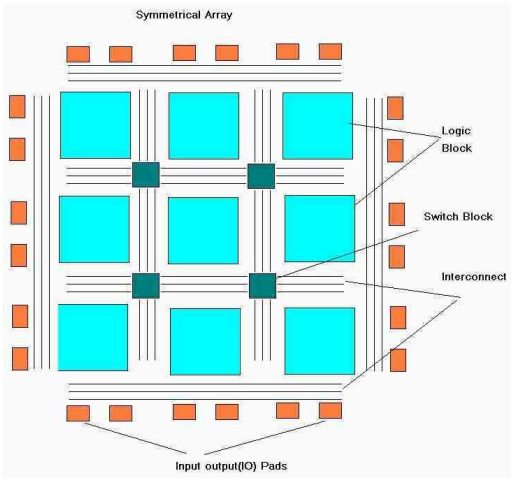
Hardware

Si posees conocimientos en la arquitectura de computadoras, es importante que tengas en cuenta que las instrucciones inicialmente se traducen a un lenguaje comprensible para las computadoras, denominado "lenguaje de máquina". Este lenguaje está compuesto por un conjunto de códigos hexadecimales que carecen de significado para nosotros, los seres humanos. Detrás de cada instrucción de código de máquina, se encuentra un proceso complejo de operaciones de hardware que se ejecutan dentro de su computadora. Por ejemplo, puede ser necesario acceder a una variable almacenada en la memoria, llevar a cabo una operación matemática que involucre la Unidad Aritmético Lógica (ALU), o guardar una variable en la memoria o en un registro, entre otras posibilidades.

Cuando un programa se ejecuta, lo que sucede en esencia es que el código de máquina se copia en la memoria, desde donde se extraen y ejecutan estas instrucciones de manera secuencial. Cada instrucción comienza con códigos de operación únicos que definen el tipo de acción que se debe realizar. Además, la instrucción puede contener datos que requieran algún tipo de manipulación. A continuación, se presenta una descripción de una arquitectura de CPU muy sencilla capaz de llevar a cabo la búsqueda, decodificación y ejecución de estas instrucciones.



Ahora que tenemos claridad al respecto, podemos adentrarnos en el tema de las FPGA, las FPGA están conformadas por numerosos módulos o bloques de construcción de pequeño tamaño que se conectan entre sí para llevar a cabo operaciones sofisticadas de manera simultánea. El código que se desarrolla para programarlas se transforma en un archivo de flujo de bits que especifica cómo se deben interconectar estos módulos de construcción.



Lo esencial que debes comprender es que al conectar entre sí estos diminutos bloques, tienes la capacidad de diseñar funcionalidades altamente complejas, es posible sintetizar diversos elementos lógicos digitales, tales como flip-flops, multiplexores, registros, compuertas AND/OR/NAND, y más. Si estás familiarizado con la lógica digital, sabrás que estos elementos son los componentes fundamentales de prácticamente todos los circuitos digitales

Cuando te dispongas a programar una FPGA, básicamente estarás escribiendo un código que establece la manera en que estos pequeños módulos se interconectan dentro de la FPGA. Este enfoque se conoce como "Síntesis de Hardware", esta técnica posibilita un alto grado de paralelismo en la arquitectura. Incluso con esta metodología, puedes llegar a sintetizar unidades de procesamiento central (CPU) y microcontroladores dentro de la misma FPGA.

La serie ZYNQ proporciona una manera de combinar el poder de PS y PL de tal manera que algunas de las tareas complejas, como el control de flujo, se pueden implementar en PS mientras que los algoritmos que requieren paralelismo se sintetizan en PL.

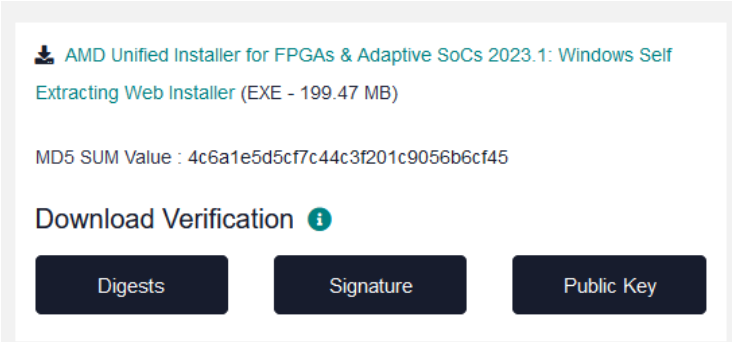
Puedes escribir cualquier tipo de algoritmo en Python en la parte PS. Ahora bien, si cree que algún algoritmo es un cuello de botella debido al comportamiento secuencial del PS, puede aprovechar el poder del paralelismo de PL y acelerar el algoritmo escribiéndolo en PL.

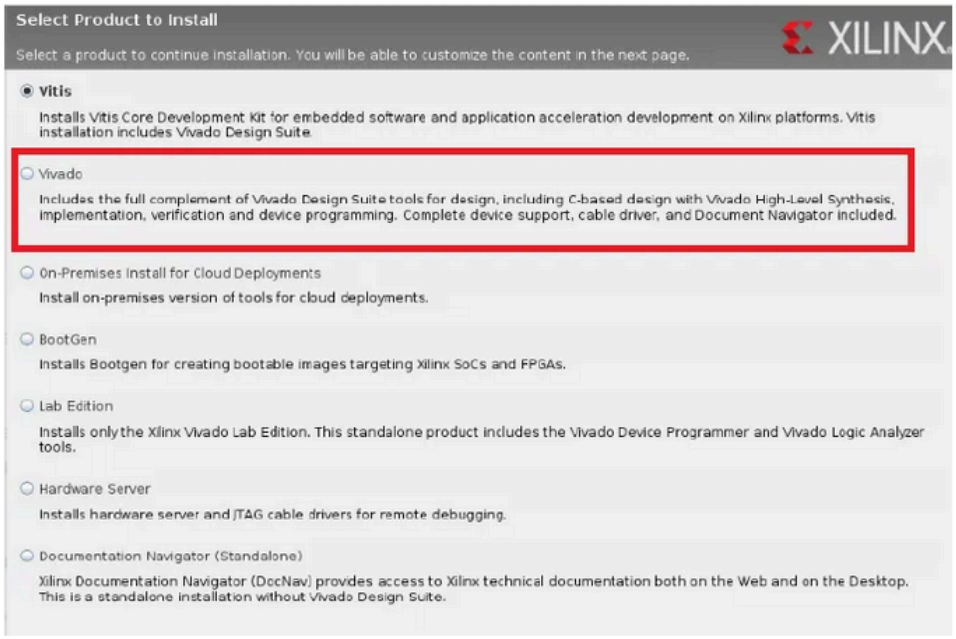
Vivado

Primero hablemos de que es Vivado, Vivado es un entorno de desarrollo integrado (IDE, por sus siglas en inglés) desarrollado por Xilinx, una empresa líder en tecnología de dispositivos lógicos programables y sistemas en chip (FPGAs y SoCs), este entorno de software se utiliza para diseñar, implementar y verificar circuitos digitales en hardware reconfigurable, lo que permite a los ingenieros y diseñadores trabajar con dispositivos FPGA de Xilinx. Con una interfaz gráfica intuitiva, facilita el diseño de hardware mediante la utilización de lenguajes de descripción de hardware como VHDL o Verilog, ofrece herramientas avanzadas para la síntesis de alto nivel, asignación de recursos, enrutamiento y generación de archivos de configuración. Vivado también proporciona potentes capacidades de depuración y verificación, incluyendo simulación funcional y temporal. Con soporte para una variedad de dispositivos Xilinx, desde FPGAs hasta SoCs con procesadores ARM, Vivado abarca todo el ciclo de vida del desarrollo, culminando en la generación de archivos binarios para la programación del hardware objetivo. En resumen, es una herramienta esencial para ingenieros y diseñadores que trabajan en el ámbito de los sistemas digitales reconfigurables. Ahora sabiendo esto ya podemos continuar.

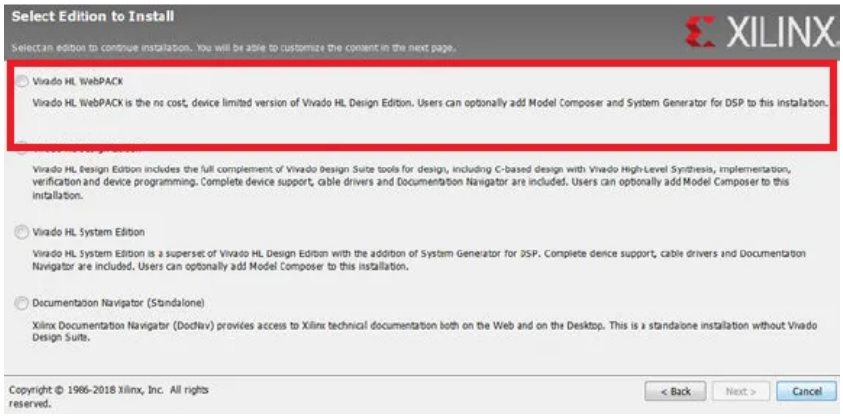
1. Descargue e instale Xilinx Vivado IDE

   Puede instalar la versión más reciente sin ningún problema, a menos claro que requiera una en específico para alguna función que quiera usar. Necesitarás crear un cuenta Xilinx.





Vivado ofrece una versión gratuita con algunas limitaciones que no afectarán nuestro tutorial. Si no tiene una licencia, seleccione 'Vivado HL WebPACK'.

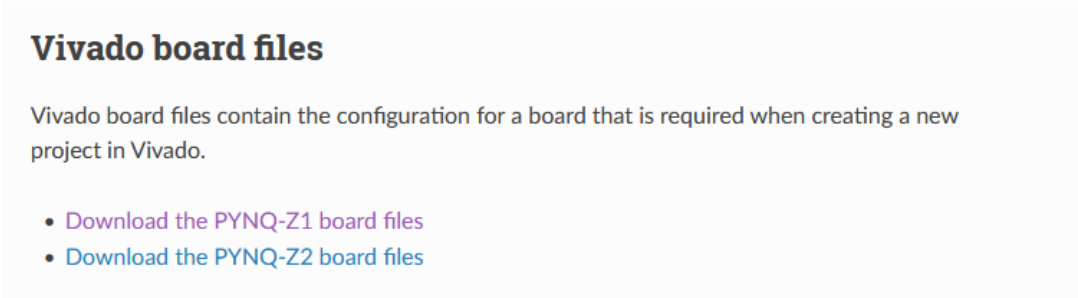


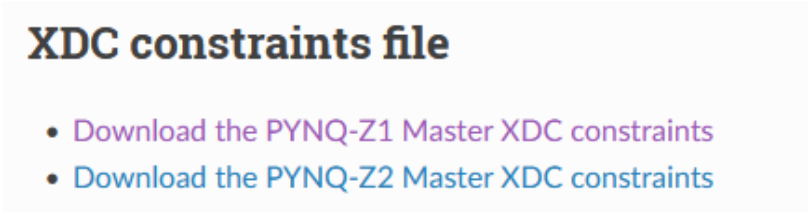
El resto del proceso es sencillo simplemente es seguir dando Next. Puede tardar un par de horas dependiendo de su máquina. Si le solicita que instale algunos controladores, instálelos ya que sin ellos, no podrás conectar tu placa de desarrollo al IDE.

2. Descargue los archivos Board

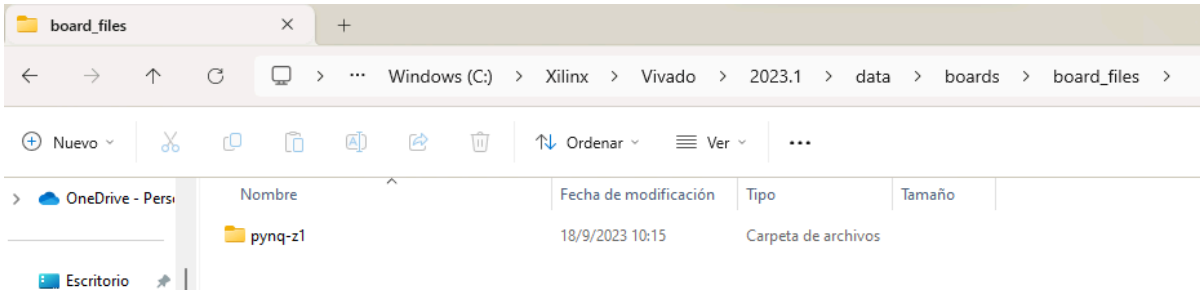
En la instalación predeterminada de Vivado IDE, no hay soporte para placas PYNQ. Sin embargo, puede descargar los archivos de configuración de la placa. Una vez que instale estos archivos de tablero, debería poder ver el tablero PYNQ en la lista de boards cuando creamos un nuevo proyecto en Vivado.

Descargue los archivos de la placa y el archivo de restricciones XDC. Ayuda a identificar los pines sin mirar el esquema.

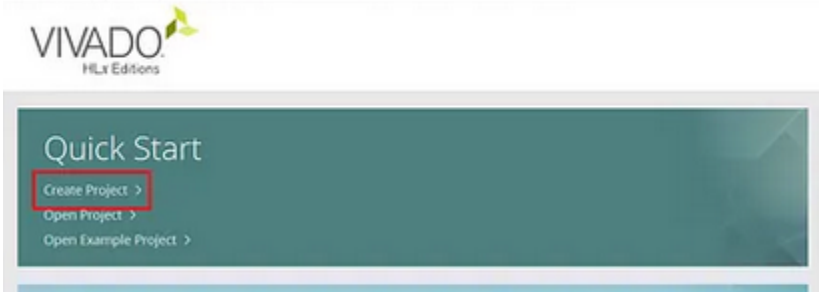


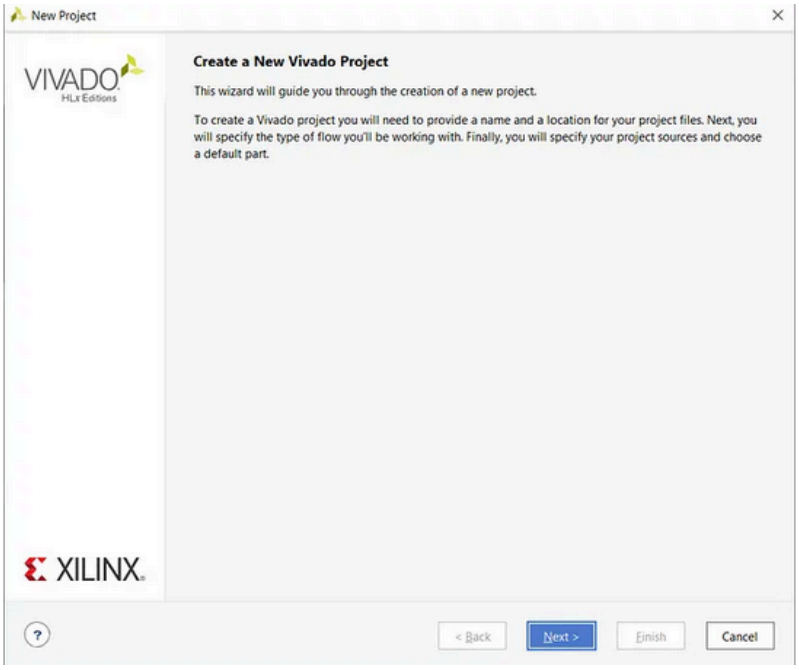


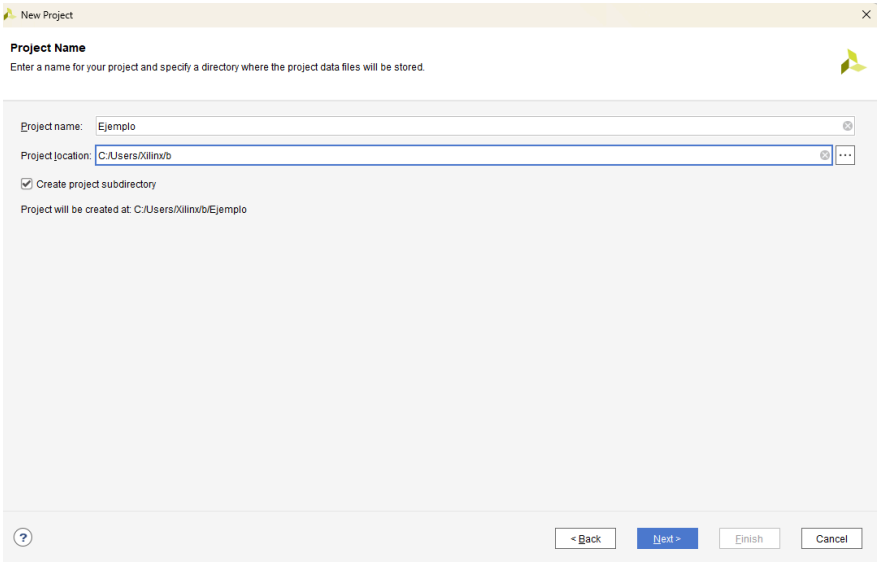
https://pynq.readthedocs.io/en/v2.3/overlay_design_methodology/board_settings.html
Esta carpeta se copia a :{Vivado install directory}\data\boards\board_files



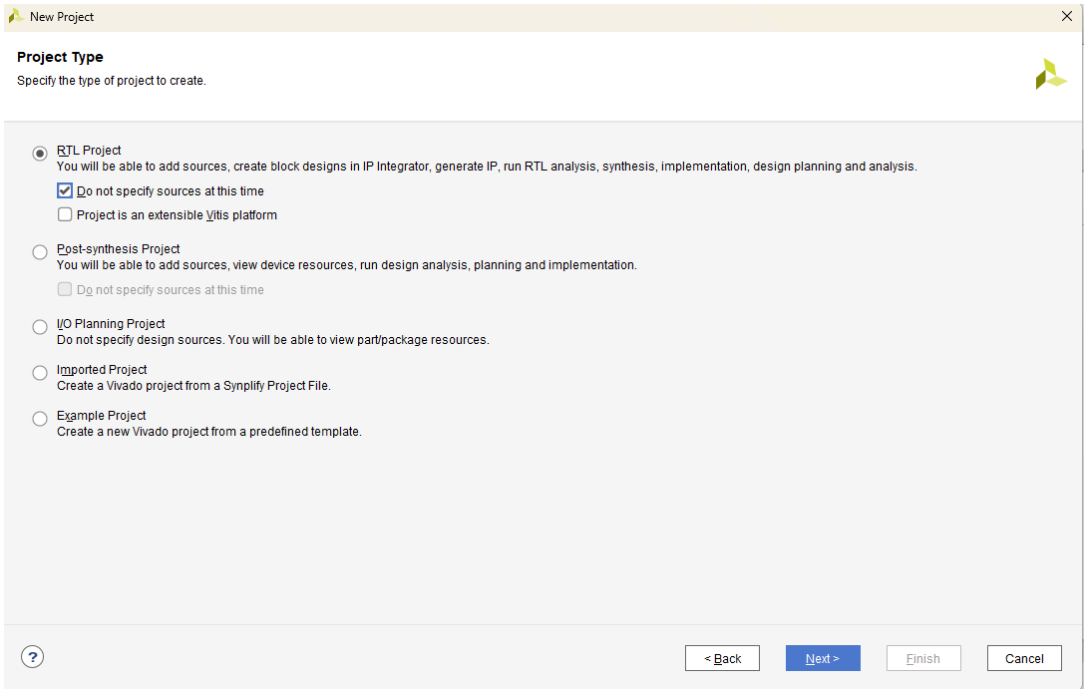
3. Crear y configurar un nuevo proyecto


VIVADO
HLx Editions
Quick Start
Create Project
Open Project
Open Example Project

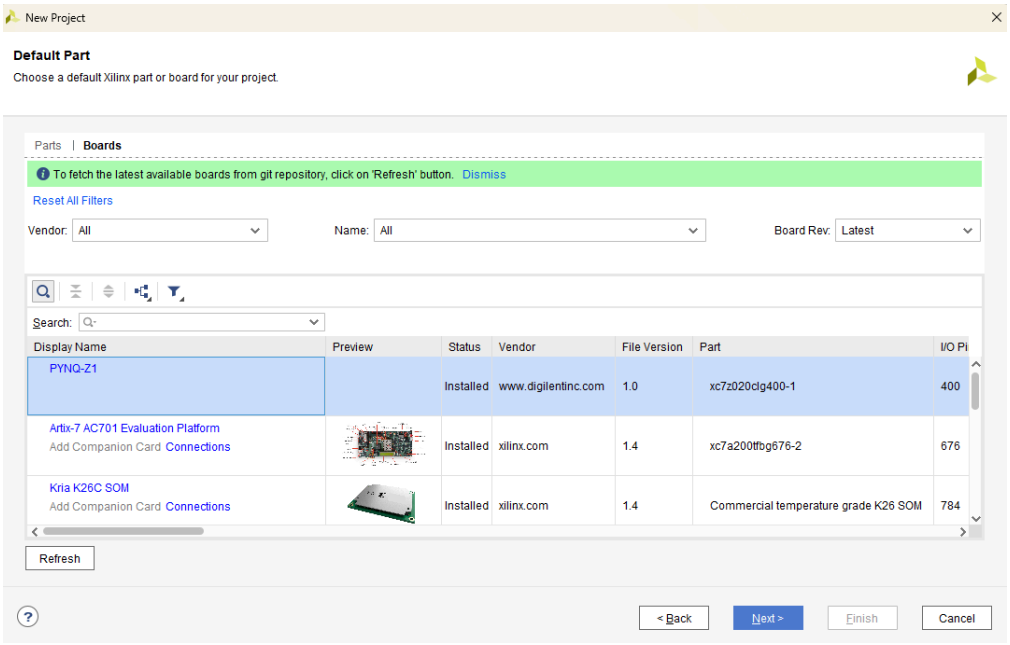

New Project
VIVADO
HLx Editions
Create a New Vivado Project
This wizard will guide you through the creation of a new project.
To create a Vivado project you will need to provide a name and a location for your project files. Next, you will specify the type of flow you'll be working with. Finally, you will specify your project sources and choose a default part.
XILINX
< Back
Next >
Finish
Cancel

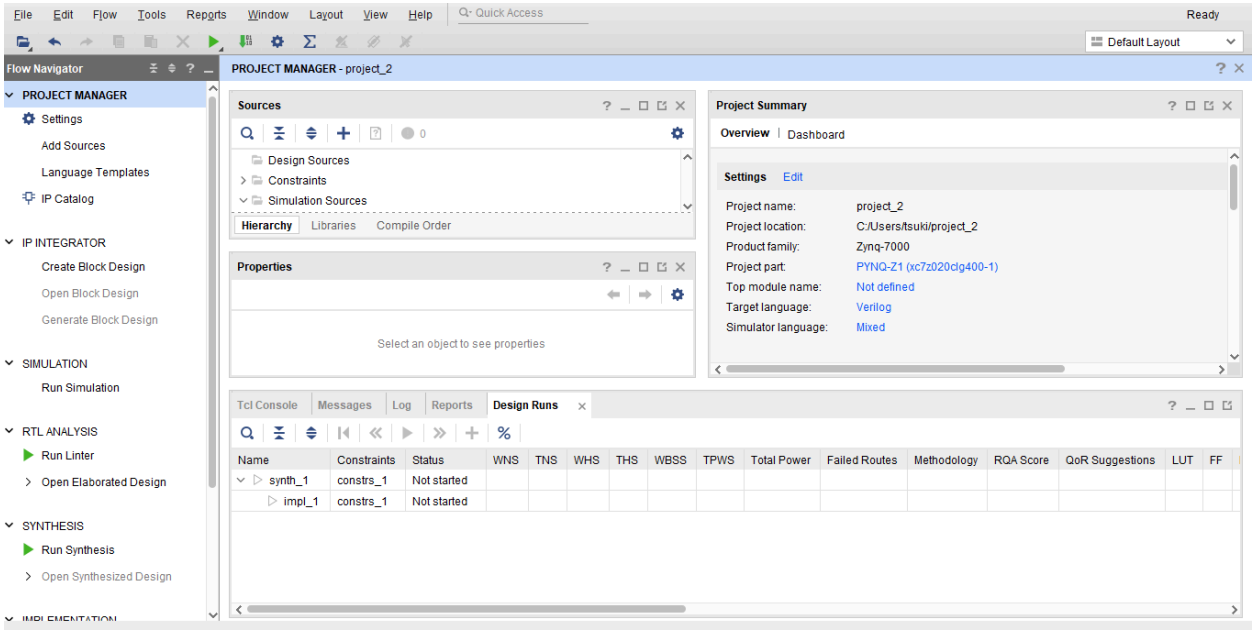


Hay diferentes tipos de proyectos disponibles. En nuestro caso, comenzaremos desde cero, es decir, creación de módulos/diseños, síntesis, implementación, etc. Por lo tanto, elegiremos 'Proyecto RTL', que significa 'Nivel de transferencia de registro'. Supongamos que no tenemos ningún archivo fuente por ahora, así que mantenga marcada la casilla de verificación 'No especificar fuentes en este momento'.



Es hora de elegir la FPGA en la que estamos a punto de trabajar. ¿Recuerdas que agregamos los board files anteriormente? Si lo ha hecho correctamente de la manera correcta, podrá ver su tablero PYNQ en la lista Tableros como se muestra. Seleccione 'PYNQ-Z1' y haga clic en Next y después en Finish en la pantalla final. Después de un tiempo, Vivado creará con éxito un proyecto para usted.





Ya desde aquí puedes realizar cualquier proyecto como en cualquier otra FPGA, en el caso de la PYNQ se fomenta la realización de Overlays por lo que lo explicaremos a continuación.

Overlays

¿Qué son los Overlays en el contexto de FPGA?

Los overlays son una característica clave en el ecosistema PYNQ que simplifica la programación y reconfiguración de FPGA. En el contexto de FPGA, un overlay se refiere a un diseño o conjunto de bloques lógicos que se superponen o cargan en la FPGA para realizar una tarea específica. Estos overlays son generalmente desarrollados en lenguajes de descripción de hardware como VHDL o Verilog y luego integrados en el entorno PYNQ para su fácil utilización, también permiten a los usuarios aprovechar al máximo la flexibilidad de una FPGA, ya que pueden ser cargados y reemplazados dinámicamente según sea necesario.

Los Overlays son diseños de hardware que se pueden cargar dinámicamente en la lógica programable (PL) del dispositivo Zynq de Xilinx, que integra un procesador ARM con una FPGA. Los Overlays permiten extender la aplicación del usuario desde el sistema de procesamiento (PS) del Zynq hasta la PL, aprovechando las ventajas de la programación en Python y las bibliotecas de código abierto.

Los Overlays se pueden utilizar para acelerar una aplicación de software o para personalizar la plataforma de hardware para una aplicación específica. Por ejemplo, el procesamiento de imágenes es una aplicación típica donde las FPGAs pueden proporcionar aceleración. Un programador de software puede utilizar un Overlay de la misma manera que una biblioteca de software para ejecutar algunas de las funciones de procesamiento de imágenes (por ejemplo, detección de bordes, umbralización, etc.) en la FPGA. Los Overlays se pueden cargar en la FPGA según se requiera, al igual que una biblioteca de

software. En este ejemplo, se podrían implementar funciones de procesamiento de imágenes separadas en diferentes Overlays y cargarlas desde Python según se necesite.

PYNQ proporciona una interfaz Python para permitir que los Overlays en la PL se controlen desde Python ejecutado en el PS. Los diseños FPGA son una tarea especializada que requiere conocimientos y experiencia en ingeniería de hardware. Los Overlays PYNQ son creados por diseñadores de hardware y envueltos con esta API Python PYNQ. Los desarrolladores de software pueden utilizar la interfaz Python para programar y controlar Overlays especializados sin necesidad de diseñar un Overlay ellos mismos. Esto es análogo a las bibliotecas de software creadas por expertos desarrolladores que son utilizadas por muchos otros desarrolladores de software que trabajan a nivel de aplicación.

Un Overlay puede contener uno o más bloques IP (Intellectual Property) que implementan funciones específicas del dominio, como procesamiento de imágenes, aprendizaje automático, visión por computadora, etc. Cada bloque IP tiene una interfaz AXI que permite la comunicación con el PS2. Los bloques IP pueden ser creados usando diferentes herramientas, como Vivado HLS, Vivado IP Integrator, etc.

Cada bloque IP dentro del Overlay tiene un controlador asociado que expone una interfaz Python al usuario. El controlador puede ser uno genérico proporcionado por PYNQ o uno personalizado creado por el diseñador del hardware. El controlador permite al usuario interactuar con el bloque IP mediante llamadas a funciones Python que leen o escriben registros o memoria.

Los bloques IP dentro del Overlay también pueden estar organizados en jerarquías, lo que facilita la modularidad y la reutilización del diseño. Una jerarquía es un grupo de bloques IP conectados entre sí que realizan una función común. Una jerarquía también puede tener un controlador asociado que expone una interfaz Python al usuario.

Ventajas de utilizar Overlays

La utilización de overlays en las tarjetas FPGA PYNQ-Z1 y PYNQ-Z2 proporciona varias ventajas significativas:

1. Facilidad de Uso: Los overlays simplifican la programación de FPGA al proporcionar interfaces de alto nivel en Python. Esto permite a los desarrolladores acceder y

controlar directamente los bloques lógicos sin necesidad de conocimientos profundos de hardware.

2. Reconfigurabilidad en Tiempo Real: Los overlays pueden ser cargados y reemplazados en tiempo real, lo que permite la adaptación dinámica del hardware a las necesidades cambiantes de la aplicación.
3. Reutilización de Diseños: Los overlays son fácilmente compartibles y reutilizables, lo que acelera el desarrollo de aplicaciones FPGA al aprovechar diseños previamente desarrollados.
4. Fomento de la Innovación: La flexibilidad y facilidad de uso de los overlays fomentan la innovación al permitir a los desarrolladores concentrarse en la lógica específica de su aplicación sin preocuparse por detalles de bajo nivel.

Carga y Utilización de Overlays

Una vez que entendemos la importancia de los overlays en las tarjetas FPGA PYNQ-Z1 y PYNQ-Z2, es crucial comprender cómo cargar y utilizar estos overlays en el entorno PYNQ. Aquí hay una descripción de los pasos clave:

1. Desarrollo del Overlay: Antes de cargar un overlay, primero debemos desarrollarlo. Los overlays se crean en lenguajes de descripción de hardware como VHDL o Verilog y se diseñan para cumplir una función específica. Una vez que el overlay está desarrollado y probado, se puede proceder a cargarlo en la FPGA.
2. Carga del Overlay en la FPGA: El entorno PYNQ proporciona herramientas y bibliotecas que permiten cargar overlays en la FPGA de manera sencilla. Esto se realiza mediante el uso de comandos Python o Jupyter Notebooks. El proceso de carga implica especificar la ruta del archivo bitstream del overlay y utilizar una función específica para cargarlo en la FPGA.
3. Interfaz con el Overlay: Una vez que el overlay se carga con éxito, los bloques lógicos y las interfaces definidas en el overlay están disponibles para su uso. Los desarrolladores pueden interactuar con el overlay a través de Python, lo que facilita la comunicación y el control de la FPGA.
4. Ejecución de la Aplicación: Finalmente, se puede desarrollar y ejecutar una aplicación que aproveche el overlay cargado en la FPGA. Esta aplicación puede

comunicarse con el overlay a través de Python y utilizar los recursos y la lógica proporcionados por el overlay para realizar tareas específicas.

Beneficios de este Enfoque

La carga y utilización de overlays en las tarjetas FPGA PYNQ-Z1 y PYNQ-Z2 proporciona un enfoque ágil y flexible para el desarrollo de sistemas FPGA. Permite a los desarrolladores adaptar rápidamente el hardware a las necesidades de su aplicación sin tener que realizar cambios de hardware físico. Esto acelera el tiempo de desarrollo y permite una mayor innovación en el diseño de sistemas basados en FPGA.

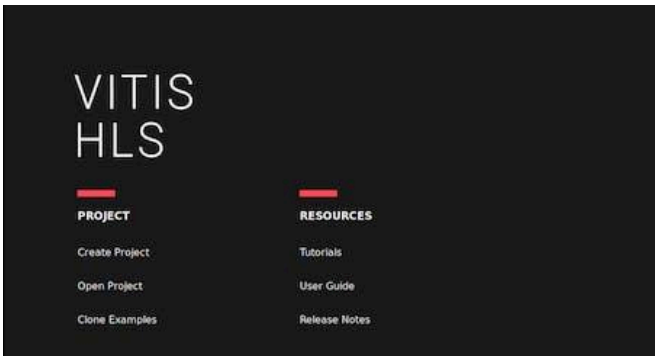
Ejemplo de Overlay

Primero hablemos de que es Vitis HLS, Vitis High-Level Synthesis (HLS) es una herramienta de Xilinx que proporciona un enfoque de alto nivel para el diseño y la implementación de hardware digital en dispositivos programables (como FPGAs), a diferencia de los métodos tradicionales de diseño de hardware mediante descripciones en lenguajes de hardware de bajo nivel, Vitis HLS permite a los desarrolladores expresar su diseño en lenguajes de alto nivel como C, C++, y OpenCL. Esto simplifica significativamente el proceso de diseño, permitiendo a los desarrolladores aprovechar las ventajas de la aceleración de hardware sin tener que ser expertos en lenguajes de descripción de hardware.

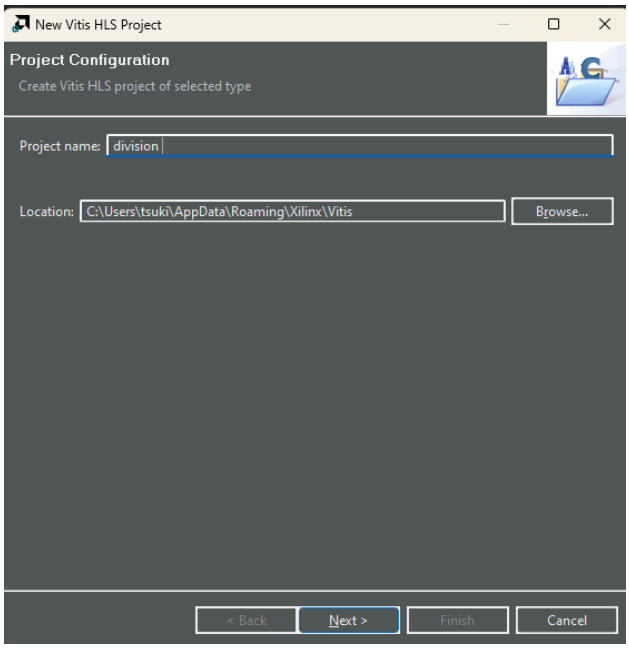
Algunas características clave de Vitis HLS incluyen la capacidad de optimizar automáticamente el código para mejorar el rendimiento y la eficiencia del hardware resultante, así como la integración con flujos de trabajo más amplios de diseño de sistemas utilizando herramientas como Vivado. En resumen, Vitis HLS es una herramienta que facilita la implementación de algoritmos en hardware reconfigurable de manera eficiente y accesible, permitiendo a los diseñadores centrarse en la funcionalidad de su código sin tener que abordar directamente los detalles de la descripción de hardware de bajo nivel. Sabiendo esto podemos continuar:

1. Crear IP en Vitis HLS

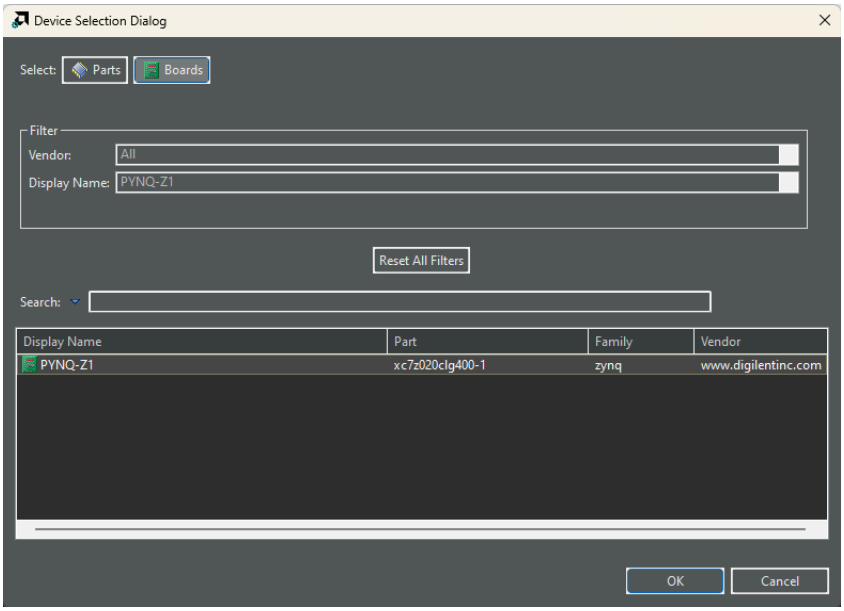
Cuando se realizo la instalacion de Vivado adicionalmente se debio haber instalado Vitsi HLS por defecto, lo abrimos:

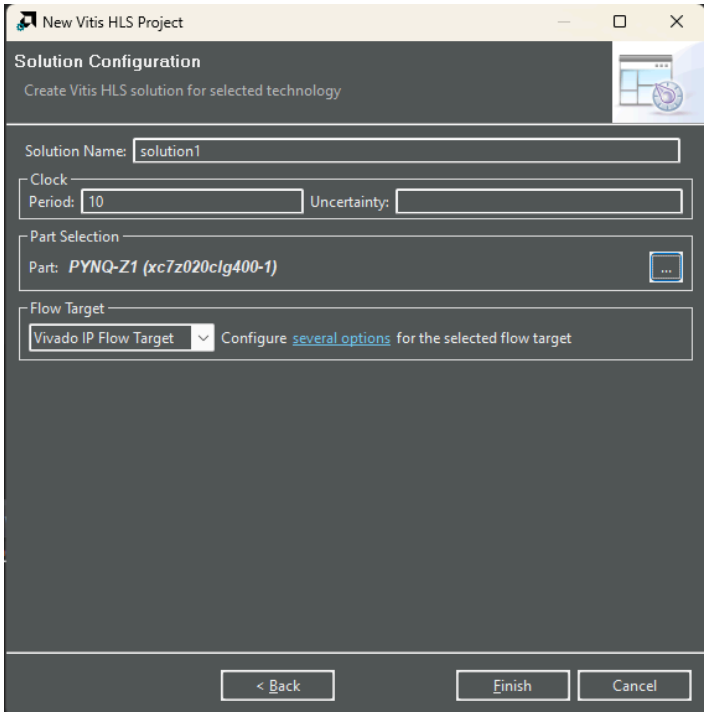


Creamos un nuevo proyecto:

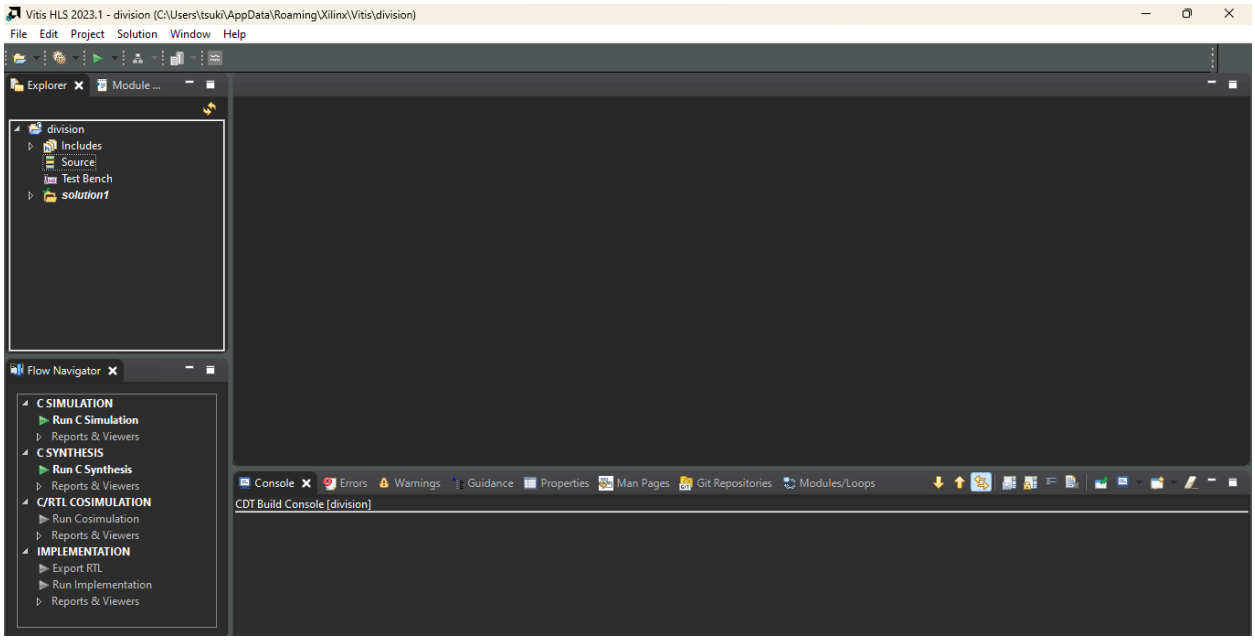


A continuación, seleccione la FPGA que se va utilizar en el proyecto:

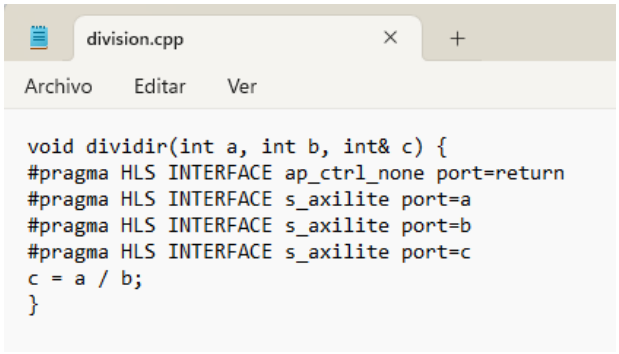




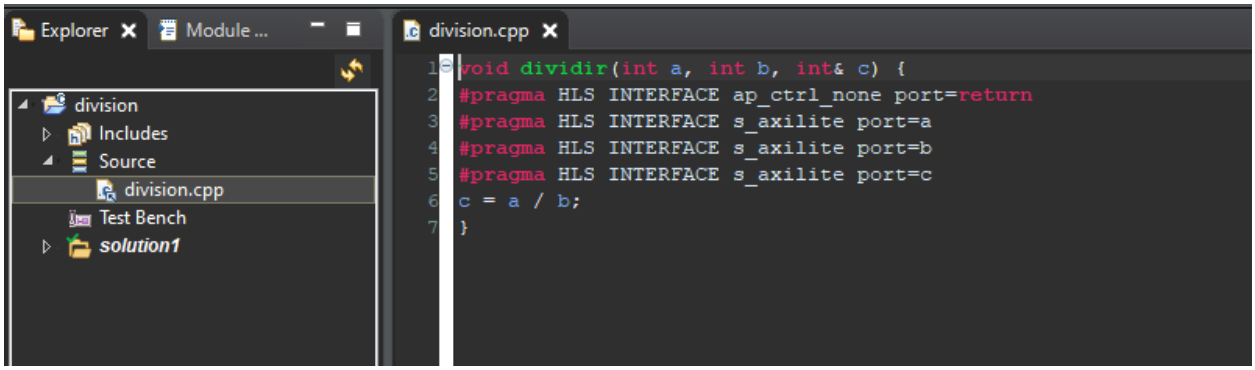
Presione Finish y HLS cargará el nuevo proyecto:



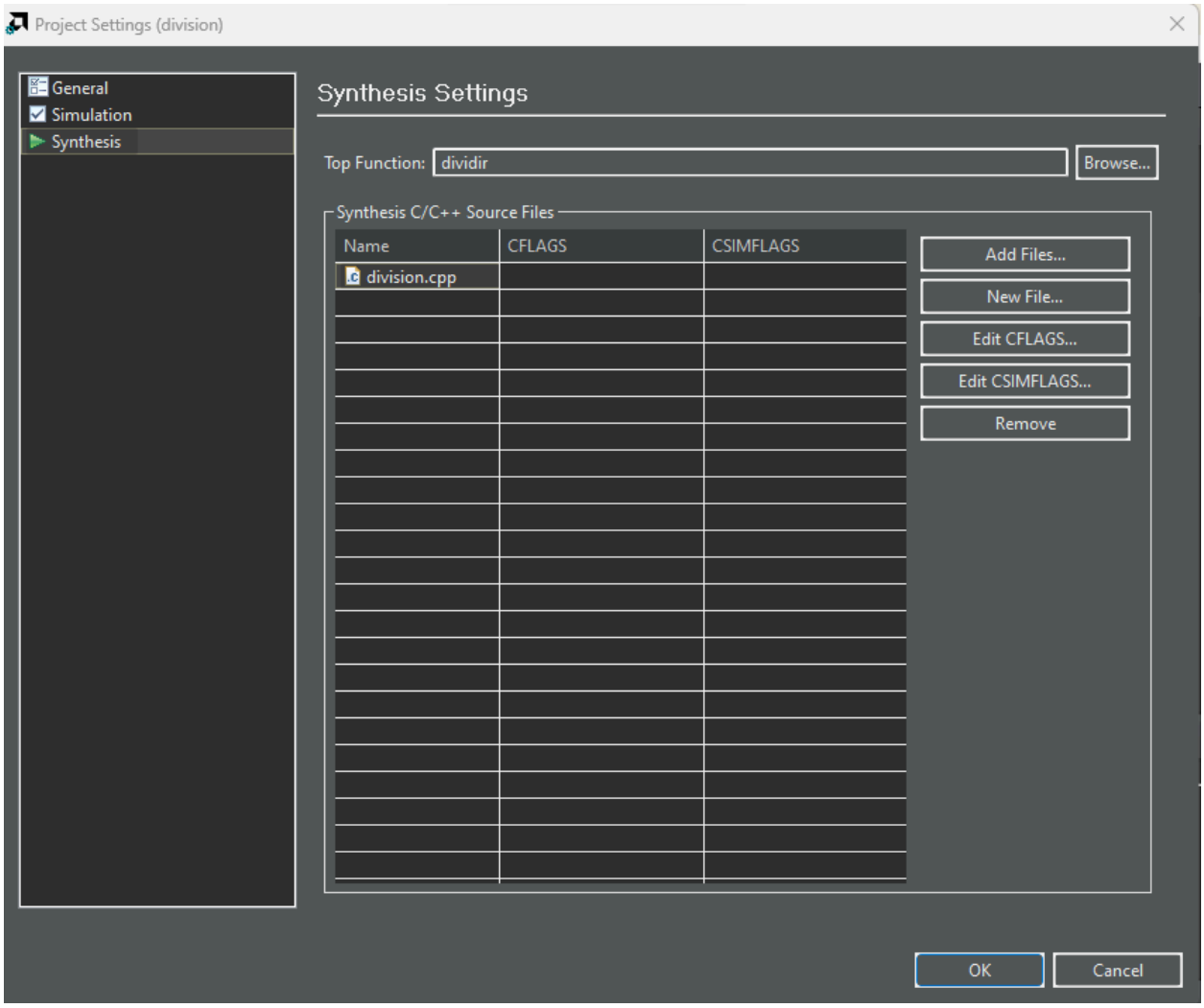
Es más fácil crear los archivos de origen C/C por separado utilizando su editor de texto preferido. Así que se codificó la función de división en C en el editor de texto y se guardo para luego importar en el nuevo proyecto de HLS:



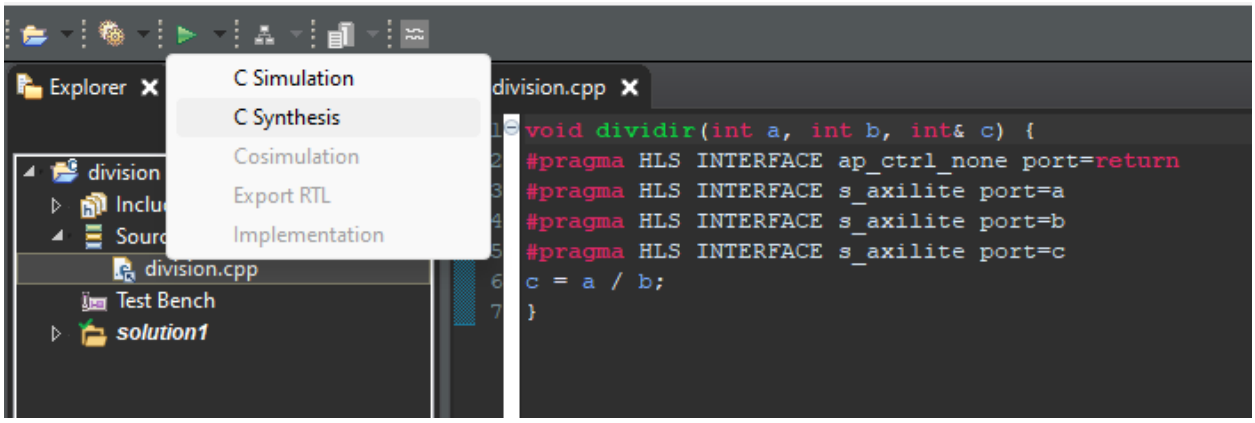
En source se sube el archivo anterior:



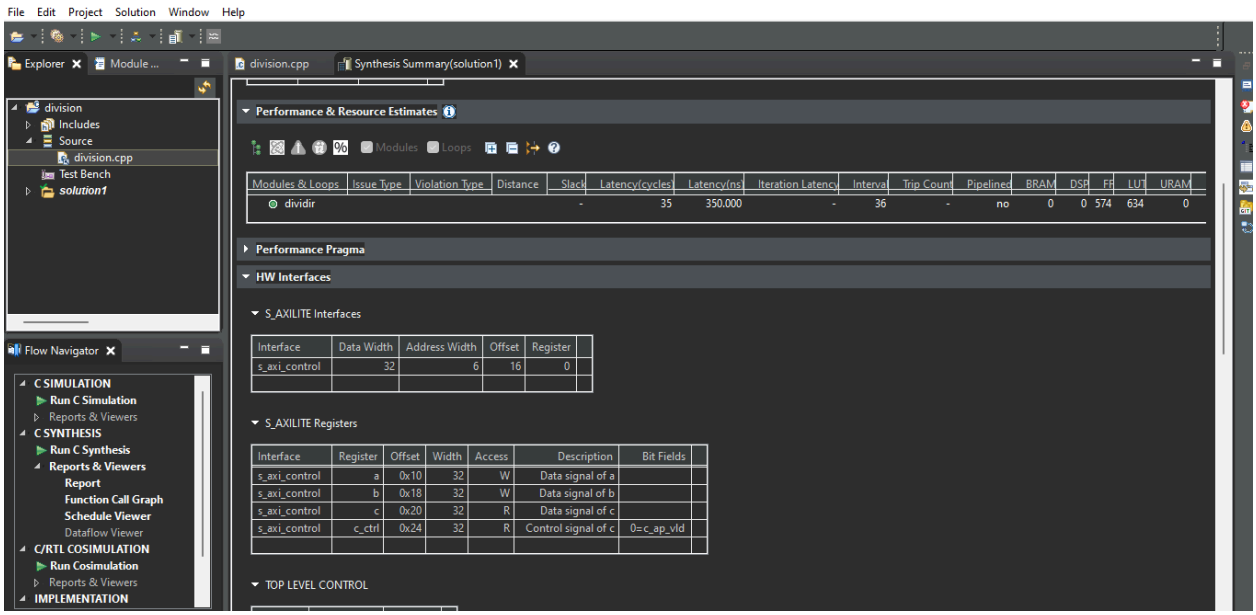
Tienes que añadirla como Top Function:



Ejecutamos C synthesis:

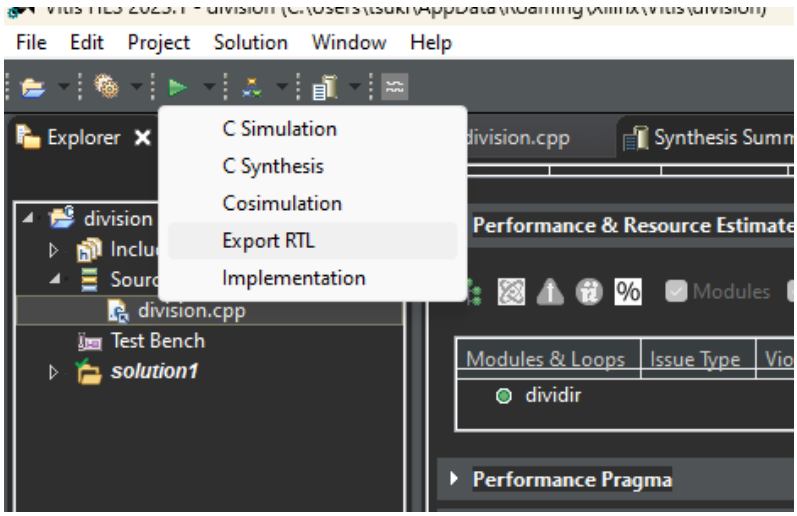


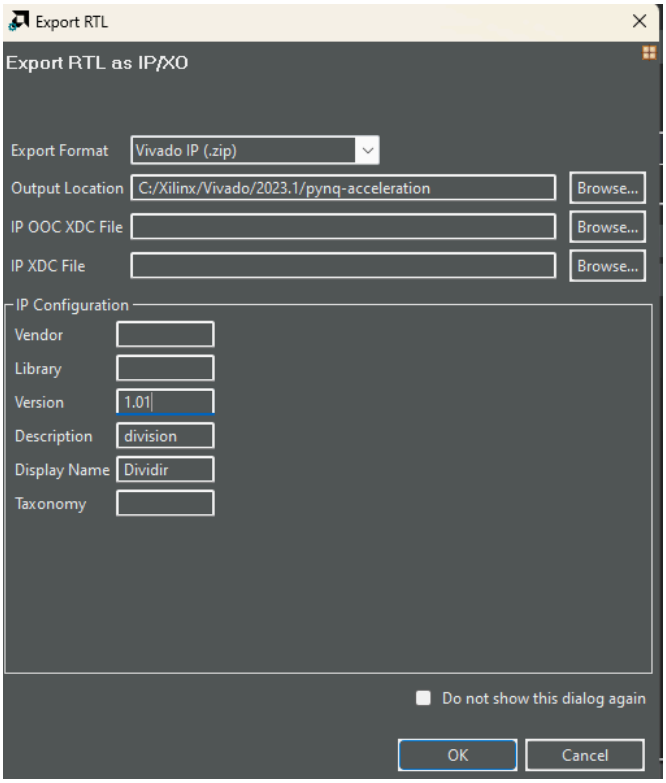
Dando como resultado:



2. Exportar RTL de HLS

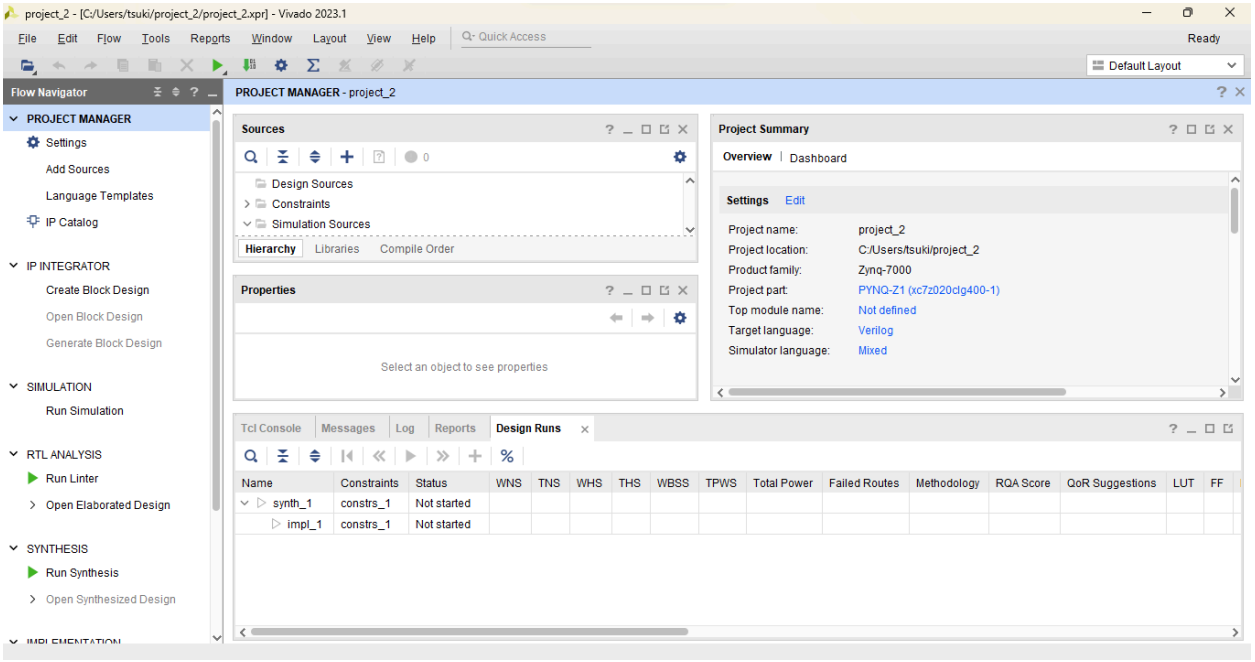
Para empaquetar la aplicación de C en una IP y exportarla para usarla en Vivado, seleccione Export RTL:



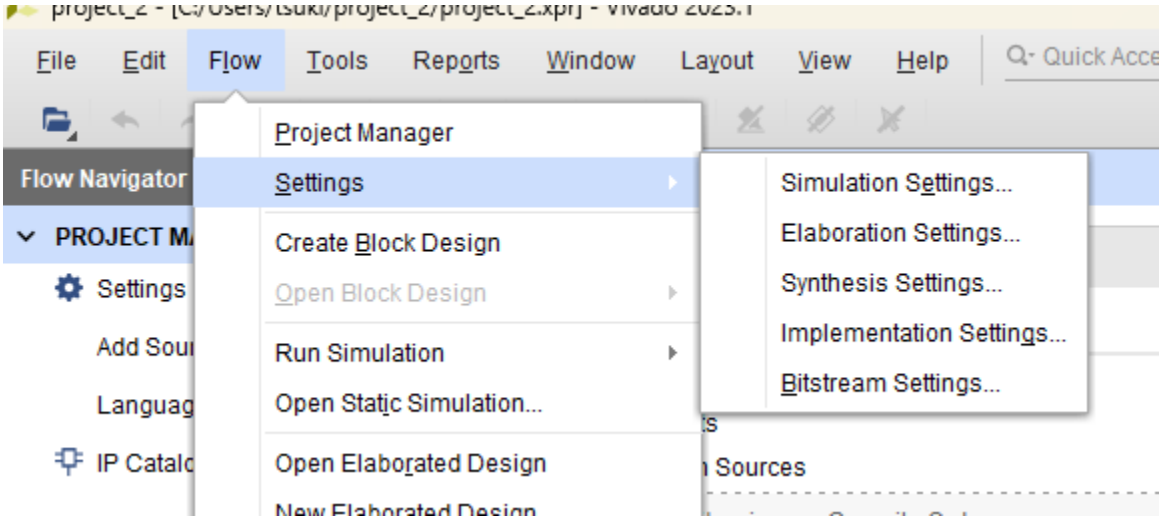


3. Crear el proyecto en Vivado

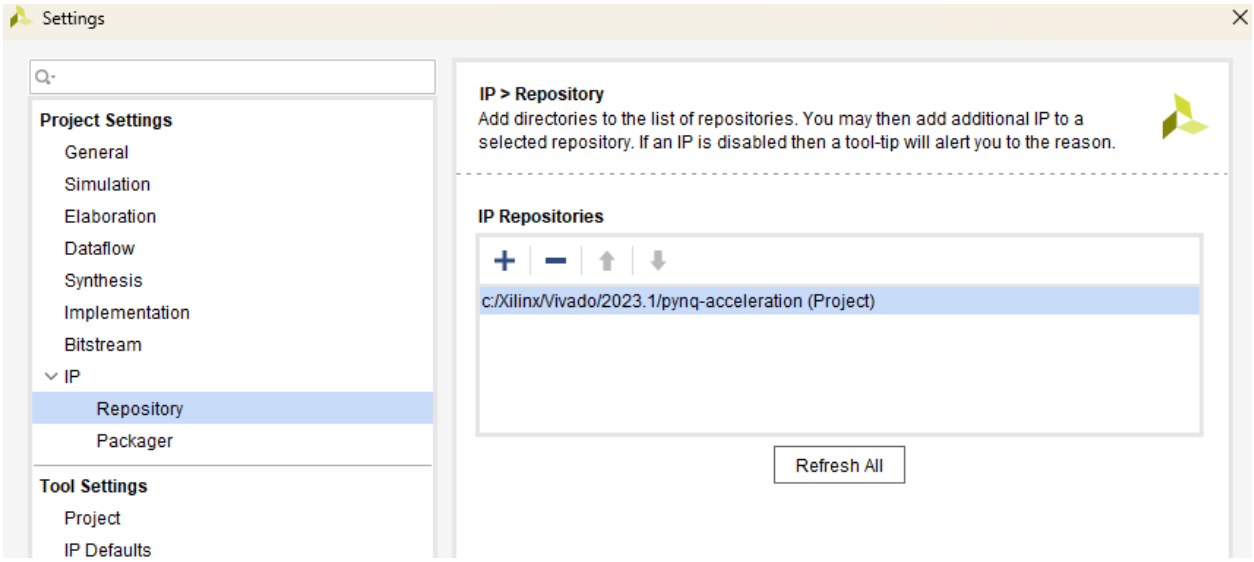
   Se siguen los mismos pasos que se explicaron en la parte de Vivado para tener esto:



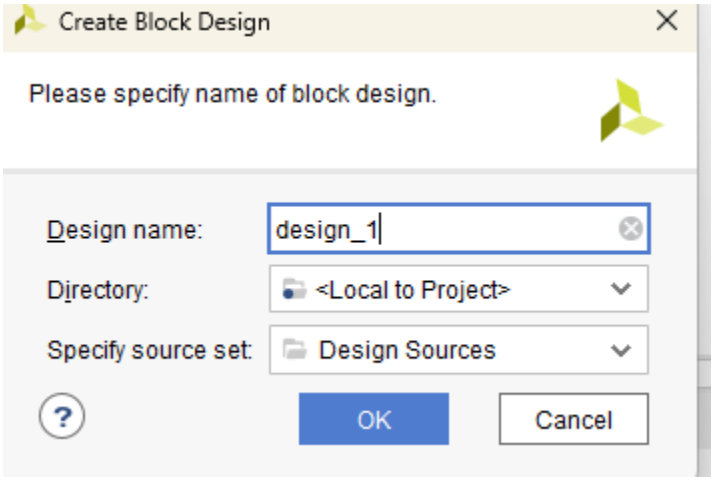
Desde aquí vamos a Simulation Settings:



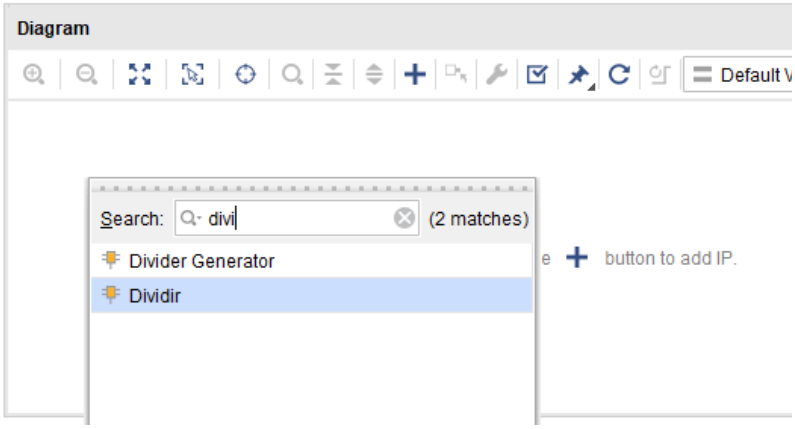
Se debe descomprimir la IP exportada (***export.zip***) de HLS en un directorio y luego añadir ese directorio al proyecto Vivado como repositorio IP:



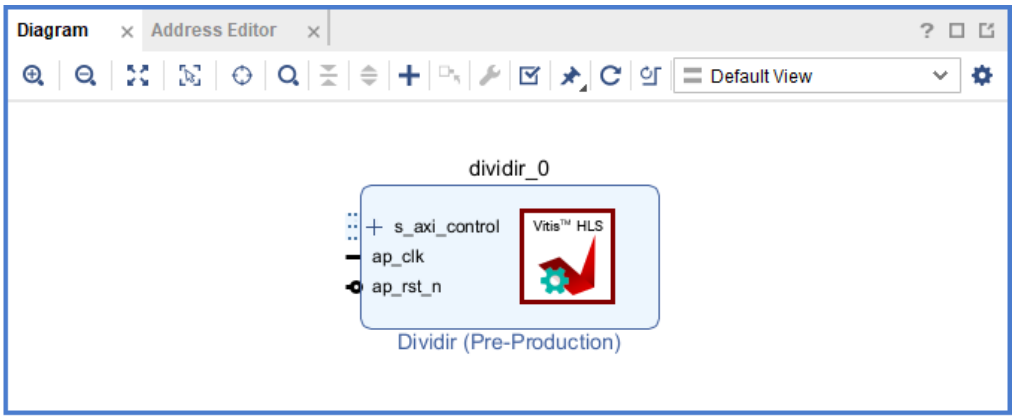
Se crea un nuevo Block Design:



Y buscamos nuestro IP:



Y ya está listo para usar en lo que necesitemos:



EJEMPLOS

Antes de empezar miremos cómo conectar una cámara USB, y usar la entrada y salida HDMI que nos provee la tarjeta:

La PYNQ incluye en su imagen un overlay básico cuyo propósito es permitir el uso de periféricos. El diseño incluye IP de hardware para controlar los periféricos en la placa de destino y conecta estos bloques de IP al Zynq PS.

```
from pynq.overlays.base import BaseOverlay
base_overlay = BaseOverlay("base.bit")
```

Configure una instancia de entrada y salida HDMI:

```
hdmi_in = base.video.hdmi_in
hdmi_out = base.video.hdmi_out
```

La interfaz de entrada HDMI se habilita mediante la función de configuración que, opcionalmente, puede tomar un parámetro de espacio de color. Si no se especifica ningún espacio de color, se utiliza BGR de 24 bits de forma predeterminada. El modo de entrada HDMI se utiliza para configurar el bloque de salida HDMI y especifica el espacio de color de salida y la resolución.

```
hdmi_in.configure()
hdmi_out.configure(hdmi_in.mode)
```

Una vez configurados los controladores HDMI, se pueden iniciar:

```
hdmi_in.start()
hdmi_out.start()
```

Para conectar una transmisión simple desde la entrada HDMI a la salida HDMI, las dos transmisiones se pueden unir:

```
hdmi_in.tie(hdmi_out)
```

Esto toma exactamente lo que entra y lo pasa directamente a la salida. Si bien la entrada y la salida están vinculadas, los fotogramas aún se pueden leer desde la entrada, pero cualquier llamada a `hdmi_out.writeframe` finalizará el vínculo:

```
frame = hdmi_in.readframe()
...
hdmi_out.writeframe(frame)
```

Esto permitiría realizar algún procesamiento en el marco de entrada HDMI antes de escribirlo en la salida HDMI.

El formato de píxeles predeterminado para las interfaces HDMI es de 24 bits, es decir, tres canales de 8 bits. Este se puede convertir a 8, 16, 24 o 32 bits.

El modo de 8 bits selecciona el primer canal en el píxel (y elimina los otros dos). El modo de 16 bits puede seleccionar los dos primeros canales o seleccionar el primero y realiza un remuestreo cromático de los otros dos, lo que da como resultado fotogramas con formato 4:2:2. El modo de 24 bits es de transferencia y no cambia el formato y el modo de 32 bits rellena la transmisión con 8 bits adicionales.

```
pixel_in = base.video.hdmi_in.pixel_pack
pixel_out = base.video.hdmi_out.pixel_unpack

pixel_in.bits_per_pixel = 8
pixel_out.bits_per_pixel = 16
pixel_out.resample = False
```

Para más información visitar:https://pynq.readthedocs.io/en/v2.4/pynq_libraries/video.html

Para leer desde una cámara USB se usa OpenCV de la siguiente forma:

```
# initialize camera from OpenCV
from pynq.drivers import Frame
import cv2
videoIn = cv2.VideoCapture(0)
videoIn.set(cv2.CAP_PROP_FRAME_WIDTH, frame_in_w);
videoIn.set(cv2.CAP_PROP_FRAME_HEIGHT, frame_in_h);
print("Capture device is open: " + str(videoIn.isOpened()))
```

Si funciona debería salir: `Capture device is open: True`

Algo que tener en cuenta es que no se puede tomar en si el video lo que se hará será ir tomando frames de la imagen y con esos frames se trabajara.

Y si queremos mostrar la salida de la cámara por HDMI seria:

```
import numpy as np
ret, frame_vga = videoIn.read()
# Display webcam image via HDMI Out
if (ret):
    outframe = hdmi_out.newframe()
    outframe[0:480,0:640,:] = frame_vga[0:480,0:640,:]
    hdmi_out.writeframe(outframe)
```

```
else:
    raise RuntimeError("Failed to read from camera.")
```

Otra opción para capturar imágenes desde una webcam seria, aqui se mostraria directamente en el jupyter notebook:

```
import cv2
from PIL import Image as PIL_Image
cap = cv2.VideoCapture(0)
_ , cv2_im = cap.read()
cv2_im = cv2.cvtColor(cv2_im,cv2.COLOR_BGR2RGB)
img = PIL_Image.fromarray(cv2_im)
img
```

Para liberar los HDMI y la camara seria:

```
videoIn.release()
hdmi_out.stop()
hdmi_in.stop()
del hdmi_in, hdmi_out
```

Filtros OpenCV

Veamos dos formas de realizar un detector de bordes, la primera con un filtro Laplaciano y la segunda usando el algoritmo de Canny, a partir de lo que ya vimos de usar una cámara.

Filtro Laplaciano:

```
import time
frame_1080p = np.zeros((1080,1920,3)).astype(np.uint8)
num_frames = 20
readError = 0
start = time.time()
for i in range (num_frames):
    # read next image
    ret, frame_vga = videoIn.read()
    if (ret):
        laplacian_frame = cv2.Laplacian(frame_vga, cv2.CV_8U)
        # copy to frame buffer / show on monitor reorder RGB (HDMI = GBR)
```

```
        frame_1080p[0:480,0:640,[0,1,2]] = laplacian_frame[0:480,0:640,
                                                           [1,0,2]]
        hdmi_out.frame_raw(bytearray(frame_1080p.astype(np.int8).tobytes()))
    else:
        readError += 1
end = time.time()

print("Frames per second: " + str((num_frames-readError) / (end - start)))
print("Number of read errors: " + str(readError))
```

Algoritmo de Canny:

```
frame_1080p = np.zeros((1080,1920,3)).astype(np.uint8)
num_frames = 20
start = time.time()
for i in range (num_frames):
    # read next image
    ret, frame_webcam = videoIn.read()
    if (ret):
        frame_canny = cv2.Canny(frame_webcam,100,110)
        frame_1080p[0:480,0:640,0] = frame_canny[0:480,0:640]
        frame_1080p[0:480,0:640,1] = frame_canny[0:480,0:640]
        frame_1080p[0:480,0:640,2] = frame_canny[0:480,0:640]
        # copy to frame buffer / show on monitor
        hdmi_out.frame_raw(bytearray(frame_1080p.astype(np.int8).tobytes()))
    else:
        readError += 1
end = time.time()

print("Frames per second: " + str((num_frames-readError) / (end - start)))
print("Number of read errors: " + str(readError))
```

Para ver la imagen usaremos matplotlib:

```
%matplotlib inline
from matplotlib import pyplot as plt
import numpy as np

plt.figure(1, figsize=(10, 10))
```

```
frame_vga = np.zeros((480,640,3)).astype(np.uint8)
frame_vga[0:480,0:640,0] = frame_canny[0:480,0:640]
frame_vga[0:480,0:640,1] = frame_canny[0:480,0:640]
frame_vga[0:480,0:640,2] = frame_canny[0:480,0:640]
plt.imshow(frame_vga[:,:,[2,1,0]])
plt.show()
```

OpenCV detector de Rostros

La detección de objetos mediante clasificadores Haar en cascada es un método eficaz de detección de objetos propuesto por Paul Viola y Michael Jones en su artículo '*Rapid Object Detection using a Boosted Cascade of Simple Features*' en 2001 (https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf). Es un enfoque basado en el aprendizaje automático en el que la función en cascada se entrena a partir de muchas imágenes positivas (en nuestro caso imágenes de rostros) y negativas (imágenes sin rostros), que luego se utiliza para detectar objetos en otras imágenes. OpenCV puede trabajar con estos Haarcascades y posee modelos previamente entrenados, aquí vamos a ver como implementar el detector de rostros mediante entrada HDMI:

```
import cv2
import numpy as np

frame = hdmi_in.readframe()

face_cascade = cv2.CascadeClassifier(
    '/home/xilinx/jupyter_notebooks/base/video/data/'
    'haarcascade_frontalface_default.xml')

gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(gray, 1.3, 5)

for (x,y,w,h) in faces: #Dibuja un rectángulo en el rostro
    cv2.rectangle(frame,(x,y),(x+w,y+h),(255,0,0),2)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = frame[y:y+h, x:x+w]
```

Para ver la imagen en el notebook

```
import PIL.Image
img = PIL.Image.fromarray(frame)
img
```

Si lo hacemos la entrada por camara:

```
import numpy as np
import cv2
ret, frame_vga = videoIn.read()

np_frame = frame_vga
face_cascade = cv2.CascadeClassifier(
    '/home/xilinx/jupyter_notebooks/base/video/data/'
    'haarcascade_frontalface_default.xml')
gray = cv2.cvtColor(np_frame, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(gray, 1.3, 5)

for (x,y,w,h) in faces:
    cv2.rectangle(np_frame,(x,y),(x+w,y+h),(255,0,0),2)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = np_frame[y:y+h, x:x+w]

# Output OpenCV results via matplotlib
%matplotlib inline
from matplotlib import pyplot as plt
import numpy as np
plt.imshow(np_frame[:,:,[2,1,0]])
plt.show()
```

Hay que tener en cuenta que el archivo .xml debe estar subido en la PYNQ, este y otros modelos podemos encontrarlos aqui: https://github.com/opencv/opencv/tree/master/data/haarcascades

Puedes crear tus propios Haarcascades, aquí dejamos algunas páginas con las que puedes guiarte:

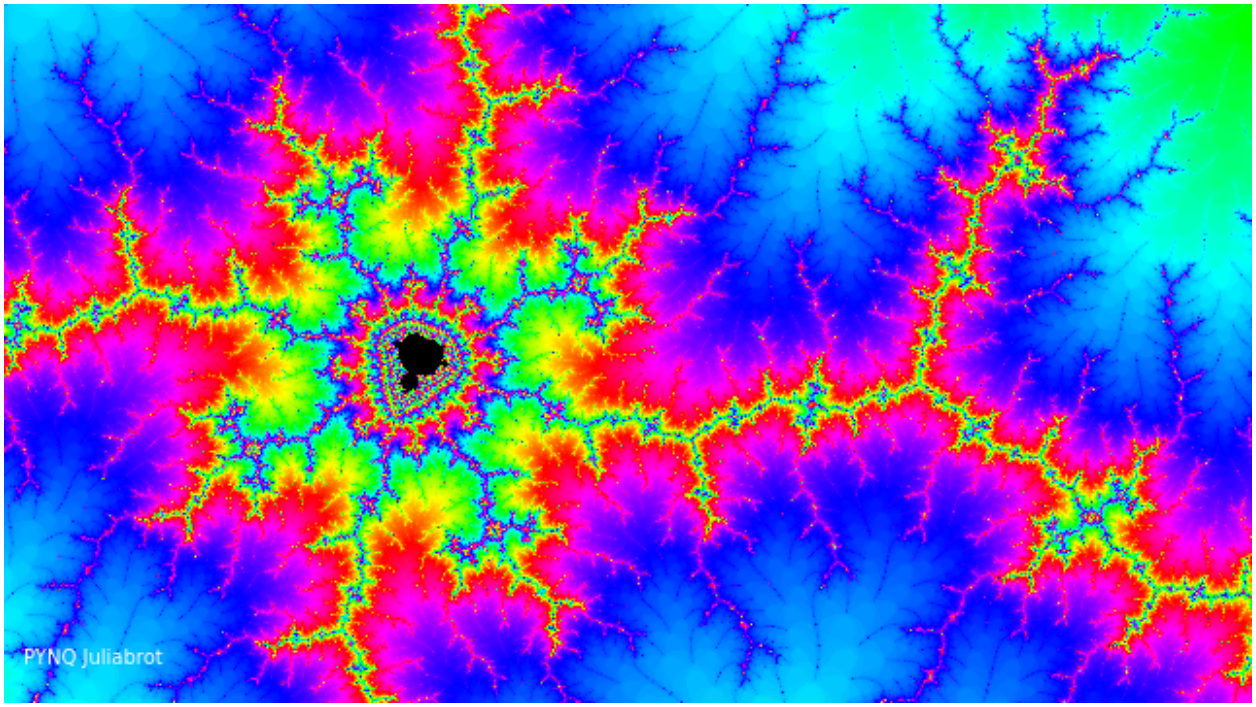
- https://omes-va.com/como-crear-tu-propio-detector-de-objetos-con-haar-cascade-python-y-opencv/
- https://medium.com/@vipulgote4/guide-to-make-custom-haar-cascade-xml-file-for-object-detection-with-opencv-6932e22c3f0e
- https://pythonprogramming.net/haar-cascade-object-detection-python-opencv-tutorial/

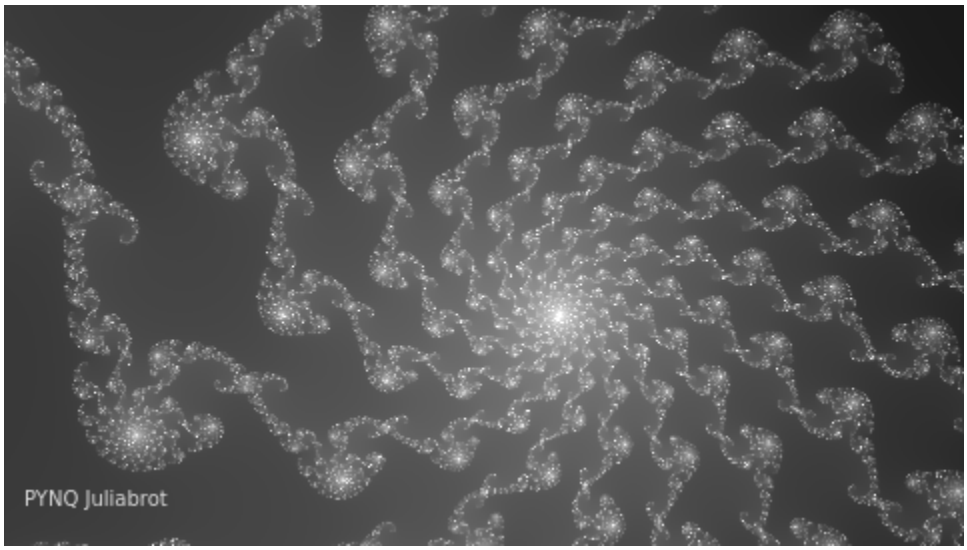
PYNQ Juliabrot Fractal Factory

Hay un proyecto interesante sobre los conjuntos de Julia, nombrados así por el matemático Gaston Julia, son una familia de conjuntos fractales que se obtienen al estudiar el comportamiento de los números complejos al ser iterados por una función. Aquí utilizan la función:

$$F_c(z) = z^2 + c$$

Donde c es un número complejo, para obtener un conjunto de Julia, tales como estos:





Se encuentra en el siguiente repositorio: https://github.com/FredKellerman/pynq-juliabrot/tree/master

BNN

Este repositorio contiene el paquete de instalación de pip para Binary Neural Networks (BNN) en PYNQ, inspirado en VGG-16, con 6 capas convolucionales, 3 capas de grupo máximo y 3 capas completamente conectadas.

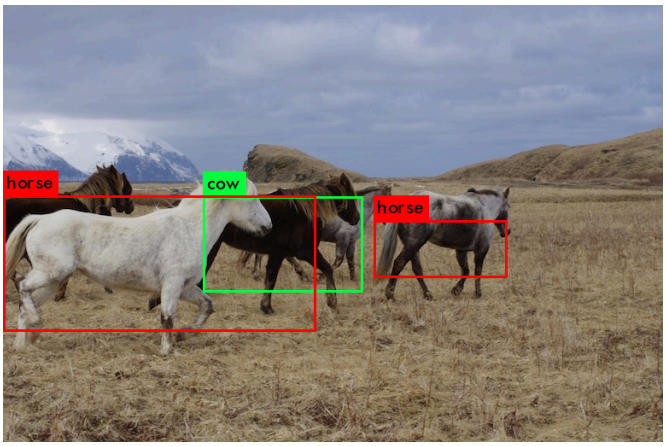
Realizan ejemplos con datasets como CIFAR-10 o MNIST.

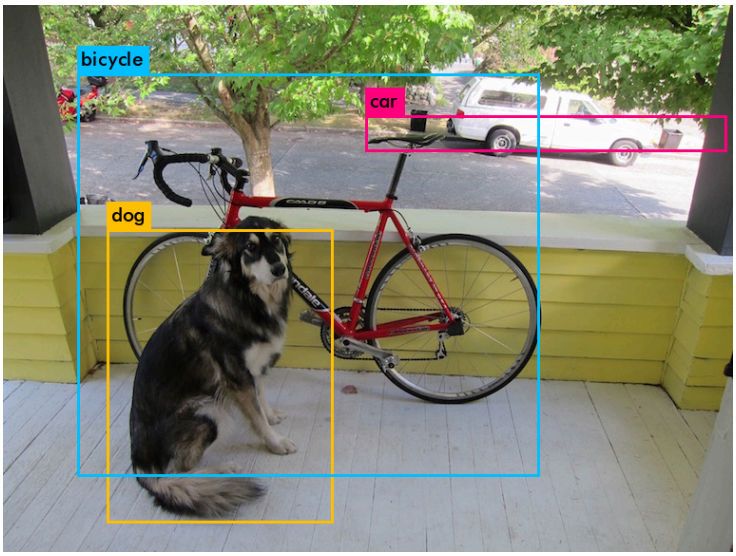
https://github.com/Xilinx/BNN-PYNQ/tree/master/notebooks

QNN

Este repositorio contiene el paquete de instalación de pip para Quantized Neural Network (QNN) en PYNQ utilizando una arquitectura de descarga multicapa (MO). Aquí se incluyen dos superposiciones diferentes, a saber, W1A2 (pesos de 1 bit, activaciones de 2 bits) y W1A3 (pesos de 1 bit, activaciones de 3 bits), que se ejecutan en la capa convolucional de lógica programable 1 y en la capa Max Pool 1 (opcional).

Además puede realizar modelos en Tiny-Yolo para reconocer múltiples objetos:





https://github.com/Xilinx/QNN-MO-PYNQ

Otros

Para ver más ejemplos visitar:

https://github.com/Xilinx/PYNQ/tree/v1.4/Pynq-Z1/notebooks/examples

http://www.pynq.io/community.html

https://github.com/Xilinx/PYNQ_Workshop