Trabajo Fin de Grado
Ingeniería Electrónica Robótica y Mecatrónica

Implementación de Fista en una FPGA para la resolución de problemas QP

Autor: Andreu Morro Gallego
Tutores: Juan Moreno Nadales y Pablo Krupa

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Ingeniería Electrónica Robótica y Mecatrónica

Implementación de Fista en una FPGA para la resolución de problemas QP

Autor:
Andreu Morro Gallego

Tutores:
Juan Moreno Nadales
Pablo Krupa

Dpto. Ingeniería de sistemas y automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Grado: Implementación de Fista en una FPGA para la resolución de problemas QP

Autor: Andreu Morro Gallego 

Tutores: Juan Moreno Nadales y Pablo Krupa

El tribunal nombrado para juzgar el Trabajo arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

Agradecimientos

Con este trabajo pongo fin a mis estudios en el grado de GIERM en la Escuela Técnica Superior de Ingeniería de Sevilla, y me gustaría aprovechar esta sección para agradecer a todas aquellas personas que me han acompañado estos años.
A mi familia, por todo el apoyo y amor que me han dado, y por no haber dudado nunca de mí, incluso en aquellos momentos en los que ni yo mismo me veía capaz.
A mis compañeros y amigos, porque todo es más fácil con personas como vosotros.
A mis profesores, no solo por enseñarnos lo correspondiente a sus asignaturas, sino también, por habernos hecho crecer como personas durante sus clases, y en especial a mis tutores Juan y Pablo por haberme ayudado y atendido siempre que se lo he pedido.

Gracias a todos, desde lo más profundo de mi corazón.

Andreu Morro Gallego

Sevilla, 2020

Resumen

El objetivo de este trabajo es implementar un algoritmo de resolución de problemas de programación cuadrática, llamado FISTA, e implementarlo en una FPGA utilizando síntesis de alto nivel.

Para ello veremos que son los problemas de programación cuadrática y una explicación sobre el algoritmo FISTA. Después tendremos una introducción sobre las FPGAs y sus ventajas sobre otros procesadores, para finalizar con una introducción a la síntesis de alto nivel y las ventajas que ofrece este tipo de técnica de diseño.

Una vez vistos los dos bloques teóricos veremos todos los pasos de implementación seguidos, empezando por diseñar el algoritmo en MATLAB, para después pasarlo a C y finalmente utilizando Vivado HLS sintetizarlo.

Una vez hecho se implementará en la FPGA.

Finalizaremos el proyecto con un análisis de los resultados obtenidos en las diferentes etapas de implementación y observaremos la correcta implementación del algoritmo en la FPGA.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Este TFC trata la resolución de problemas de Programación cuadrática (QP) utilizando el algoritmo FISTA[1][1] (Fast Iterative Shrinking-Threshold Algorithm) e implementado en una FPGA (Field-Programmable Gate Array).

Empezamos respondiendo a dos preguntas: ¿Por qué resolución de problemas QP? Y ¿Por qué implementarlo en una FPGA?

Problemas QPs

La programación cuadrática (QP) es el nombre que se le da a un procedimiento que minimiza un problema de optimización convexo sujeto a restricciones lineales de igualdad y desigualdad cuya función objetivo es una función cuadrática. La importancia de la programación cuadrática es debida a que un gran número de problemas aparecen de forma natural como cuadráticos, por ejemplo, optimización por mínimos cuadrados, pero además es importante porque aparece como un subproblema frecuentemente para resolver problemas no lineales más complicados [2]. Las técnicas propuestas para solucionar los problemas cuadráticos tienen mucha similitud con la programación lineal.

En concreto, en el ámbito de control muchos MPC (model predictive control) se resuelven resolviendo problemas QP, en concreto, los MPC basados en un modelo lineal [3].

Los algoritmos de control basados en MPC requieren resolver problemas de optimización en tiempo real, por lo que resulta de interés el desarrollo de algoritmos de optimización eficientes y su implementación en sistemas embebidos capaces de resolverlos en tiempos suficientemente pequeños.

FPGA

Los sistemas embebidos[2] constan de componentes de hardware y software diseñados para realizar una función específica y a menudo tienen restricciones de tiempo real o fiabilidad que van mucho más allá de la computación estándar. Para satisfacer estas demandas a nivel de hardware, los sistemas embebidos tradicionales suelen incorporar microcontroladores, procesadores de señales digitales (DSP) o arrays programables de puertas lógicas (FPGA). Respecto al software, tradicionalmente hacen falta varios lenguajes o herramientas para programar o configurar cada elemento de hardware. Como resultado, los equipos de diseño embebido tradicional requieren miembros con numerosos conocimientos en diseño de hardware y software capaces de integrar el hardware y el software en sistemas embebidos típicos.

Las FPGAs son chips programables que implementan funcionalidad de hardware personalizado. La llegada de herramientas de diseño de un nivel superior ha permitido a los usuarios que tienen amplia experiencia con el diseño de hardware digital aprovechar la tecnología FPGA. Las ventajas de FPGA son la fiabilidad y el determinismo del hardware sin el coste inicial y la rigidez del diseño de circuitos integrados de aplicación específica (ASIC). Además, las FPGA son más ventajosas que los procesadores multinúcleo que ejecutan muchas menos instrucciones en paralelo que lo que consiguen las FPGA existentes y que requieren pilas de software y controladores más sofisticados, lo que disminuye la fiabilidad en comparación con las FPGAs.

Por lo tanto, estas ventajas convierten las FPGAs en el elemento de hardware ideal para la creación rápida de prototipos y para conseguir un alto rendimiento en el diseño embebido. La tecnología FPGA ofrece a los diseñadores una vía más rápida al mercado con menor coste. Además, la capacidad de cargar nueva lógica y redefinir las conexiones en el tejido de la FPGA permite tener en cuenta usos futuros de diseños y beneficiarse

[1] FISTA es un algoritmo de resolución de QPs que se explicará en detalle en el bloque **2-Programación cuadrática,** concretamente en el apartado **2.7. Algoritmo FISTA**

[2] Un sistema embebido o empotrado es un sistema de computación diseñado para realizar una o algunas funciones dedicadas.

de actualizaciones más robustas sin necesidad de modificar sustancialmente el hardware.

Dentro de los algoritmos de resolución de QPs se ha elegido FISTA porque es un método rápido de primer orden muy utilizado [4].

Por estas razones en este TFC se ha optado por realizar esta implementación.

Para implementar el algoritmo en una FPGA primero se ha diseñado en MATLAB, después se ha pasado a C, se ha implementado y simulado en el microprocesador de la placa y finalmente utilizando síntesis de alto nivel (HLS) con la herramienta Vivado HLS se ha creado un módulo de propiedad intelectual (IP) y se ha implementado en la FPGA.

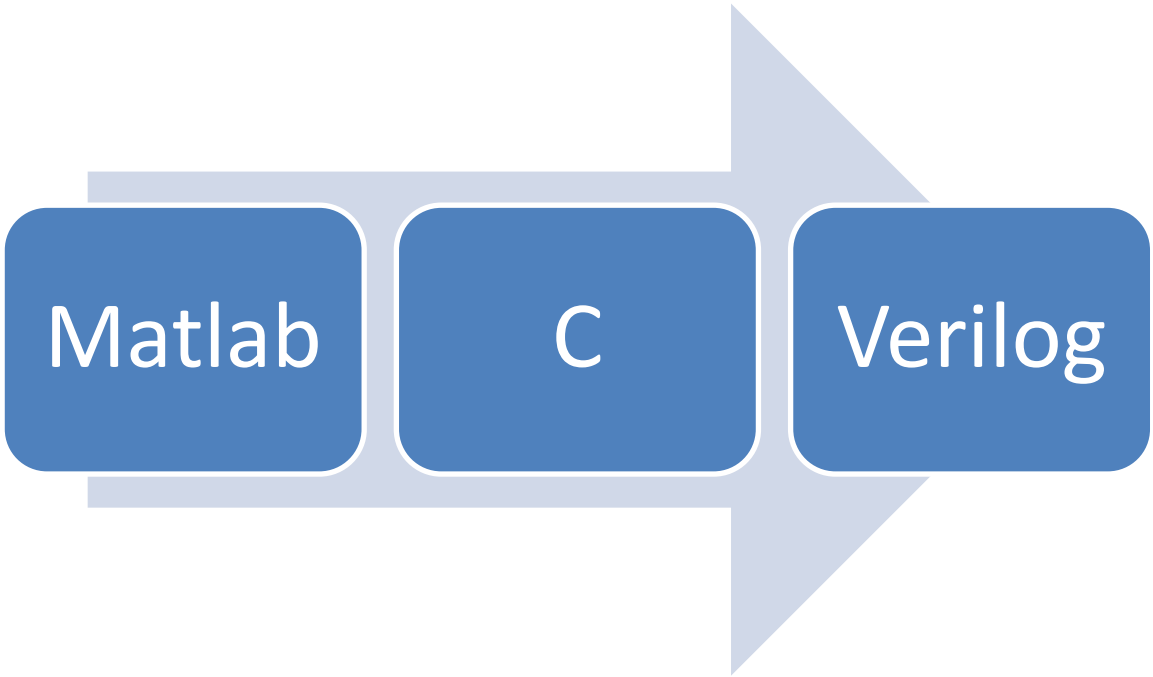

Figura 1-1 Proceso de Implementación del algoritmo FISTA

El trabajo está dividido en dos bloques teóricos. En el primero llamado 2-Programación Cuadrática veremos primeramente el origen de este, los diferentes métodos de resolución de los problemas QPs y finalmente una explicación del algoritmo FISTA. En el segundo bloque llamada 3-FPGAs veremos una pequeña introducción a lo que son las FPGAs y su arquitectura básica, después comentaremos las principales ventajas que ofrecen las FPGAs frente a otras plataformas y finalmente explicaremos que es la síntesis de alto nivel (HLS) y las ventajas que ofrece.

Después en el capítulo 4-Programacion e implementación del algoritmo veremos el diseño del algoritmo tanto en MATLAB como en C, después veremos cómo se ha utilizado Vivado HLS para la síntesis de alto nivel y finalmente implementarlo en la FPGA.

Finalmente, en el capítulo 5-Resultados y conclusiones veremos los resultados obtenidos de dicha implementación.

2 PROGRAMACIÓN CUADRÁTICA

Para comprender mejor la programación cuadrática (QP), vamos a ver primero lo que son los problemas de optimización, puesto que los problemas QP son un tipo de problemas de optimización, para después poder ver en detalle el problema QP que se ha resuelto utilizando el algoritmo FISTA.

2.1. Historia

Durante la segunda guerra mundial, la administración militar de Reino Unido llamó a un grupo de científicos de diferentes disciplinas científicas para usar su conocimiento y proveer asistencia tanto en problemas estratégicos, como tácticos de la guerra. Estos resultados obtenidos por los científicos británicos, motivaron a E.E.U.U a empezar actividades similares. La metodología aplicada por estos científicos para alcanzar sus objetivos se llamó O.R. (Operational Research).

La investigación operativa, conocida popularmente como O.R. es una adición reciente a un largo listado de herramientas científicas que brindan una nueva perspectiva a muchos problemas de gestión. La investigación operativa agrega mayor sofisticación hacia la resolución de problemas de gestión. Busca la determinación del mejor curso de acción (óptimo) de un problema de decisión bajo el factor limitante de recursos limitados. En consecuencia, O.R. se ha convertido en una herramienta versátil en el campo de gestión y su potencial para uso futuro es muy sustancial. O.R. demuestra una técnica científica eficaz para resolver tales problemas de toma de decisiones de industria y negocios modernos [5].

2.2. Problema de optimización

Una palabra clave asociada con O.R. es “optimización”. Optimización significa determinar el mejor curso de acción entre las diferentes alternativas disponibles en un problema de toma de decisiones. Puede considerarse como un proceso de encontrar el valor óptimo de una función (generalmente llamada función objetivo) bajo un conjunto dado de circunstancias (a menudo llamado “restricciones”).

Los problemas de optimización se pueden encontrar en casi todos los ámbitos de actividad humana. Esto ocurre en casi todas las ingenierías como Civil, Mecánica, Eléctrica, Telecomunicación, Química etc. Esto también ocurre en la industria económica y negocios. De hecho, el desarrollo de técnicas de optimización está siendo aplicado en todos los ámbitos donde las decisiones deben ser tomadas en una situación compleja que puede ser representada por un modelo matemático [5].

2.3. Solución a problemas de optimización

La solución de un problema de optimización generalmente implica tres fases: i) fase de modelado, ii) solución del modelo matemático, y iii) validación de los resultados y su implementación. De estos tres, la primera fase es la más importante. Un modelo incorrecto produce una solución incorrecta. Sin embargo, las otras dos fases también son igualmente importantes puesto que proporcionan la base para obtener la solución óptima y su implementación.

En la mayoría de los casos, el problema de optimización que nace de una descripción tiene que transformarse en un modelo matemático, a la que una o más de las técnicas disponibles de optimización pueden ser aplicadas [14]. Antiguamente, debido a la disponibilidad limitada de recursos, la tendencia era introducir aproximaciones y supuestos en el modelo matemático, para que pudieran resolverse convenientemente utilizando algunas técnicas de optimización. Sin embargo, la solución de éste no era precisa y el modelo matemático simplificado

a menudo no cumplía con las especificaciones del usuario. Ésta fue una de las principales razones por las que inicialmente estos métodos no eran muy utilizados. Sin embargo, con la disponibilidad de recursos informáticos en forma de ordenadores personales, y al mismo tiempo el desarrollo de técnicas computacionales más robustas y eficientes de optimización, el escenario ahora ha cambiado [5].

En concreto, en el ámbito de resolución de problemas QP, existen una gran variedad de metodologías de resolución entre las que destacan:

- **El Método del Punto Interior** [6]**.**
- **Conjunto Activo** [7].
- **Lagrangiana Aumentada** [8].
- **Gradiente Conjugado** [9].
- **Gradiente proyectado** [10].
- **Extensiones del algoritmo simplex** [7].

2.4. Descripción del problema QP

Consideramos el siguiente problema QP:

$$\min_{z} \frac{1}{2} z^T H z + q^T z \tag{2-1a}$$

$$s.a. \; z \in Z \tag{2-1b}$$

$$Gz = b \tag{2-1c}$$

Donde:

- $z \in \mathbb{R}^{n_z}$ son las variables de decisión
- $G \in \mathbb{R}^{m_z x n_z}$.
- b es un vector real m-dimensional
- q es un vector real n-dimensional

Con las siguientes asunciones realizadas:

- La matriz hessiana H $\in \mathbb{R}^{m_z \times n_z}$ es diagonal y definida positiva.
- $Z = \{z \in \mathbb{R}^{n_z} \mid \underline{z} \leq z \leq \bar{z}\}$ define un conjunto de restricciones en caja.
- $n_Z > m_z$ y el rango de G $= m_z$.
- Existe $\hat{z} \in ri(Z)$ tal que $G\hat{z} = b$, donde $ri(\cdot)$ representa un interior relativo.
- Dejando que $J(z) = \frac{1}{2} z^t H z + q^T z$ y z^*sean la función de coste y solución óptima de las ecuaciones (2-1), respectivamente. Denotamos el coste óptimo $J^* = J(z^*)$.

2.5. Algoritmo FISTA

Esta sección está basada en el artículo [11].

El algoritmo FISTA es un algoritmo de resolución de QPs perteneciente a la clasificación del gradiente conjugado y basado en el método de Nesterov [1].

Para comprender como funciona el algoritmo FISTA, primeramente, es necesario explicar lo que es la formulación dual.

2.5.1. Formulación Dual

Considerando el problema QP expuesto en (2-1) con las asunciones realizadas anteriormente. La función lagrangiana $L: \mathbb{R}^{n_z} \times \mathbb{R}^{m_z} \to \mathbb{R}$, la función dual $\Psi: \mathbb{R}^{m_z} \to \mathbb{R}$, y $z(\lambda)$ se pueden definir como

$$L(z,\lambda) = \frac{1}{2} z^T H z + q^T z - \lambda^T (Gz - b) \tag{2-2}$$

$$\psi(\lambda) = \inf_{z \in Z} L(z,\lambda) \tag{2-3}$$

$$z(\lambda) = argmin_{z \in Z} L(z,\lambda) \tag{2-4}$$

Donde $\lambda \in \mathbb{R}^{mz}$ es la variable dual.

Debido a las asunciones realizadas, se cumplen las condiciones de Slater [12], por lo que se da la dualidad fuerte, es decir, $J^* = \psi(\lambda^*)$ y $z^* = z(\lambda^*)$ con

$$\lambda^* = arg\, \max_{\lambda} \psi(\lambda) \tag{2-5}$$

Definición 1: Dado el problema QP descrito en (2-1) con matriz hessiana $H \in D_+^{nz}$ y matriz de restricción de igualdad $G \in \mathbb{R}^{m_z \times n_z}$, denotamos W_H como

$$W_H = GH^{-1}G^T \in \mathbb{R}^{m_z \times m_z} \tag{2-6}$$

Con las asunciones realizadas en (2-1) y [13, Lemma 3.1], tenemos que $\psi(\cdot)$ es una función fuertemenete suave, por lo tanto, es continuamente diferenciable con un gradiente Lipschitz. Además, se puede demostrar que el parámetro de suavidad viene dado por la matriz W_H dada en la definición 1, es decir, que $\psi(\cdot)$ satisface,

$$\psi(\lambda + \Delta\lambda) \geq \psi(\lambda) - \Delta\lambda^T (Gz(\lambda) - b) - \frac{1}{2}\lambda^T W_H \Delta\lambda \tag{2-7}$$

Por lo tanto, dado a λ_k un valor de $\lambda_{k+1} = \lambda_k + \Delta\lambda_k$ tal que $\psi(\lambda_{k+1}) \geq \psi(\lambda_k)$ puede ser obtenido tomando $\Delta\lambda$ como solución de

$$\Delta\lambda_k = \arg\max_{\Delta\lambda} -\Delta\lambda^T (G(z_k) - b) - \frac{1}{2}\lambda^T W_H \Delta\lambda \tag{2-8}$$

Cuya solución puede ser obtenida resolviendo el siguiente sistema de ecuaciones

$$W_H \Delta\lambda_k = -(Gz(\lambda_k) - b) \tag{2-9}$$

El $\psi(\lambda^*)$ óptimo es alcanzado cuando $\psi(\lambda_{k+1}) = \psi(\lambda_k) = \psi(\lambda^*)$, que ocurre cuando $\Delta\lambda = 0$,

O, equivalentemente cuando $Gz(\lambda_k) - b = 0$. Pero dado que esta condición no se da en la práctica, tomamos la solución subóptima z_k que satisface

$$\|Gz(\lambda_k) - b\| \leq \epsilon \tag{2-10}$$

Siendo $\epsilon > 0$ un parámetro de exactitud y $\|\cdot\|$ cualquier norma.

Cabe remarcar que si se mantienen las asunciones realizadas en el aparado (2-1), los elementos de $z(\lambda)$, llamados como $z(\lambda)_i \; para \; i = 1, \ldots \ldots n_z$, se puede calcular resolviendo los siguientes problemas de optimización, ya que no hay acoplamiento de variables $z(\lambda)_i$:

$$z(\lambda)_i = \arg\min_{\hat{z}\in\mathbb{R}} \frac{1}{2} H_{i,i}\hat{z}^2 + (q_i - \sum_{j=1}^{m_z} G_{j,i}\lambda_j)\hat{z} \tag{2-11a}$$

$$\text{s.a. } \underline{z_i} \leq \hat{z} \leq \bar{z}_i \tag{2-11b}$$

El cual tiene la siguiente solución explicita:

$$z(\lambda)_i = \max\{\min\{\frac{-\left(q_i - \sum_{j=1}^{m_z} G_{i,j}\lambda_j\right)}{H_{i,i}}, \bar{z}_i\}, \underline{z_i}\} \tag{2-17}$$

2.5.2 Algoritmo Fista

Finalmente procedemos a explicar el algoritmo paso a paso:

Paso 0: definir λ_o, normalmente, $\lambda_o = 0$.

Paso 1: inicializar las variables $k = 0, \eta_o = \lambda_o, t_o = 1$.

Paso 2: Repetir.

Paso 3: Incrementamos k, $k = k + 1$.

Paso 4: obtener $z(\lambda_{k-1})$ utilizando la ecuación (2-17).

Paso 5: obtener $\Delta\lambda_{k-1}$. Resolviendo el sistema de ecuaciones (2-9)

Paso 6: calcular $\eta_k = \lambda_{k-1} + \Delta\lambda_{k-1}$.

Paso 7: Calcular $t_k = \frac{1}{2}\left(1 + \sqrt{1 + 4t_{k-1}^2}\right)$.

Paso 8: Calcular $\lambda_k = \eta_k + \frac{t_{k-1}-1}{t_k}(\eta_k - \eta_{k-1})$.

Paso 9: Calcular el residuo $\Gamma = Gz(\lambda_k) - b$.

Paso 10: Hasta que $\|\Gamma\| < \epsilon$.

Salida: $z^* = z(\lambda_k)$.

3 FPGA

Las FPGAs fueron introducidas hace casi dos décadas y media. Desde entonces han experimentado un rápido crecimiento y se han convertido en un medio de implementación popular para circuitos digitales. Los grandes avances tecnológicos han permitido una mejora en la capacidad lógica de las FPGAs y las ha convertido en un método alternativo de implementación para diseños grandes y complejos. Además, la naturaleza programable de sus recursos lógicos y de enrutamiento ha tenido un gran impacto en la calidad del área, la velocidad y el consumo de energía del dispositivo [14].

3.1. Introducción

Las FPGAs son dispositivos fabricados con silicio que pueden ser programados eléctricamente para convertirse en casi cualquier tipo de circuito digital o sistema. Para producciones de bajo a medio volumen, las FPGAs proveen soluciones más baratas y en menor tiempo que los circuitos integrados de aplicación específica (ASIC) los cuales normalmente requieren muchos recursos tanto de dinero como de tiempo para obtener el primer dispositivo. Mientras que las FPGAs toman menos tiempo para programarse que por ejemplo un ASIC y su precio oscila entre unos cientos o unos miles de dólares. También para variaciones de requerimientos las FPGAs pueden ser parcialmente reconfiguradas, mientras el resto de FPGA sigue ejecutándose. La mayor ventaja que presentas las FPGAs frente a su contraparte los ASICs es la flexibilidad. Esto provoca que las FPGAs sean más grandes y consuman más energía que los ASICs. Estas desventajas surgen en gran parte por la interconexión de enrutamiento que presentan las FPGAs que corresponde casi el 90% del área total de ésta [14].

3.2. Conceptos básicos sobre las FPGAs

Normalmente las FPGAs se componen de:

- Bloques de lógica programable (CLB) donde se implementan las funciones lógicas.
- Enrutamiento programable que conecta estas funciones lógicas
- Bloques de entrada/salida (I/O) que están conectados a estos bloques lógicos a través del enrutamiento interconectado hechos fueran del chip.

Un ejemplo de esto puede ver en la figura 3-1, donde las celdas lógicas están interconectadas mediante las fuentes de enrutamiento programable. Los bloques entrada/salida están situados en las periferias de la celda y también están interconectadas mediante las fuentes de enrutamiento.

El término “programable/reconfigurable” en las FPGAs hace referencia a la habilidad de implementar nuevas funciones en el chip después de que su fabricación sea completa [15].

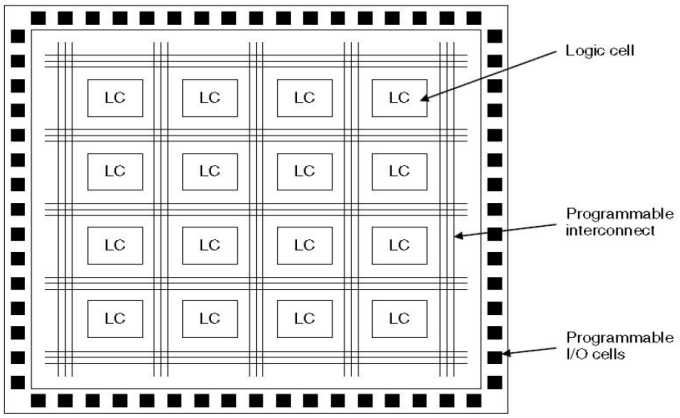

Figura 3-1 Esquema de la arquitectura FPGA.

3.2.1 Celda lógica programable Básica

Una celda lógica programable básica está compuesta por tres elementos principales: una LUT (Look-Up Table), un biestable y un multiplexor. La tecnología en la que está basada la construcción de estos elementos básicos de lógica CMOS. Dependiendo del fabricante del dispositivo la configuración de una celda programable varía (principalmente en el número y capacidad de cada uno de estos elementos de lógica), pero todos tienen una configuración parecida a la siguiente figura:

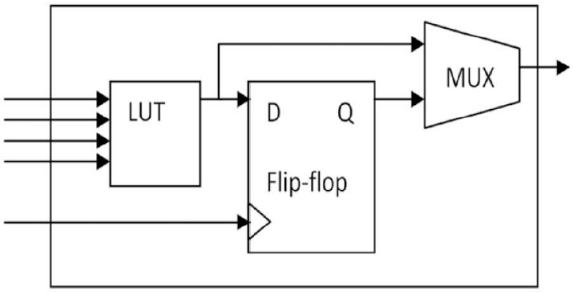

Figura 3-2: Esquema de una celda básica programable

LUTs

Una LUT es un pequeño bloque que actúa como una memoria ROM. Se trata de un bloque con cuatro entradas y una salida que tiene la capacidad de generar con look_ups cualquiera de las cuatro funciones básicas (AND/OR/XOR/NOT), aunque dependiendo del tamaño de la LUT se puede implementar cualquier función lógica compuesta por estas funciones básicas. Una vez se programa la memoria de configuración de la FPGA, la LUT se configura para implementar la función lógica deseada [15].

Biestables

Los biestables o Flip-Flops son componentes que permiten guardar un bit de información en cada momento. Son muy importantes de cara a realizar un diseño secuencial ya que permiten guardar el estado del sistema en el ciclo de tiempo anterior y hacerlo accesible para los elementos lógicos. En la figura se puede observar el diseño mediante puertas lógicas de un biestable.

En este tipo de biestables el dato D se guardará en el momento en el que haya un flanco de la señal de reloj por medio de la señal Q, que no variará hasta que no haya otro flanco de reloj. El uso de varios biestables interconectados entre ellos puede formar lo que se conoce como registro [15].

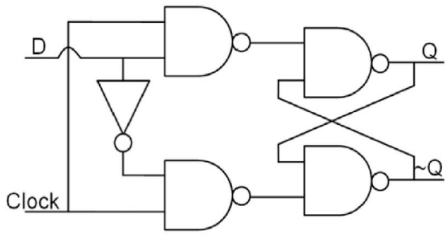

Figura 3-3: Diseño mediante puertas lógicas de biestable

Multiplexor

Un multiplexor es un bloque combinacional con dos o más entradas, que tendrá como máximo una señal menos que, de entrada, escogida mediante una señal de selección.

3.2.2 Celda de entrada/salida programable (IOB)

Se trata de otro de los componentes básicos de una FPGA. Se utiliza para conectar los bloques lógicos básicos

al exterior. Se pueden configurar como entradas, salidas, bidireccionales o desconectados. Puede manejar diferentes tipos de especificaciones eléctricas de entrada/salida (TTL, 3.3V, CMOS o PCIe) y puede añadir resistencias de pull-up o pull-down [15].

3.2.3 Matriz de interconexiones

Es necesario que haya una interconexión entre los bloques configurables y los IOBs para que la comunicación entre estos sea posible. Es por eso que para que las conexiones sean eficientes, se disponen de CLBs y las IOBs de manera matricial (en FPGAs modernas se pueden encontrar distribuciones tridimensionales, como es el caso de la tecnología *stacked silicon interconnect* (SSI) de Xilinx). Los IOBs estarán los más cerca posible del borde del circuito. Los cables que interconectarán Les con IOBs serán configurables y generalmente las herramientas de configuración de la propia FPGA se encargan de realizar estas interconexiones [15].

3.2.4 Memoria interna

Los dispositivos de lógica programable, como las FPGAs son configurados a través de un archivo de configuración que es enviado desde el ordenador hasta la placa. Dentro de la placa, la configuración será leída y guardada en la memoria interna del dispositivo. Esta memoria puede ser tanto volátil como no volátil [15].

Memoria volátil

Con este tipo de memoria, la información de configuración solamente será retenida mientras el dispositivo esté conectado a una fuente de energía. Cuando se produzca una desconexión de la red, los datos de configuración serán perdidos. Las FPGAs fabricadas por Intel y Xilinx, que copan alrededor del 80% del mercado utilizan memoria volátil [15].

Memoria no volátil

Hay algunos modelos de FPGA que utilizan memorias no volátiles para conservar la configuración en el momento en el que se corta el suministro eléctrico. Las FPGA más modernas pueden incorporar una memoria Flash para la retención de esta configuración. Las FPGAs más conocidas de tipo Flash son Microsemi y LATICCE. Son dispositivos más pequeños y generalmente están orientados al bajo consumo [15].

3.3. Ventajas del Uso de FPGAs

Hay ciertas ventajas en usar las FPGAs sobre un microprocesador como un ASIC en un prototipo o un diseño de producción limitada. Las principales ventajas son su gran flexibilidad, la reusabilidad y su mayor rapidez de adquisición.

Las principales ventajas que ofrecen las FPGAs son:

- Mejor rendimiento: Generalmente una CPU no puede ejecutar procesos en paralelo, esto da a las FPGAs una ventaja ya que pueden realizar el procesamiento y cálculo en paralelo a una velocidad más rápida. Un programa diseñado cuidadosamente en una FPGA normalmente, puede ejecutar cualquier función más rápido que una CPU que ejecuta código de forma secuencial.
- Coste: Como las FPGAs pueden ser reprogramables una y otra vez resultan extremadamente rentables a largo plazo.
- Prototipado: Como se menciona anteriormente las FPGAs son reprogramables y reutilizables. Esto las convierte en la perfecta elección para la creación de prototipos, especialmente para los propósitos de validación de un ASIC. Es importante determinar si el diseño del ASIC está funcionando, ya que son muy costosos de fabricar y si se termina con un chip que necesita modificaciones, se tendrá que invertir una cantidad importante de tiempo y dinero para rediseñarlo. Con la reprogramabilidad de las FPGAs se puede testar el funcionamiento y determinar la configuración ideal en un solo chip. Una vez que se ha finalizado la etapa de prototipado y determinado cual es la mejor solución se puede utilizar la FPGA como un sistema ASIC.
- Tiempo de comercialización más rápido: otra ventaja de las FPGAs es que permite terminar el desarrollo del producto en un periodo de tiempo muy corto, lo que significa un tiempo de

comercialización más corto. Las herramientas de diseño de FPGAs son fáciles de usar y no requieren una gran curva de aprendizaje.

- Aplicaciones en tiempo real: Las FPGAs son perfectas para sistemas de tiempo crítico debido a su arquitectura de procesamiento más eficiente. Como tales, son ideales para aplicaciones en tiempo real, ya que pueden realizar más procesamiento en un período de tiempo más corto en comparación con otras alternativas en el mercado.

3.4. Placa de Desarrollo e implementación

Para el desarrollo e implementación del algoritmo FISTA se ha utilizado la placa: **Zybo Z7-20: Zynq-7000 ARM/FPGA SoC Development Board,** que forma parte de la familia de placas con CPU integrada y modelo Zynq-2000 explicadas en el apartado anterior.

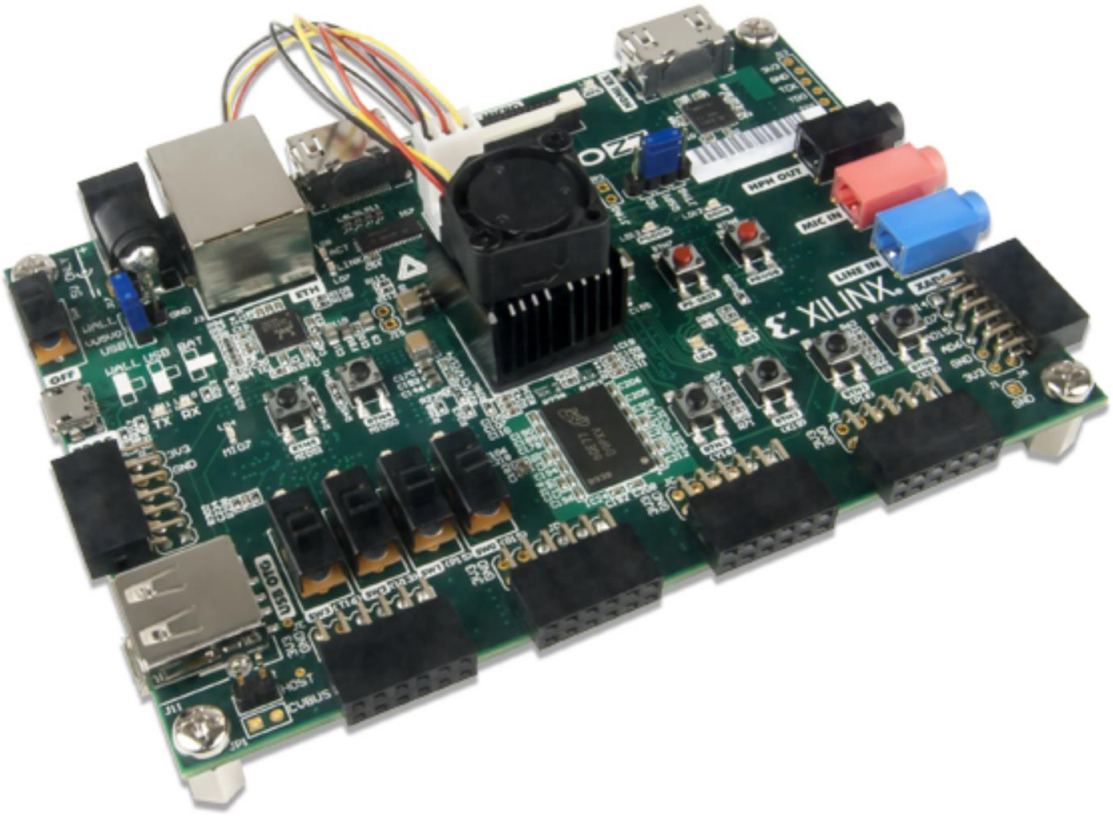

Figura 3-4 Placa Zybo Z7-20: Zynq-7000 ARM/FPGA SoC Development Board

La Zybo Z7 es una placa con software integrado y una placa de desarrollo de circuitos digitales con muchas funcionalidades listas para usar, construido alrededor de la familia Xilinx Zynq-7000. La familia Zynq se basa en la arquitectura Xilinx All Programmable System-on-Chip (AP SoC), que integra estrechamente un procesador ARM Cortex-A9 de doble núcleo con lógica Xilinx 7-series Field Programmable Gate Array (FPGA). La Zybo Z7 rodea al Zynq con un rico conjunto de periféricos multimedia y de conectividad para crear una formidable computadora de placa única, incluso antes de considerar la flexibilidad y potencia agregadas por la FPGA. El conjunto de funciones con capacidad de video del Zybo Z7, que incluye un conector Pcam compatible con MIPI CSI-2, entrada HDMI, salida HDMI y ancho de banda DDR3L alto, fue elegido para convertirlo en una solución asequible para las aplicaciones de visión integrada de alta gama. Los conectores Pmod del Zybo Z7 facilitan la instalación de hardware adicional, lo que permite el acceso al catálogo de Digilent de más de 70 placas periféricas Pmod, incluidos controladores de motor, sensores, pantallas y más [16].

La principal característica de esta placa, como se menciona arriba, es que es un SoC, que son especialmente útiles para el módulo que se va a diseñar en el capítulo 4, ya que este se puede integrar como esclavo del

procesador.

3.5. Lenguaje de descripción Hardware

Tradicionalmente se han utilizado lenguajes de descripción hardware, como Verilog o VHDL, pero en la actualidad, a pesar de ser menos eficiente, es preferible utilizar la síntesis de alto nivel, porque se ahorra tiempo de diseño y se aumenta la productividad.

Un lenguaje de descripción de hardware (HDL, hardware description language) es un lenguaje de programación especializado que se utiliza para definir la estructura, diseño y operación de circuitos electrónicos, y más comúnmente, de circuitos electrónicos digitales, como el convertidor analógico-digital o cualquier antena satelital. Así, los lenguajes de descripción de hardware hacen posible una descripción formal de un circuito electrónico, y posibilitan su análisis automático y su simulación. Los lenguajes de descripción de hardware consisten en una descripción textual con expresiones, declaraciones y estructuras de control. Sin embargo, una importante diferencia entre los HDL y otros lenguajes de programación está en que el HDL incluye explícitamente la noción de tiempo. Así, los HDL pueden ser usados para escribir especificaciones "ejecutables" de hardware. Es decir, un programa escrito en HDL hace posible que el diseñador de hardware pueda modelar y simular un componente electrónico antes de que este sea construido físicamente. Es esta posibilidad de "ejecución" de componentes lo que hace que a veces los HDL se vean como lenguajes de programación convencionales, cuando en realidad se debería clasificarlos más precisamente como lenguajes de modelado [17].

El modelado de comportamiento abstracto y el modelado de la estructura hardware son dos aspectos de los lenguajes de descripción Hardware (HDL). El modelado de comportamiento abstracto hace referencia a un lenguaje cuando es declarativo con el objetivo de facilitar la descripción del comportamiento hardware de un circuito el cual está siendo implementado, por otro lado, el modelado de la estructura hardware es aquella estructura que puede ser modelada en un HDL independientemente al lenguaje de descripción hardware usado para modelar el comportamiento del diseño, teniendo la posibilidad de ser modelado y representado en distintos niveles de abstracción durante el proceso de diseño.
La diferencia principal entre los modelos con altos y bajos niveles de abstracción es que mientras que los modelos con niveles bajos de abstracción incluyen más detalles, los modelos de bajos niveles de abstracción describen el comportamiento del diseño de forma abstracta. Gracias a los lenguajes de descripción hardware se puede representar diagramas lógicos y circuitos con diferentes grados de complejidad. Además, representan sistemas digitales de manera legible tanto para las máquinas como para las personas.
La metodología de diseño en un lenguaje de descripción de hardware habla de la manera en la que los procesos, destacando la complejidad y la abstracción del diseño, se ordenan para lograr que el costo y el tiempo de desarrollo sea el menos posible, de esta manera se logra ofrecer una mejor calidad, garantizando la confiabilidad del producto obtenido. Los pasos a seguir para llevar a cabo una metodología óptima son: Definir el nivel de abstracción inicial, realizar una descomposición jerárquica, definir la estructuración de los nuevos niveles jerárquicos, desarrollar la arquitectura necesaria, y finalmente, seleccionar la tecnología que se va a utilizar.
El diseño Bottom-Up y Top-Down son dos diseños diferentes que se llevan a cabo para orientar el orden de las acciones denominado flujo de diseño. El diseño Top-Down es el más utilizado, gracias a que sus descripciones son independientes de la tecnología, esto lleva a un aumento en la reutilización del diseño en diferentes casos. El diseño Bottom-Up no es un diseño recomendado ya que en la mayoría de los casos ha resultado ser ineficiente en diseños complejos y dependiente de la tecnología.
Usando HDL se puede describir la operación del sistema con diferentes niveles de abstracción:

- Nivel de conmutadores, se utiliza para describir el circuito en términos de transistores y cables.
- Nivel de compuertas, para describir el circuito en términos de compuertas lógicas y elementos de almacenamiento como flip-flops.
- Nivel de flujo de dato, los cuales describen el circuito en términos de flujo de datos entre registros.
- Nivel algorítmico o comportamental, incluye instrucciones de alto nivel como lazos, comandos de decisión y otros, su comportamiento es similar a un programa en un lenguaje de alto nivel como C.

Dentro de las características principales de los HDL se encuentran que permiten modelizar el concepto de tiempo, esta característica es fundamental para llevar a cabo la descripción de sistemas electrónicos; describe actividades que ocurren de forma simultánea, permite tanto la construcción de una estructura jerárquica como la

descripción de módulos con acciones que se evalúan en forma secuencial. La construcción de una estructura jerárquica posibilita combinar descripciones estructurales y de flujo de datos con descripciones de comportamiento.
Existen variedad de ventajas al utilizar HDL como herramienta para sistemas de diseño digital. Gracias a sus herramientas de especificación es posible su uso para la especificación general de un sistema en hardware y software, de igual manera describe el hardware, sus sistemas, subsistemas y componentes. Las herramientas de diseño proporcionan mejor documentación y facilita el proceso de reúso, adicional a esto presenta independencia tecnológica y posibilidad de parametrización. Al hablar de herramientas de simulación cabe destacar la facilidad para generar vectores de test complejos, como también la disponibilidad existente de modelos de distintos componentes de fabricantes variados en HDL normalizados.
Así como se encuentran ventajas al llevar a cabo un proyecto con un lenguaje de descripción hardware, existen distintas desventajas. Entre estas se encuentra que su uso involuntariamente pierde interés sobre el aspecto físico del diseño ya que se enfoca principalmente a su funcionalidad. La necesidad de adquirir conocimientos desde cero de este lenguaje es otra desventaja ya que supone un esfuerzo de aprendizaje debido a que es una nueva metodología que se debe estudiar y aprender a llevar. Es necesaria la utilización de nuevas herramientas como simuladores y sintetizadores de HDL.
Las aplicaciones enfocadas en el lenguaje de descripción de hardware se basan en modelización, simulación, síntesis, y reusabilidad. La modelización se enfoca en la creación del diseño, para llegar a esto se lleva a cabo un diseño de flujo de datos que nos ayuda a hacernos una idea de los componentes que se necesitan en el diseño. La simulación se lleva a cabo para verificar el correcto funcionamiento de los modelos en cualquier etapa del diseño además de corregir fallos de forma teórica, sin necesidad de que el circuito este implementado. Para la síntesis se usan los HDL como entrada en las herramientas de síntesis, como salida se obtiene el diseño del circuito. Reusabilidad se basa en la capacidad que tiene el código que se ha creado en ser usado en otros programas diferentes, es decir, el reúso de este trabajo [17].

3.6. Síntesis de alto nivel

La síntesis de alto nivel (HLS) es un paradigma de diseño clave para realizar cualquier sistema, especialmente aquellos de elevada complejidad. En realidad, a medida que progresa la tecnología y los sistemas aumentan su complejidad el uso de modelos de alto nivel permite a los diseñadores ser más productivos porque diseñan los sistemas en lugar de los circuitos. Coincidiendo así con la tendencia de la industria que ofrece un número cada vez mayor de sistemas integrados en comparación a los circuitos integrados.

El potencial de la síntesis de alto nivel se relaciona con dejar detalles de implementación a los algoritmos y herramientas de diseño, incluida la capacidad de determinar la precisión y sincronización de las operaciones, transferencias de datos y almacenamiento. La optimización de alto nivel, acoplada con síntesis de alto nivel, puede proporcionar a los diseñadores la estructura de concurrencia óptima para un flujo de datos y las limitaciones tecnológicas correspondientes, proporcionando así el equilibrio en la compensación entre latencia y uso de recursos.[18]

3.5.1 Introducción

La complejidad de los circuitos integrales digitales se ha incrementado. Los diseñadores a menudo han tenido que adaptarse al desafío de proporcionar soluciones comercialmente aceptables con una optimización de recursos. Gracias al incremento del factor integral ofrecido por la tecnología de nodos, la complejidad en los últimos SoCs ha alcanzado decenas de millones de puertas.

Por lo tanto, con el incremento de la densidad de los chips de las FPGAs es posible implementar algoritmos más sofisticados. Pero programar una FPGA utilizando un lenguaje de nivel de transferencia de dato (RTL) es una forma poco eficiente desde el punto de vista del consumo de tiempo y es propenso a errores. Para utilizar la posibilidad de reprogramabilidad de la FPGA para un espacio de diseño rápido y un tiempo de comercialización también rápido, se vuelve cada vez más necesario aumentar un nivel de abstracción desde el diseño estructural hasta el diseño de comportamiento. Por eso la síntesis de alto nivel ha irrumpido con fuerza.

La síntesis de alto nivel es el proceso de transformar un programa o código escrito en un lenguaje de alto como C, C++ o SystemC en una descripción hardware en HDL. La síntesis de alto nivel transforma las

especificaciones sin tiempo en especificaciones temporales que luego, automáticamente se sintetizan en puertas. El poder de la síntesis de alto nivel es que permite la exploración del espacio de diseño a nivel de sistema y reduce los esfuerzos de diseño y verificación [19].

3.5.2 Ventajas que ofrece HLS

La principal ventaja de utilizar síntesis de alto nivel es que los equipos de diseño aceleran enormemente el tiempo de diseño y al mismo tiempo se reduce el esfuerzo general de verificación, por esto es una técnica muy utilizadas en los tiempos actuales.

Es muy difícil crear un modelo RTL con la suficiente eficiencia sin errores y problemas durante la fase de verificación. La síntesis de alto nivel aborda la raíz de este problema al proporcionar una ruta libre de errores de la etapa de especificaciones hasta RTL.

Cuando se trabaja en un alto nivel de abstracción, se necesitan muchos menos detalles para la descripción. Por ejemplo, a nivel funcional, los programadores no necesitan preocuparse por los detalles de implementación. como jerarquía, procesos, relojes o tecnología. Son libres de enfocarse solo en el comportamiento deseado. Esto hace que la descripción sea mucho más fácil de escribir. Con menos líneas de código, el riesgo de los errores se reduce en gran medida, por lo que es más fácil verificar exhaustivamente el modelo.

Una vez escrito y verificado el modelo de alto nivel, HLS automatiza el proceso de implementación RTL. Pero si por un lado las herramientas HLS eliminan las intervenciones manuales y los errores, no eliminan la intervención humana. Es decir, aún deben tomarse decisiones. Con síntesis de alto nivel, los programadores mantienen el control; ellos toman las decisiones y la herramienta HLS las implementa. Los programadores simplemente tienen una forma más eficiente y productiva de hacer su trabajo. Por ejemplo, el diseñador decide el nivel adecuado de paralelismo para una arquitectura óptima y restringe la herramienta HLS en consecuencia. A su vez, la herramienta se encarga de asignar y programar los recursos de hardware, construyendo la ruta de datos y las estructuras de control para producir una implementación optimizada. Con HLS, el RTL se obtiene más rápidamente, acortando la fase de creación. A su vez, se reduce la sobrecarga de depuración y se reduce la carga de verificación [20].

3.5.3 Etapas de HLS

Este apartado está basado en el artículo [21].

Generalmente las etapas de la síntesis de alto nivel son (figura 3-5).

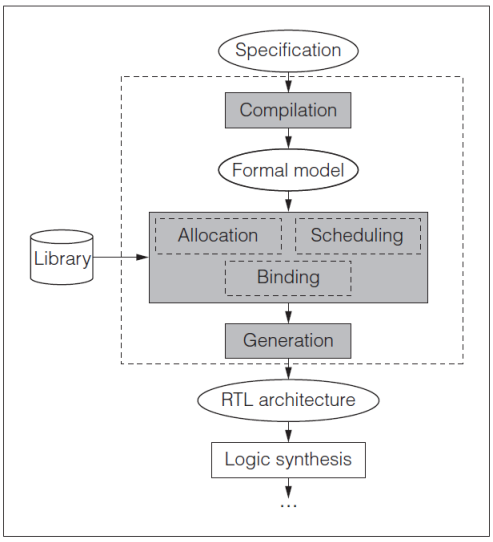

Figura 3-5 Etapas síntesis de alto nivel

- **Compilación:** La síntesis de alto nivel siempre empieza con la compilación de una función específica. Este primer paso, tradicionalmente incluye optimizaciones de código.

- **Asignación:** La asignación define el tipo y el número de recursos hardware (por ejemplo, unidades funcionales, componentes de almacenamiento o conectividad) necesarios para satisfacer las limitaciones de diseño. Según la herramienta que se utilice para la síntesis de alto nivel, se pueden agregar algunos componentes durante las etapas de planificación y vinculación. Por ejemplo, los componentes de conectividad. Los componentes son seleccionados de la librería RTL. Es importante seleccionar, al menos una vez, un componente para cada operación en la especificación del modelo. La librería debe incluir las características de los componentes (como área, retraso y potencia).
- **Planificación:** Todas las operaciones requeridas en la especificación del modelo deben programarse en ciclos. Dependiendo del componente funcional al que la operación está mapeada, las operaciones se pueden programar dentro de un ciclo de reloj o para varios ciclos. Las operaciones se pueden encadenar (la salida de una operación alimenta directamente una entrada de otra operación). Las operaciones pueden programarse para ejecutarse en paralelo siempre que no haya dependencias de datos entre ellos y hay recursos suficientes disponible al mismo tiempo.
- **Vinculación:** Cada operación definida en las especificaciones del modelo debe estar vinculada a una de las unidades funcionales capaces de ejecutar la operación.
- **Generación:** Una vez tomadas las decisiones en la etapa de vinculación, planificación y asignación, el objetivo del paso de generación de la arquitectura RTL es aplicar todas las decisiones de diseño tomadas y generar un RTL del modelo sintetizado.

3.7. Vivado HLS

Para realizar la síntesis de alto nivel se ha utilizado la herramienta de Xilinx Vivado HLS. Vivado HLS, es la modificación y personalización realizada por Xilinx de las herramientas de SAN AutoESL AutoPilot que a su vez se deriva del entorno XPilot de la Universidad de California Los Ángeles [22][23]. Vivado HLS sintetiza el modelo algorítmico en C/C++ y SystemC generando un bloque de propiedad intelectual (IP) o exportando el código RTL en VHDL/Verilog para síntesis e implementación en una FPGA de Xilinx (figura 3-6). Utiliza el entorno de compilación de código abierto LLVM [24] y se aprovecha de sus continuas optimizaciones. Vivado HLS soporta C, C++ y SystemC, lo que permite realizar la simulación del código C realizando un mínimo conjunto de modificaciones.
Vivado HLS realiza la síntesis del algoritmo, donde se produce la transformación de las funciones en la arquitectura RTL, incluyendo las tareas de planificación, asignación y enlazado de unidades funcionales. Estas operaciones afectan a la síntesis de las interfaces y son afectadas a su vez por esta.
La síntesis de las interfaces realiza la transformación de los argumentos y parámetros de las funciones en puertos RTL con el protocolo de comunicación elegido para la comunicación con otros bloques del diseño. Afecta a variables globales, argumentos de las funciones de mayor nivel de jerarquía y valores retornados por las funciones.
Las principales técnicas de optimización usadas por Vivado HLS son las siguientes:

- Optimización del flujo de datos o entre funciones (Dataflow pipelining)
- Segmentación de las funciones
- Optimización de latencias
- Enrollado y desenrollado de bucles
- Segmentación de bucles
- Modificación de los intervalos de iniciación
- Mapeado de arrays a memorias
- Limitación o compartición de recursos

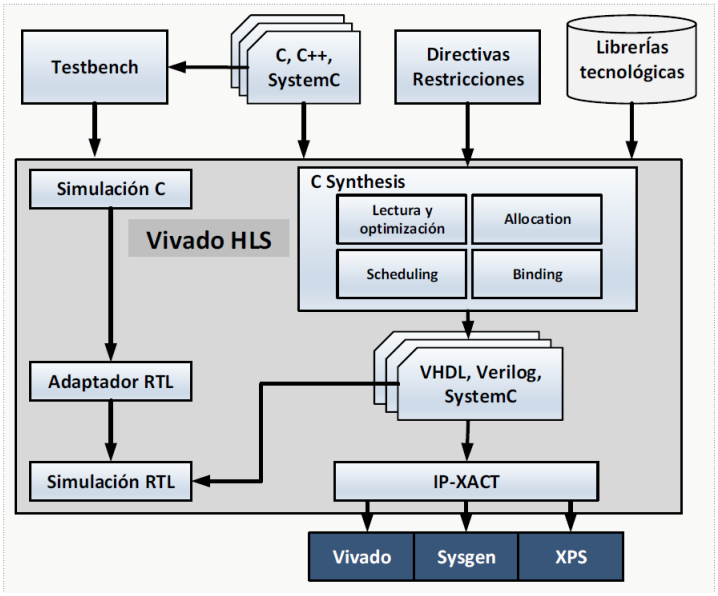

Figura 3-6 Flujo de diseño en Vivado

4 DISEÑO E IMPLEMENTACIÓN DEL ALGORITMO

Tal como se ha explicado en la introducción, el diseño del algoritmo ha pasado por varias etapas antes de poder ser implementando en la placa.

A continuación, se van a comentar las características principales del código para cada etapa.

4.1. Programación en MATLAB y C

Debido a que tanto el código en C como en MATLAB son muy parecidos se van a explicar juntos en la misma sección comentando las principales diferencias entre ellos.

Para evitar la confusión entre los símbolos matemáticos descritos en el apartado **2.6.2 Algoritmo FISTA** y el programa en MATLAB y C se ha hecho una tabla de equivalencias entre los símbolos y las variables:

Símbolo Matemático	Nombre en MATLAB y C
λ_o	Lambda_k
η_o	eta_k
t_o	t_k
$z(\lambda_{k-1})$	z_ka
$\Delta\lambda_{k-1}$	deltalambda_ka
Γ	residuo
ϵ	epsilon
z^*	z_salida
$\underline{z}$	UB
$\bar{z}$	LB

Tabla 4-1 Correspondencia símbolos con programa

Para evitar colocar muchas líneas de código se van a comentar las características principales de cada paso del algoritmo. El código completo puede consultarse en el anexo A y en el anexo B siendo los códigos en MATLAB y en C respectivamente.

Las variables de entrada del algoritmo son matriz G, matriz H, vector q, vector b, vector UB y vector LB.

Los pasos seguidos para el algoritmo FISTA están explicados en el apartado 2.5.2.

- El paso 1 es idéntico para los dos códigos y consiste en inicializar las variables lambda_k, k, eta_k y t_k.
- Para el paso 2 se ha utilizado un bucle **while**[3]para los dos códigos. Para MATLAB se ha utilizado la condición de que la norma del residuo sea mayor que la variable épsilon, es decir mientras que la norma

[3] El bucle while es un ciclo repetitivo basado en los resultados de una expresión lógica.

del residuo es mayor que el valor de épsilon dado el bucle **while** se mantiene activo:

```
while(norm(residuo) > epsilon)
```

Para el codigo en C se ha utilizado un bucle **while,** pero se ha añadido la condición de que el bucle solo se ejecute una serie limitada de veces con la variable kmax. Es el algoritmo puede salir del bucle **while** de dos maneras diferentes, la primera igual que en MATLAB si la norma del residuo es mayor que la variable epsilon o si se ha alcanzado el número máximo de repeticiones.

```
while((norma > epsilon) && (k < kmax))
```

- El paso tres simplemente consiste en incrementar la variable k para ambos códigos.

```
k=k+1;
```

- Para el paso cuatro se ha utilizado un doble bucle **for**[4] en los dos códigos para calcular z_ka como se indica en la ecuación (2-17). La diferencia reside en que en a la hora de calcular el máximo y mínimo en MATLAB se han utilizado las funciones **max** y **min** respectivamente, mientras que en C al no contar con estas funciones se ha utilizado el condicional **if-else**[5]**.**

- El paso cinco consiste en calcular deltalambda_ka resolviendo la ecuación (2-9).

  En Matlab:

```
Wh=(G/H)*G';
```

```
deltalambda_ka=(inv(Wh))*(-(G*z_ka'-b));
```

  Mientras que en C se deben utilizar dobles bucles **for** para las operaciones matriciales, como puede verse en el anexo B

- El paso seis consiste en calcular eta_k y es idéntico para ambos códigos con la salvedad que en C se requiere un bucle **for** para sumar los dos vectores lambda_ka y deltalambda_ka

```
eta_k=lambda_ka+deltalambda_ka;
```

- El paso siete es idéntico para ambos códigos y consiste en calcular la variable t_k.

```
t_k=(1+sqrt(1+4*t_ka^2))/2;
```

- En el paso ocho calculamos el vector lambda_ka y como en el paso seis en MATLAB es inmediato, mientras que en C requerimos de bucles **for**.

```
lambda_k=eta_k+((t_ka-1)/t_k)*(eta_k-eta_ka);
```

- El paso nueve consiste en calcular el residuo en MATLAB es inmediato y en C se requieren bucles **for** para los productos matriciales

```
residuo=G*z_k'-b;
```

- Y finalmente el paso diez consiste en calcular la norma del residuo, para continuar o salir del bucle **while**, en MATLAB se calcula dentro de la propia condición del bucle con la función **norm**, mientras que en C se calcula con un bucle **for.**

```
for(int i=0;i<120;i++){
      aux6=aux6+(residuo[i]*residuo[i]);
}
norma=sqrt(aux6);
```

  Si finalmente no se cumplen alguna de las condiciones del bucle **while** copiamos los valores de z_k al vector z_salida.

[4] El bucle for es un bucle que repite un bloque de instrucciones dadas un número predeterminado de veces.
[5] El condicional if-else es una estructura que nos posibilita definir las acciones que se deben llevar a cabo si se cumple cierta condición y también determinar las acciones que se deben ejecutar si no se cumple.

4.2 Implementación utilizando HLS

Como se ha comentado anteriormente para realizar la síntesis de alto nivel se ha utilizado la herramienta de Xilinx Vivado HLS y se han seguido las siguientes etapas:

- Simulación en el procesador de la placa.
- Una vez nos aseguramos que el código en C implementado en el procesador de la placa nos da el mismo resultado que en C procedemos a realizar la cosimulación C/RTL.
- Finalmente, una vez hecha la cosimulación y el resultado ha sido satisfactorio procedemos a realizar la síntesis y exportación del módulo IP.

4.2.1 Simulación en el procesador

Primeramente, nos dirigimos a la aplicación Vivado HLS en este caso se ha utilizado la versión 2019.2.

Pulsamos el botón “create new proyect” como se muestra en la figura 4-1.

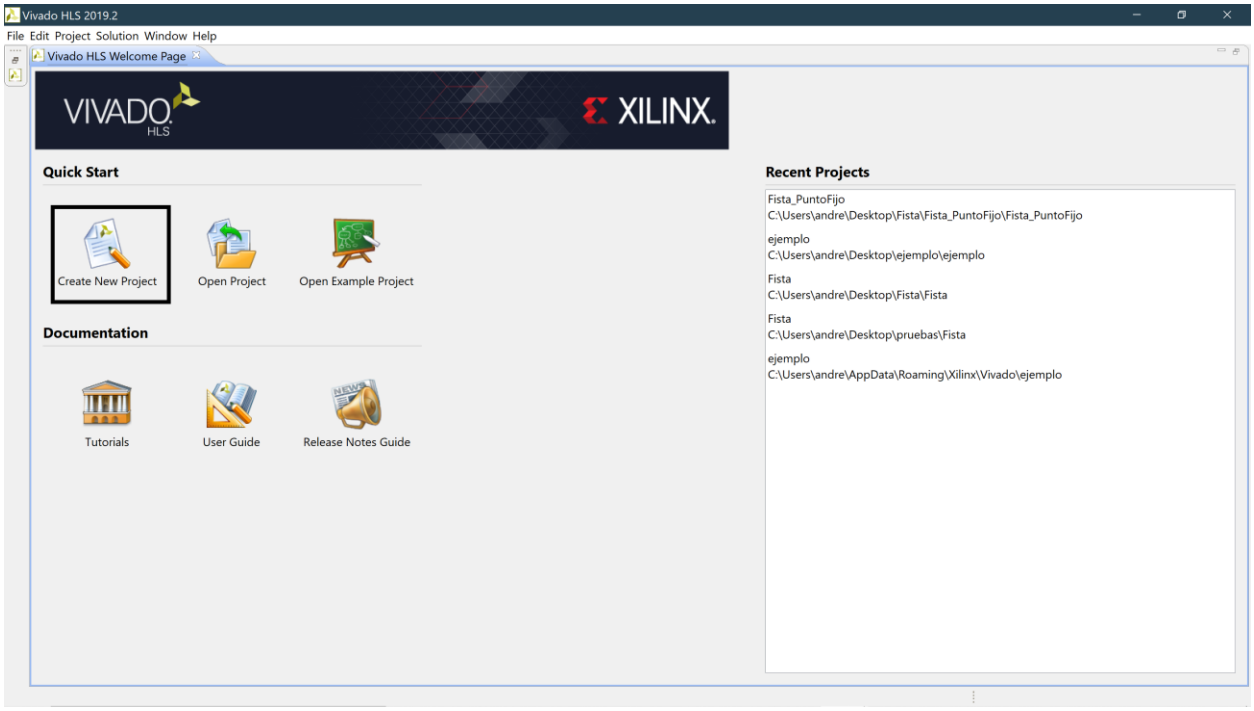

Figura 4-1 Selección de nuevo proyecto

Una vez apretado el botón nos aparecerá una ventana para seleccionar el nombre del proyecto y la localización de este como se muestra en la figura 4-2.

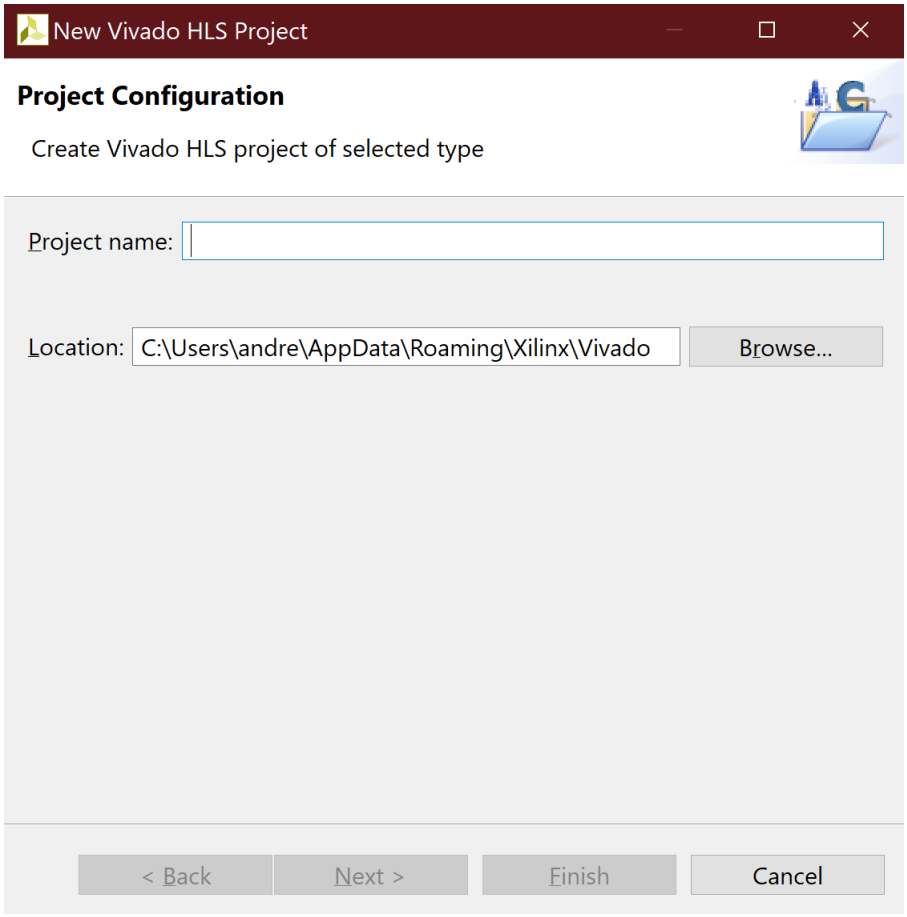

Figura 4-2 Selección de nombre del proyecto y localización

Una vez nombrado y localizado nos preguntara si queremos agregar algún archivo. En esta ventana podríamos añadir nuestros archivos de C, pero se ha optado por no añadir ningún archivo y añadirlos más tarde como se explicará a continuación.

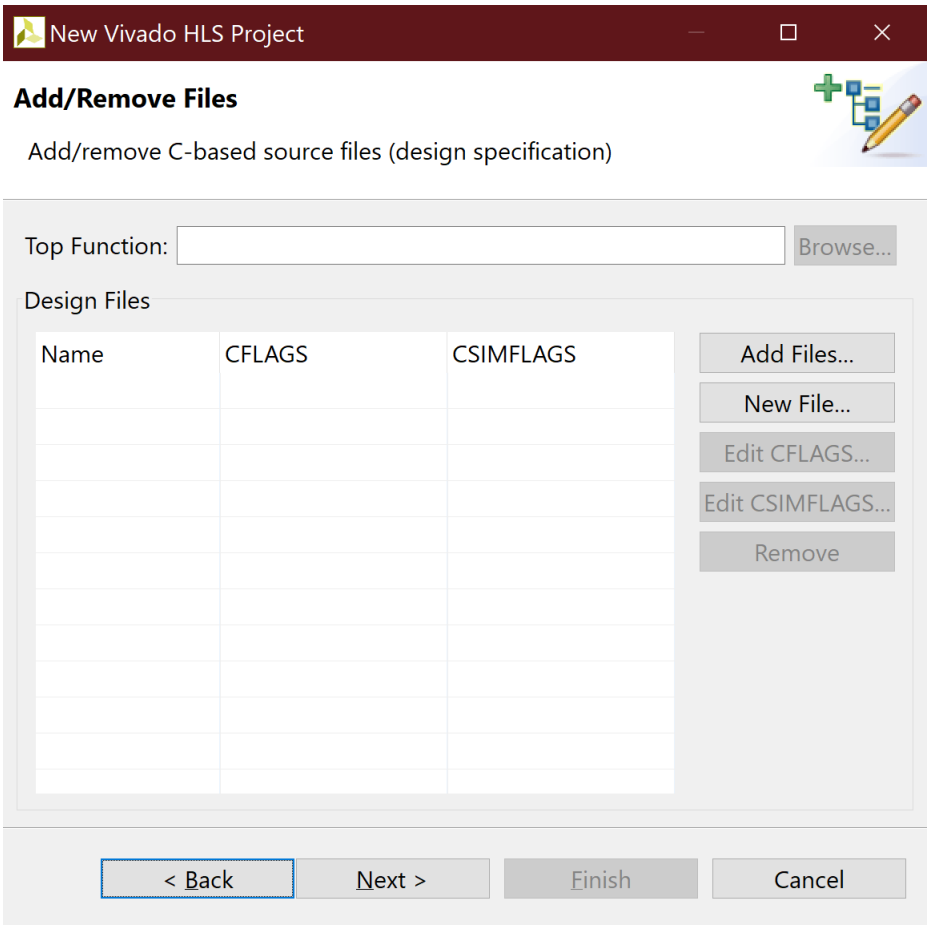

Figura 4-3 añadir ficheros

Ahora nos aparecerá una ventana (figura 4-4) que nos pregunta que nombre queremos darle a la solución de la síntesis de alto nivel y además podremos seleccionar la placa que queramos apretando en el botón de los tres puntos.

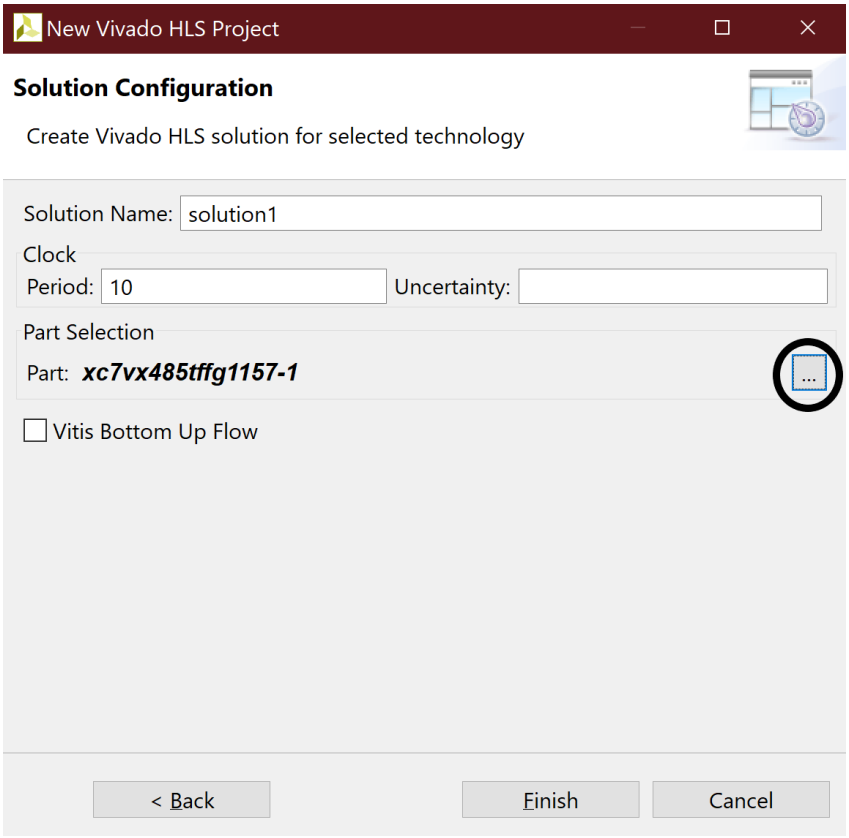

Figura 4-4 selección de nombre de solución HLS

Y procedemos a seleccionar la placa explicada en el apartado 3.3. como se muestra en la figura 4-5.

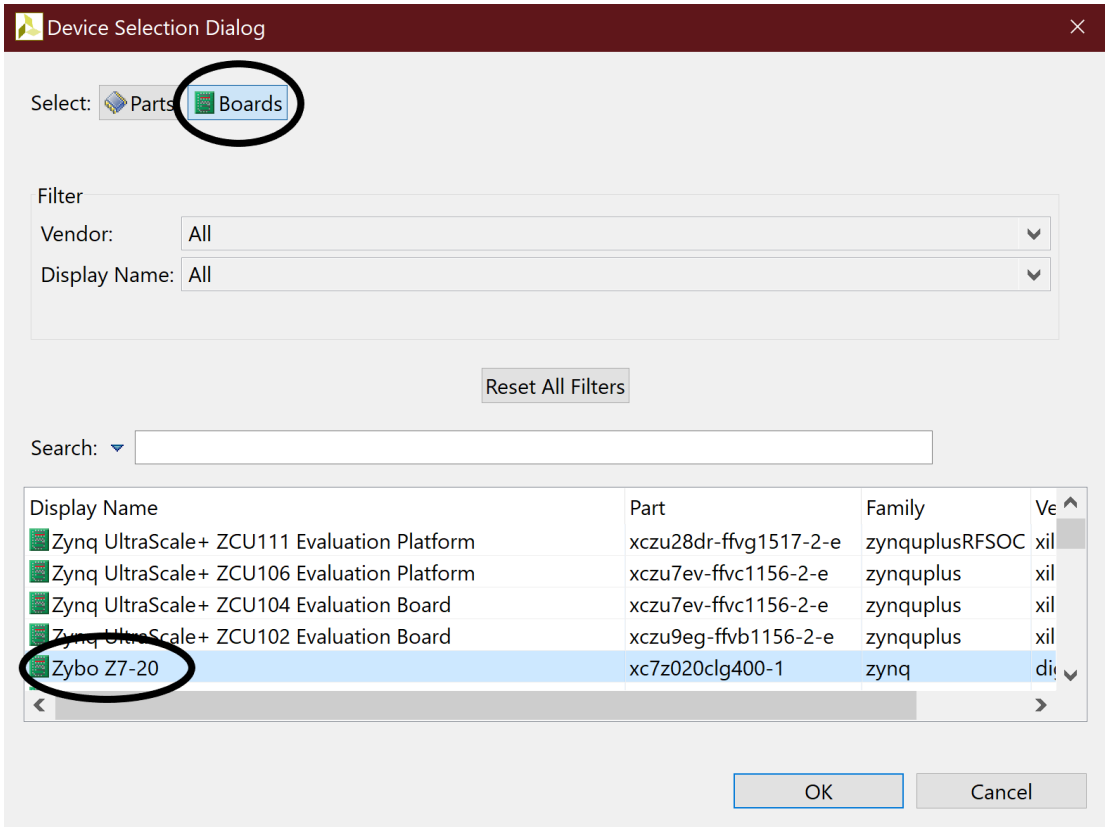

Figura 4-5 Selección de la placa

Una vez seguidos los pasos correctamente nos aparecerá una ventana (figura 4-6).

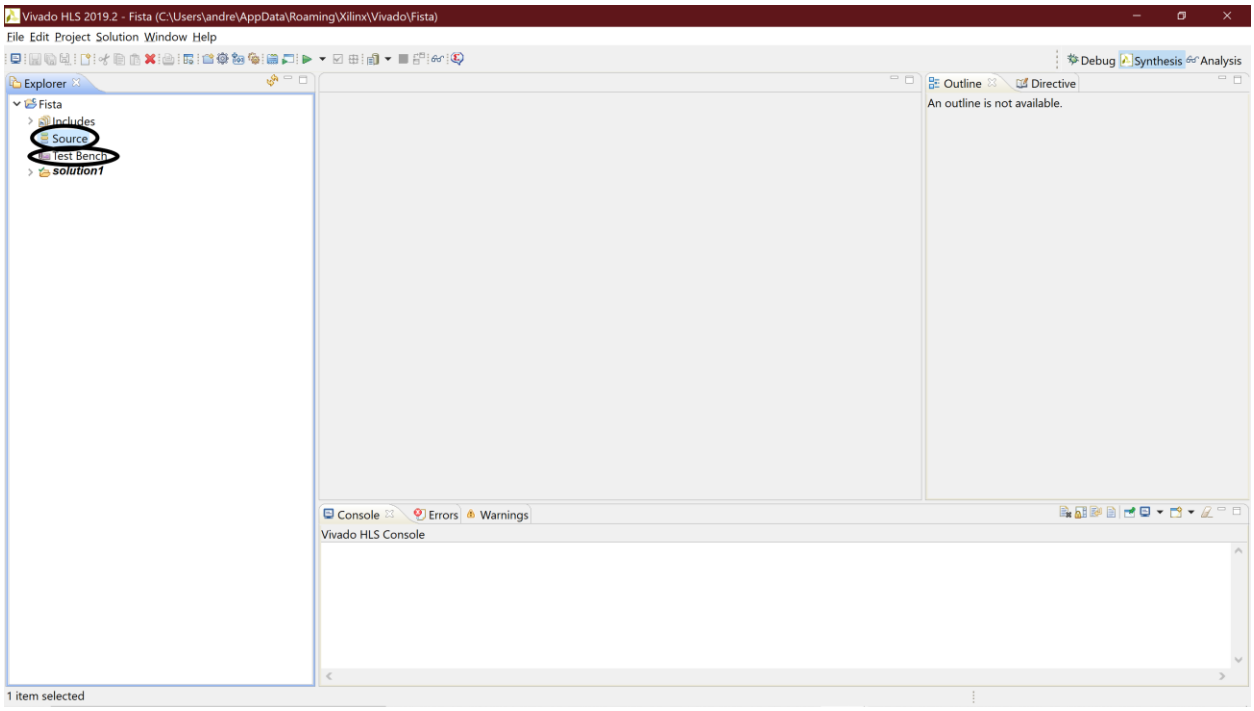

Figura 4-6 Pantalla principal

Ahora procedemos a crear los archivos de programación para ello arriba en la izquierda en "**source**" damos clic derecho con el ratón y seleccionamos "**new file**" y nos preguntará el nombre y la localización del archivo nuevo. Es importante escribir el terminal .cpp después del nombre y guardarlo en el mismo directorio que el proyecto.

Repetimos lo mismo para "**Test Bench**".

Una vez hecho esto hay que explicar la modificación que se han hecho del código en C descrito en el anexo B.

Para poder probar la simulación el código se ha dividido en dos, el apartado **source** contiene el código del algoritmo llamado "Fista.cpp" exactamente igual que en C, el código completo puede consultarse en el anexo C. Las únicas dos diferencias son que el algoritmo FISTA se ha convertido en una función que será llamada en el archivo de **Test Bench**, proporcionándole las entradas descritas en la sección 4.1 añadiendo épsilon y kmax.

```
void fista_puntofijo(fix_t epsilon, int kmax, fix_t G[filas][columnas], fix_t
H[columnas], fix_t UB[filas], fix_t LB[columnas], fix_t b[filas], fix_t
q[columnas], fix_t Wh_inv[filas][filas])
```

y los datos tienen formato de punto fijo.

4.2.1.1 Punto Fijo

Las FPGAs utilizan el formato de punto fijo porque reduce el nivel de carga computacional respecto a otros formatos, como por ejemplo el punto flotante. En la figura 4-7 se puede ver cómo está formado un número en punto fijo

Una palabra digital L bits se particiona en:

L* = S + *QE* + *QF

QE : bits para la parte entera

QF = Qi : bits para la parte fraccionaria

S : un bit de signo, 0 positivo, 1, negativo

Un número *X* positivo se puede representar en formato de punto entero como:

$$X=\sum_{i=-QE}^{QF} b_i r^{-i}=\left(b_{-QE},\ldots,b_{-1},b_0\ldots,b_{QF}\right) \quad 0\le b_i\le\left(r-1\right)$$

Figura 4-7 Formato punto fijo

Es decir, el formato en punto fijo requiere elegir el número de bits de la parte entera de forma que no se produzca desbordamiento en las operaciones, mientras que el número de bits de la parte decimal nos va a especificar la precisión de las cifras decimales, cuanto mayor sea el número de bits mayor precisión

El archivo de **Test Bench** llamado “Fista_TB.cpp” contiene las entradas del algoritmo y una función main que llama a la función de “Fista.cpp”, el código completo está en el Anexo D.

Una vez tengamos preparados los archivos de **Source** y **Test Bench** podemos proceder a simularlos en el procesador apretando el botón “run C simulation” como se muestra en la figura 4-8.

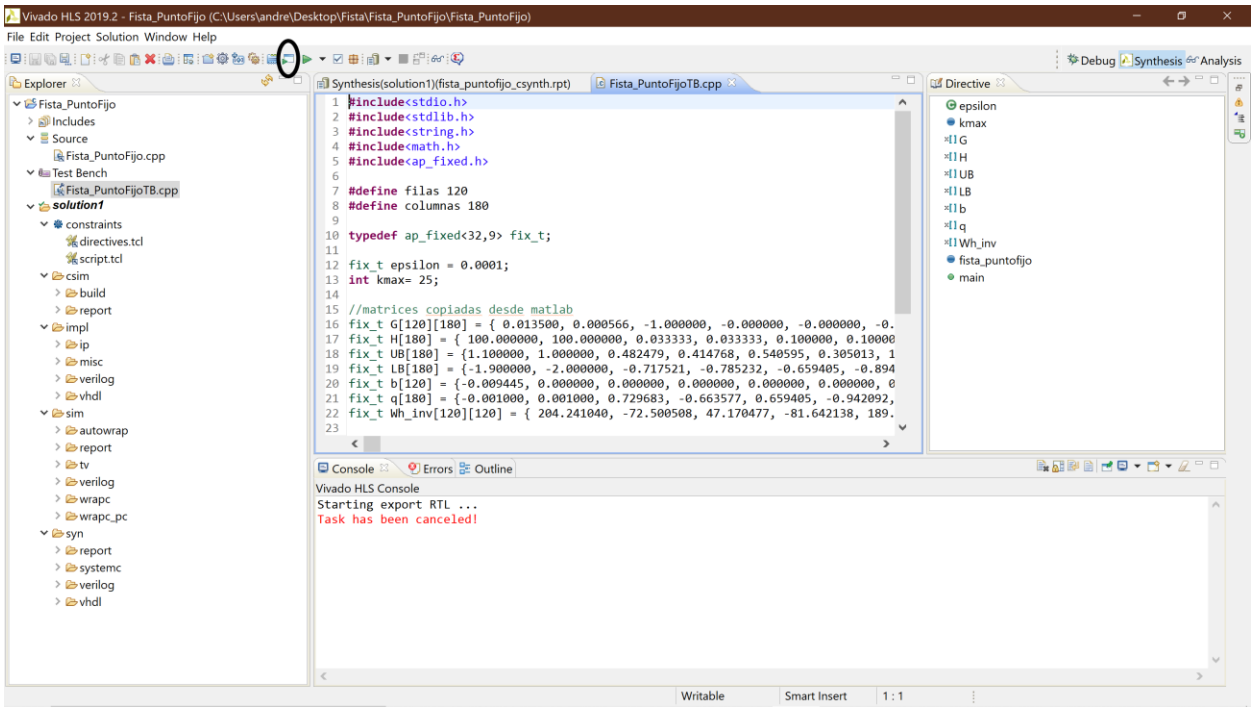

Figura 4-8 simulación en C

Nos aparecerá una ventana (figura 4-9), le damos a “ok” y nos simulará el código en C.

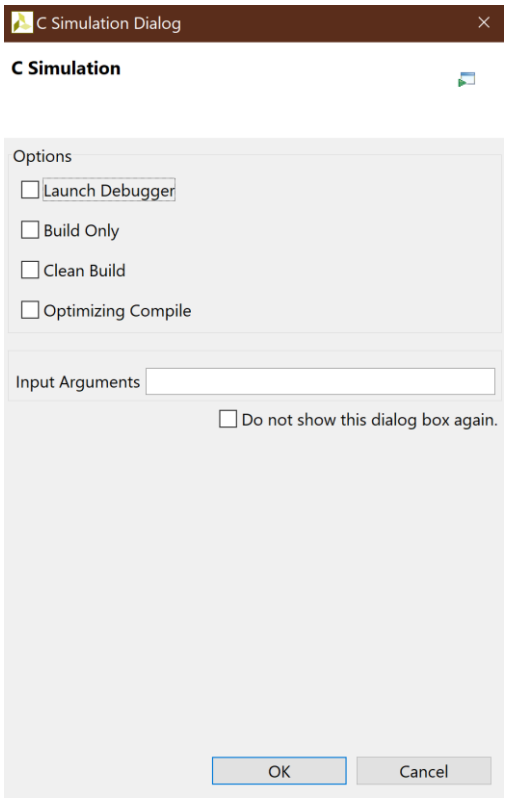

Figura 4-9 simulación en C

Una vez terminada nos aparecerá una ventana como en la pantalla principal, tal y como se muestra en la figura

4-10. Con los resultados de la simulación.

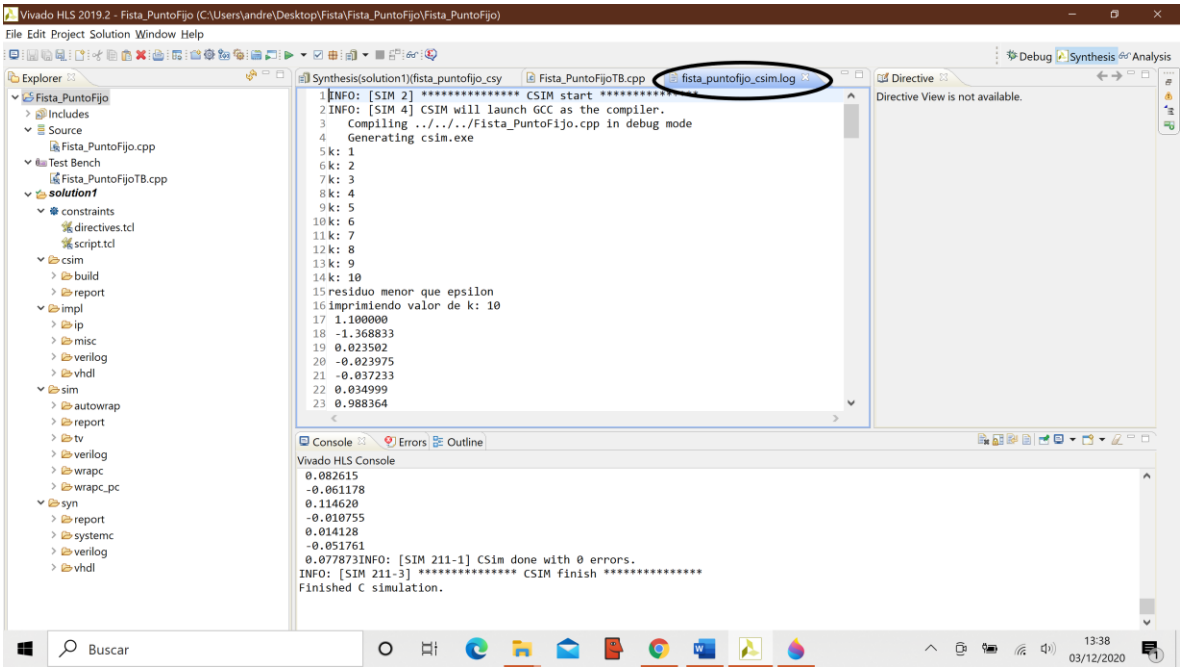

Figura 4-10 resultados de la simulación en C

4.2.2 Cosimulación C/RTL

Una vez que nos aseguramos que los resultados de la simulación en C son satisfactorios procedemos a realizar la cosimulación en C/RTL. Es decir, el programa se encarga de comparar el código en C con el diseño RTL.

Para ello debemos pulsar el icono “C/RTL cosimulation” que se muestra en la figura 4-11.

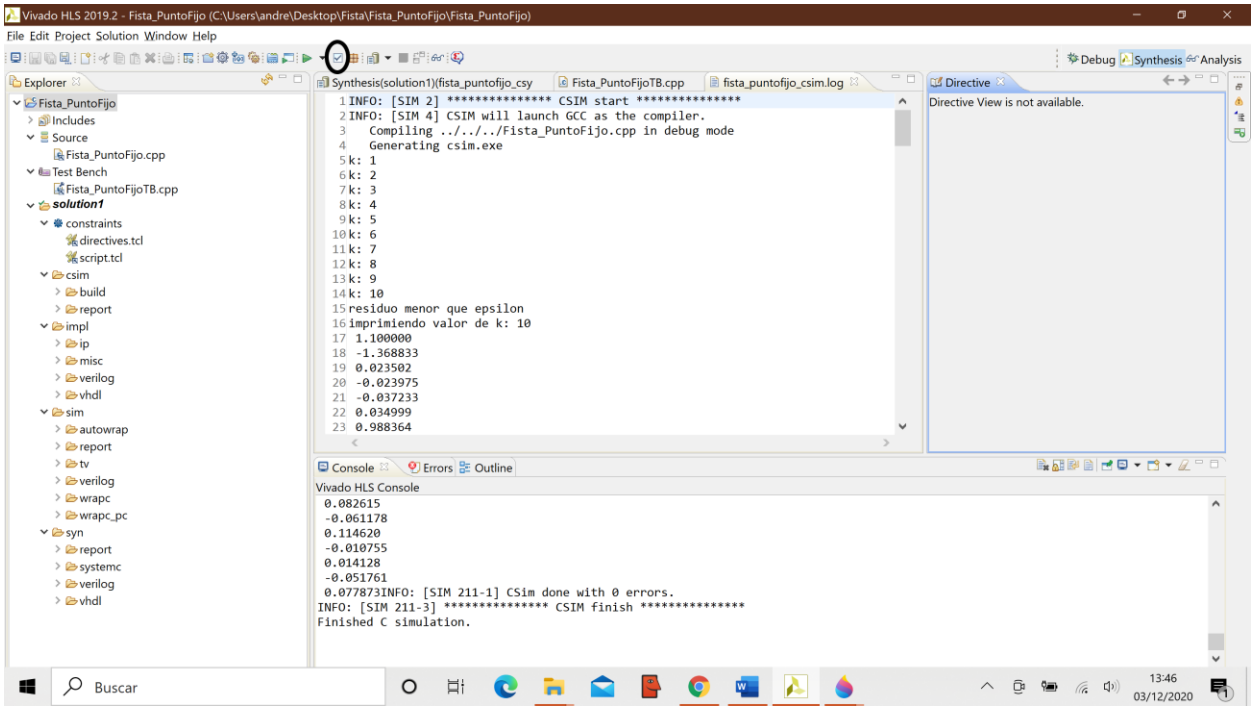

Figura 4-11 Cosimulación C/RTL

Una vez apretado el icono nos aparecerá una ventana (figura 4-12). Debemos seleccionar “vivado simulator” y el lenguaje de programación, verilog en este caso.

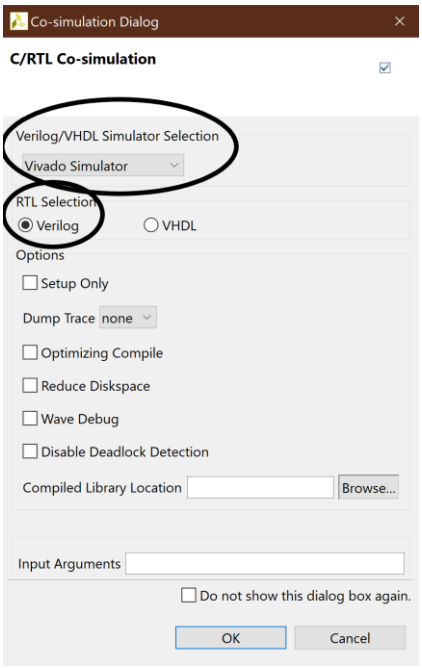

Figura 4-12 Selección de características de la cosimulación

Por último, una vez finalizada la cosimulación se nos abrirá una pestaña en el menú principal con la solución de la cosimulación tal como se muestra en la figura 4-13.

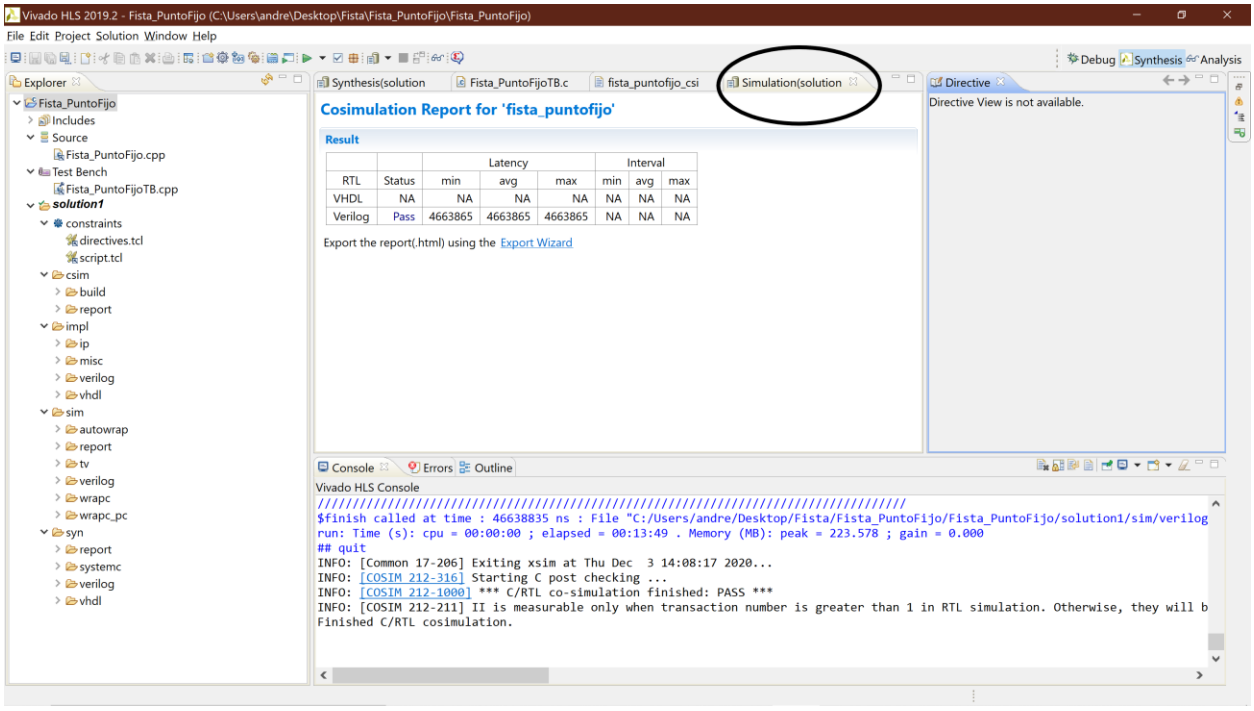

Figura 4-13 Solución de la cosimulación

4.2.3 Síntesis en C y exportación RTL

Una vez finalizada la cosimulación si el resultado ha sido el mismo podremos sintetizar el código y después exportarlo para crear el módulo IP.

Para ello primeramente debemos configurar la función y las variables de entrada como “s_axilite”.

Para ello en la ventana principal en la pestaña **directive** debemos hacer clic derecho con el ratón y seleccionar "insert directive" tal como se muestra en la figura 4-14.

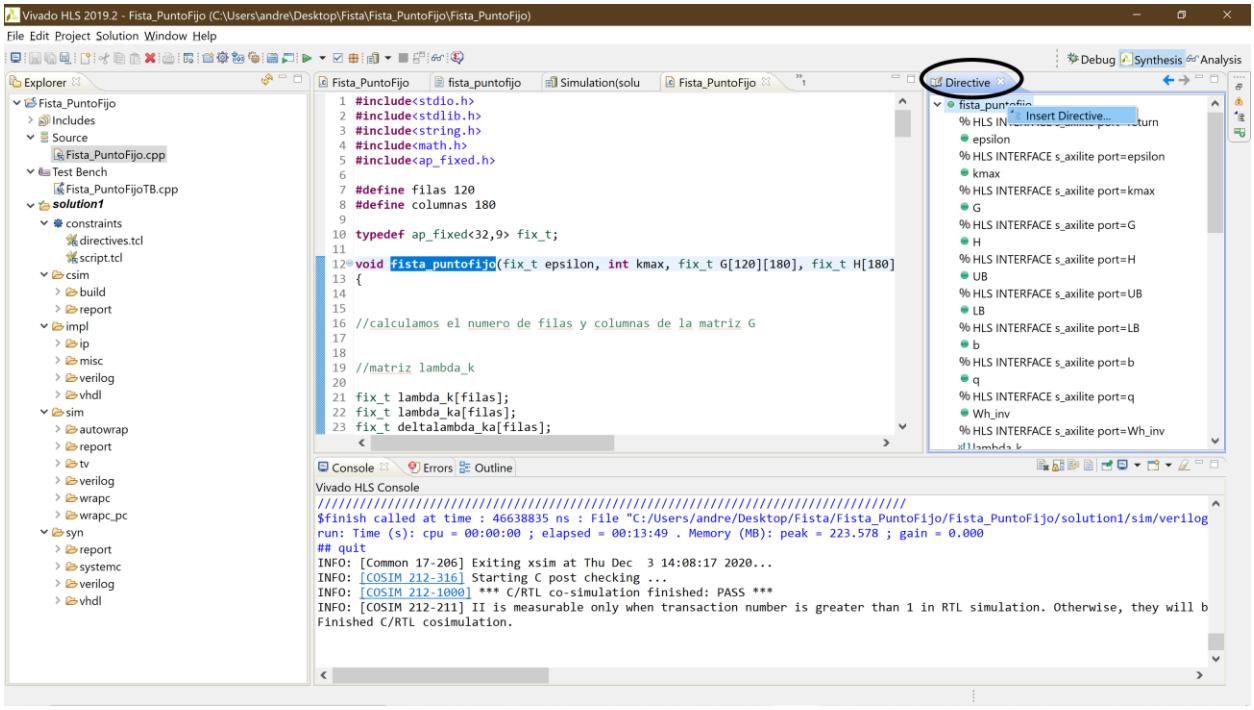

Figura 4-14 Insertar directiva

Una vez hecho se nos abrirá una ventana (figura 4-15) debemos seleccionar "INTERFACE" y "s_axilite".

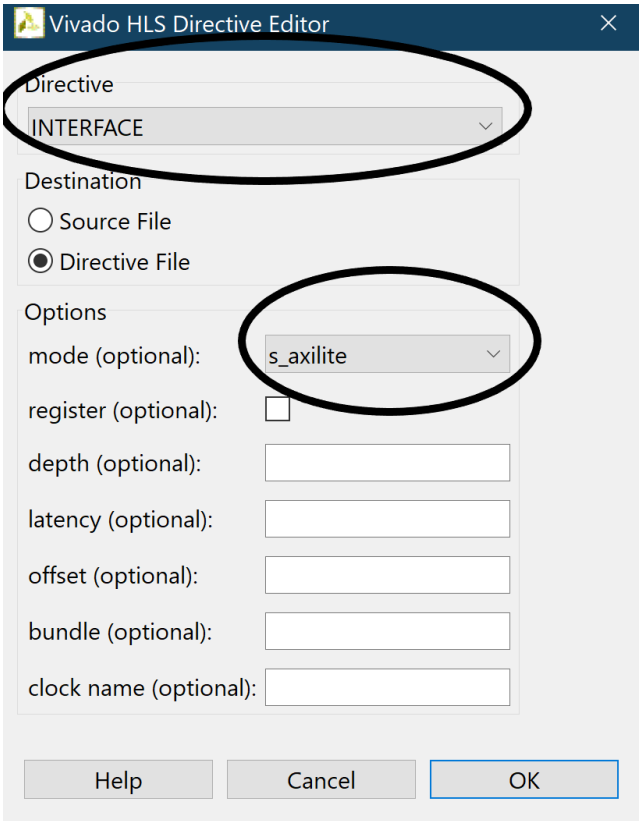

Figura 4-15 Selección de Modo

Este paso se debe repetir para cada entrada de la función.

Una vez hecho debemos asegurarnos de que se sintetice la función deseada, para ello en la ventana principal

debemos pulsar el icono llamado “Project settings” como se muestra en la figura 4-16.

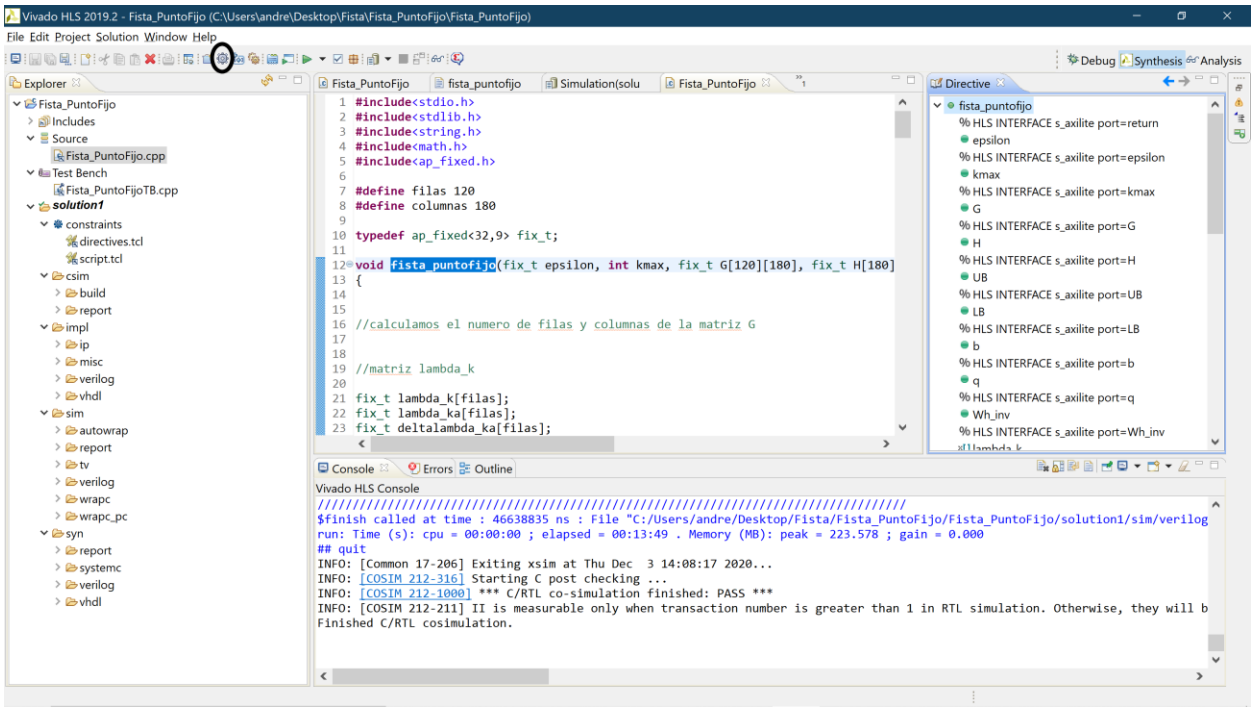

Figura 4-16 selección ajustes

En la ventana emergente en el apartado **synthesis** debemos asegurarnos que en **top Function** esté seleccionada el nombre de nuestra función, en este caso “fista_puntofijo” como se indica en la figura 4-17.

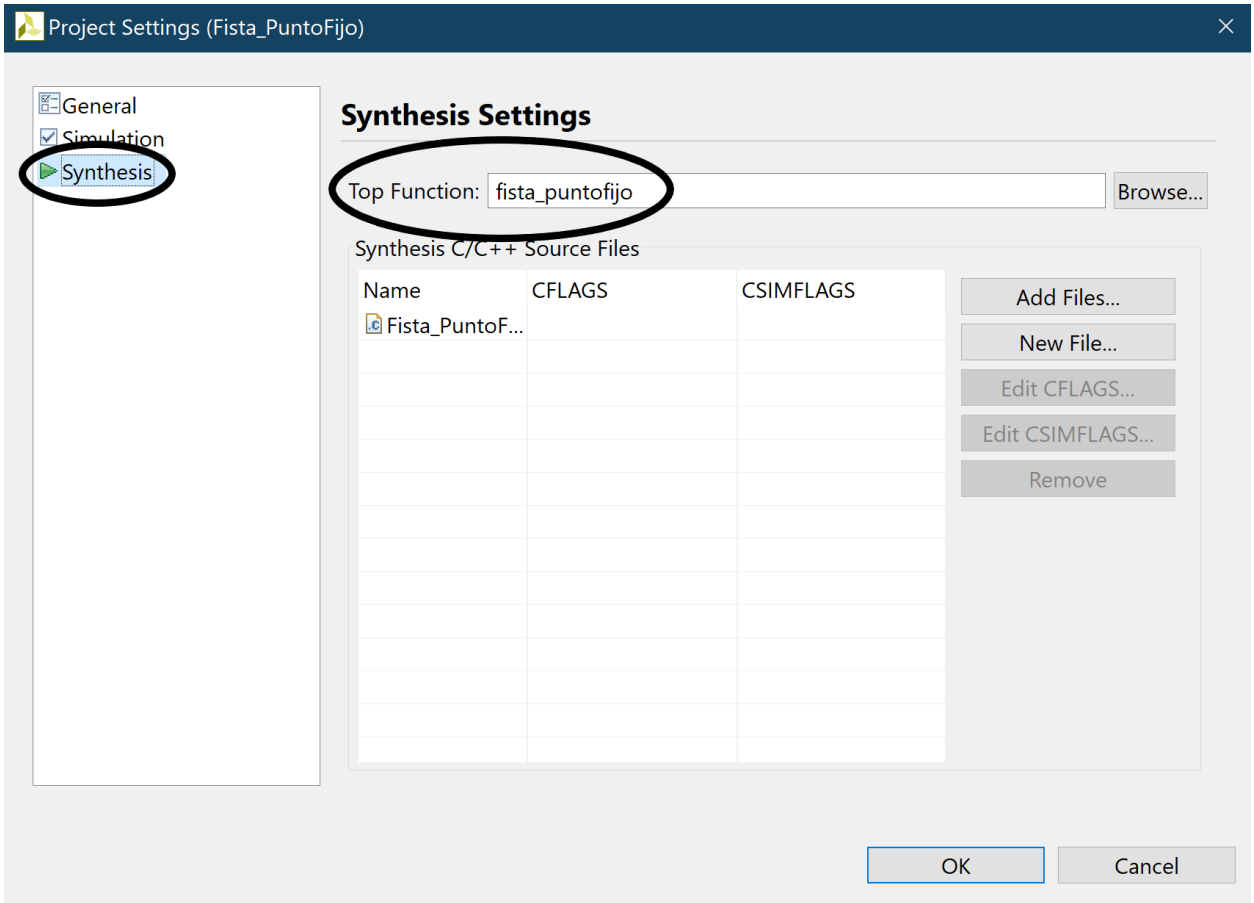

Figura 4-17 Selección de la función

Una vez realizada esta tarea podemos seleccionar el botón llamado “Run C Synthesis” en la pantalla principal tal como se muestra en la figura 4-18.

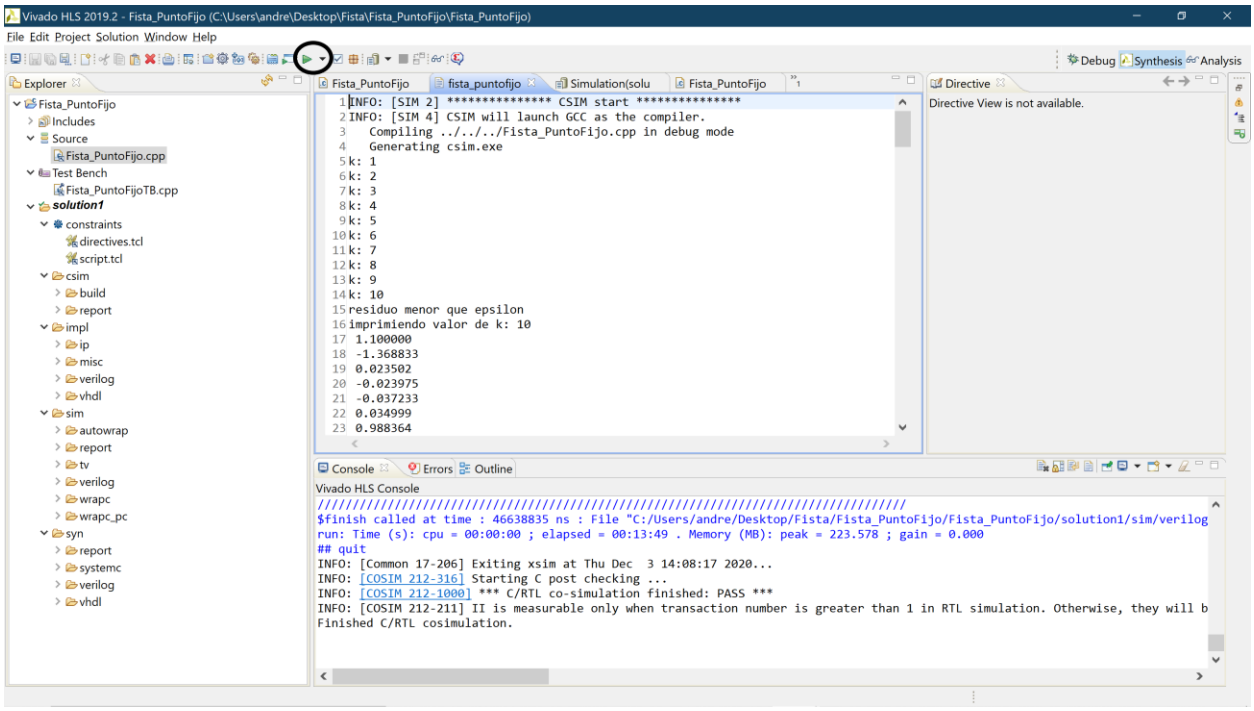

Figura 4-18 Ejecutar Síntesis

Al terminar la síntesis se nos abrirá una ventana con información de los resultados de la síntesis, que se comentarán en el capítulo 5-resultados, y a la izquierda la solución de la síntesis que contiene los códigos en verilog.

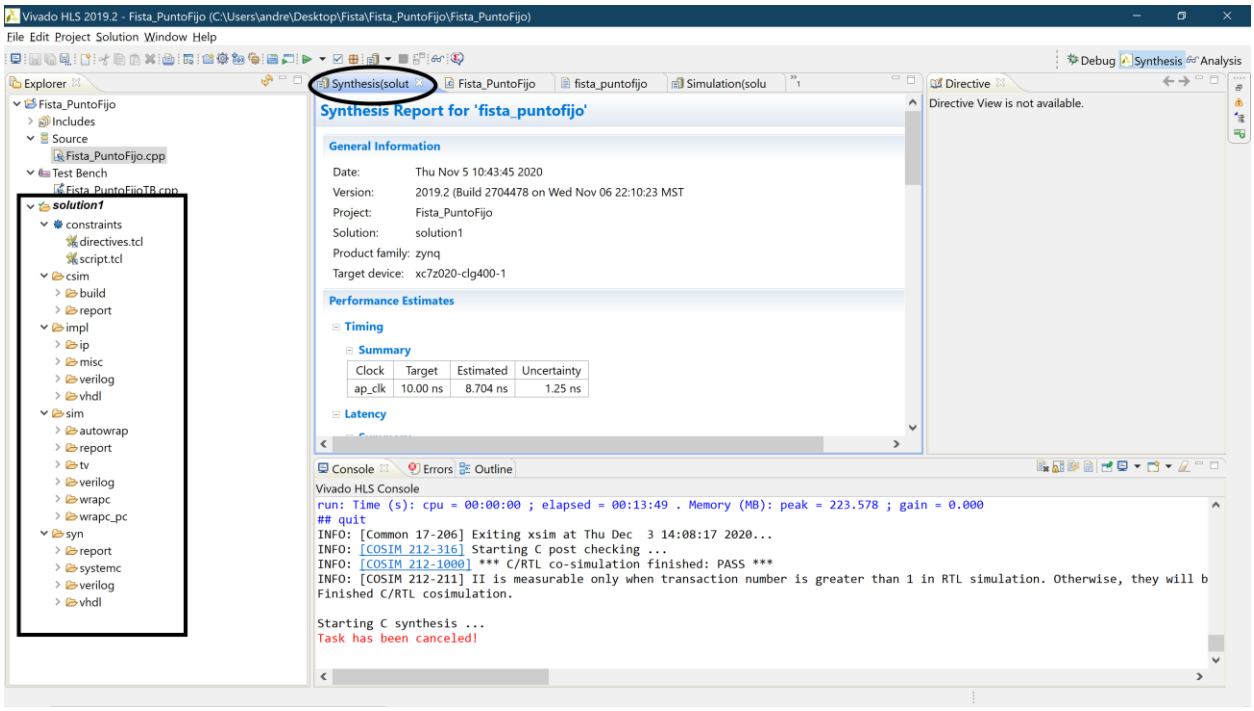

Figura 4-19 Resultados de la síntesis

Finalmente, si la simulación ha resultado exitosa podemos proceder a exportar el algoritmo y crear el módulo

IP, para ello vamos al icono llamada “export RTL” en el menú principal. Como se muestra en la figura 4-20.

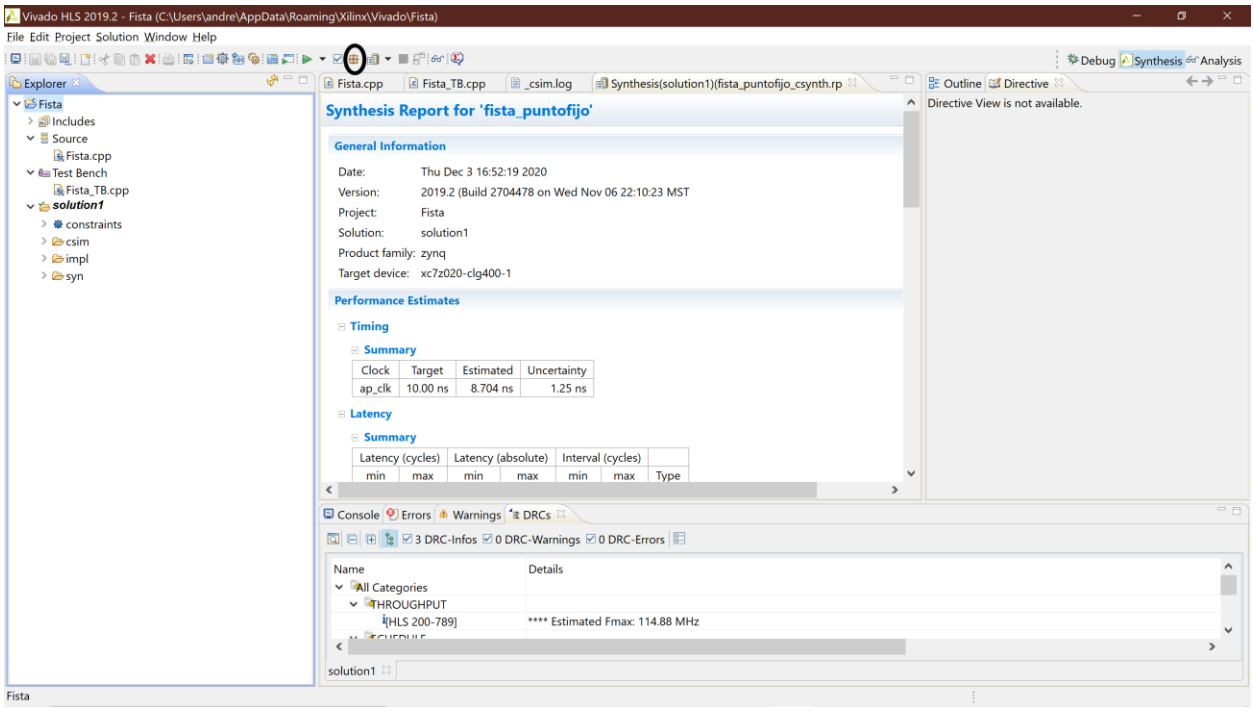

Figura 4-20 Exportación RTL

Se nos abrirá una ventana (figura 4-21) y debemos seleccionar IP Catalog y el lenguaje verilog

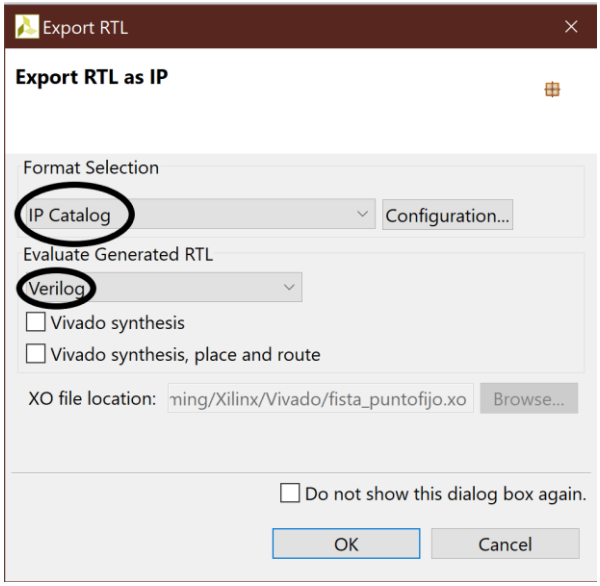

Figura 4-21 Selección parámetros exportación

Y si se han realizado todos los pasos correctamente ya tenemos nuestro módulo IP lista para implementar en la placa.

4.3. Implementación del módulo IP en la placa y generación del Bitsream

Primero de todo nos dirigimos a la aplicación Vivado en este caso la versión Vivado 2019.2 y creamos un nuevo proyecto como en la figura 4-22.

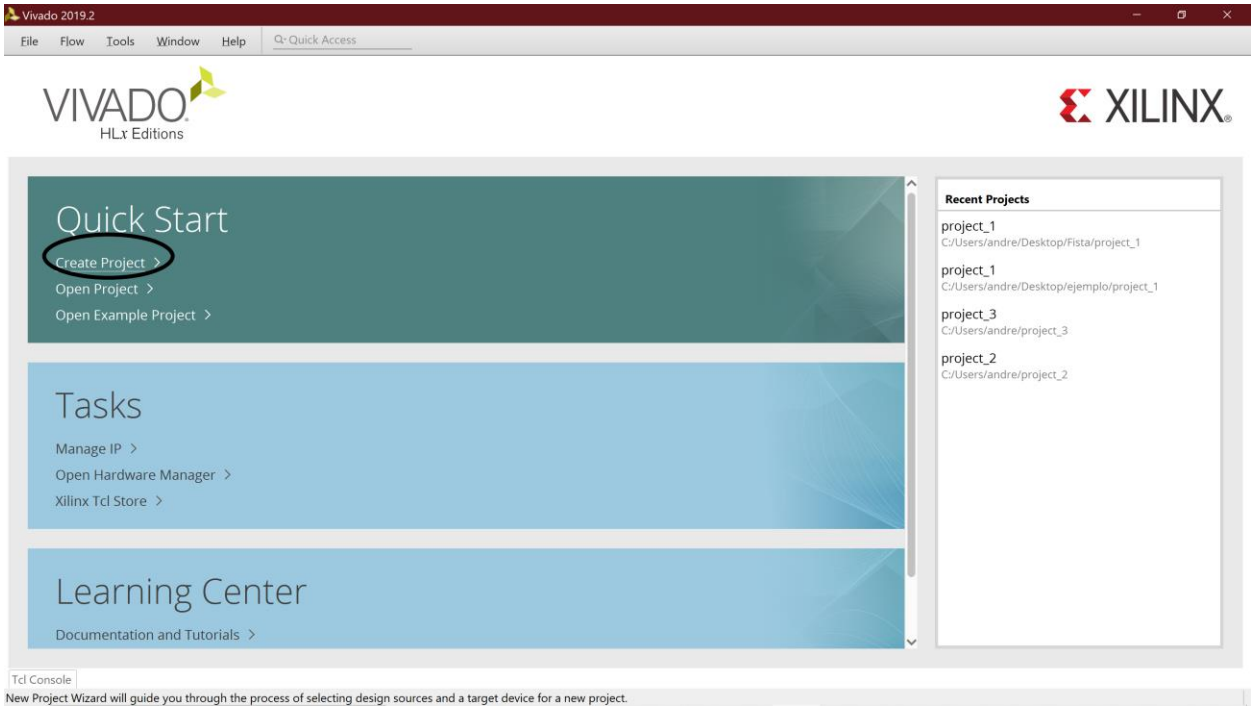

Figura 4-22 Creación de un nuevo proyecto

Una vez hecho nos aparecerá una pantalla de presentación, pulsamos en el botón “next” y nos aparecerá una pantalla (figura 4-23) donde debemos nombrar y localizar nuestro nuevo proyecto.

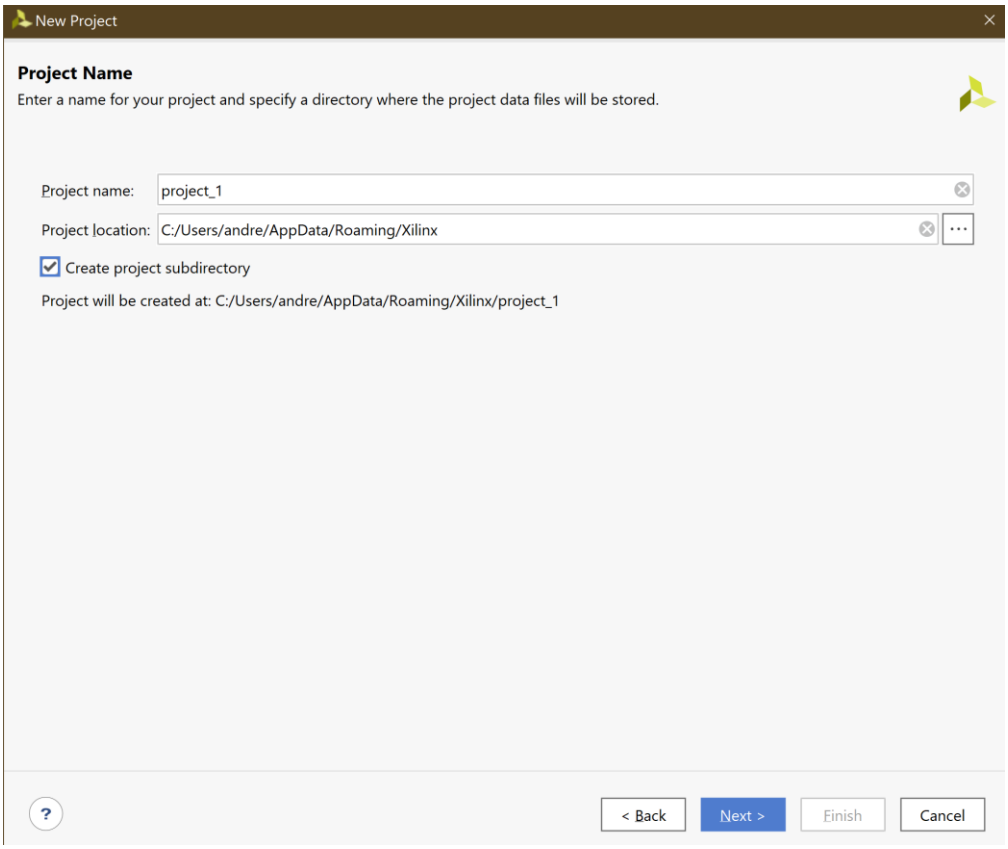

Figura 4-23 selección de nombre y ruta del proyecto

Apretamos “Next” y nos aparecerá una nueva ventana en la que debemos seleccionar la primera opción RTL_proyect (figura 4-24).

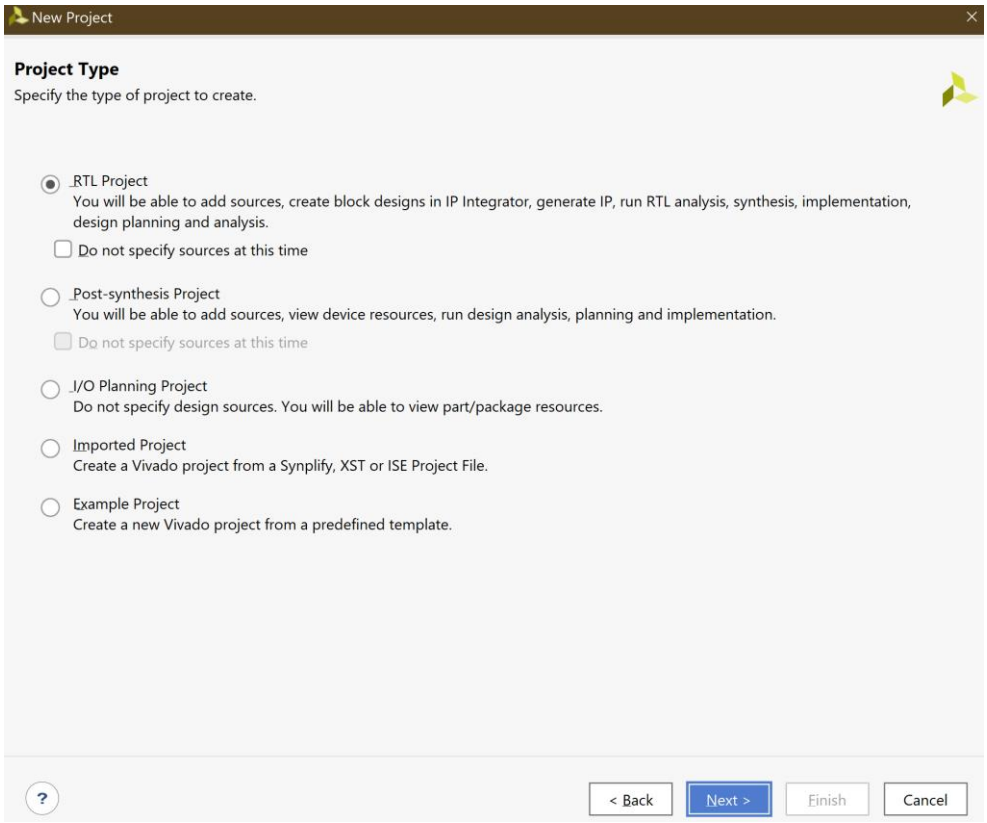

Figura 4-24 Selección RTL_project

Apretamos “Next” en las dos siguientes ventanas porque no queremos añadir fuentes ni restricciones (figura 4-25).

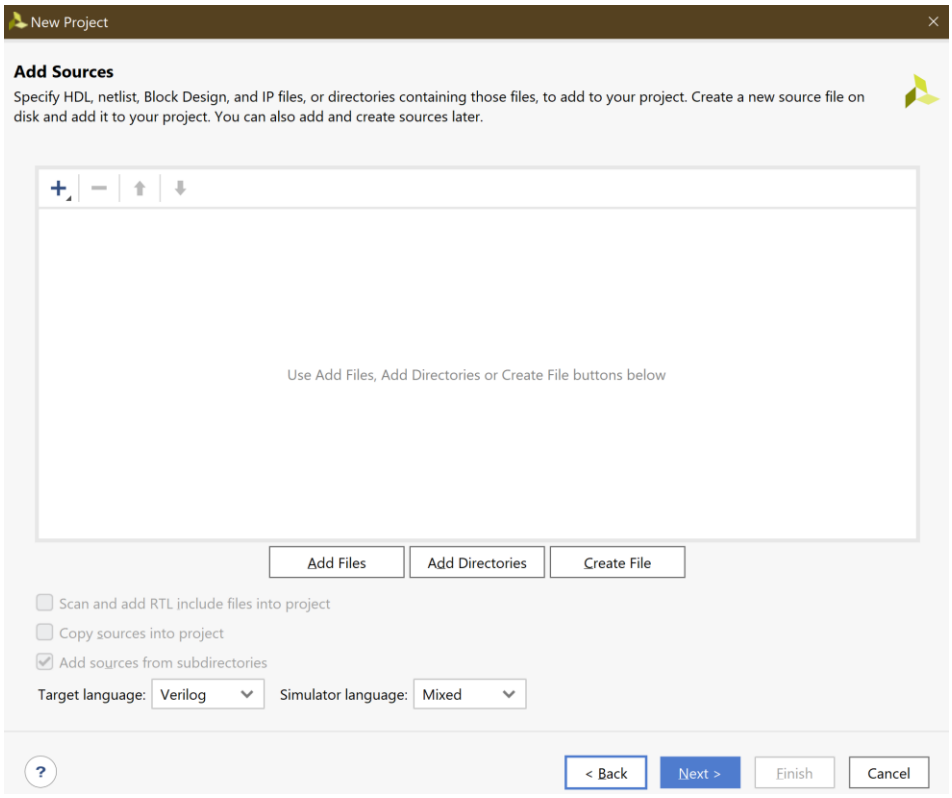

Figura 4-25 Selección de fuentes y restricciones

Y finalmente tal y como se hizo en Vivado HLS seleccionamos la placa **zybo z20** y apretamos “Next” (figura 4-26).

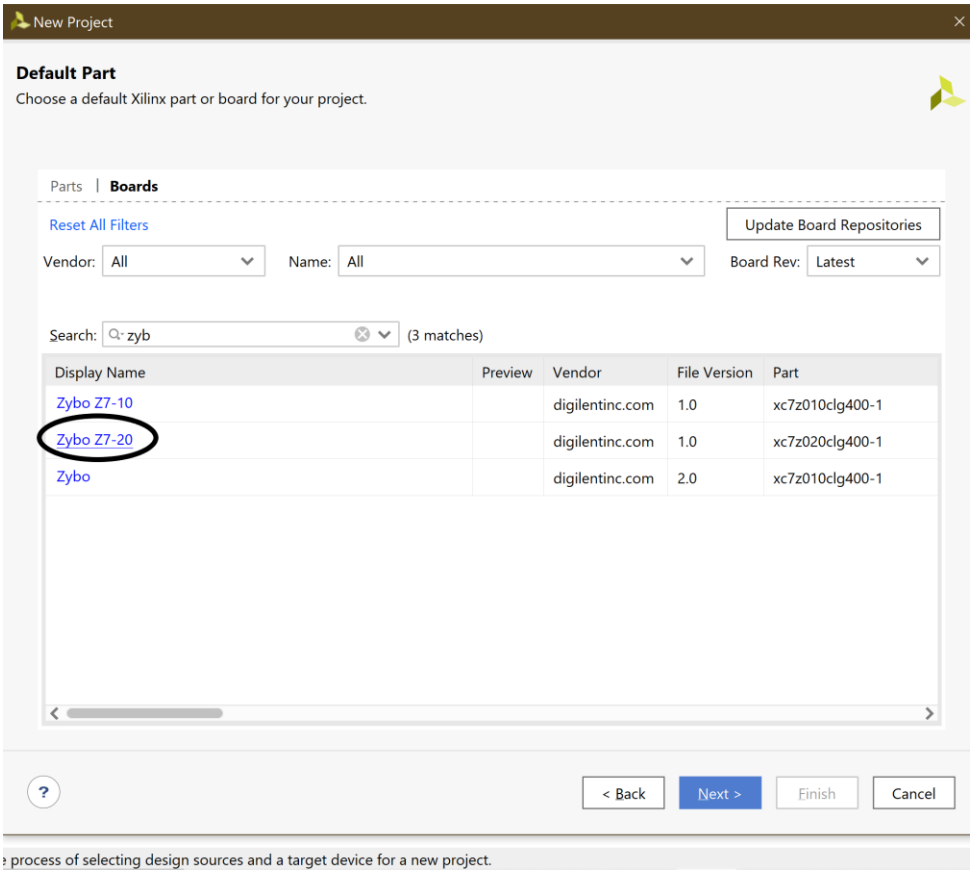

Figura 4-26 selección de la placa

Nos aparecerá una ventana con un resumen y si todo es correcto apretamos en “Finish” (figura 4-27).

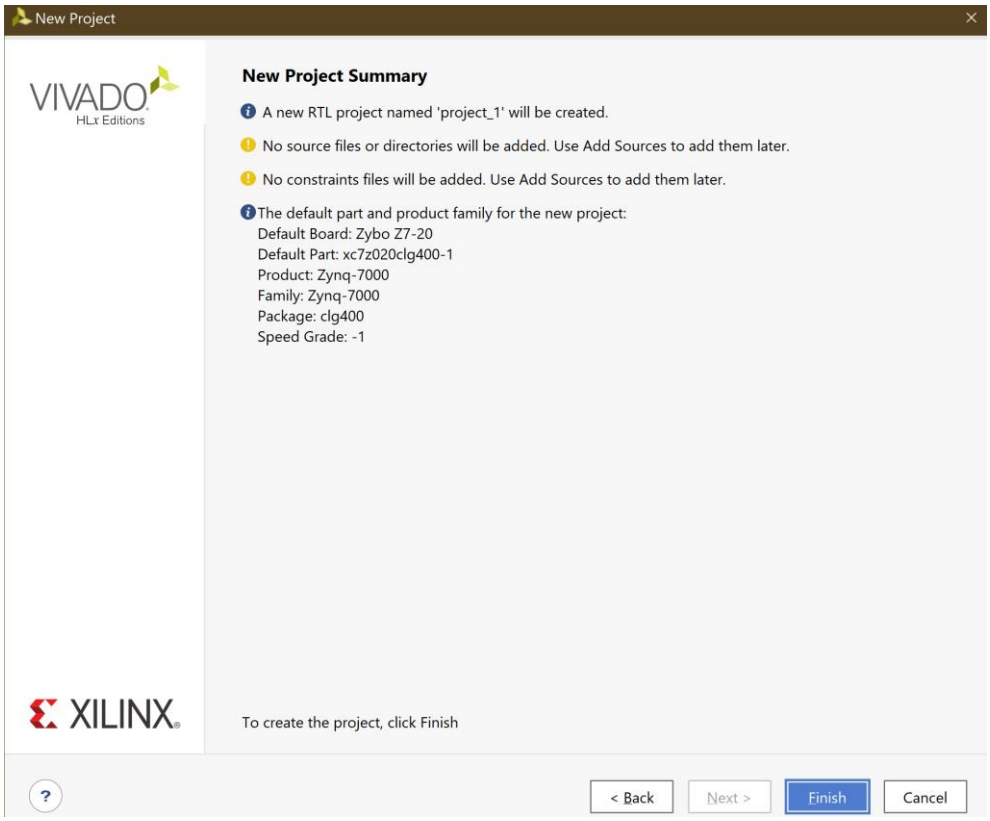

Figura 4-27 Resumen de la creación del proyecto

Finalmente, nos aparecerá una ventana principal (figura 4-28).

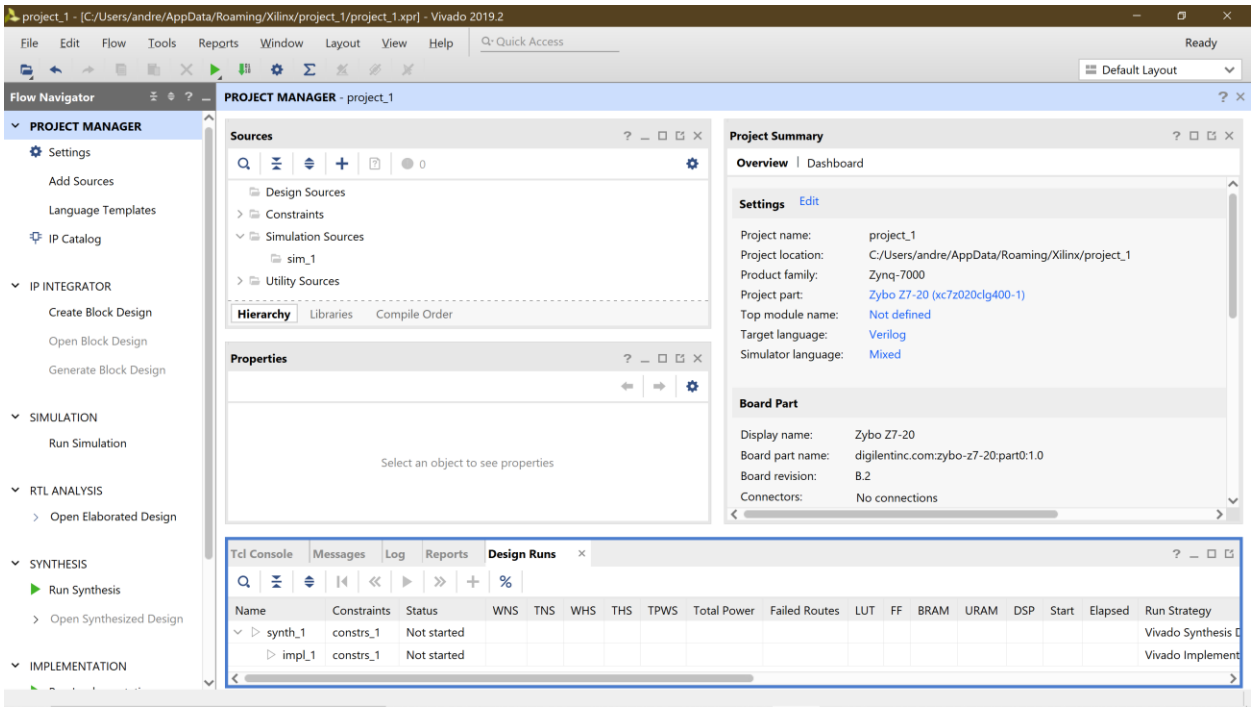

Figura 4-28 Menú principal de Vivado

Ahora procedemos a implementar el módulo IP que creamos anteriormente, primero como se muestra en la figura 4-29 seleccionamos Create block Design.

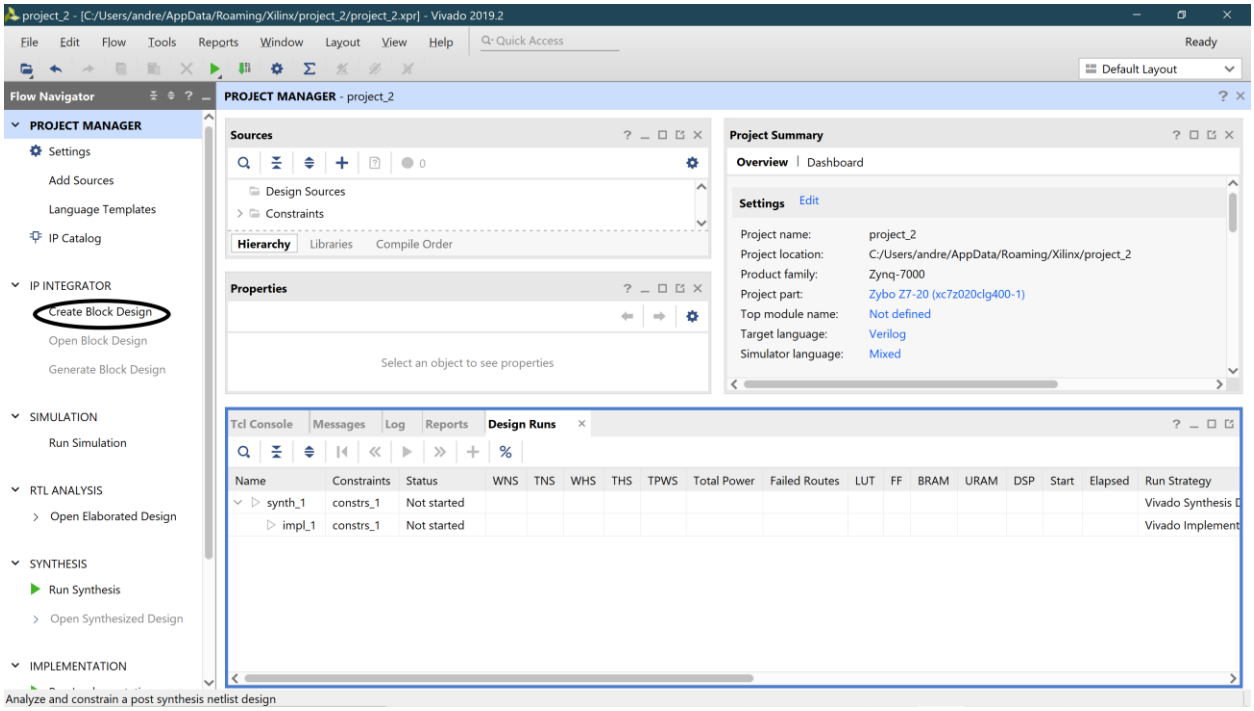

Figura 4-29 creación de nuevo bloque

Nos aparecerá una ventana emergente y apretamos “Ok”

Y en la ventana de la derecha damos clic derecho con el ratón a “Add Repositoy” (figura 4-30).

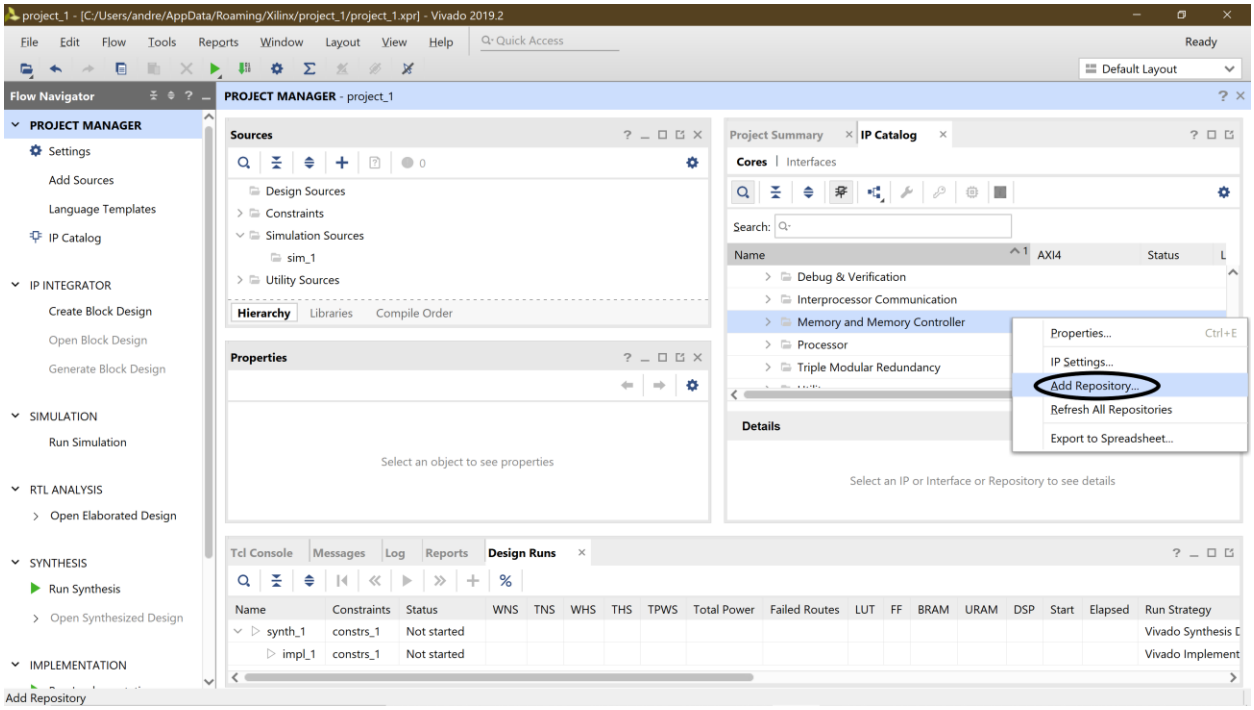

Figura 4-30 Añadir repositorio

Seleccionamos la carpeta donde guardamos la exportación realizada en Vivado HLS (figura 4-31).

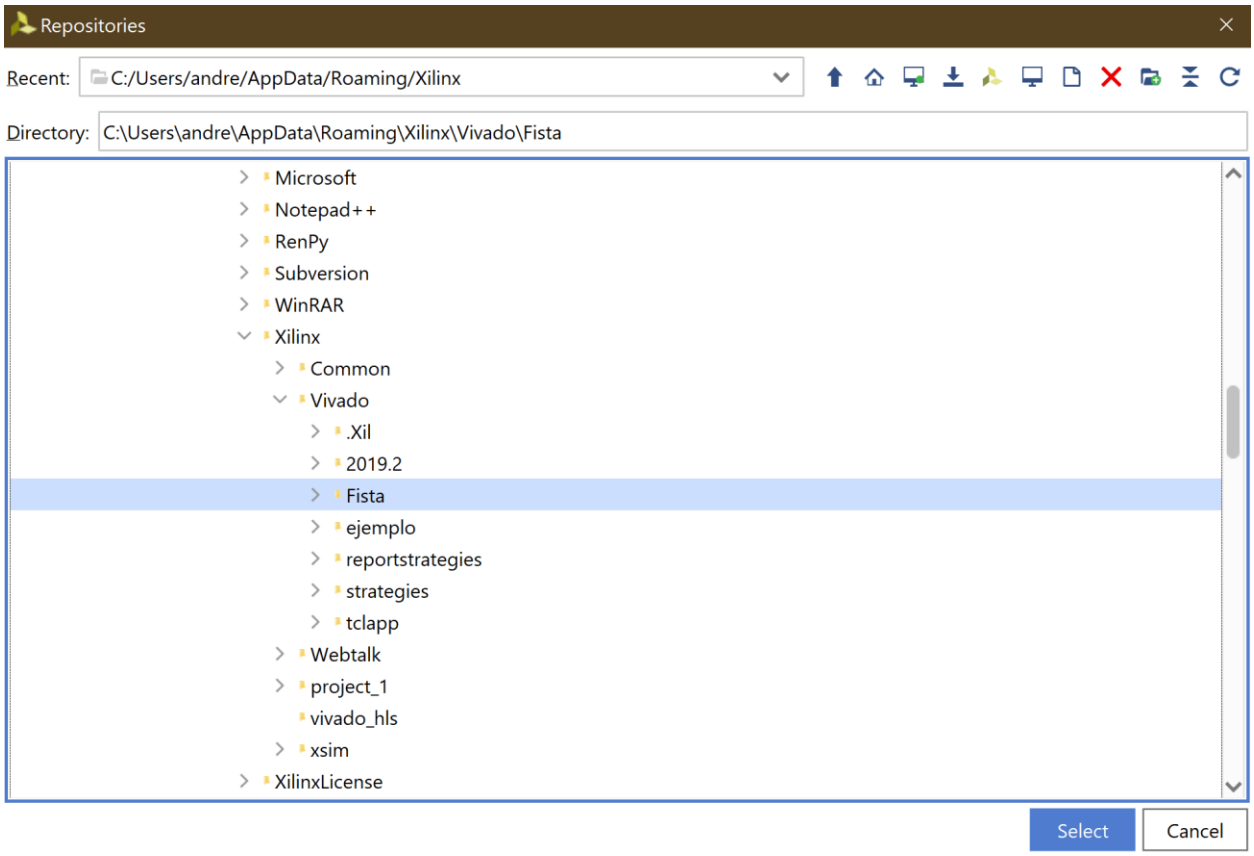

Figura 4-31 Selección de carpeta

Y el programa nos debería detectar automáticamente el repositorio como se muestra en la figura 4-32.

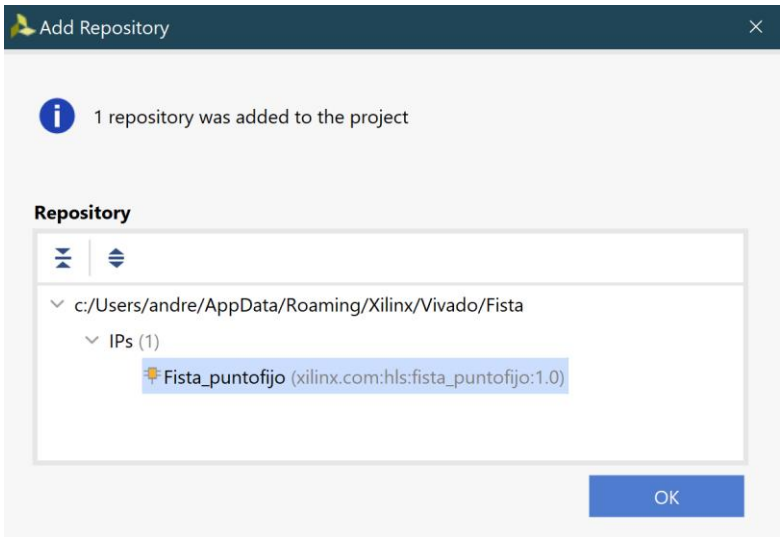

Figura 4-32 Localización del repositorio

Ahora ya podemos buscar y seleccionar nuestro IP como en la figura 4-33.

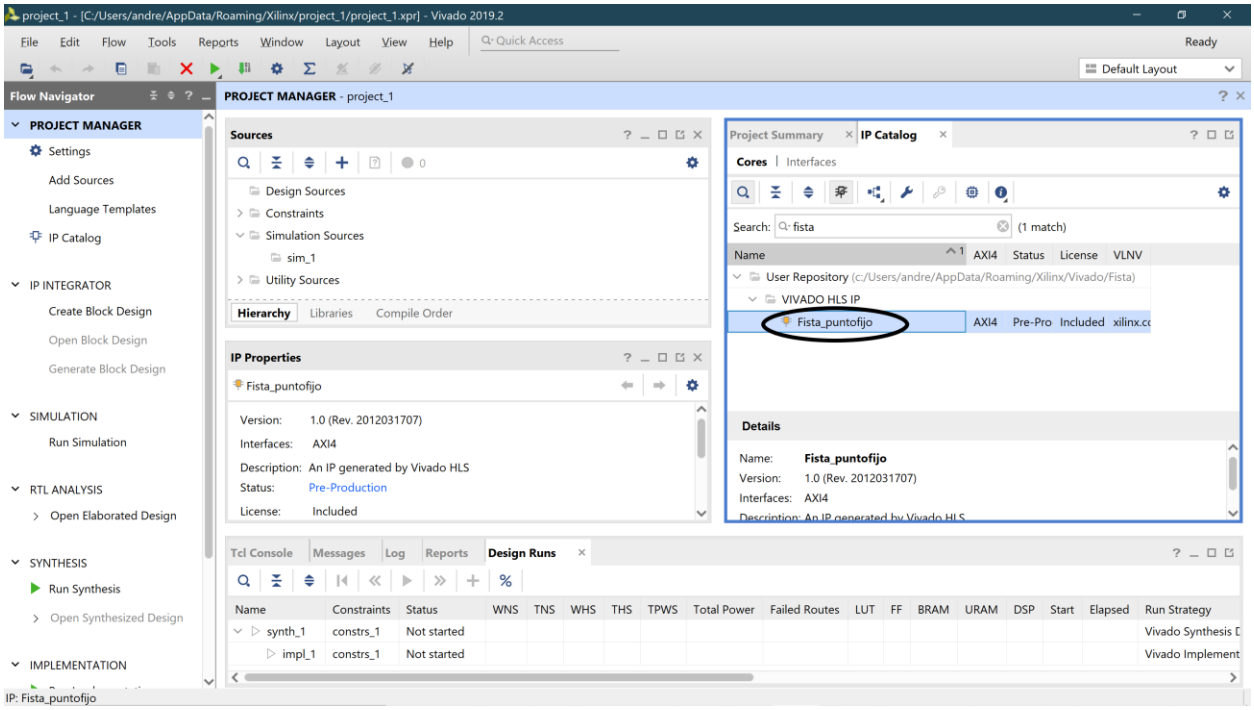

Figura 4-33 selección del módulo IP

Ahora en la ventana **diagram** ya podemos agregar nuestro módulo IP (figura 4-34).

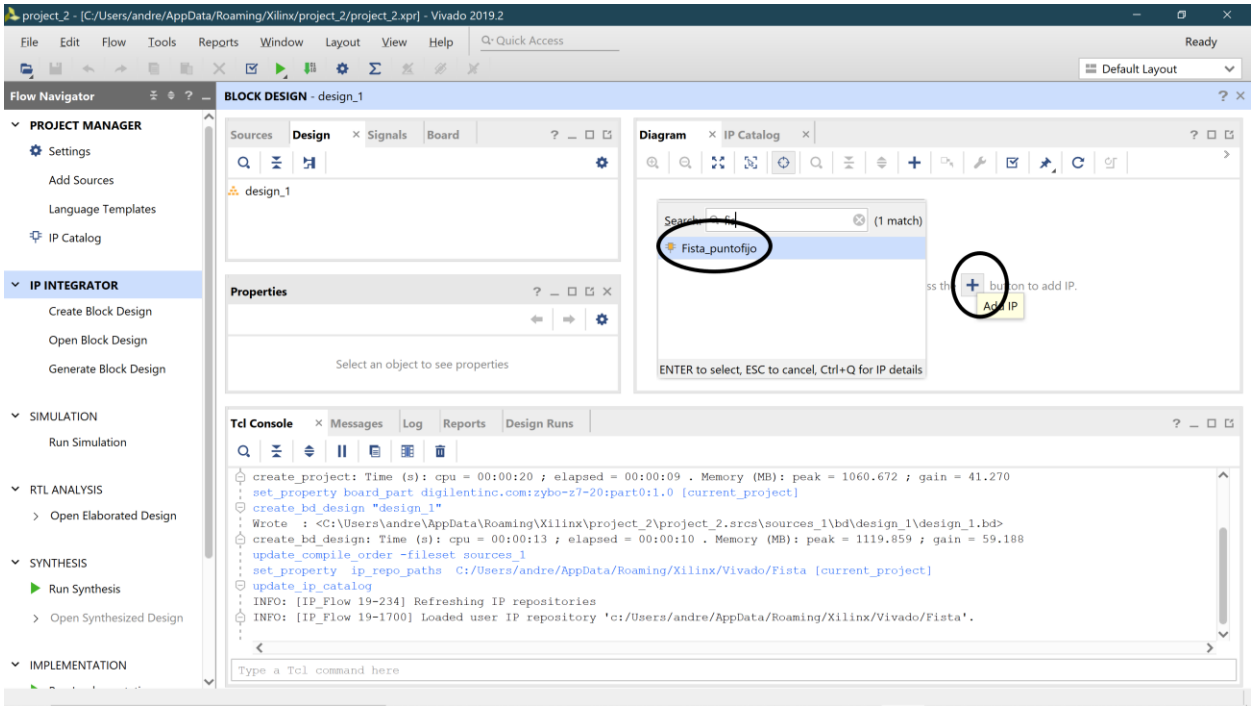

Figura 4-34 Añadir módulo IP

Ahora ya tenemos el módulo IP agregado (figura 4-35).

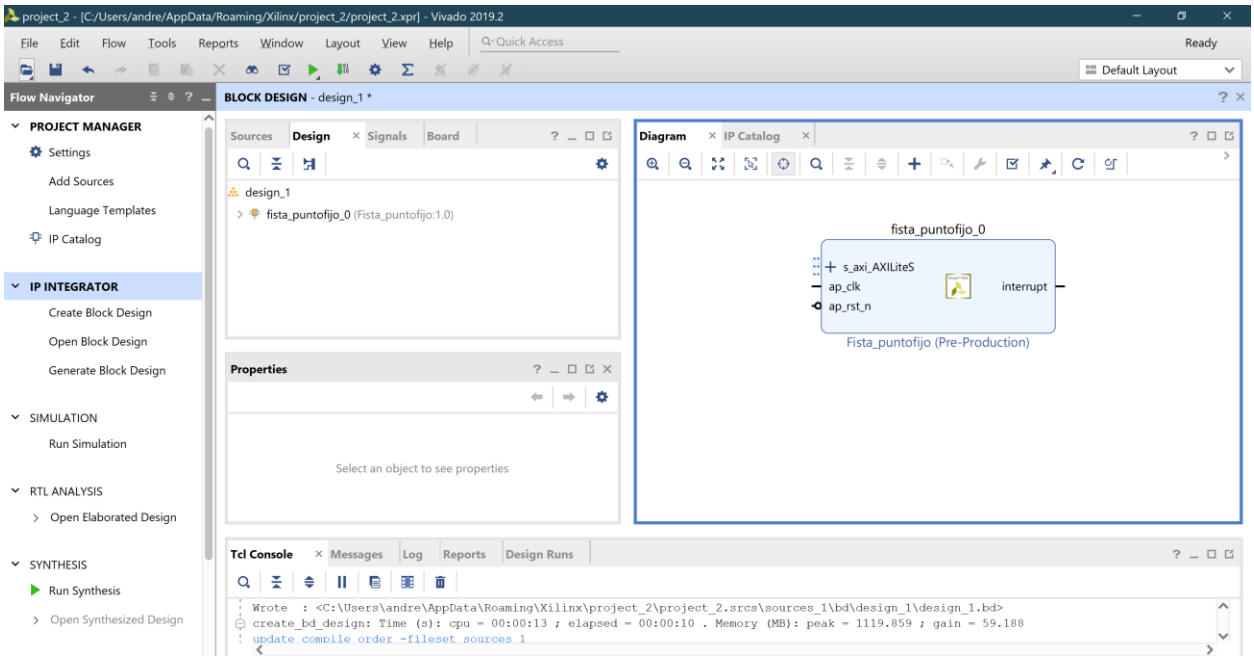

Figura 4-35 Módulo IP añadido

En el siguiente paso debemos conectar nuestro módulo IP con el procesador **zynq** de la placa, para ello repetimos el paso anterior. Y seleccionamos la opción “Run Connection Automation” (figura 4-36).

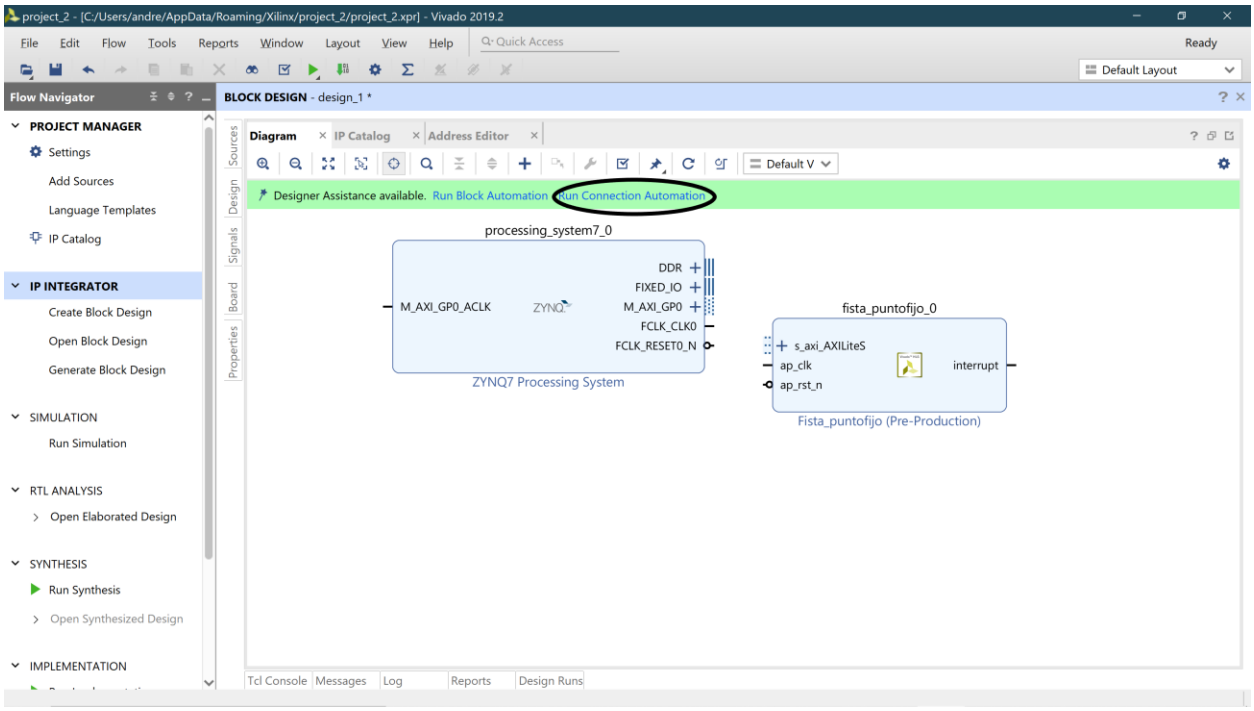

Figura 4-36 conexión del procesador con el módulo IP

En la figura 4-37 podemos ver la conexión realizada.

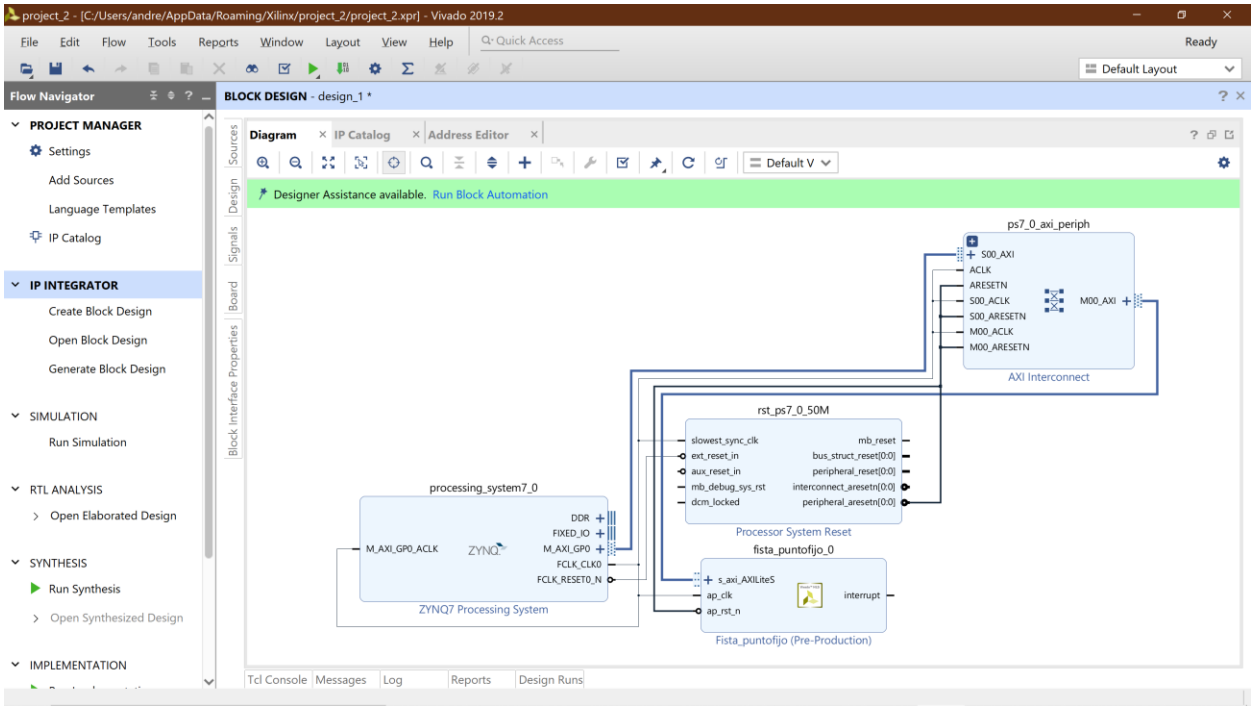

Figura 4-37 Conexión realizada

Ahora repetimos el mismo proceso para la opción “Run Block Automation”.

Una vez hecho validamos el diseño como se muestra en la figura 4-38.

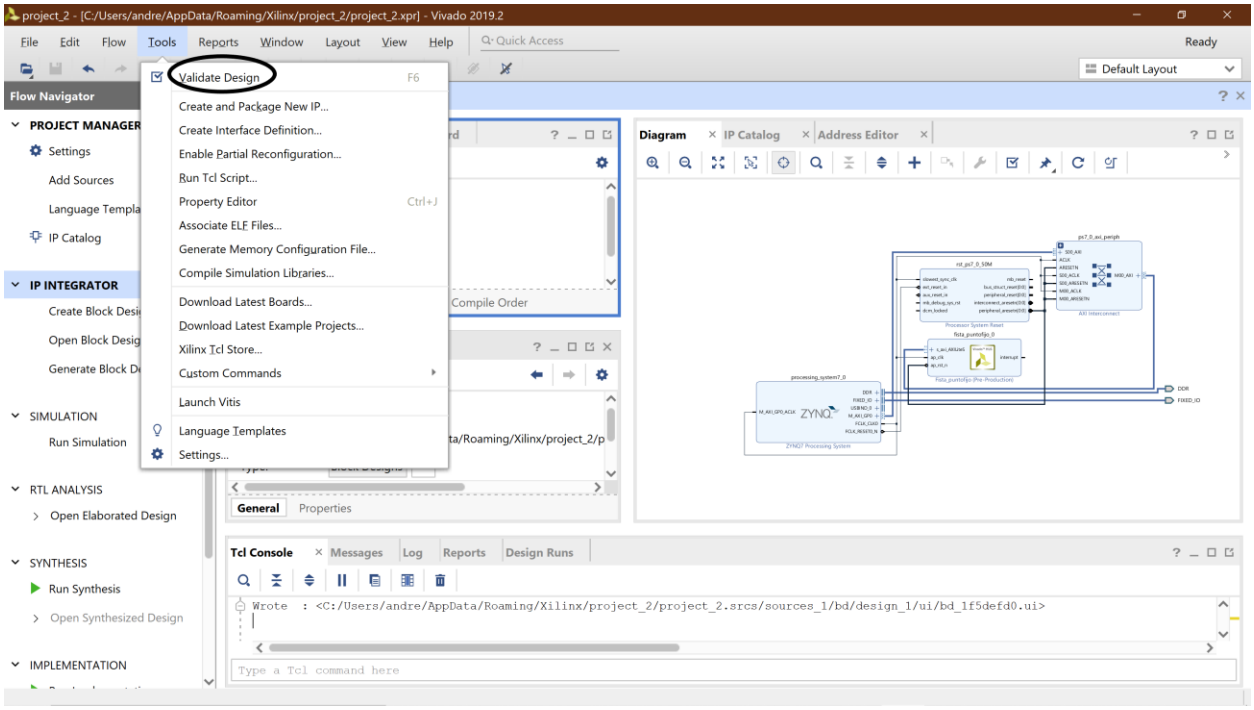

Figura 4-38 Validación del diseño

Una vez validado el diseño, procedemos a ir a la pestaña “source” y apretamos la opción “Create HDL Wrapper” (figura 4-39).

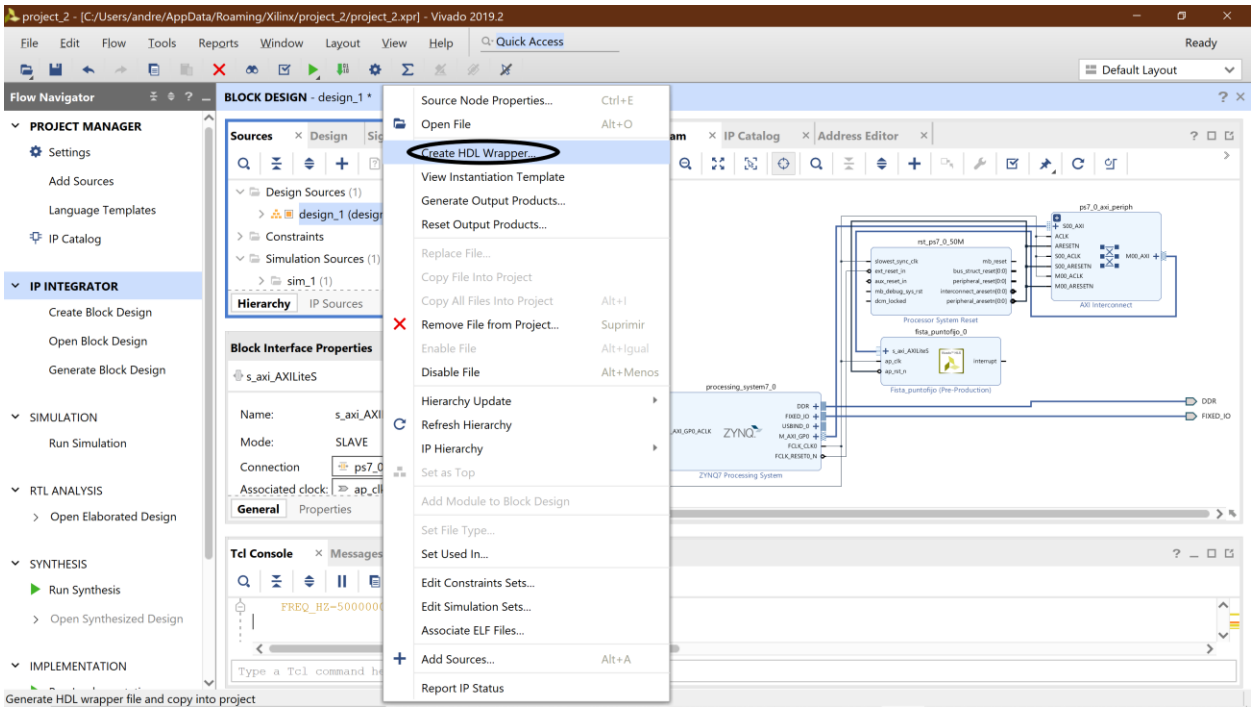

Figura 4-39 Creación de HDL Wrapper

Le damos ok a la ventana emergente seleccionando la segunda opción (figura 4-40).

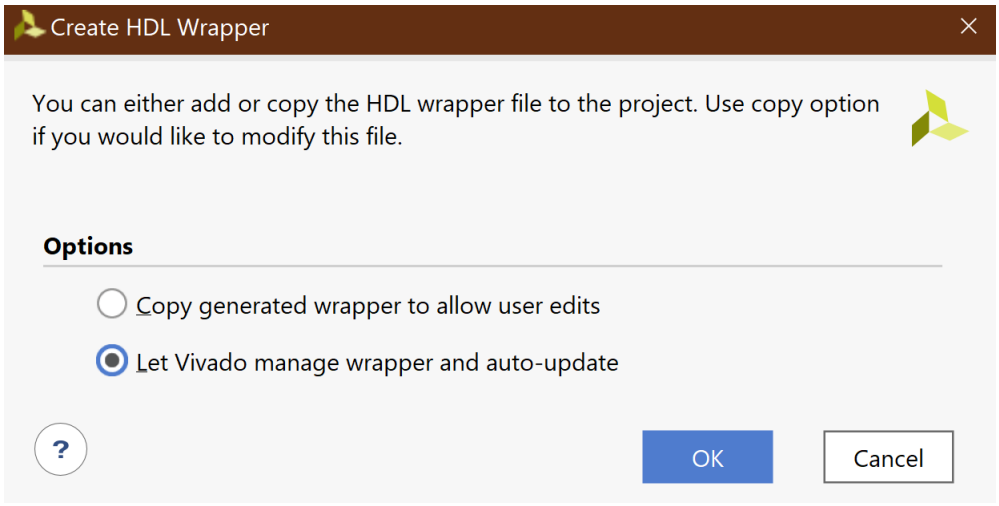

Figura 4-40 selección de HDL Wrapper

Y finalmente ya podemos ejecutar el último paso y generar el Bitstream (figura 4-41).

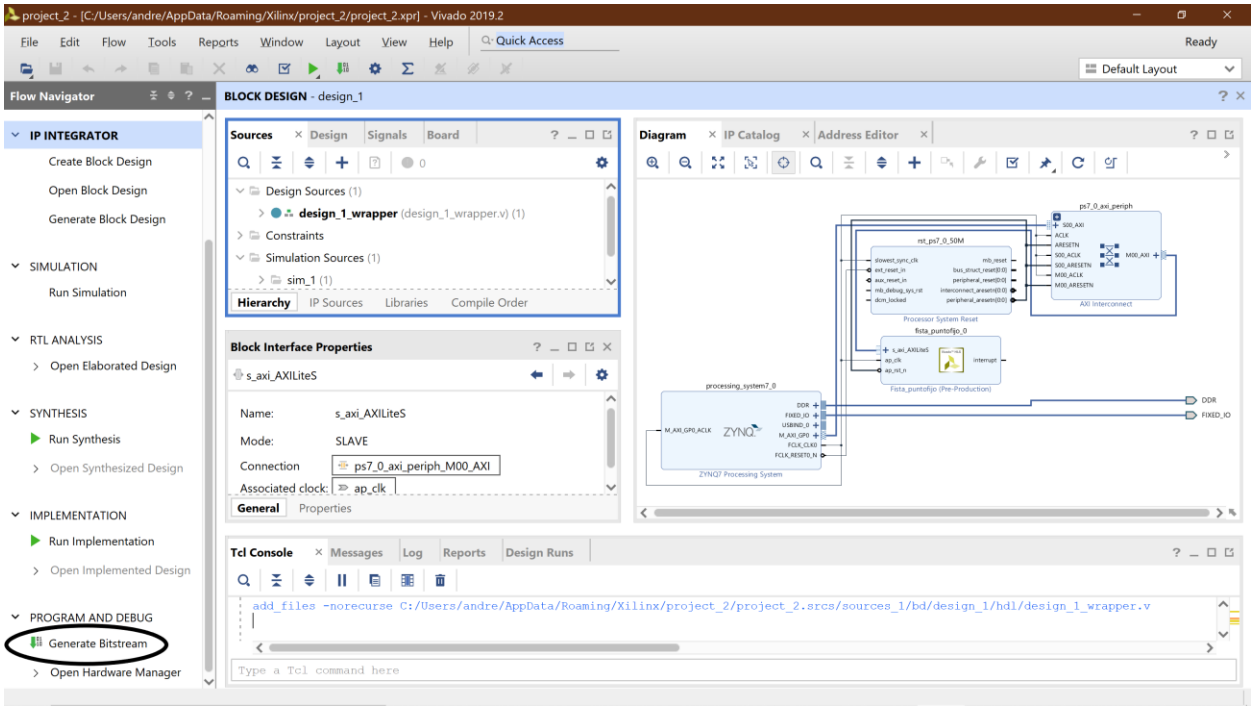

Figura 4-41 Generación del Bitstream

Le damos clic a “ok” en la ventana emergente y el programa empezará a generar el Bitstream (figura 4-42).

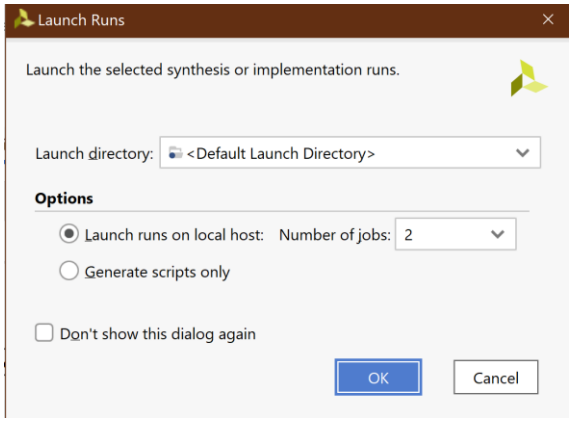

Figura 4-42 selección de opciones Bitstream

5 RESULTADOS Y CONCLUSIONES

En este capítulo se van a mostrar los resultados obtenidos en la programación del algoritmo en MATLAB y C, la simulación en C en Vivado HLS, los resultados de la síntesis y finalmente los resultados de la implementación del algoritmo en la placa tras realizar el Bitstream.

5.1. Resultados del algoritmo en MATLAB y C

La salida del programa tanto en MATLAB como en C es un vector n-dimensional, cuyo tamaño depende del número de columnas de la matriz G. En este caso z_salida tiene un tamaño de 180 elementos y se ha elegido una tolerancia de salida $\varepsilon = 1e-4$ para ambos códigos, para poder comprobar si la salida es correcta se ha comparado la solución con otro solver de una función de MATLAB llamada **quadprog ()**. Para ello se ha calculado la norma infinita de la diferencia de los dos resultados como se muestra a continuación:

```
max(abs(z_quad2-z_salida))

ans =

    9.2523e-04
```

Se debe destacar que el solver **quadprog ()**, utiliza una tolerancia de salida elegida por defecto, que no tiene por qué coincidir con el valor de la mencionada arriba.

Como se puede observar la diferencia entre los resultados es mínima, por lo tanto, podemos concluir que el algoritmo funciona correctamente en MATLAB.

Para comprobar el funcionamiento del algoritmo en C se ha repetido el paso anterior comparando la salida del algoritmo en C con la de MATLAB como se puede ver a continuación.

```
max(abs(z_salida-z_salidaC))

ans =

    3.9123e-05
```

Como vemos existe una pequeña diferencia entre los resultados, esto es debido a la declaración del tipo de formato de datos de uno y otro.

Todo lo comentado anteriormente se puede ver con mayor claridad en la figura 5-1.

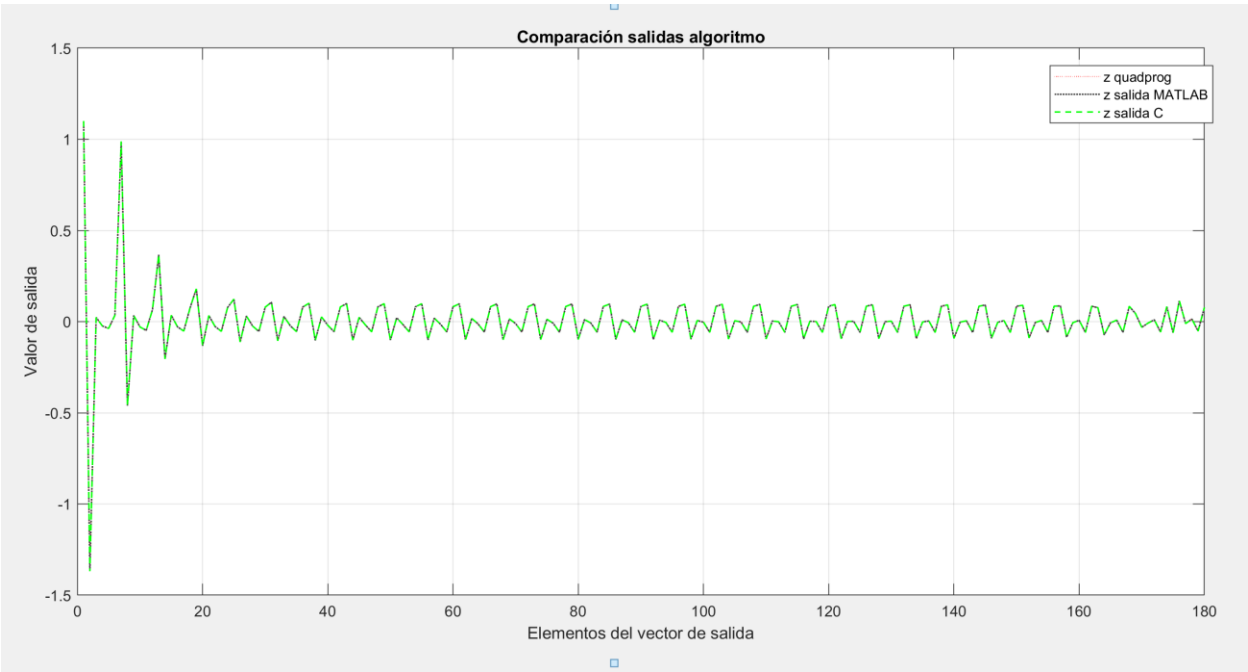

Figura 5-1 Comparación salidas del algoritmo

Como podemos ver las diferencias son despreciables, por lo tanto, podemos concluir que tanto el código en MATLAB como el código en C funcionan correctamente.

5.2. Resultados obtenidos en Vivado HLS

En este apartado destacan tres resultados principalmente:

- Resultados obtenidos por la simulación en C.
- Resultados obtenidos de la cosimulación C/RTL.
- Resultados obtenidos tras realizar la síntesis.

5.2.1 Resultados obtenidos tras la simulación en el procesador

Como se explicó en el apartado 4.2.1 una vez realizada la cosimulación se abrirá una pestaña llamada "_csim.log" (figura 4-10) con los datos obtenidos de la simulación, si repetimos lo hecho anteriormente y comparamos la salida de la simulación con MATLAB obtenemos el siguiente resultado:

```
max(abs(z_salida-z_salidaSimC))

ans =

    6.3123e-05
```

Como vemos la diferencia es mínima por lo tanto podemos concluir que el algoritmo funciona correctamente.

5.2.2 Resultados obtenidos tras la cosimulación

Como se explicó en el apartado 4.2.2 una vez finalizada la cosimulación se abrirá una nueva pestaña llamada "simulation(solution)" (figura 4-13). Esta ventana contiene una tabla que nos indica si la simulación C/RTL ha arrojado el mismo resultado o no, como se puede ver en la figura 5-2.

Cosimulation Report for 'fista_puntofijo'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	4663865	4663865	4663865	NA	NA	NA

Export the report(.html) using the Export Wizard

Figura 5-2 Resultados de la cosimulación

Como podemos ver en "Status" indica "Pass" esto significado que el resultado ha sido el mismo, por lo tanto, el algoritmo funciona correctamente.

5.2.3 Resultado de la síntesis

Como se explicó en el apartado 4.2.3 una vez finalizada la síntesis se abrirá una ventana llamada "Synthesis(solution)" (figura 4-18). Esta ventana nos muestra las estimaciones que hace el programa nuestro código, hay dos que resultan interesantes.

La primera muestra el tiempo esperado de ejecución (figura 5-3).

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.704 ns	1.25 ns

Figura 5-3 Tiempos de ejecución aproximados

La segunda muestra la utilización de la placa (figura 5-4).

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	32	80	4984	-
FIFO	-	-	-	-	-
Instance	106	25	4679	6007	-
Memory	11	-	0	0	0
Multiplexer	-	-	-	1582	-
Register	-	-	3401	-	-
Total	117	57	8160	12573	0
Available	280	220	106400	53200	0
Utilization (%)	41	25	7	23	0

Figura 5-4 Estimación uso de la placa

Se debe destacar que estas dos tablas son aproximaciones y después en el apartado 5.3 se mostrarán los resultados reales obtenidos tras la generación del Bitstream.

5.3 Resultados tras la implementación en la placa

Tras generar el Bitstream como se indica en el apartado 4.3 (figura 4-41), nos aparecerá una ventana emergente como se muestra en la figura 5-5 le damos a "Ok" seleccionando la primera opción y podremos ver todos los resultados obtenidos de la implementación.

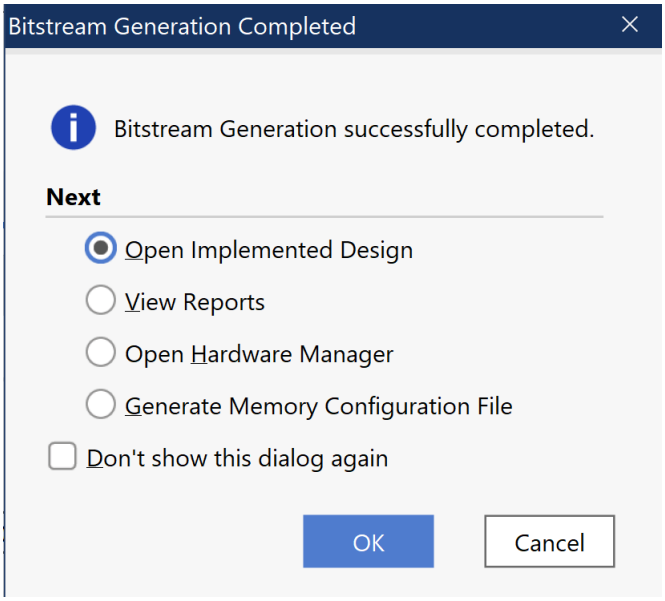

Figura 5-5 Finalización del Bitstream

Se abrirá una ventana en la que podemos ver las conexiones realizadas en el interior de la placa (figura 5-6).

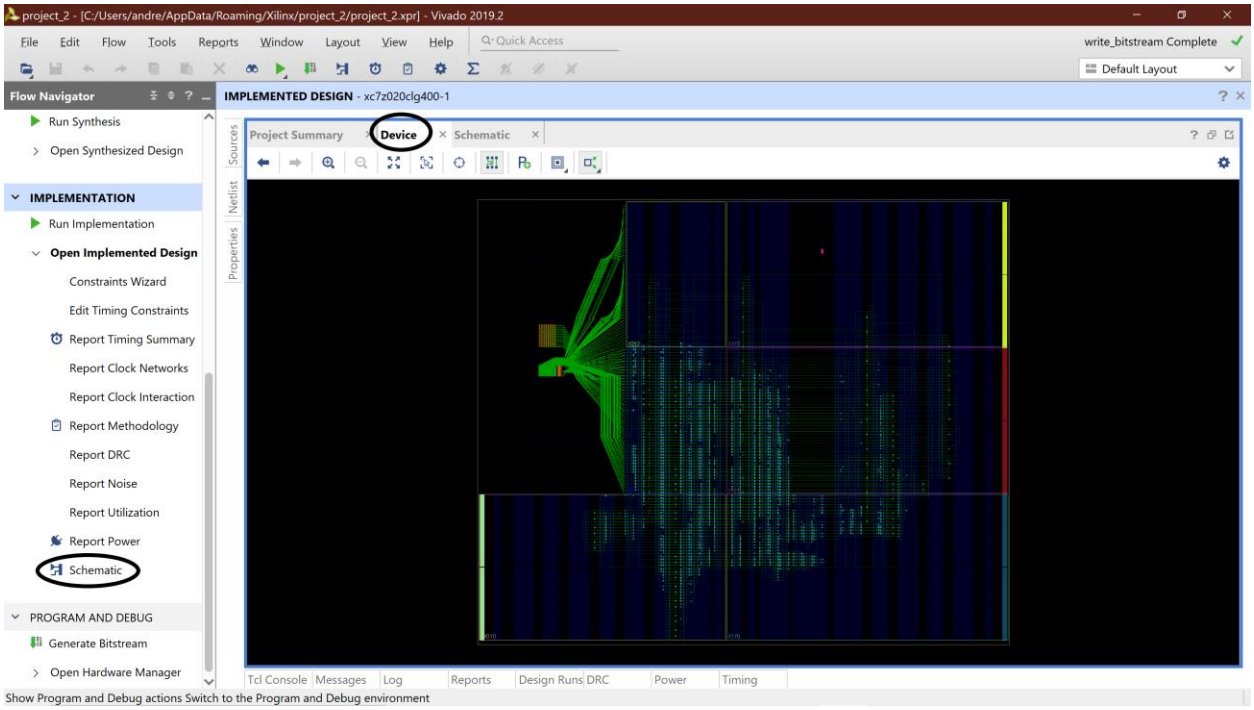

Figura 5-6 Conexiones realizadas en la FPGA

Si hacemos zoom lo suficiente se pueden llegar a ver las conexiones a nivel de puertas lógicas.

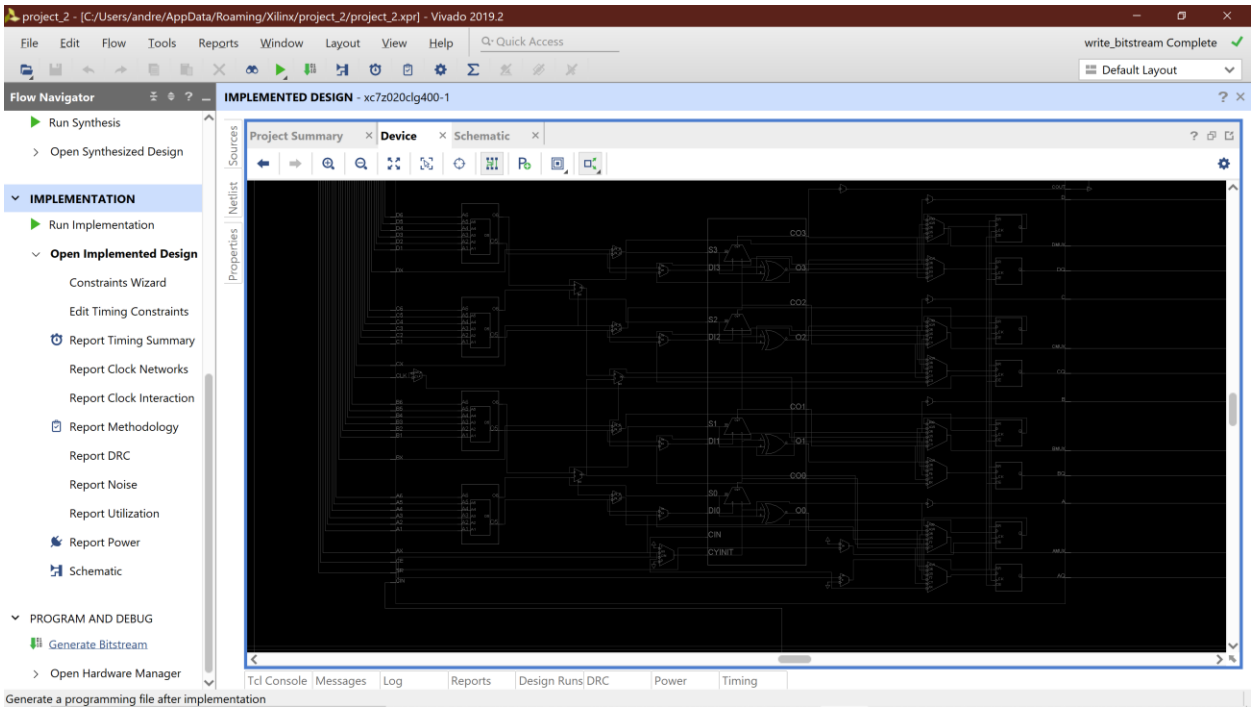

Figura 5-7 Nivel de conexiones en puertas lógicas

En la sección “IMPLEMENTATION” de la izquierda podemos ver diferentes análisis realizados tras la implementación en la FPGA. Vamos a destacar dos principalmente que son “Report Power” y “Report utilization” (figura 5-8).

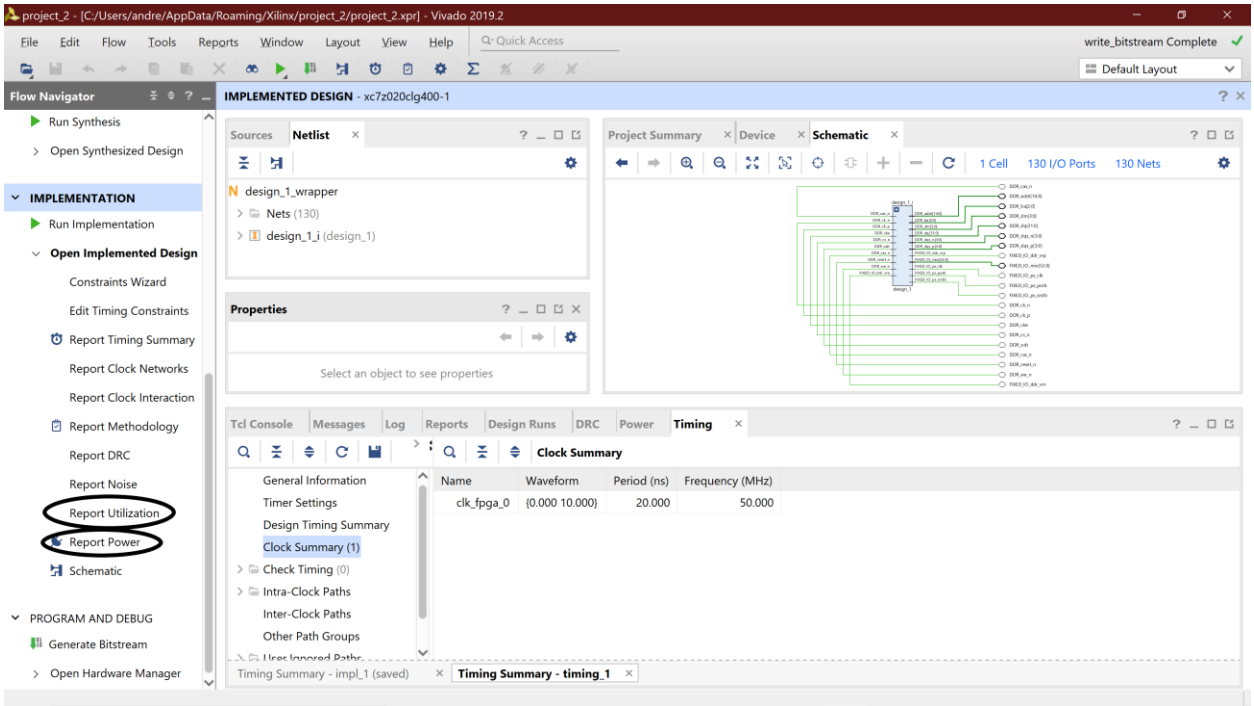

Figura 5-8 Características principales tras la implementación

En el análisis de consumo de energía podemos ver datos interesantes como son la temperatura máxima que alcanza el chip de la FPGA, o el consumo de energía de esta (Figura 5-9).

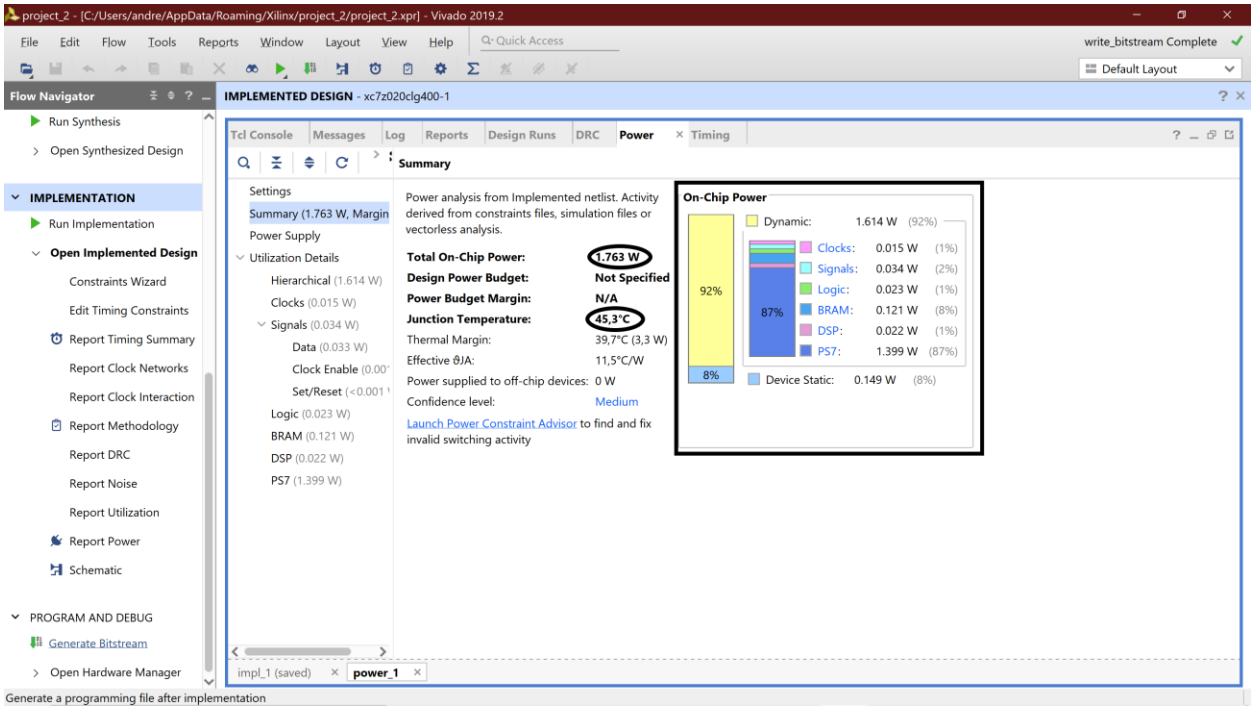

Figura 5-9 Análisis de consumo de Energía de la FPGA

Y en el análisis de recursos podemos ver cuanta memoria de la FPGA consume el algoritmo (figura 5-10).

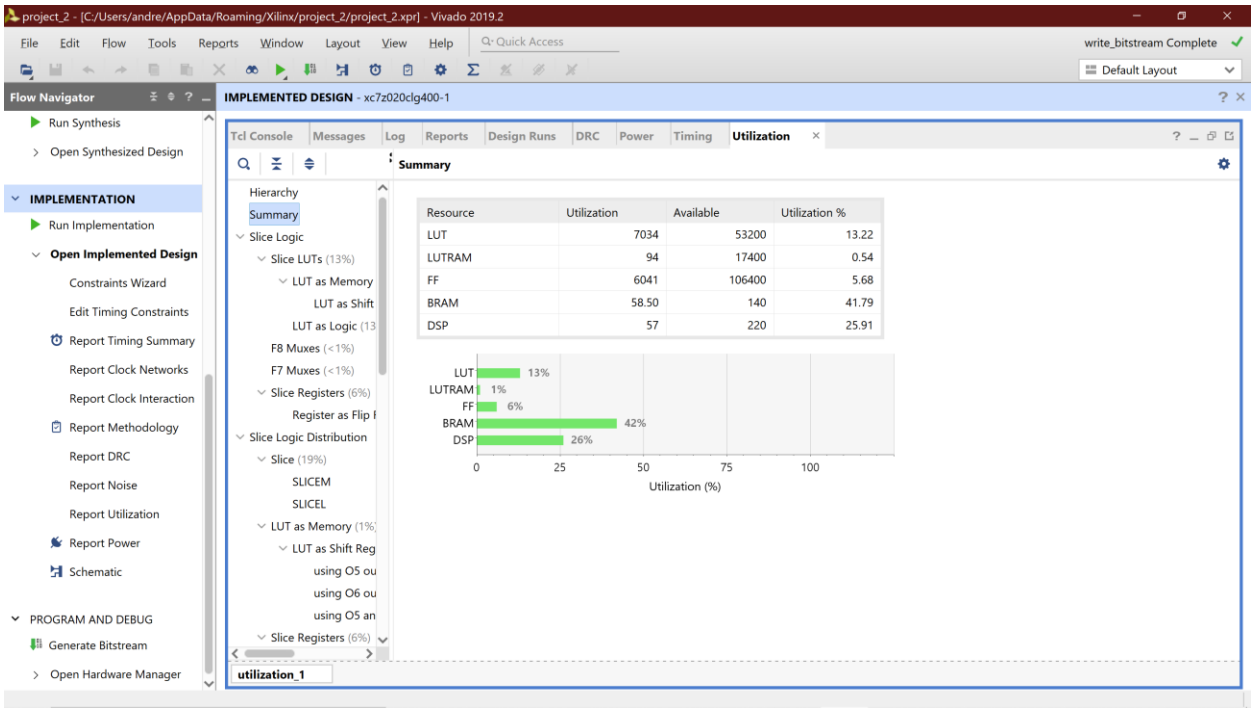

Figura 5-10 Análisis de consumo de recursos

Como podemos ver hay algunas diferencias con los resultados obtenidos tras la síntesis en Vivado HLS, esto se debe a que en Vivado HLS son estimaciones, mientras que estas son las reales.

5.4 Conclusiones finales

Se ha comprobado el correcto funcionamiento del algoritmo tanto en el código de MATLAB y en C y finalmente de la simulación C. Y después se ha comprobado que el diseño RTL ha ofrecido el mismo resultado, finalmente

se ha comprobado que tras la implementación en la FPGA no ha habido problemas de consumo ni de utilización de recurso, por lo tanto, podemos concluir que el algoritmo funciona y se puede implementar sin problemas en la placa real **Zybo z20**.

6 REFERENCIAS

[1] BECK, Amir; TEBOULLE, Marc. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences*, 2009, vol. 2, no 1, p. 183-202.

[2] BADER, B. W. Constrained and unconstrained optimization. 2009.

[3] CAMACHO, Eduardo F.; ALBA, Carlos Bordons. *Model predictive control*. Springer Science & Business Media, 2013.

[4] KIM, Donghwan; FESSLER, Jeffrey A. Another look at the fast iterative shrinkage/thresholding algorithm (FISTA). *SIAM Journal on Optimization*, 2018, vol. 28, no 1, p. 223-250.

[5] KHAN, Ashfaq Ahmad. *Stochastic Quadratic Programming*. 2011. Tesis Doctoral. Aligarh Muslim University.

[6] KIM, Seung-Jean, et al. An interior-point method for large-scale l_1-regularized least squares. *IEEE journal of selected topics in signal processing*, 2007, vol. 1, no 4, p. 606-617.

[7] MURTY, Katta G.; YU, Feng-Tien. *Linear complementarity, linear and nonlinear programming*. Berlin: Heldermann, 1988.

[8] DELBOS, Frédéric; GILBERT, Jean Charles. Global linear convergence of an augmented Lagrangian algorithm for solving convex quadratic optimization problems. 2003.

[9] DAI, Yu-Hong. Nonlinear conjugate gradient methods. *Wiley Encyclopedia of Operations Research and Management Science*, 2010.

[10] ROSEN, Jo Bo. The gradient projection method for nonlinear programming. Part I. Linear constraints. *Journal of the society for industrial and applied mathematics*, 1960, vol. 8, no 1, p. 181-217.

[11] KRUPA, Pablo; LIMON, Daniel; ALAMO, Teodoro. Implementation of Model Predictive Control in Programmable Logic Controllers. *IEEE Transactions on Control Systems Technology*, 2020.

[12] BERTSEKAS, Dimitri P. *Convex optimization theory*. Belmont: Athena Scientific, 2009.

[13] BECK, Amir; TEBOULLE, Marc. A fast dual proximal gradient algorithm for convex minimization and applications. *Operations Research Letters*, 2014, vol. 42, no 1, p. 1-6.

[14] FAROOQ, Umer; MARRAKCHI, Zied; MEHREZ, Habib. *Tree-based heterogeneous FPGA architectures: application specific exploration and optimization*. Springer Science & Business Media, 2012.

[15] PANG, Aiken; MEMBREY, Peter. *Beginning FPGA: Programming Metal*. Apress, 2016.

[16] https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/start

[17] APARICIO OLMEDO, Aldo Alejandro; PONLUISA MARCALLA, Neiser Fernando. *Estudio comparativo de los lenguajes HDL y su aplicación en la implementación del Laboratorio de Sistemas Digitales Avanzados mediante FPGAS en la EIE-CRI*. 2014. Tesis de Licenciatura.

[18] COUSSY, Philippe; MORAWIEC, Adam (ed.). *High-level synthesis: from algorithm to digital circuit*. Springer Science & Business Media, 2008.

[19] HENDRIKS, Johan, et al. High Level Synthesis: Performance analysis and code optimization. *Master thesis of Eindhoven University of Technology*, 2012.

[20] FINGEROFF, Michael. *High-level synthesis: blue book*. Xlibris Corporation, 2010.

[21] COUSSY, Philippe, et al. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 2009, vol. 26, no 4, p. 8-17.

[22] CONG, Jason, et al. Platform-based behavior-level and system-level synthesis. En *2006 IEEE International SoC Conference*. IEEE, 2006. p. 199-202.

[23] CHEN, Deming, et al. xpilot: A platform-based behavioral synthesis system. *SRC TechCon*, 2005, vol. 5.

[24] P CARBALLO, Pedro. *Aportaciones a la metodología de diseño basada en Síntesis de Alto Nivel. Aportaciones al diseño de IPs para procesado de eventos complejos y codificación de vídeo*. 2016. Tesis Doctoral.

7 ANEXOS

8.1. Anexo A código del Algoritmo FISTA en MATLAB

```
%Algoritmo FISTA
clear all
load('QPejemplo.mat')
% salida----->valor óptimo de la variable Z
%Tambien calcular la variable lambda

epsilon=1e-4;

%%valores que necesito calcular mz, lambda,

 mz=rank(G);
[m,n]=size(G);

%%inicio del algoritmo


%paso 1: inicialización de variables

lambda_k(1:m)=0;
lambda_k=lambda_k';
k=0;
eta_k=lambda_k;
t_k=1;
residuo=1;

%%para eliminar warnings
z_ka(1:n)=0;
z_k=z_ka;

%% paso 2:repetir hasta

while(norm(residuo) >  epsilon)

%paso 3 incrementar k

k=k+1;
lambda_ka=lambda_k;
eta_ka=eta_k;
t_ka=t_k;
%paso 4 obtener z(lambda(k-1))

for i=1:n
 sum=0;
for j=1:1:mz
    sum=sum+G(j,i)*lambda_ka(j);
%z_ka(i)= max(min((-(q(i)-G(j,i)*lambda_ka(j))/H(i,i)),UB(i)),LB(i));

end
z_ka(i)= max(min((-(q(i)-sum)/H(i,i)),UB(i)),LB(i));
end
```

```
%paso 5 obtener deltalambda(k-1)

Wh=(G/H)*G';
deltalambda_ka=(inv(Wh))*(-(G*z_ka'-b));

%paso 6 obtener eta(k)

eta_k=lambda_ka+deltalambda_ka;

%paso 7 calcular t(k)

t_k=(1+sqrt(1+4*t_ka^2))/2;

%paso 8 calcular lambda(k)

lambda_k=eta_k+((t_ka-1)/t_k)*(eta_k-eta_ka);

%paso 9 residuo

 for i=1:n
     sum=0;
 for j=1:1:mz
     sum=sum+G(j,i)*lambda_k(j);

 %z_k(i)= max(min((-(q(i)-G(j,i)*lambda_k(j))/H(i,i)),UB(i)),LB(i));

 end
 z_k(i)= max(min((-(q(i)-sum)/H(i,i)),UB(i)),LB(i));
end
residuo=G*z_k'-b;

% paso 10 repetir hasta que norma residuo<=epsilon

end

z_salida=z_k';
```

8.2. Anexo B Programación en C

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

#define filas 120
#define columnas 180

float epsilon = 0.0001;
int kmax= 25;

int main(){


//matriz lambda_k

float lambda_k[filas];
float lambda_ka[filas];
float deltalambda_ka[filas];

int k=0;

//matriz eta

float eta_k[filas];
float eta_ka[filas];

float t_k = 1;
float t_ka;
float residuo[filas];

float z_k[columnas];
float z_ka[columnas];
float z_salida[columnas];


float norma = 1;
float aux;
float minimo;

// paso 1 inicializacion de variables

for(int i=0;i<filas;i++){
      lambda_k[i]=0;
      eta_k[i]=0;
}

//inicio del algoritmo
//paso 2

while((norma > epsilon) && (k < kmax)){

//paso 3 incrementar k

k = k+1;
printf("k: %d \n",k);

//copiamos matrices

for(int i=0;i<filas;i++){
```

```
lambda_ka[i] = lambda_k[i];
eta_ka[i] = eta_k[i];
}
t_ka = t_k;


//paso 4 obtener z (lambda (k-1))

for(int i = 0; i<columnas; i++){
      float sum1=0;
      for(int j=0; j<filas;j++){
            sum1 = sum1 + G[j][i]*lambda_ka[j];
      }
      aux=(-(q[i]-sum1))*H[i];


      //cálculo del minimo y máximo

      if(aux>UB[i]){
            minimo = UB[i];
      }
      else{
            minimo = aux;
      }
      if(minimo>LB[i]){
            z_ka[i]=minimo;
      }
      else{
            z_ka[i]=LB[i];
      }

}

//paso 5 obtener deltalambda(k-1)

//deltalambda_ka=(inv(Wh))*(-(G*z_ka'-b));

//multiplicacion G*z_ka'
float producto2[120];


        // Dentro recorremos las filas de la primera (A)
        for (int i = 0; i < 120; i++) {
            float aux2 = 0;
            // Y cada columna de la primera (A)
            for (int j = 0; j < 180; j++) {
                // Multiplicamos y sumamos resultado
                aux2 += G[i][j] * z_ka[j];
            }
            // Lo acomodamos dentro del producto
            producto2[i] = aux2;
        }

//resta producto3 -b
float aux4[120];

for(int i=0; i<120;i++){
      aux4[i]=-(producto2[i]-b[i]);
}

//multiplicacion inv(wh)*aux4
```

```
        // Dentro recorremos las filas de la primera (A)
        for (int i = 0; i < 120; i++) {
            float aux2 = 0;
            // Y cada columna de la primera (A)
            for (int j = 0; j < 120; j++) {
                // Multiplicamos y sumamos resultado
                aux2 += Wh_inv[i][j] * aux4[j];
            }
            // Lo acomodamos dentro del producto
            deltalambda_ka[i] = aux2;
        }

//paso 6 obtener eta(k)
//eta_k=lambda_ka+deltalambda_ka;

for(int i=0;i<120;i++){
      eta_k[i]=lambda_k[i]+deltalambda_ka[i];
      }

//paso 7 calcular t(k)

t_k=(1+sqrt(1+4*(t_ka*t_ka)))/2;

//paso 8 calcular lambda(k)
//lambda_k=eta_k+((t_ka-1)/t_k)*(eta_k-eta_ka);

for(int i=0;i<120;i++){

      lambda_k[i]=eta_k[i]+((t_ka-1)/t_k)*(eta_k[i]-eta_ka[i]);
      }

//paso 9 residuo

 for (int i=0;i<columnas;i++){

     float sum2=0;
 for (int j=0;j<filas;j++){

     sum2=sum2+G[j][i]*lambda_k[j];
 }
 //z_k(i)= max(min((-(q(i)-G(j,i)*lambda_k(j))/H(i,i)),UB(i)),LB(i));

 aux=(-(q[i]-sum2))*H[i];

      //calculo del minimo y maximo

      if(aux>UB[i]){
            minimo = UB[i];
      }
      else{
            minimo = aux;
      }
      if(minimo>LB[i]){
            z_k[i]=minimo;
      }
      else{
            z_k[i]=LB[i];
      }

}
```

```
//multiplicacion G*z_k'

//residuo=G*z_k'-b;
float producto5[120];


        // Dentro recorremos las filas de la primera (A)
        for (int i = 0; i < 120; i++) {
            float aux10 = 0;
            // Y cada columna de la primera (A)
            for (int j = 0; j < 180; j++) {
                // Multiplicamos y sumamos resultado
                aux10 += G[i][j] * z_k[j];
            }
            // Lo acomodamos dentro del producto
            producto5[i] = aux10;

        }



for(int i=0;i<120;i++){
      residuo[i]=producto5[i]-b[i];

}

//paso 10 repetir hasta que norma residuo<=epsilon
float aux6=0;

for(int i=0;i<120;i++){
      aux6=aux6+(residuo[i]*residuo[i]);
}
norma=sqrt(aux6);
//printf("\n norma: %f \n\n",norma);
}

for(int i=0;i<180;i++){

z_salida[i]=z_k[i];
}
if(k==kmax){
      printf("Numero de iteraciones maximas alcanzadas\n");
}
else{
      printf("residuo menor que epsilon\n");
}

printf("imprimiendo valor de k: %d",k);
for(int i=0;i<180;i++){

      printf("\n %f",z_salida[i]);
}

return 0;

}
```

8.3. Anexo C codigo Fista.cpp

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

#define filas 120
#define columnas 180

typedef ap_fixed<32,9> fix_t;

void fista_puntofijo(fix_t epsilon, int kmax, fix_t G[filas][columnas], fix_t
H[columnas], fix_t UB[filas], fix_t LB[columnas], fix_t b[filas], fix_t
q[columnas], fix_t Wh_inv[filas][filas])
{


//calculamos el numero de filas y columnas de la matriz G


//matriz lambda_k

fix_t lambda_k[filas];
fix_t lambda_ka[filas];
fix_t deltalambda_ka[filas];

int k=0;

//matriz eta

fix_t eta_k[filas];
fix_t eta_ka[filas];

fix_t t_k = 1;
fix_t t_ka;
fix_t residuo[filas];

fix_t z_k[columnas];
fix_t z_ka[columnas];
fix_t z_salida[columnas];


fix_t norma = 1;
fix_t aux;
fix_t minimo;

//inicializacion de variables

for(int i=0;i<filas;i++){
      lambda_k[i]=0;
      eta_k[i]=0;
}

//inicio del algoritmo
//printf("\nempezando algoritmo\n");

while((norma > epsilon) && (k < kmax)){

//paso 1 incrementar k
```

```
//printf("incrementamos k\n");
k = k+1;
printf("k: %d \n",k);

//copiar matrices

//printf("copiamos matrices\n");
for(int i=0;i<filas;i++){

lambda_ka[i] = lambda_k[i];
eta_ka[i] = eta_k[i];
}
t_ka = t_k;
//printf("matrices copiadas\n");

//paso 3 obtener z (lambda (k-1))

//printf("empezando paso3\n");
for(int i = 0; i<columnas; i++){
      fix_t sum1=0;
      for(int j=0; j<filas;j++){
            sum1 = sum1 + G[j][i]*lambda_ka[j];
      }
      //z_ka(i)= max(min((-(q(i)-sum)/H(i,i)),UB(i)),LB(i));
      aux=(-(q[i]-sum1))*H[i];


      //calculo del minimo y maximo

      if(aux>UB[i]){
            minimo = UB[i];
      }
      else{
            minimo = aux;
      }
      if(minimo>LB[i]){
            z_ka[i]=minimo;
      }
      else{
            z_ka[i]=LB[i];
      }

}

//paso 4 obtener deltalambda(k-1)

//deltalambda_ka=(inv(Wh))*(-(G*z_ka'-b));

//printf("empezando paso 4\n");


//multiplicacion G*z_ka'
fix_t producto2[120];


        // Dentro recorremos las filas de la primera (A)
        for (int i = 0; i < 120; i++) {
            fix_t aux2 = 0;
            // Y cada columna de la primera (A)
            for (int j = 0; j < 180; j++) {
                // Multiplicamos y sumamos resultado
                aux2 += G[i][j] * z_ka[j];
```

```
            }
            // Lo acomodamos dentro del producto
            producto2[i] = aux2;
        }

//resta producto3 -b
fix_t aux4[120];

for(int i=0; i<120;i++){
      aux4[i]=-(producto2[i]-b[i]);
}

//multiplicacion inv(wh)*aux4


        // Dentro recorremos las filas de la primera (A)
        for (int i = 0; i < 120; i++) {
            fix_t aux2 = 0;
            // Y cada columna de la primera (A)
            for (int j = 0; j < 120; j++) {
                // Multiplicamos y sumamos resultado
                aux2 += Wh_inv[i][j] * aux4[j];
            }
            // Lo acomodamos dentro del producto
            deltalambda_ka[i] = aux2;
        }

//paso 5 obtener eta(k)
//eta_k=lambda_ka+deltalambda_ka;

//printf("Empezando paso 5\n");
for(int i=0;i<120;i++){
      eta_k[i]=lambda_k[i]+deltalambda_ka[i];
      }

//paso 6 calcular t(k)

//printf("Empezando paso 6\n");

t_k=(1+sqrt(1+4*((float)t_ka*(float)t_ka)))/2;

//t_k = (fix_t) t_k;

//paso 7 calcular lambda(k)
//lambda_k=eta_k+((t_ka-1)/t_k)*(eta_k-eta_ka);

//printf("Empezando paso 7\n");

for(int i=0;i<120;i++){

      lambda_k[i]=eta_k[i]+((t_ka-1)/t_k)*(eta_k[i]-eta_ka[i]);
      }

//paso 8 residuo

//printf("Empezando paso 8 \n");

 for (int i=0;i<columnas;i++){

     fix_t sum2=0;
 for (int j=0;j<filas;j++){
```

```
      sum2=sum2+G[j][i]*lambda_k[j];
 }
 //z_k(i)= max(min((-(q(i)-G(j,i)*lambda_k(j))/H(i,i)),UB(i)),LB(i));

 aux=(-(q[i]-sum2))*H[i];

      //calculo del minimo y maximo

      if(aux>UB[i]){
            minimo = UB[i];
      }
      else{
            minimo = aux;
      }
      if(minimo>LB[i]){
            z_k[i]=minimo;
      }
      else{
            z_k[i]=LB[i];
      }

}


//multiplicacion G*z_k'

//residuo=G*z_k'-b;
fix_t producto5[120];


        // Dentro recorremos las filas de la primera (A)
        for (int i = 0; i < 120; i++) {
            fix_t aux10 = 0;
            // Y cada columna de la primera (A)
            for (int j = 0; j < 180; j++) {
                // Multiplicamos y sumamos resultado
                aux10 += G[i][j] * z_k[j];
            }
            // Lo acomodamos dentro del producto
            producto5[i] = aux10;

        }



for(int i=0;i<120;i++){
      residuo[i]=producto5[i]-b[i];

}

//paso 9 repetir hasta que norma residuo<=epsilon
//printf("empezamos paso 9\n");

fix_t aux6=0;

for(int i=0;i<120;i++){
      aux6=aux6+(residuo[i]*residuo[i]);
}
norma=sqrt((float)aux6);
//norma =(fix_t) norma;
//printf("\n norma: %f \n\n",norma);
}
```

```
for(int i=0;i<180;i++){

z_salida[i]=z_k[i];
}
if(k==kmax){
      printf("Numero de iteraciones maximas alcanzadas\n");
}
else{
      printf("residuo menor que epsilon\n");
}

printf("imprimiendo valor de k: %d",k);
for(int i=0;i<180;i++){

      printf("\n %f",(float)z_salida[i]);
}

return;

}
```

8.4 Anexo D Código de Fista_TB.cpp

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

#define filas 120
#define columnas 180

typedef ap_fixed<32,9> fix_t;
int main()
{

      fista_puntofijo(epsilon,kmax,G,H,UB,LB,b,q,Wh_inv);


return 0;
}
```