Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

Diseño de un Filtro de Kalman e Implementación en una FPGA Basada en Vivado HLS

Autor: Daniel Calvo Guillén

Tutor/es: Ignacio Bravo Muñoz

2016

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Diseño de un filtro de Kalman e implementación en una FPGA basada en Vivado HLS

Autor: Daniel Calvo Guillén

Tutor/es: Ignacio Bravo Muñoz

TRIBUNAL:

Presidente: Alfredo Gardel Vicente

Vocal 1º: Ignacio Fernández Lorenzo

Vocal 2º: Ignacio Bravo Muñoz

Calificación: ……

Fecha: ……

Índice general

1. Introducción

1.1. Resumen

En este proyecto se aborda el diseño de un filtro de Kalman y su implementación en una FPGA siguiendo una metodología basada en Vivado HLS. El objetivo principal es evaluar la herramienta y su metodología de trabajo para el diseño de aplicaciones complicadas de desarrollar siguiendo la metodología tradicional de descripción hardware.

También se explora la posibilidad de exportar los diseños realizados en Vivado HLS para utilizarlos como periféricos del bus AXI, presente en la familia de dispositivos Zynq-7000 de Xilinx, e integrarlos en un sistema dirigido por un microprocesador.

Palabras clave:

- HLS
- Kalman
- FPGA
- ZedBoard
- Zynq

1.2. Abstract

This Project addresses the design of a Kalman filter and its implementation in a FPGA following a methodology based on Vivado HLS. The main aim is to evaluate the tool and its working methodology towards the design of applications hard to develop following the traditional methodology of hardware description.

Besides, it is explored the possibility of exporting the designs carried out in Vivado HLS to use them as peripheral designs of AXI bus, present in the Zynq-7000 devices family of Xilinx, and integrating them in a system run by a microprocessor.

Key words:

- HLS
- Kalman
- FPGA
- ZedBoard
- Zynq

1.3. Resumen extendido

Vivado HLS[1] es una herramienta desarrollada por Xilinx que permite el desarrollo de diseños para FPGAs[2] utilizando lenguaje de alto nivel (C/C++ y SystemC). Dada una aplicación o función descrita en uno de estos lenguajes, la herramienta realiza una síntesis del comportamiento descrito en el código, y sintetiza una solución hardware que lo imite.

De esta forma, es posible dar un nuevo enfoque a los problemas que se pretenden resolver utilizando hardware reprogramable, pues los lenguajes de alto nivel permiten un mayor nivel de abstracción del diseñador.

Este proyecto pretende evaluar cómo sería el flujo de diseño en alto nivel de una aplicación mínimamente exigente para poner a prueba a la herramienta. Se plantea, por tanto, el diseño de un filtro de Kalman.

El filtro de Kalman es un estimador de estados para sistemas dinámicos modelados en el espacio de estados. La particularidad que éste presenta frente a otros estimadores es que en su formulación incluye los posibles ruidos o incertidumbres que puedan afectar tanto al propio sistema como al sensor utilizado para medir sus salidas. Los ruidos son tenidos en cuenta a la hora de calcular la ganancia del estimador. Esta ganancia se calcula de tal modo que haga mínima la covarianza del error en la estimación, es decir, que haga óptimos los valores estimados de los estados del sistema.

Se ha escogido este algoritmo como conejillo de indias para Vivado HLS debido a que en su formulación se plantea una inversión de matrices, una tarea nada trivial, y mucho menos para abordarla mediante descripción hardware.

Una vez se haya realizado el diseño en Vivado HLS, la herramienta ofrece la posibilidad de exportarlo a Vivado como un IP, e implementarlo en una FPGA. Entre las muchas posibilidades de exportación o implementación, es posible hacerlo como si fuese un periférico para los sistemas basados en microprocesador, como los dispositivos de la familia Zynq o Microblaze.

Utilizar Vivado y Vivado HLS juntos permite la implementación de una solución mixta (software y hardware coordinados) en una única tarjeta y codificarlo todo en lenguaje de alto nivel.

En este trabajo se ha explorado también esta faceta de la herramienta, y se han exportado los diseños realizados en HLS como periféricos de Zynq, elaborando un

[1] High Level Síntesis.
[2] Field Programmable Gate Array

sistema sencillo para la implementación del filtro de Kalman basado en un microprocesador. El sistema ha sido montado sobre la tarjeta de evaluación ZedBoard.

En el Capítulo 2. se detallan los pasos seguidos en el desarrollo del filtro de Kalman en Vivado HLS y su implementación en la tarjeta ZedBoard, así como todos los conocimientos previos que se estiman oportunos para la compresión del trabajo realizado.

2. Memoria

2.1. Fundamentos teóricos

Introducción:

Los estados de un sistema son todas aquellas variables que suministran una completa representación de comportamiento interno del mismo. Lejos de ser una definición rigurosa, es suficiente para la introducción al filtro de Kalman. Por ejemplo, los estados de un motor eléctrico pueden ser las corrientes que circulan por los devanados, la posición y velocidad del eje. Los estados de un satélite pueden ser su posición, velocidad y posición angular. Fuera del mundo de la ingeniería, se puede definir un sistema económico en el que sus estados sean la renta per cápita, el desempleo y la tasa de crecimiento económico. En un sistema biológico, los estados podrían ser el nivel de azúcar en la sangre, el ritmo cardiaco y la temperatura corporal.

La estimación de estos estados se puede aplicar a casi todas las áreas de la ingeniería y la ciencia, o al menos a aquellas disciplinas que cuenten con sistemas modelados matemáticamente. La estimación de los estados ocultos de un sistema tiene un sinfín de aplicaciones, y por ello la teoría del espacio de estado y la estimación del estado han sido ampliamente investigada en las últimas décadas, desde que se empezara a desarrollarse en los años 50 y 60.

Concretando, para el mundo de la ingeniería la estimación de los estados el clave en al menos dos aspectos:

A menudo, el conocimiento de los estados es necesario para la implementación de un controlador mediante realimentación de los estados.

Los estados de un sistema pueden ser de interés por sí mismos. Por ejemplo, si se quiere saber si un sistema está funcionando correctamente, puede ser necesario saber su condición interna; o si se quiere organizar de forma inteligente las actividades de un satélite, puede ser necesario conocer su posición.

La alternativa escogida para solucionar la estimación de los estados de un sistema es el Filtro de Kalman. El filtro fue desarrollado por Rudolf E. Kalman en 1960, y permite identificar el estado oculto (no medible) de un sistema dinámico lineal cuando éste se encuentra en presencia de ruido blanco. El punto fuerte de este algoritmo es que, partiendo de que el ruido presente debe ser blanco y el sistema lineal, permite la

obtención de la ganancia de realimentación de forma óptima y, por lo tanto, haciendo óptima la estimación del estado.

En sus primeros años de vida, el algoritmo del filtro de Kalman fue utilizado como parte del sistema de guía en el programa Apollo y en la implementación de sistemas de navegación astronáutica. El filtro permitió resolver el problema de la fusión de datos asociado al combinar las mediciones del radar y del sensor inercial, logrando una aproximación global de la trayectoria de la nave.

Hoy en día, el filtro de Kalman se utiliza en los sistemas de control moderno, en el seguimiento y navegación de todo tipo de vehículos, y en el diseño de predictivo de estimación de los mismos.

A grandes rasgos, el algoritmo funciona propagando a través del tiempo la media y la covarianza del estado, siendo la primera la propia estimación del estado. Esta propagación sigue la misma dinámica que el sistema modelado. Con cada medida realizada, se corrige o actualiza tanto la media como la covarianza.

Éste es un algoritmo recursivo y, por lo tanto, no es necesario conocer toda la información generada desde su puesta en marcha, únicamente la referente a la última iteración.

Filtro de Kalman:

A continuación, se derivan las ecuaciones del filtro de Kalman. Supongamos que tenemos un sistema lineal en tiempo discreto definido de la siguiente manera:

$$\begin{aligned} x_k &= A_{k-1}x_{k-1} + B_{k-1}u_{k-1} + w_{k-1} \\ y_k &= C_k x_k + v_k \end{aligned} \qquad \textbf{(2.1)}$$

Donde x representa el estado del sistema, u las entradas al sistema e y las salidas medibles, o directamente mediciones, del sistema. Las matrices A, B, y C proporciona un modelo matemático lineal de la dinámica del sistema.

El ruido de proceso $\{w_k\}$ y el de medida $\{u_k\}$ son ruidos blancos, de media cero, incorrelados y cuyas matrices de covarianza Q_k y R_k, respectivamente, son conocidas:

$$\begin{gathered} w_k \sim (0, Q_k) \\ v_k \sim (0, R_k) \\ E[w_k w_k^T]^3 = Q_k \delta_k \\ E[v_k v_k^T] = R_k \delta_k \\ E[v_k w_k^T] = 0 \end{gathered} \quad \textbf{(2.2)}$$

Donde δ_k es la función delta de Kronecker. Nuestro objetivo es estimar x_k basándonos en nuestro conocimiento de la dinámica del sistema y en la disponibilidad de mediciones ruidosas $\{y_k\}$. La cantidad de información que esté disponible dependerá del problema que pretendamos resolver. Si hemos tomado todas las medidas hasta el instante *k*, incluyendo ese instante, podemos realizar una estimación *a posteriori*, la cual denotaremos como $\hat{x}_k^+$. Esta estimación se forma al procesar el valor esperado de $\hat{x}_k$ junto con todas las mediciones realizadas hasta el instante *k*, instante incluido:

$$\hat{x}_k^+ = E[x_k | y_1, y_2, y_3, \cdots, y_k] = Estimación\ a\ posteriori \quad \textbf{(2.3)}$$

Si, por el contrario, disponemos de todas las medidas hasta el instante *k*, sin estar este incluido, nuestra estimación es *a priori,* $\hat{x}_k^-$*:*

$$\hat{x}_k^- = E[x_k | y_1, y_2, y_3, \cdots, y_{k-1}] = Estimación\ a\ priori \quad \textbf{(2.4)}$$

Tanto $\hat{x}_k^+$ como $\hat{x}_k^-$ son estimaciones del mismo valor de x_k; simplemente $\hat{x}_k^-$ es una estimación previa a conocer la medida en el instante *k*, y $\hat{x}_k^+$ es posterior a la medición de y_k.

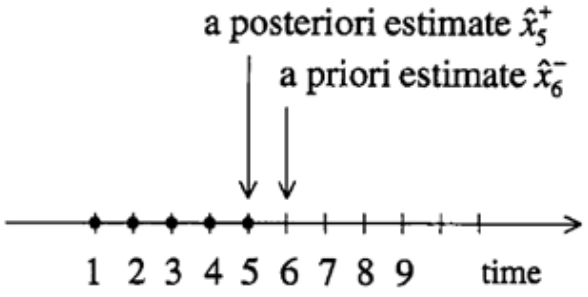


Figura 2.1. Relación entre estimación a priori y a posteriori.

[3] El operador $E[x]$ representa la media de x, $E[x] = \frac{1}{n}\sum_{i=0}^{n} x_i$

La covarianza del error de estimación P_k será:

$$P_k^- = E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^T]$$
$$P_k^+ = E[(x_k - \hat{x}_k^+)(x_k - \hat{x}_k^+)^T] \quad \textbf{(2.5)}$$

Donde P_k^- es la covarianza del error de estimación *a priori*, y P_k^+ la covarianza del error de estimación *a posteriori*.

La Figura 2.2 muestra la evolución de la estimación en cada instante. Antes de procesar las mediciones, se realiza una estimación *a priori* ($\hat{x}_k^-$) con su correspondiente covarianza (P_k^-) y una vez obtenidas las medidas, se actualiza esa estimación, formado la estimación *a posteriori* ($\hat{x}_k^+$ y P_k^+).

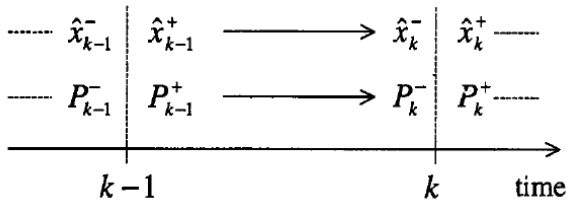


Figura 2.2. Proceso de estimación a lo largo del tiempo.

Ahora bien, ¿Cómo se propaga la estimación a lo largo del tiempo? Puesto que tanto el estado del sistema como su medida son variables aleatorias al estar influidas por el ruido de proceso (w_k) y el ruido de medida (v_k) respectivamente, el filtro de Kalman interpreta el estado estimado como si fuese la media del estado del sistema, y lo propaga a lo largo del tiempo como tal.

$$\hat{x}_k^- = E[(x_k)]$$
$$= A_{k-1}\hat{x}_{k-1}^+ + B_{k-1}u_{k-1} \quad \textbf{(2.6)}$$

Por otro lado, para averiguar cómo se propaga la covarianza de la estimación, podemos utilizar las ecuaciones (2.1) y (2.6).

$$(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^T = (A_{k-1}x_{k-1} + B_{k-1}u_{k-1} + w_{k-1} - \hat{x}_k^-)(\cdots)^T$$
$$= [A_{k-1}(x_{k-1} - \hat{x}_{k-1}^+) + w_{k-1}][\cdots]^T \quad \textbf{(2.7)}$$
$$= A_{k-1}(x_{k-1} - \hat{x}_{k-1}^+)A_{k-1}^T + w_{k-1}w_{k-1}^T$$
$$+ A_{k-1}(x_{k-1} - \hat{x}_{k-1}^+)w_{k-1}^T + w_{k-1}(x_{k-1} - \hat{x}_{k-1}^+)A_{k-1}^T$$

Por lo tanto, podemos obtener una expresión que permita calcular la covarianza de $\hat{x}_k$. Como $(x_{k-1} - \hat{x}_{k-1})$ no guarda relación con w_{k-1}, podemos llegar a la siguiente conclusión:

$$\begin{aligned} P_k^- &= E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^T] \\ &= A_{k-1}P_{k-1}^+ A_{k-1}^T + Q_{k-1} \end{aligned} \quad \textbf{(2.8)}$$

La ecuación (2.8) se denomina la ecuación discreta de Lyapunov, y es vital para el desarrollo de filtro de Kalman.

Gracias a las ecuaciones (2.6) y (2.8), sabemos cómo evoluciona o se propaga en el tiempo el estado estimado y su covarianza, es decir, que conocemos las ecuaciones de actualización de tiempo que permiten pasar del instante $(k-1)^+$ al instante k^-, tal y como se muestra en la Figura 2.2. En este paso, actualizamos el valor de nuestra estimación únicamente basándonos en nuestro conocimiento del sistema.

Habiendo desarrollado las ecuaciones de actualización de tiempo (2.6) y (2.8), necesitamos desarrollar las ecuaciones de actualización de medida, que permitan integrar las medidas realizadas a la estimación y, de manera óptima, hallar $\hat{x}_k^+$.

Para ello, buscaremos una forma estimación recursiva y lineal, que haga mínimo (óptimo) el error de estimación.

Conseguir un algoritmo que permita actualizar la información *a priori* una mediante combinación lineal y de una manera recursiva reducirá es coste computacional y aumentara la sencillez del filtro, pues solo será necesario conocer la última iteración

Un estimador recursivo lineal puede describirse así:

$$\begin{aligned} \hat{x}_k &= \hat{x}_{k-1} + K_k(y_k - C_k\hat{x}_{k-1}) \\ y_k &= C_k x_k + v_k \end{aligned} \quad \textbf{(2.9)}$$

que adaptado al filtro de Kalman resultaría:

$$\begin{aligned} \hat{x}_k^+ &= \hat{x}_k^- + K_k(y_k - C_k\hat{x}_k^-) \\ y_k &= C_k x_k + v_k \end{aligned} \quad \textbf{(2.10)}$$

Es decir, que si en (2.9) pasábamos del instante *k-1* al *k,* en (2.10) pasamos de una estimación *a priori* a una estimación *a posteriori*, que incluye más información de la estimación del estado para un mismo instante.

El criterio a seguir para la búsqueda de una estimación óptima será el de mínimos cuadrados, pues establece que el valor más probable de una cantidad desconocida es aquel que hace que la suma de los cuadrados de las diferencias entre el valor esperado y el computado sea mínima.

Necesitamos encontrar una ganancia del estimador K_k que haga óptima la estimación de x_k, siguiendo el criterio antes comentado. Por ello, el valor de K_k deseado será el que haga mínima la covarianza del error de estimación.

$$\begin{aligned} J_k &= E[(x_1 - \hat{x}_1^+)^2] + \cdots + E[(x_n - \hat{x}_n^+)^2] \\ &= E\left({\varepsilon_{x1,k}^+}^2 + \cdots + {\varepsilon_{xn,k}^+}^2\right) \\ &= E\left({\varepsilon_{x,k}^+}^T \varepsilon_{x,k}^+\right) \\ &= TrP_k^+ \end{aligned} \quad \textbf{(2.11)}$$

La expresión J_k recibe el nombre de función de coste. ε_k es el error de estimación en el instante *k*, y P_k es la covarianza del error de estimación, y si la desarrollamos desde su definición, en combinación con la ecuación (2.9), podemos obtener una manera de hallarla de forma recursiva:

$$\begin{aligned} P_k^+ &= E\left(\varepsilon_{x,k}^+ {\varepsilon_{x,k}^+}^T\right) \\ &= E\{[(I - K_k C_k)\varepsilon_{x,k}^- - K_k v_k][\cdots]^T\} \\ &= (I - K_k C_k)E\left(\varepsilon_{x,k}^- {\varepsilon_{x,k}^-}^T\right)(I - K_k C_k)^T - K_k E\left(v_k\, {\varepsilon_{x,k}^-}^T\right)(I - K_k C_k)^T \\ &\quad - (I - K_k C_k)E\left(\varepsilon_{x,k}^- {v_k}^T\right)K_k^T + K_k E(v_k {v_k}^T)K_k^T \end{aligned} \quad \textbf{(2.12)}$$

Puesto que el error de estimación a priori $\varepsilon_{x,k}^-$ es independiente del ruido de medida en el instante *k*, sabemos que:

$$E\left(v_k\, {\varepsilon_{x,k}^-}^T\right) = E(v_k)E\left(\varepsilon_{x,k}^-\right) = 0 \quad \textbf{(2.13)}$$

Combinando (2.12) y (2.13), obtenemos la siguiente expresión:

$$P_k^+ = (I - K_k C_k)P_k^-(I - K_k C_k)^T + K_k R_k K_k^T \quad \textbf{(2.14)}$$

Ya hemos encontrado una forma recursiva de determinar la covarianza del error de estimación, que además nos adelanta, de forma intuitiva, que a mayor ruido de medida R_k, mayor es la incertidumbre en la estimación.

Ahora necesitamos encontrar un valor de K_k que haga la función de coste J_k tan pequeña como sea posible. Para ello, realizaremos el siguiente desarrollo:

$$\begin{aligned} \frac{\partial J_k}{\partial K_k} &= 2(I - K_k C_k)P_k^-(-C_k^T) + 2K_k R_k = 0 \\ K_k R_k &= (I - K_k C_k)P_k^- C_k^T \\ K_k(R_k + C_k P_k^- C_k^T) &= P_k^- C_k^T \\ K_k &= P_k^- C_k^T (R_k + C_k P_k^- C_k^T)^{-1} \end{aligned} \quad \textbf{(2.15)}$$

Mediante (2.15) obtenemos el valor que hace óptima la corrección de la estimación a priori. Podemos simplificar (2.14) sustituyendo el valor de K_k. Si definimos la variable auxiliar $S_k = C_k P_k^- C_k^T + R_k$ obtenemos:

$$\begin{aligned} P_k^+ &= (I - P_k^- C_k^T S_k^{-1} C_k) P_k^- (I - P_k^- C_k^T S_k^{-1} C_k)^T + P_k^- C_k^T S_k^{-1} R_k P_k^- C_k {S_k^{-1}}^T \\ &= P_k^- - P_k^- C_k^T S_k^{-1} C_k P_k^- - P_k^- C_k^T S_k^{-1} C_k P_k^- + P_k^- C_k^T S_k^{-1} C_k P_k^- C_k^T S_k^{-1} C_k P_k^- \\ &\quad + P_k^- C_k^T S_k^{-1} R_k S_k^{-1} C_k P_k^- \\ &= P_k^- - 2P_k^- C_k^T S_k^{-1} C_k P_k^- + P_k^- C_k^T S_k^{-1} S_k S_k^{-1} C_k^T P_k^- \\ &= P_k^- - 2P_k^- C_k^T S_k^{-1} C_k P_k^- + P_k^- C_k^T S_k^{-1} C_k P_k^- \\ &= P_k^- - P_k^- C_k^T S_k^{-1} C_k P_k^- \\ P_k^+ &= (I - K_k C_k) P_k^- \end{aligned} \quad \textbf{(2.16)}$$

Con esta última ecuación, el filtro de Kalman queda completamente desarrollado en su formulación básica.

Resumiendo, el algoritmo resulta:

- El sistema viene dado por:

$$\begin{gathered} x_k = A_{k-1} x_{k-1} + B_{k-1} u_{k-1} + w_{k-1} \\ y_k = C_k x_k + v_k \\ w_k \sim (0, Q_k) \\ v_k \sim (0, R_k) \\ E[w_k w_k^T] = Q_k \delta_k \\ E[v_k v_k^T] = R_k \delta_k \\ E[v_k w_k^T] = 0 \end{gathered} \quad \textbf{(2.17)}$$

- El filtro de Kalman viene dado por las siguientes ecuaciones:

$$\begin{gathered} P_k^- = A_{k-1} P_{k-1}^- A_{k-1}^T + Q_{k-1} \\ K_k = P_k^- C_k^T (R_k + C_k P_k^- C_k^T)^{-1} \\ \hat{x}_k^- = A_{k-1} \hat{x}_{k-1}^+ + B_{k-1} u_{k-1} = estimacion\ a\ priori \\ \hat{x}_k^+ = \hat{x}_k^- + K_k (y_k - C_k \hat{x}_k^-) = estimación\ a\ posterirori \\ P_k^+ = (I - K_k C_k) P_k^- \end{gathered} \quad \textbf{(2.18)}$$

Finalizado el desarrollo teórico, hemos de recalcar algunas cuestiones sobre el algoritmo:

- Si $\{w_k\}$ y $\{v_k\}$ son ruidos blancos, de distribución Gaussiana, incorrelados y de media cero, el filtro de Kalman es la solución óptima.
- Si $\{w_k\}$ y $\{v_k\}$ son ruidos blancos, incorrelados y de media cero, el filtro de Kalman es la mejor solución lineal para abordar el problema.

- Si $\{w_k\}$ y $\{v_k\}$ son ruidos coloreados y correlados, el filtro de Kalman admite modificaciones solucionar el problema.
- Si el sistema no el lineal, existen formulaciones del filtro de Kalman que permiten obtener una solución aproximada.

En el 2.4 se abordará la implementación en Vivado HLS del algoritmo expuesto en esta sección, y se discutirán algunas consideraciones realizadas a la hora de llevarlo a cabo.

2.2. Vivado HLS

Introducción

Dispositivos de hardware reconfigurable como las FPGAs ofrecen una gran capacidad de cómputo debido al gran paralelismo de los sistemas que se pueden implementar en ellos. No obstante, la metodología tradicional de diseño, basada en lenguajes de descripción hardware (VHDL y Verilog), es costosa y complicada, pues obliga al diseñador a desarrollar las aplicaciones en un nivel de abstracción bajo. De ahí nace la búsqueda de herramientas alternativas que simplifiquen las metodologías y el flujo de diseño para FPGAs, como son las herramientas de síntesis de comportamiento.

La primera generación de estas herramientas de síntesis de comportamiento nació en 1994, y usaban como lenguajes de entradas VHDL y Verilog. El nivel de abstracción era de procesos parcialmente síncronos. Estas herramientas de síntesis nunca llegaron a asentarse en el mercado, pues su metodología no podía igual el diseñar las aplicaciones en el nivel de abstracción que permitían los lenguajes de alto nivel.

En 2004 surgió la nueva generación de herramienta de síntesis de comportamiento, que permitían la síntesis de circuitos electrónicos especificados en lenguaje C a diseños RTL. El cambio de lenguaje ofrecía la abstracción precisa, flexibilidad en el código y un mayor poder de expresión.

Vivado HLS (High Level Synthesis) es una herramienta desarrollada por Xilinx que permite realizar diseños hardware para la implementación en FPGAs codificando en lenguajes de alto nivel (C/C++ y SystemC). Es decir, dada una aplicación o especificación codificada en estos leguajes, la herramienta hace una interpretación de su comportamiento, y genera un diseño hardware que lo imita. Este diseño es altamente parametrizable, pudiendo el usuario introducir directivas que desemboquen en arquitecturas con un distinto grado de paralelismo, recursos consumidos y potencia requerida.

La Síntesis de Alto Nivel tiende puentes entre los mundos del diseñador (hardware) y el programador (software), lo que implica ciertos beneficios:

- Aumenta la productividad del diseñador, pues le permite trabajar a un mayor nivel de abstracción al crear el diseño hardware sin tener que utilizar una metodología de descripción hardware.

- Permite mejorar los sistemas de computación de los programadores, ya que las partes más críticas, en cuanto a coste de computación se refiere, pueden ahora implementarlas en una FPGA.

Utilizar una metodología basada en Vivado HLS permite:

- Diseñar el hardware a un nivel de abstracción mayor, ahorrando tiempo de desarrollo.
- Verificar la funcionalidad de tu diseño en lenguaje de alto nivel, mucho más rápido que en los lenguajes de descripción hardware tradicionales
- Crear múltiples implementaciones de tu algoritmo en C, lo que permite explorar y comparar las múltiples soluciones y así hallar la más óptima.
- El código C es fácilmente exportable a diferentes FPGAs. Cada solución del proceso de síntesis está orientado a un dispositivo en concreto. Cambiar de dispositivo es tan fácil como modificar los ajustes de la solución y volver a sintetizarla.

Vivado HLS ofrece, siguiendo las líneas que marcan las nuevas tecnologías que aparecen en el mercado, la posibilidad de integrar sus productos de salida en Vivado para su análisis o uso. Concretamente, una de las opciones más interesantes es la posibilidad de importar los diseños realizados en HLS como periféricos para sistemas basados en microprocesadores (Zynq y Microblaze).

Etapas de la Síntesis de Alto Nivel

La síntesis del código de alto nivel se realiza en los siguientes pasos:

- Programación de las operaciones:

  Determina que operaciones se realizan en cada ciclo de reloj basándose en la frecuencia del mismo, el tiempo que tardaría cada operación en función de la FPGA a la que está orientado el proyecto y las directivas de optimización.

- Enlazado:

  Determina el recurso hardware que será utilizado en cada operación programada.

- Extracción de la lógica de control:

  Genera una máquina de estados finitos (FSM) que permita secuenciar las operaciones en el diseño RTL.

  La Figura 2.3 muestra los dos primeros pasos para un diseño sencillo de síntesis. La función de alto nivel *foo* recibe como argumentos 4 variables de 8 bits cada una, y retorna el valor del producto entre los dos primeros argumentos más la suma de los dos últimos.

```
int foo(char x, char a, char b, char c) {
  char y;
  y = x*a+b+c;
  return y
}
```

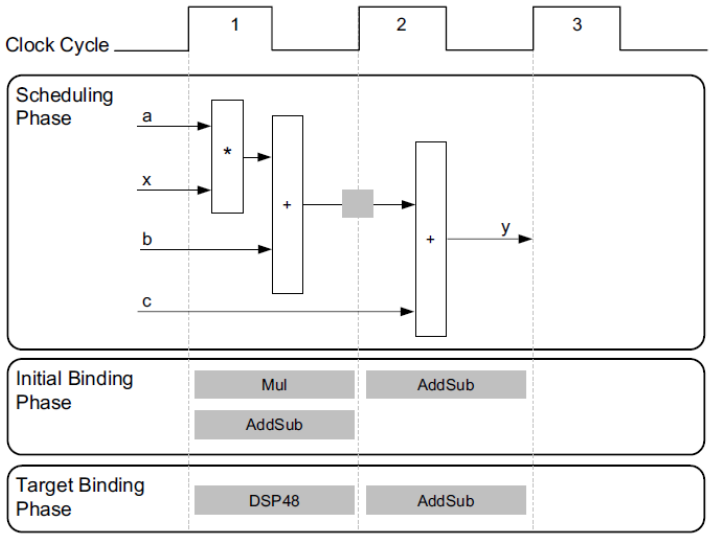


Figura 2.3. Programado y enlazado para la función dada.

La multiplicación y la primera suma es capaz de realizarla en un único flanco de reloj, y como no puede incluir la segunda suma, almacena el resultado en un buffer, simbolizado por un cuadrado gris, para en el siguiente ciclo de reloj realizar la operación. Para la implementación de las operaciones de multiplicación y suma emplea el recurso hardware DSP48 (multiplicador hardware), y para la segunda suma un sumador.

La función *foo* que ahora aparece en la Figura 2.4 introduce dentro de un bucle la funcionalidad mostrada en la Figura 2.3. Esta vez, uno de los argumentos es un array de enteros, y los resultados los almacena en otro array. HLS genera una máquina de estado capara controlar el flujo del diseño.

```
void foo(int in[3], char a, char b, char c, int out[3]) {
  int x,y;
  for(int i = 0; i < 3; i++) {
    x = in[i];
    y = a*x + b + c;
    out[i] = y;
  }
}
```

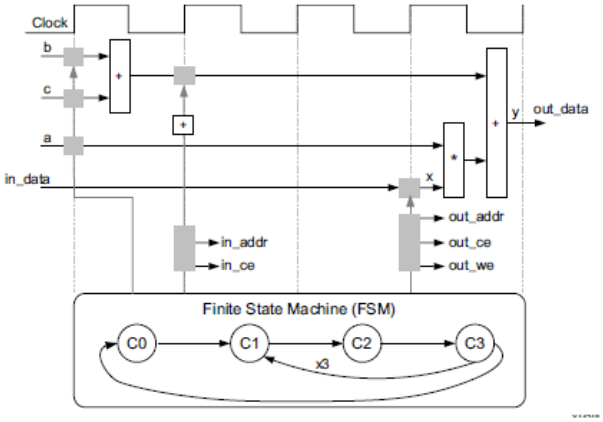


Figura 2.4. lógica de control para la función dada.

HLS hará la siguiente interpretación del código C a sintetizar:

- La síntesis sigue una jerarquía. En cada proyecto es necesario especificar la función de mayor jerarquía (top-function). Cada función especificada en C se sintetizará como un bloque o instancia dentro de la jerarquía RTL. De esta manera, la top-function determina el encapsulado exterior del diseño final.
- Los argumentos de la top-function serán implementados como puertos del bloque RTL. El número de puertos destinados a la implementación de cada argumento dependerá de tipo de argumento (un solo dato, arrays, estructuras), el recurso hardware con el que se va a implementar (RAM, AXI-Stream, . . .) y a las interfaces elegidas (ap_cntrl_hs, ap_cntrl_none, s_axilite, . . .).
- Los bucles se mantienen enrollados por defecto. Esto quiere decir que en la síntesis se creará únicamente la lógica para una sola iteración, y esta se ejecutará el número de veces necesaria. Mediante las directivas de optimización es posible desenrollar los bucles, lo que permite realizar todas las iteraciones en paralelo.
- Los arrays de variables se implementarán mediante memorias RAM. Si alguno de los argumentos de la función top son arrays, la síntesis creara los puertos necesarios para acceder a una memoria RAM fuera del diseño.

La Síntesis de Alto Nivel genera la implementación óptima del código C basándose en su comportamiento, las restricciones y todas las directivas que se apliquen. Las directivas permiten modificar y controlar el comportamiento del bloque y los puertos de

entrada/salida, y generar así varias implementaciones que permitan cumplir los requisitos del usuario.

Para evaluar si esos requisitos se han cumplido, Vivado HLS suministra reportes de la síntesis que permiten evaluar el diseño RTL. En esa información se incluye:

- Área utilizada: cantidad de recursos hardware utilizados para la implementación basados en los recursos básicos disponibles en la FPGA. Estos recursos son la look-up tables (LUTs), los registros, los bloques RAM y los DSP48s.
- Latencia: número de ciclos de reloj que el diseño necesita para computar todos los datos de salida.
- Intervalo de iniciación: número de ciclos de reloj ante de que el bloque pueda aceptar nuevas entradas.
- Latencia de iteración del bucle: número de ciclos de reloj que se emplea en realizar una iteración de un bucle.
- Intervalo de iniciación del bucle: número de ciclos de reloj que tarda la siguiente iteración del bucle en empezar a procesar datos.
- Latencia del bucle: número de ciclos de reloj que se emplean en completar todas las iteraciones del bucle.

HLS también permite simular el producto de la síntesis junto con el código de alto nivel, lo que permite comprobar si el diseño RTL se comporta como se ha descrito en C.

La herramienta también permite exportar el diseño RTL realizado en la síntesis como un IP al catálogo de IPs de Vivado.

Metodología de trabajo de Vivado HLS

En los diseños realizados con Vivado HLS se sigue el siguiente flujo de trabajo:

1. Compilar, ejecutar y depurar el código C.
2. Sintetizar el algoritmo C en una implementación RTL. Opcionalmente (y muy recomendado), se pueden aplicar directivas que modifican el hardware implementado.
3. Analizar los reportes que suministra HLS.
4. Verificar la implementación RTL
5. Empaquetar la implementación RTL en una de las múltiples formas de IP.

Entradas y salidas

La herramienta requiere los siguientes ficheros de entrada:

- La función descrita en C, C++ o SystemC. Es la entrada primaria a Vivado HLS, y puede contener una jerarquía de sub-funciones.
- Restricciones (Constrains). Son necesarias y contienen el periodo del reloj, su incertidumbre (por defecto el 12.5% del periodo si no se ha especificado) y la FPGA a la que va orientada.
- Directivas. Son optativas, y dirigen el proceso de síntesis a una implementación concreta.
- C test bench y ficheros asociados. Vivado HLS utiliza el test bench C para simular la función C antes de sintetizarla y para comprobar automáticamente las salidas de la implementación RTL cuando se utiliza la RTL/C co-simulación.

Tras el proceso de síntesis, Vivado HLS devuelve los siguientes archivos:

- Ficheros en formato HLD que describen la implementación RTL. Es la salida primaria de la herramienta. Utilizando la síntesis lógica se puede sintetizar el RTL en una implementación al nivel de puertas lógicas y generar el fichero bitstream. EL diseño RTL está disponible tanto en VHDL como en Verilog.
  Vivado HLS empaqueta los archivos de la implementación RTL como un bloque IP ser utilizado en otras herramientas de Xilinx y añadirlos a un sistema hardware.
- Ficheros en formato SystemC. Estos ficheros se proporcionan principalmente como modelos de simulación y no se incluyen en los IPs empaquetados.
- Reportes. Contienen la información de los procesos de síntesis.

En la Figura 2.5 se muestra gráficamente el flujo de archivos de entrada y salida cuando se trabaja con Vivado HLS.

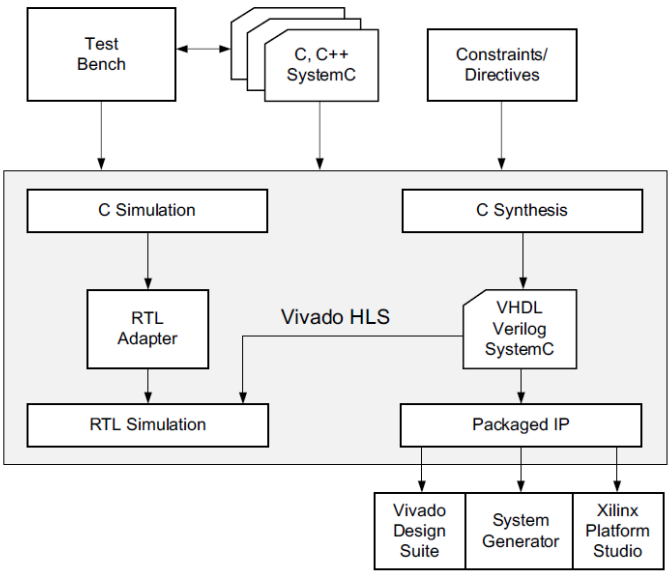


Figura 2.5. Flujo de diseño en Vivado HLS.

Test bench, lenguajes soportados y librerías C

En todos los programas descritos en C, la función de más alto nivel jerárquico (top-function) es aquella que se llama *main()*. Utilizando la metodología de HLS se puede especificar como top-function para la síntesis cualquier función de jerarquía inferior a *main()*. No es posible seleccionar la función *main()* como top function para la síntesis. A parte de éstas, existen otras reglas:

- Solo se permite una top-function para la síntesis.
- Todas las funciones bajo la top-function en la jerarquía también son sintetizadas.
- Si se quiere sintetizar también funciones por encima de top-function, debes conjuntarlas todas en un único nivel jerárquico, que además debe ser el más alto.

Para ahorrar tiempo y no sintetizar un código C que no funciona, HLS incluye la posibilidad de depurarlo mediante el test bench. El test bench contiene la función *main()*, desde la que se hace una llamada a top-function de la síntesis.

Esta forma de depurar es propia del diseño hardware. En el test bench se generan todos los estímulos que se estimen oportunos para poner a prueba la top-function de la síntesis, comprobando el correcto funcionamiento de la función al examinar las salidas.

Vivado HLS soporta casi en su totalidad los lenguajes ANSI-C, C++ y SystemC, salvando las siguientes operaciones:

- Asignación dinámica de memoria. Las FPGAs tienen recursos fijos, y no soportan la creación dinámica y liberación de memoria.
- Operaciones de sistemas operativos. Todos los datos de las FPGAs deben ser leídos de los puertos de entrada o escritos en los de salida. Las operaciones de sistemas operativos como la lectura y escritura de ficheros y las consultas al OS como la fecha y la hora no están soportadas.

Las librerías C contienen funciones y construcciones optimizadas para la implementación en FPGAs. El uso de estas librerías asegura una alta calidad en los resultados, es decir, que la salida final es un diseño de altas prestaciones que asegura el consumo óptimo de los recursos. Estas librerías son:

- Tipos de datos de recisión arbitraria.
- Operaciones matemáticas
- Funciones de vídeo

- Funciones de IPs de Xilinx, incluyendo la transformada rápida de Fourier (FFT) y la respuesta finita al impulso (FIR)
- Funciones de recursos FPGAs que ayudan a maximizar el uso de los recursos de registros de desplazamiento LUT

Síntesis, optimizaciones y análisis

Vivado HLS es una aplicación basada en proyectos. Cada proyecto incluye un conjunto de códigos C y puede tener múltiples soluciones. Cada solución puede tener distintas restricciones y diferentes directivas. Los resultados de cada solución se pueden analizar y comparar.

Los pasos a seguir cuando se diseña con Vivado HLS son:

1. Crear un proyecto con una solución inicial.
2. Verificar que el código C se ejecuta sin errores.
3. Lanzar la síntesis y obtener un conjunto de resultados.
4. Analizar esos resultados.

Después del análisis, se puede crear una nueva solución con diferentes restricciones y directivas y sintetizarla. Este proceso se puede repetir hasta conseguir un diseño con las prestaciones deseadas.

Optimización

Vivado HLS permite aplicar diferentes directivas de optimizaciones, incluyendo la siguientes:

- Ordenar a una tarea a que se ejecute de forma segmentada (pipeline), de manera que pueda empezar la siguiente ejecución sin que la actual haya acabado.
- Especificar una latencia para la finalización de funciones, bucles o regiones.
- Especificar el máximo número de recursos a emplear.
- Sobrecargar las dependencias de datos inherentes e implícitas en el código para permitir determinadas operaciones.
- Seleccionar el protocolo de entradas/salidas para asegurar que el diseño final pueda conectarse a otros bloques hardware con el mismo protocolo.

Análisis

Cuando la síntesis se completa, Vivado HLS genera automáticamente reportes de la síntesis, que ayudan a comprender las prestaciones de la implementación.

Verificación RTL

El test bench C también puede utilizarse para comprobar si el diseño RTL es funcionalmente idéntico al código C original. Vivado HLS utiliza el valor de retorno de la simulación C y la C/RTL co-simulación para comprobar si tanto el diseño RTL como la función C funcionan igual. Si ambos devuelven cero, HLS interpreta que el funcionamiento del RTL es válido e igual al de la función original. Si no, HLS comunica que la simulación ha fallado.

Vivado HLS soporta los siguientes simuladores RTL:

- Vivado Simulator (XSim).
- QuestaSim simulator.
- VCS.
- NCSim.
- ISE Simulator (ISim).
- Riviera.
- Open SystemC Initiative (OSCI).

Exportación RTL

Los archivos de salida RTL pueden ser exportados y empaquetados como un IP en cualquiera de los siguientes formatos de Xilinx, para su en las diferentes herramientas de la marca:

- Vivado IP Catalog. Importar al catálogo de IPs de Vivado para utilizarlos en Vivado Desing Suite.
- System Generator para DSP. Importar al System Generator y usarlos tanto en ISE Desing Suite como en Vivado Desing Suite.
- Pcore para Embedded Developement Kit (EDK). Importar al Xilinx Platform Studio (XPS).

2.3. System on Chip. ZedBoard.

Zynq-7000 all programable Soc.

La solución final del filtro de Kalman generada por Vivado HLS se ha implementado sobre la plataforma ZedBoard de Xilinx, perteneciente a la familia Zynq-700.

Esta familia de dispositivos está basada en la arquitectura “All programable SoC”. Estos productos integran un sistema de procesamiento (PS) basado en el Cortex-A9 de ARM y lógica programable (PL), con un nivel de integración de 28 nm, en un mismo dispositivo.

Zynq-7000 All Programmable SoC								
	Device Name	**Z-7010**	**Z-7015**	**Z-7020**	**Z-7030**	**Z-7035**	**Z-7045**	**Z-7100**
	Part Number	**XC7Z010**	**XC7Z015**	**XC7Z020**	**XC7Z030**	**XC7Z035**	**XC7Z045**	**XC7Z100**
Processing System	Processor Core	Dual-core ARM® Cortex™-A9 MPCore™ with CoreSight™						
	Processor Extensions	NEON™ & Single / Double Precision Floating Point for each processor						
	Maximum Frequency	667 MHz (-1); 766 MHz (-2); 866 MHz (-3)			667 MHz (-1); 800 MHz (-2); 1 GHz (-3)			667 MHz (-1) 800 MHz (-2)
	L1 Cache	32 KB Instruction, 32 KB data per processor						
	L2 Cache	512 KB						
	On-Chip Memory	256 KB						
	External Memory Support(1)	DDR3, DDR3L, DDR2, LPDDR2						
	External Static Memory Support(1)	2x Quad-SPI, NAND, NOR						
	DMA Channels	8 (4 dedicated to Programmable Logic)						
	Peripherals(1)	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO						
	Peripherals w/ built-in DMA(1)	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO						
	Security(2)	RSA Authentication, and AES and SHA 256-bit Decryption and Authentication for Secure Boot						
Processing System to Programmable Logic Interface Ports (Primary Interfaces & Interrupts Only)		2x AXI 32b Master 2x AXI 32-bit Slave 4x AXI 64-bit/32-bit Memory AXI 64-bit ACP 16 Interrupts						
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix®-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Kintex®-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA
	Programmable Logic Cells (Approximate ASIC Gates)(3)	28K Logic Cells (~430K)	74K Logic Cells (~1.1M)	85K Logic Cells (~1.3M)	125K Logic Cells (~1.9M)	275K Logic Cells (~4.1M)	350K Logic Cells (~5.2M)	444K Logic Cells (~6.6M)
	Look-Up Tables (LUTs)	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Extensible Block RAM (# 36 Kb Blocks)	240 KB (60)	380 KB (95)	560 KB (140)	1,060 KB (265)	2,000 KB (500)	2,180 KB (545)	3,020 KB (755)
	Programmable DSP Slices (18x25 MACCs)	80	160	220	400	900	900	2,020
	Peak DSP Performance (Symmetric FIR)	100 GMACs	200 GMACs	276 GMACs	593 GMACs	1,334 GMACs	1,334 GMACs	2,622 GMACs
	PCI Express® (Root Complex or Endpoint)(4)	—	Gen2 x4	—	Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs						
	Security(2)	AES and SHA 256b for Boot Code and Programmable Logic Configuration, Decryption, and Authentication						

Figura 2.6. Familia Zynq-7000

La familia Zynq-700 ofrece la flexibilidad y escalabilidad de una FPGA, siendo a su vez tan potente y fácil de usar como los ASICs y ASSPs tradicionales. Todos los miembros de la familia tienen integrado en el silicio el mismo PS, mientras que los recursos disponibles en la PL y los recursos de entrada/salida varía entre los distintos dispositivos.

La arquitectura de Zynq-7000 permite la implementación de lógica a medida en la PL y de software a medida en el PS. La integración de lógica programable junto con un procesador permite niveles de actuación que sistemas basados en dos chips no pueden alcanzar debido al limitado ancho de banda de entradas/salidas, su latencia y la potencia requerida.

El PS y la PL pertenecen a diferentes dominios de potencia, lo que permite al usuario apagar la PL y así reducir el consumo de energía. El procesador siempre se inicializa primero, y es el encargado de inicializar la configuración de la PL.

Es necesario recalcar que, a diferencia de otras soluciones que ofrece Xilinx para los SoC (MicroBlaze), el PS de Zynq es un hard-core, es decir, el procesador está integrado en el silicio y es, por decirlo de alguna manera, fijo. Por lo tanto, la arquitectura de Zynq-7000 está pensada para que el PS pueda funcionar sin necesidad de una FPGA que le proporcione soporte.

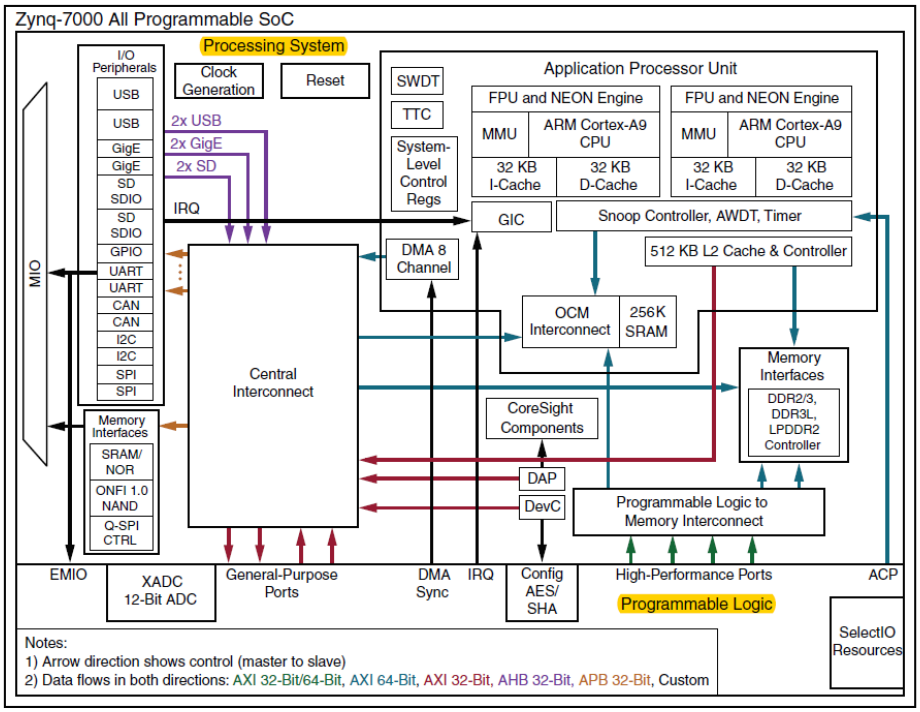


Figura 2.7. Esquema funcional Zynq-7000

Como se observa en la Figura 2.7, Zynq se divide en dos grandes conjuntos, el PS y la PL. En los siguientes apartados se detallan los componentes básicos de ambas partes.

Sistema de procesamiento (PS):

Como se muestra en la Figura 2.7, en el PS podemos distinguir cuatro grandes conjuntos:

- Application processor unit (APU)
- Memory interface
- I/O Peripherals
- Interconect

Applicaton processor unit (APU)

Los aspectos principales de la APU son:

- Dual-core ARM Cortex-A9 MPCores. Los núcleos operan en un margen de frecuencias entre 866 MHz y 667 MHz. Pueden funcionar en modo “single processor”, “dual processor” y “asymmetric dual processor”. Ambos poseen una FPU (single y double) que funciona por encima de 2 MFLOPS/MHz.
  Cada procesador cuenta con 32 KB de memoria caché de nivel 1 junto con una unidad MMU.
- EL Accelerator coherency port (ACP) asegura accesos coherentes desde la PL hasta la memoria de la CPU.
- 512 KB de memoria caché de nivel dos compartida por ambos núcleos.
- 256 KB de memoria RAM on-chip (Dual-Port)
- DMA de 8 canales.
- General interrupt controller (GIC).
- 3 timers Watch dog (uno por cada CPU y otro para el sistema completo).
- 2 triple timers/contadores (TTC).

Memory interfaces

La unidad de interfaz de memoria incluye un controlador de memoria dinámica y módulos de interfaz de memoria estática:

- Interfaces de memoria dinámica: el controlador de la memoria DDR puede ser configurado para suministrar accesos de 16 o 32 bits a un espacio de direcciones de memoria de 1GB usando una configuración rango único de 8, 16 o 32 bits. Tanto el controlador como el dispositivo físico de encuentran en el PS, junto con un set de pines dedicados.

El controlador de memoria es un dispositivo multi-puerto, y permite a la PL y al PS tener accesos compartidos a memoria común. Presenta 4 puertos AXI esclavos para ello. Todos ellos son de 64 bits. Uno está dedicado a ambas CPUs, dos al acceso desde la PL y el ultimo es compartido por el resto de maestros AXI a través del bloque central interconect.

- Interfaces de memoria estática: el controlador está preparado para soportar las siguientes memorias externas:
    - SRAM de 8 bits hasta un máximo de 64 MB
    - NOR flash paralela de 8 bits hasta un máximo de 64 MB
    - NAND flash ONFi 1.0 con un bit ECC
    - NOR flash serie SPI de 1, 2, 4 (quad-SPI) y 8 (two quad-SPI) bits.

Periféricos I/O (IOP)

La unidad IOP contiene los periféricos dedicados a la comunicación de datos. Los aspectos claves de estos periféricos son:

- 2 10/100/1000 tri-mode Ethernet MAC.
- 2 USB 2.0
- 2 CAN 2.0B
- 2 SD/SDIO
- 2 full-duplex puertos SPI
- 2 UARTs
- 2 interfaces I2C esclavos y maestros.
- Hasta 118 bits GPIO

Los IOP se conectan con dispositivos externos a través de 54 pines multiusos (MIO). Cada periférico puede asignarse a un grupo pre definido de pines, permitiendo el uso simultáneo de los dispositivos. Sin embargo, los MIO son insuficientes si se quiere utilizar todos los IOP, por lo que muchos de los interfaces del IOP están disponibles en la PL.

Interconect

La APU, los interfaces de memoria y los IOP están conectados entre sí y a la PL a través del ARM AMBA AXI interconect. Este dispositivo es no bloqueante, y permite múltiples transacciones maestro-esclavo simultaneas.

Lógica programable (PL)

Los aspectos básicos de la lógica programable de la que dispone Zynq-7000 son los siguientes:

- CLBs: 8 LUTs por CLB para implementación de lógica aleatoria o memoria distribuida. La memoria basada en LUTS puede ser configurada como RAM de 64x1 o 32x2 bits o como registros de desplazamiento. Cada CLB contiene 16 flip-flops y 2 sumadores de 4 bits en cascada para la implementación de funciones aritméticas.
- 36 Kb de memoria RAM true dual-port. Hasta un máximo de 36 bits de ancho. Puede configurarse como bloques RAM duales de 18 Kb.
- DSP slices: 18x25 multiplicadores con signo, con acumuladores de 48 bits
- Bloques I/O programables, pudiendo soportar diferentes estándares (LVCMOS, LVDS y SSTL)
- Dos ADCs de 12 bits (XADC) que permiten medir la tensión y la temperatura de la pastilla, y con 17 canales de entrada diferenciales.

ZedBoard

La tarjeta empleada en la implementación de nuestro diseño es la ZedBoard. Se trata de una tarjeta de evaluación, orientada a el aprendizaje y el desarrollo de diseños. El núcleo de la tarjeta es el Zynq-7000 AP SoC XC7Z020-CLG484-1.

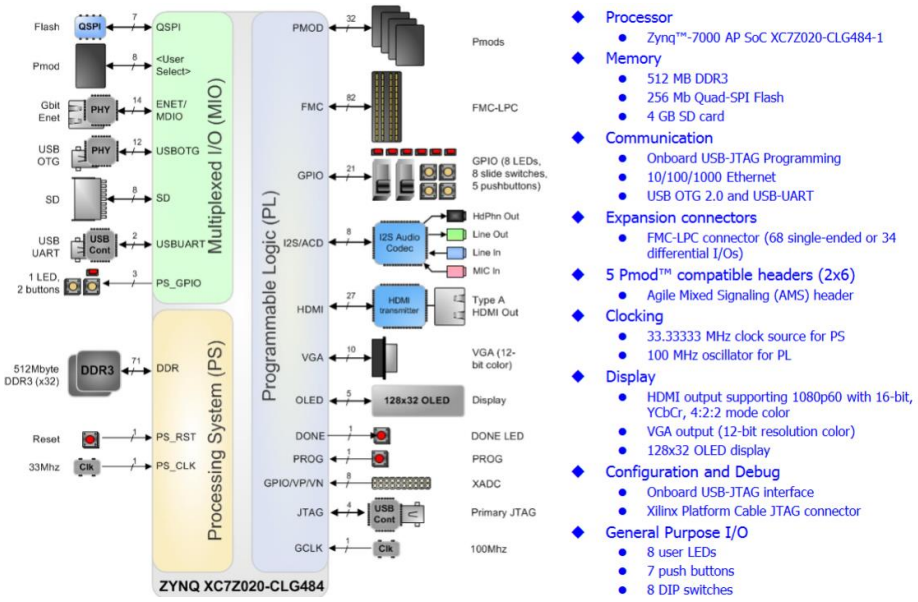


Figura 2.8. Diagrama de bloques de la ZedBoard.

En un sistema como el que propone Zynq, en el que gran parte de la plataforma (PL) no está definida y puede variar de un diseño a otro, es necesario contar con un patrón o protocolo que permita interconectar todo el sistema y dotarlo de cierta coherencia. En Zynq, esa función la realiza el bus AXI4.

La tarjeta de desarrollo ZedBoard monta una versión reducida del bus AXI4, el AXI4-LITE, con el que podemos comunicarnos con el PS desde la PL. Cualquier diseño con el que queramos extender las funcionalidades del PS debe entenderse con el bus AXI4-LITE, incluso los diseños generados con Vivado HLS.

2.4. Implementación del filtro de Kalman

Introducción

En los capítulos anteriores se han tratado los aspectos básicos y necesario para la realización del diseño. Sabemos el qué (filtro de Kalman), el cómo (Vivado HLS), y el dónde (ZedBoard), pero antes de comenzar, debemos hacer algunas consideraciones.

Éste trabajo pretende abordar el desarrollo de diseños RTL utilizando la herramienta de síntesis de Vivado, que permite la codificación en lenguaje de alto nivel. En concreto buscamos que ese diseño tenga la exigencia necesaria que haga, al menos, plantearse su utilización. Por ello, hemos tratado de abordar la implementación del filtro de Kalman.

Todas las posibles formulaciones del algoritmo de Kalman tienen algo en común, y es la necesidad de invertir por lo menos una matriz, algo nada trivial para implementar en hardware. Es por tanto un candidato ideal para Vivado HLS.

En este capítulo se desarrolla la metodología y los diseños seguidos para la implementación del filtro de Kalman para un sistema lineal invariante en el tiempo (LTI).

Partiendo de los conocimientos expuestos en el apartado 2.1, se propondrá un sistema LTI sencillo basado en una red RLC, que permita realizar un estudio teórico y una primera aproximación del filtro en Matlab.

Teniendo en cuenta nuestro objetivo es un bloque hardware, debemos valorar si los datos se codificarán en coma flotante (mejor precisión, pero mayor consumo de recursos) o en coma fija (ahorramos recursos a costa de perder precisión en el cálculo). La clase *fi* de Matlab nos ayudará a simular y ponderar ambas soluciones.

Una vez estemos satisfechos con los resultados obtenidos, podremos empezar a diseñar el bloque en HLS. La herramienta nos permite depurar nuestro código de alto nivel, de manera que se pueda comprobar su funcionalidad antes de la síntesis.

Posteriormente, se realizarán múltiples soluciones, tanto en coma fija como en coma flotante, aplicando diferentes directivas de optimización y así, obtener el diseño óptimo que se ajuste a nuestras necesidades.

El diseño definitivo se exportará a Vivado y SDK para generar una aplicación para la tarjeta de evaluación ZedBoard. Este es el último paso en el proceso de diseño, y nos permitirá conocer la en detalle el periférico generado por HLS.

Como extra, contamos con dos sistemas reales, que servirán de sujeto de pruebas para nuestro bloque. El primero de ellos es el robot de propósito general Pioneer P3-DX

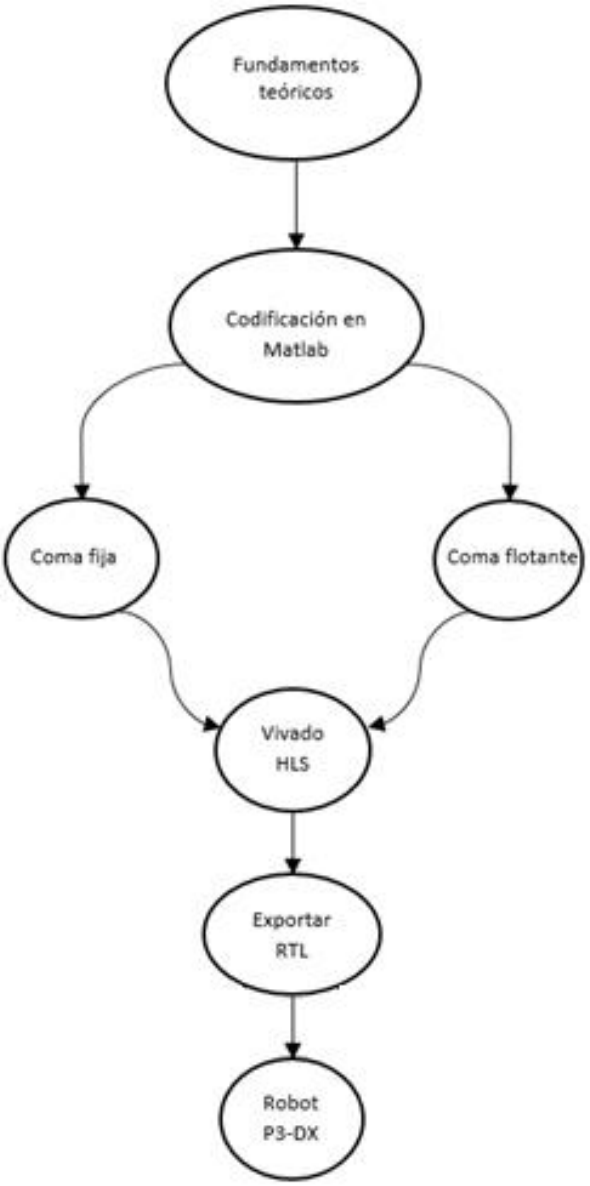
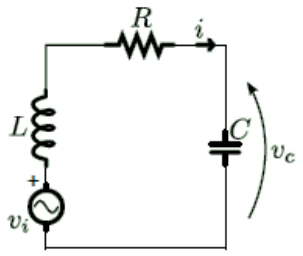


Figura 2.9. Metodología empleada en el trabajo.

Evaluación teórica del filtro de Kalman

Para la elaboración de un primer algoritmo que implemente la funcionalidad del filtro de Kalman, nos hemos servido de un sistema sencillo, basado en la siguiente malla eléctrica:



R=	0.3Ω
L=	$3mH$
C=	$1mF$
T_s	=2ms

Figura 2.10. Esquema eléctrico del sistema.

Dado un sistema:

$$\begin{gathered} x_k = Ax_{k-1} + \mathrm{B}u_{k-1} + w_{k-1} \\ y_k = \mathrm{C}x_k + v_k \\ w_k \sim (0, \mathrm{Q}) \\ v_k \sim (0, \mathrm{R}) \\ E[w_k w_k^T] = \mathrm{Q}\delta_k \\ E[v_k v_k^T] = \mathrm{R}\delta_k \\ E[v_k w_k^T] = 0 \end{gathered} \quad \textbf{(2.19)}$$

Donde:

$$A = \begin{pmatrix} -\frac{R}{L} & -L \\ \frac{1}{C} & 0 \end{pmatrix}$$

$$B = \begin{pmatrix} \frac{1}{L} \\ 0 \end{pmatrix}$$

$$C = (0 \quad 1)$$

$$x = \begin{pmatrix} i \\ v_c \end{pmatrix}$$

$$u = v_i$$

$$w \; y \; v = \; ruido \; blanco, gaussiano \; de \; media \; cero$$

$$Q = \begin{pmatrix} 0.008^2 & 0 \\ 0 & 0.004^2 \end{pmatrix}$$

$$R = 0.08^2 \quad \textbf{(2.20)}$$

Puesto que se busca una implementación en un sistema digital, el sistema es discretizado mediante el método *'zoh'*.

```
sysd =

  a =                      b =              c =
           x1       x2              u1          x1  x2
   x1   0.2975  -0.4787      x1  0.4787      y1   0   1
   x2    1.436   0.4411      x2  0.5589

Sample time: 0.002 seconds
Discrete-time state-space model.
```

Figura 2.11. Resultado obtenido en Matlab al discretizar la planta.

Si el sistema fuese completamente ideal, es decir, sin presencia de ruido en la medida ($v_k = 0$ para todo valor de k) y el modelo de la planta describiese a la perfección la dinámica del sistema, sin introducir ninguna incertidumbre ($w_k = 0$ para todo valor de k), la respuesta al escalón sería:

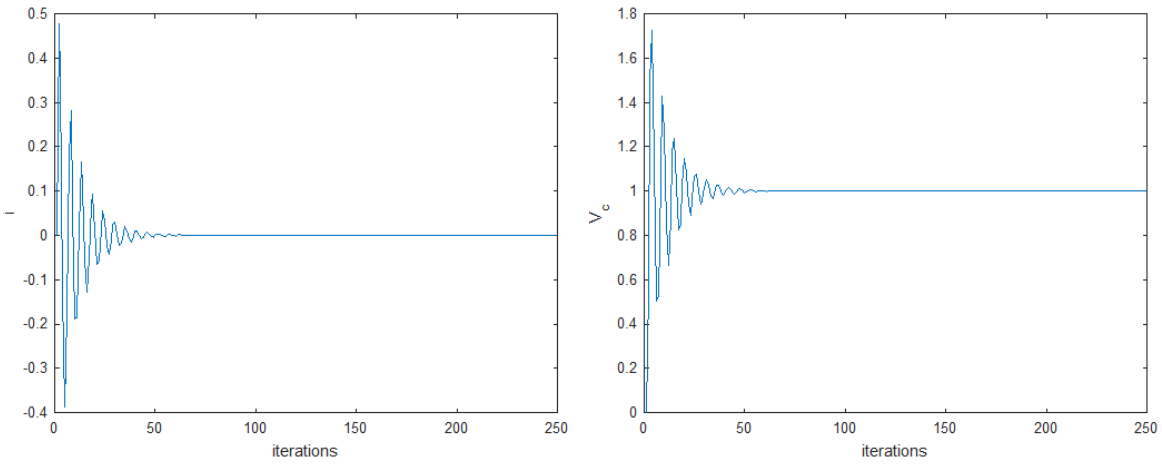


Figura 2.12. Corriente por la malla (izquierda) y tensión en el condensador (derecha).

Pero esto no se da en la realidad. Cualquier sensor o instrumento de media, por mucha calidad que tenga, siempre introduce ruido en la medida. En nuestro sistema, el único estado medible es la tensión en el condensador. Si añadimos ruido de medida v_k, con una desviación estándar de 0.08 Voltios, obtenemos las siguientes mediciones:

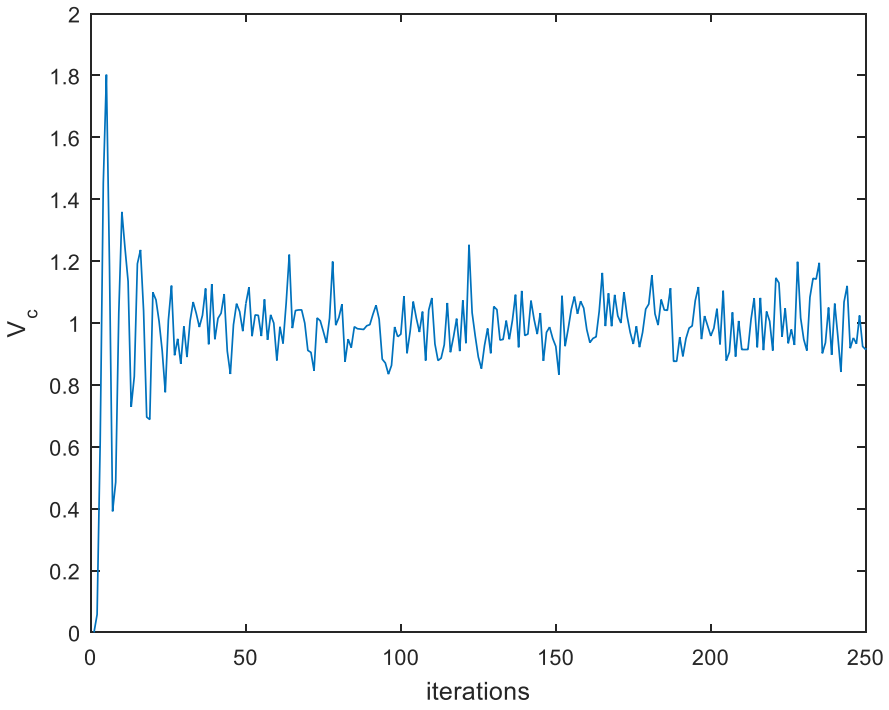


Figura 2.13. Resultado de medir la tensión en el condensador en presencia de ruido de medida.

Como se puede observar en la Figura 2.13, la medida del estado se ve enmascarada por el ruido, y si éste debe ser tratado de forma adecuada si se quieren utilizar los datos obtenidos.

A parte del ruido de medida, tenemos otra fuente de incertidumbre en nuestro sistema. El modelo utilizado para describir el comportamiento dinámico del sistema no es fiable en su totalidad, e introduce un error, representado por el ruido de proceso w_k.

Para un sistema sencillo como el que plantea la red RLC utilizada, supondremos que el modelo es más fiable que la medida y, por lo tanto, las aportaciones de w_k serán menores que las de v_k ($Q < R$).

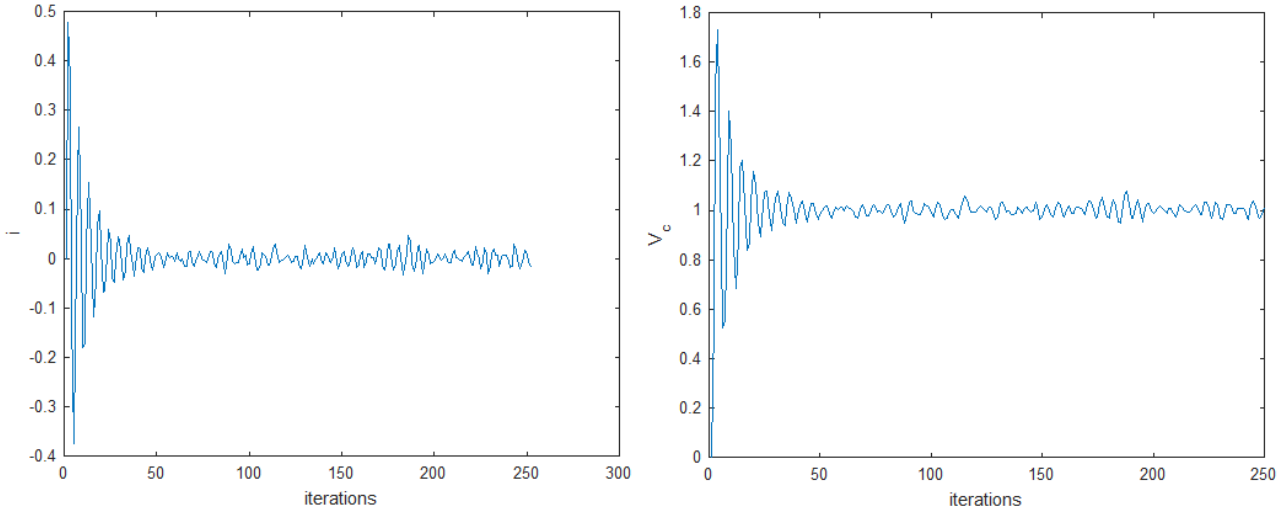


Figura 2.14. Corriente por la malla (izquierda) y tensión en el condensador (derecha) en presencia de ruido de proceso (d.e. w1 = 0.008, d.e. w2 = 0.004).

Ante esta tesitura, entra en acción el filtro de Kalman. Como ya se comentó en el Capítulo 2.1, el filtro es la mejor solución para la estimación de los estados en presencia de ruido blanco, gaussiano de media 0, que es justo el caso planteado.

El filtro de Kalman obedecerá a las siguientes ecuaciones:

$$\begin{aligned} P_k^- &= AP_{k-1}^- A^T + \mathrm{Q} \\ K_k &= P_k^- C^T (R_k + \mathrm{C} P_k^- C^T)^{-1} \\ P_k^+ &= (I - K_k C) P_k^- \\ \hat{x}_k^- &= \mathrm{A}\hat{x}_{k-1}^+ + \mathrm{B}u_{k-1} \\ \hat{x}_k^+ &= \hat{x}_k^- + K_k(y_k - \mathrm{C}\hat{x}_k^-) \end{aligned} \qquad \textbf{(2.21)}$$

En el algoritmo (2.21) podemos distinguir dos partes diferenciadas. La primera se encarga de computar la ganancia que hace óptima la estimación de los estados, así como de actualizar y corregir la covarianza del error en la estimación. La segunda propaga la media del estado la corrige cuando dispone de una nueva medida. Por ello, a la hora de implementarlo en Matlab, usaremos una función para cada una, denominadas "*Kalman_gain"* y *"Kalman_filter"* respectivamente.

Como resultado de aplicar el filtro, obtenemos las siguientes estimaciones:

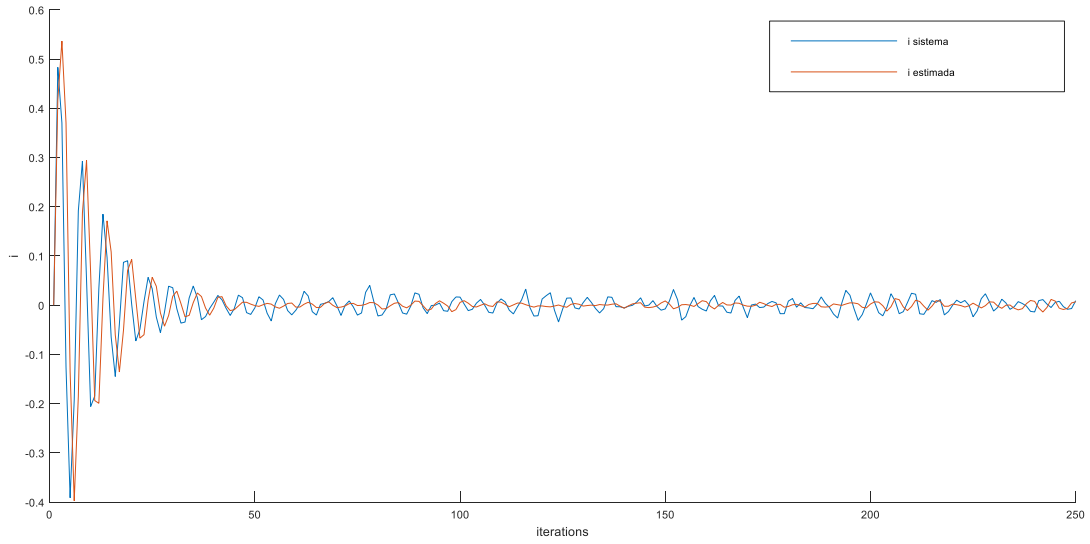


Figura 2.15. Corriente por la malla (azul), y su valor estimado (rojo)

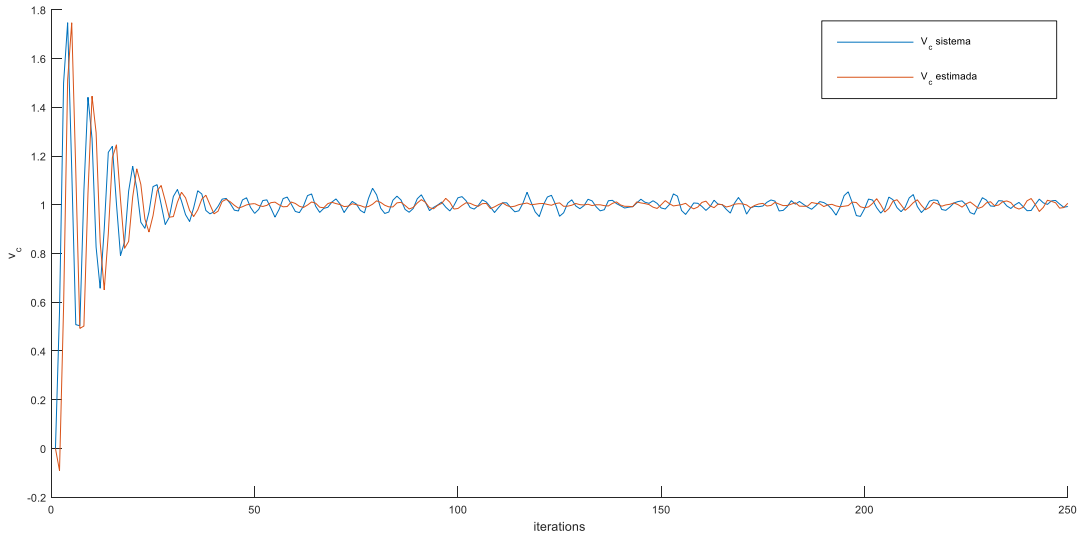


Figura 2.16. Tensión en el condensador (azul) y su valor estimado (rojo).

La Figura 2.15 y la Figura 2.16 muestran la estimación de los estados del sistema frente a la evolución temporal de los mismos. Se aprecia un tímido efecto de filtrado respecto del ruido de proceso.

Sin embargo, en la Figura 2.17 comparamos la estimación de la tensión en el condensador con las medidas realizadas. En este caso, sí se aprecia el filtrado del ruido de medida.

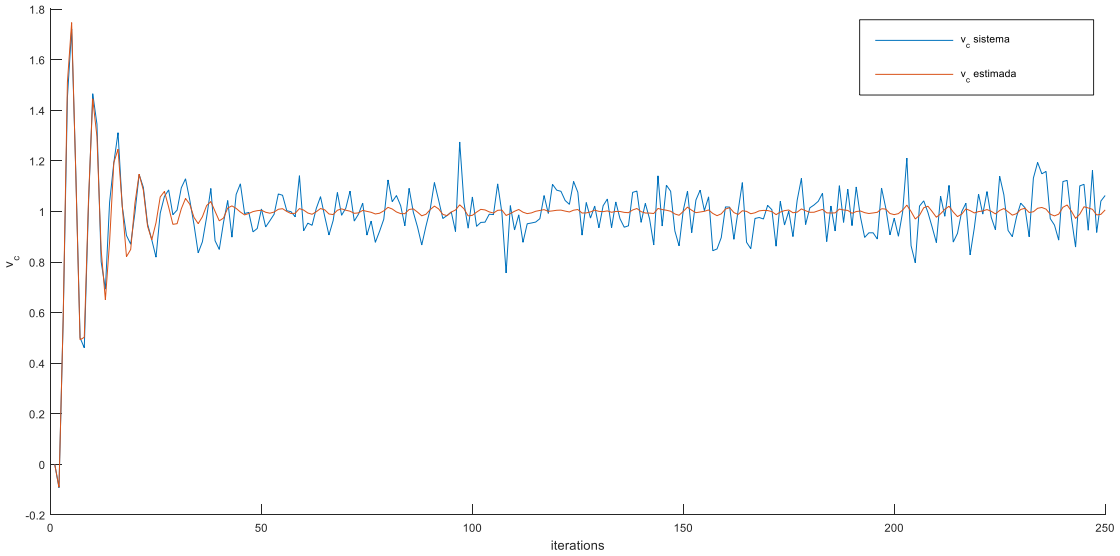


Figura 2.17. Tensión en el condensador medida (azul) y su valor estimado (rojo).

Ahora nos detendremos en la evolución de la covarianza del error estimado P. Al tener dos estados en la planta, P es una matriz cuadrada 2x2, siendo los coeficientes de la diagonal principal la varianza del estado estimado correspondiente. Siguiendo su evolución, obtenemos las siguientes gráficas:

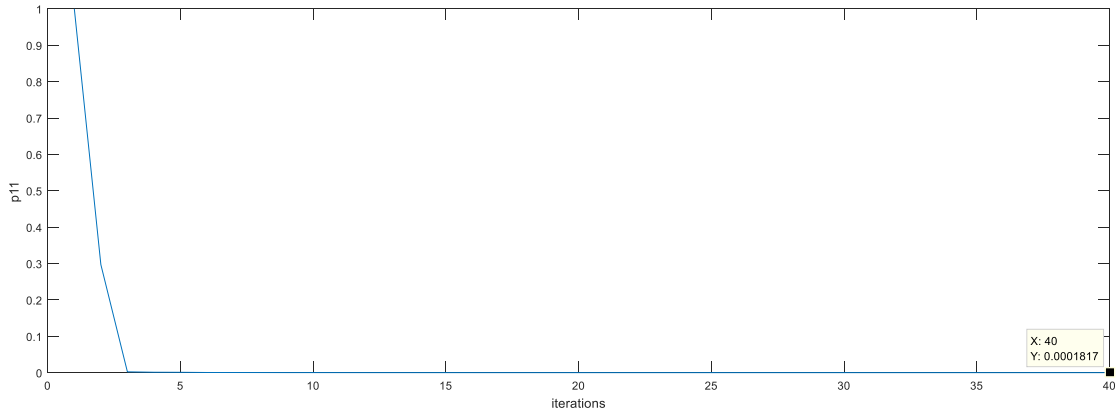


Figura 2.18. Covarianza del error de estimación de la corriente por la malla (p11)

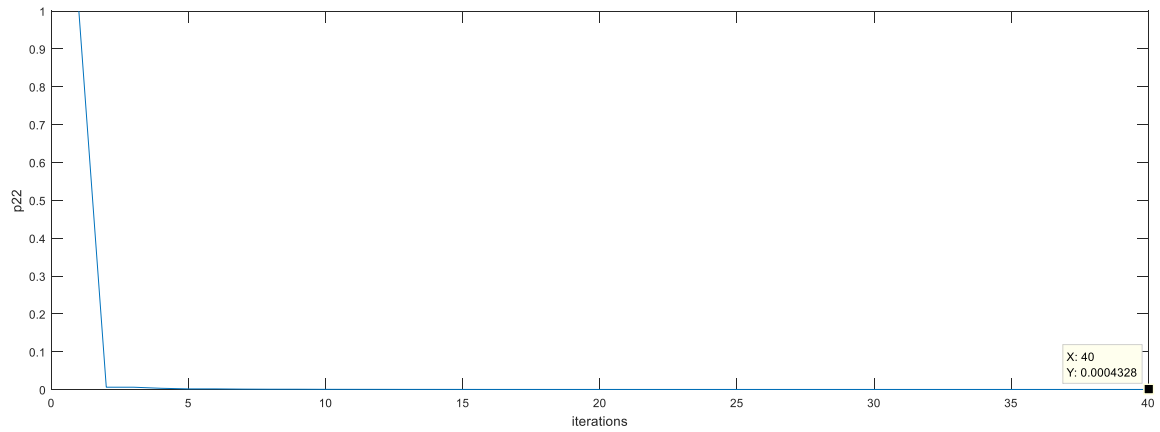


Figura 2.19. Covarianza del error en la estimación de la tensión en el condensador(p22).

Al igual que los estados, P sufre un transitorio desde su valor inicial, en este caso la matriz identidad, hasta su valor en régimen permanente. Para este sistema y con los niveles de ruidos propuestos, P alcanza el valor en régimen permanente a partir de 40 iteraciones.

En el filtro de Kalman, la covarianza P y la ganancia del estimador K están íntimamente ligados, como se muestra en la ecuación (2.21). Por ello, es de esperar que K también alcance un valor constante en régimen permanente.

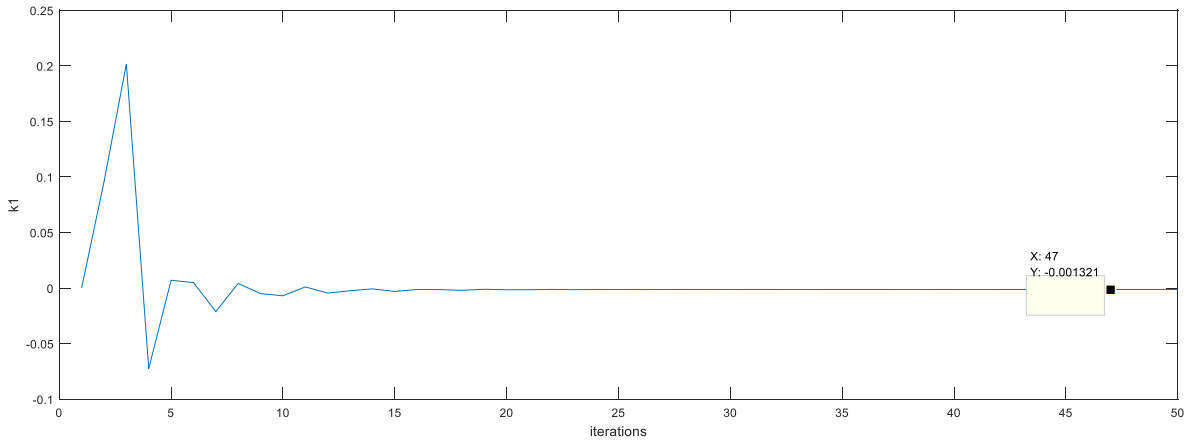


Figura 2.20. Primer coeficiente de la matriz de ganancia K.

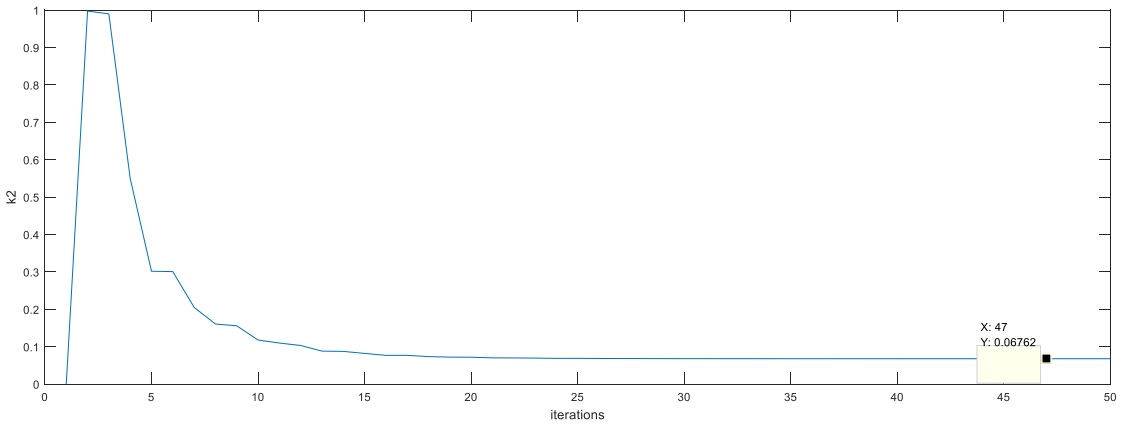


Figura 2.21. Segundo coeficiente de la ganancia K.

Para sistemas LTI, el hecho de que a partir de cierto número de iteraciones tanto la ganancia de Kalman como la covarianza de la estimación permanezcan constantes presenta una ventaja fundamental.

El peso computacional del filtro de Kalman lo lleva, principalmente, el cálculo de la ganancia de Kalman. Sabiendo que ésta alcanza su valor final después de un determinado número de iteraciones, podemos dejar de procesarla ganancia ahorrarnos su computo, ganando en tiempo de ejecución.

Es más, podemos ejecutar la parte del algoritmo encargada del cálculo de K (función "Kalman gain") offline, hallar su valor en régimen permanente, y utilizar ese valor en la corrección de la estimación, es decir, utilizar como ganancia del estimador para todo instante el valor final de K.

Esto implicaría que, durante el transitorio de los estados, la estimación de los mismos no sería óptima, pues no se estaría utilizando una ganancia que haga mínima el error en ese momento.

Por otro lado, las consecuencias de esto son reducidas. Por ejemplo, en nuestro sistema K tarda aproximadamente 40 iteraciones en estabilizarse. Con un periodo de muestreo de 2 ms, tardaríamos 0.08 segundos en empezar a tener estimaciones óptimas de los estados, que es un tiempo fácilmente salvable.

Esta forma de estimación es la que se va a implementar, y por esto el dividir el algoritmo en dos funciones. Así mismo, también se van a desarrollar dos periféricos en Vivado HLS, repartiendo de la misma manera la funcionalidad del filtro.

Codificación en coma fija del filtro de Kalman

En todo diseño hardware, debemos considerar qué formato de datos es el que mejor rendimiento nos ofrece. Habitualmente, codificar los datos en coma flotante (32 o 64 bits) ofrece un alto grado de precisión a costa de consumir más recursos. Codificar en coma fija te permite llegar a un compromiso entre precisión y recursos, pues el diseñador puede fijar el tamaño de la palabra y el número de bits asignados para representar el numero entero y su parte decimal.

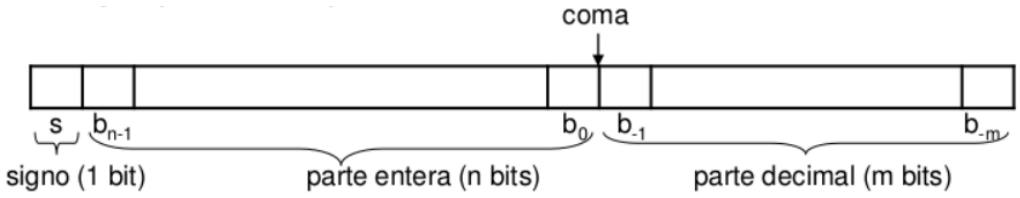


Figura 2.22. Formato en coma fija.

Antes de llevar nuestra aplicación a Vivado HLS, debemos determinar el ancho de palabra mínimo que nos permita mantener cierta resolución. Para ello, nos serviremos de la clase *fi* que suministra Matlab.

Esta clase nos permite asignar el tamaño de palabra, la fracción de la palabra dedicada a la parte decimal, si es un dato con signo o sin él y el modo de redondeo entre otras muchas cosas.

Existen tres tipos de propiedades para todo objeto fi:

- Propiedad *data:* este tipo de propiedades almacenan el valor del objeto *fi* en diferentes formatos de datos (binario, decimal, double, hexadecimal, entero y su valor real).
- Propiedad *fimath:* determinan las propiedades matemáticas del objeto *fi*, así como el modo de redondeo y desborde.
- Propiedad *numerictype:* caracterizan el número representado por el objeto *fi*, como su signo y su número de bits de parte decimal y entera.

Para evaluar la funcionalidad del filtro de Kalman empleando datos en coma fija, creamos otras dos funciones en Matlab, *"kalman_filter_fixed"* y *"Kalman_gain_fixed"*, hermanas de las creadas para la evaluación teórica. Su única distinción es que internamente recurren a la clase fi a la hora de realizar las operaciones.

Con estas funciones, podemos someter al sistema a una batería de pruebas, que nos permitan determinar el tamaño de la palabra y el número de bits dedicados a la parte decimal de manera experimental.

Sobre la práctica, se observa que se requieren un mínimo de 6 bits (uno de signo y el resto para codificar el dato) para poder representar debidamente la inversa de la matriz en el cálculo de la ganancia. De no suministrar el tamaño mínimo, el valor de la matriz inversa sería erróneo, pues saturaría.

El modo de redondeo se ha ajustad (*ceiling*)o de tal modo que aproxime al número inmediatamente superior representable. Si no lo hiciéramos, podría aproximar a cero la matriz que necesitamos invertir, produciendo una indeterminación matemática.

Solo queda por determinar el número de bits decimales necesarios. Sobre la misma situación, repetimos la estimación con diferente número de bits decimales, y evaluamos el error.

Se entiende por error el error cuadrático medio entre el valor estimado en coma flotante y el valor estimado en coma fija, normalizado al valor de coma fija.

$$e_{cm} = \sqrt[2]{\sum_{0}^{n} \frac{\left(estimacion_{coma\ flotante} - estimacion_{coma\ fija}\right)^2}{n}} \quad \textbf{(2.22)}$$

De esta manera, obtenemos la siguiente gráfica:

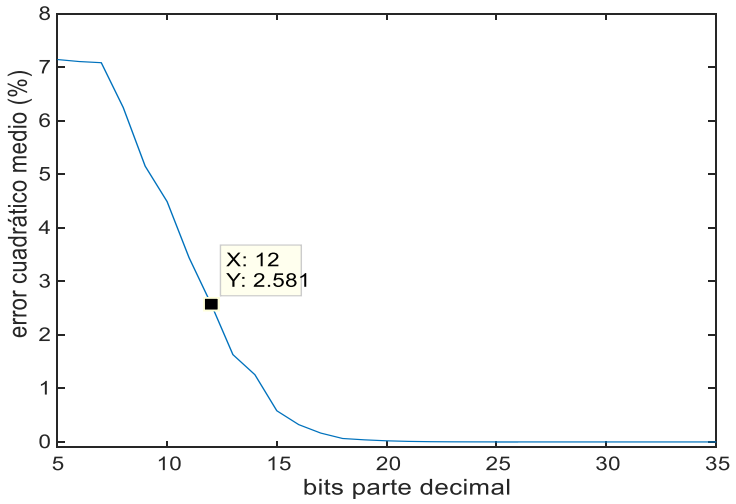


Figura 2.23. Error cuadrático medio en función del número de bits decimales.

Analizando los resultados mostrados en la Figura 2.23 y la Figura 2.24, observamos que cuando el error cuadrático medio es inferior al 3%, la estimación en coma fija es, al menos, equiparable a la estimación en coma flotante. No obstante, ¿De dónde sale ese valor promediado del error?

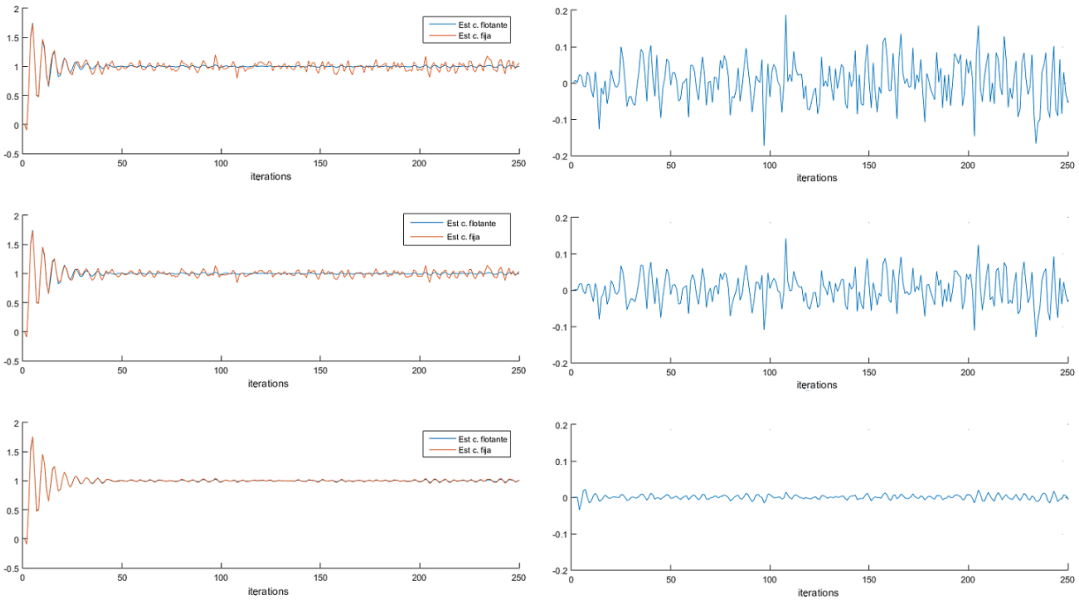


Figura 2.24. Estimación en coma flotante y fija (izquierda) y el error cometido (derecha) para 8 bits decimales (arriba), 10 bits decimales (centro) y 12 bits decimales (abajo).

La Figura 2.24 compara de manera visual las diferencias entre los diferentes anchos de palabra, manteniendo constante la parte entera. Según aumentamos el número de bits decimales, el error disminuye y la estimación en coma fija se va aproximando a la estimación en coma flotante.

El error cuadrático medio promedia el error que se comete en cada instante del tiempo. Se observa que el error cometido es mínimo si utilizamos 12 bits de parte decimal y 6 bits de parte entera. Determinamos así nuestro formato en coma fija, 18 bits de ancho de palabra, 6 bits de parte entera y 12 de parte decimal.

Vivado HLS

Una vez validado el algoritmo a utilizar, tanto en coma fija como en coma flotante, procedemos a generar un diseño RTL exportable a Vivado mediante la herramienta Vivado HLS.

Al igual que en Matlab, dividiremos la funcionalidad completa en dos bloques o periféricos. Uno computará el valor de la ganancia de Kalman K y la covarianza de la estimación P, y el otro realizará las estimaciones.

Por mucho que Vivado HLS permita diseñar bloques RTL en lenguajes de alto nivel, la codificación siempre irá orientada al hardware. Por ello, la herramienta incorpora librerías de precisión arbitraria, que permiten al usuario trabajar con diferentes anchos de palabra, a partes de los definidos en el estándar de C (8, 16, 32 y 64 bits). En relación

a este trabajo, en esas librerías se define la clase *"ap_fixed"*, que permite la utilizar el formato de dato en coma fija, de manera muy similar a la clase *fi* de Matlab.

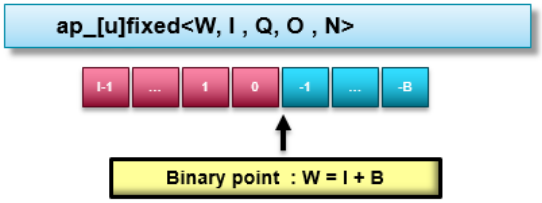


Figura 2.25. Clase ap_fixed de Vivado HLS.

Los parámetros de la clase son:

- W: ancho de la palabra.
- I: ancho de la parte entera. Si se utiliza *"ap_ufixed"* (unsigned), I es el ancho de la parte entera. Si por el contrario se utiliza *"ap_fixed"*, el ancho de la parte entera es I-1.
- Q: define el modo de redondeo. En este trabajo se ha utilizado *"AP_RND",* con el que se redondea al número infinitamente mayor.
- O: determina como debe comportarse la clase cuando el número de bits necesarios para representar un dato excede el especificado (overflow). En este trabajo se ha utilizado *"AP_SAT_SYM"*, con el que la representación satura al máximo de lo representable para números positivos, y al mínimo para números negativos.
- N: determina el número de bits que se utilizan cuando el modo de overflow es envoltorio (wrapped).

A continuación, se detallan los diseños realizados para cada periférico.

Kalman gain

El cálculo de la ganancia del estimador es la parte más crítica de todo el filtro, y la que se lleva la mayor carga computacional. Como ya se comentó anteriormente, nosotros hemos dividido el algoritmo en dos funciones, lo que se traduce en dos diseños RTL. *"Kalman_gain"* se encarga de computar la ganancia del estimador K y la covarianza del error en la estimación P, y *"Kalman_filter"* de realizar las estimaciones y corregirlas con cada nueva medida. En este apartado se desarrollan los diseños empleados en el primero de ellos.

A la hora de plantear el diseño del filtro de Kalman en general, se ha buscado cierto carácter de propósito general, y por ello los dos periféricos que implementarán la

funcionalidad del filtro de diseñarán para poder utilizarse en sistemas de hasta 10 estados, 10 entradas y 10 mediciones. Por lo tanto, todas las matrices serán de 10x10, y los vectores de 10 elementos.

Los diseños del filtro de Kalman van dirigidos a implementarse en la lógica programable de la tarjeta de evaluación ZedBoard, concretamente como periféricos del bus AXI4_LITE.

Para este periférico se realizarán soluciones tanto en coma fija como en coma flotante, siempre la solución que alcance un mejor compromiso entre recursos consumidos y la actuación que proporciona.

El diseño se basa principalmente, casi únicamente, en la multiplicación e inversión de matrices. Para la primera operación se ha incorporado una función al proyecto, mientras que para la inversión de matrices se ha utilizado la librería de álgebra lineal que proporciona Vivado HLS, concretamente la función *"cholesky_inverse"*. Esta función invierte matrices utilizando el método de descomposición de Cholesky.

Este método establece que, sea A una matriz simétrica definida positiva, entonces se puede descomponer en el producto de una matriz triangular por su transpuesta.

$$A = LL^T \ (L\ es\ una\ matriz\ triangular\ inferior) \quad \textbf{(2.23)}$$

La matriz inversa de A será:

$$A^{-1} = L^{-1}L^{-T} \quad \textbf{(2.24)}$$

Siendo la inversa de un matriz triangular más sencilla de calcular. En el apartado 5.1 se detalla cómo invertir matrices mediante este método.

En el filtro de Kalman, es necesario invertir esta matriz:

$$A = \mathrm{C}P_k^- C^T + R_k \quad \textbf{(2.25)}$$

Para poder aplicar la descomposición de Cholesky, A debe ser simétrica definida (semidefinida) positiva. Esto se cumplirá siempre que R_k y P_k^- lo sean, y puesto que ambas son matrices de covarianza, y los coeficientes de las diagonales principales son incorrelados entre sí, esta condición siempre se cumple.

Antes de realizar ninguna síntesis, HLS permite depurar el código de alto nivel, de manera que la aplicación queda validada y los errores no pasan al hardware. Nosotros buscamos que el diseño a realizar compute los datos lo más parecido a lo conseguido en Matlab, y para asegurarnos, realizamos las mismas pruebas sobre el sistema. Transferimos datos entre Matlab y Vivado HLS mediante archivos de texto.

Una vez validado, ya estamos en situación de empezar a realizar la síntesis de comportamiento y optimizar el hardware.

HLS proporciona un reporte de síntesis por cada función sintetizada, siendo el de la función top el que informa de los aspectos generales que tendrá nuestro diseño.

Inicialmente sintetizamos la función *"Kalman_gain"* sin aplicar ninguna directiva de optimización. Cuando finaliza, la herramienta nos devuelve la información del hardware generado.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.73	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
102472	122182	102473	122183	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	206
FIFO	-	-	-	-
Instance	7	14	3297	5360
Memory	6	-	0	0
Multiplexer	-	-	-	444
Register	-	-	386	-
Total	13	14	3683	6010
Available	280	220	106400	53200
Utilization (%)	4	6	3	11

Figura 2.26. Reporte de síntesis sin optimizaciones (Solution 1, coma flotante).

Para todas las soluciones, el periodo de la señal de reloj se ha fijado en 10 ns. Según la información suministrada, el periodo mínimo estimado es de 8.73 ns, con una incertidumbre de 1.25 ns, por lo que, en el peor de los casos, el periodo mínimo sería de 9.98 ns, que sigue por debajo del fijado.

La latencia es otro parámetro clave. Indica el número de ciclos de reloj que tarda el diseño en procesar todas las salidas. El intervalo indica el número de ciclos de reloj que pasan desde que empieza a procesar datos hasta que puede volver a recibir datos. En este caso, el diseño tarda un máximo de 122182 ciclos ($122182*10^{-9}$ = 1.2 ms) en procesar todas las salidas, y al ciclo de reloj siguiente ya podría volver a recibir nuevas entradas.

La función top, que en este caso es *"Kalman_gain"*, emplea 206 Look-Up Tables (LUT) para realizar expresiones y 444 en implementar multiplexores, junto con 386 Flip-Flops para registros y 6 bloques BRAM de 18 Kbits. El resto de recursos que aparecen en la Figura 2.26 son utilizados para la implementación de las instancias que la herramienta ha considerado necesarias para la síntesis.

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
grp_kalman_gain_cholesky_inverse_top_fu_393	kalman_gain_cholesky_inverse_top	4	7	2410	3721
kalman_gain_faddfsub_32ns_32ns_32_5_full_dsp_U17	kalman_gain_faddfsub_32ns_32ns_32_5_full_dsp	0	2	205	390
kalman_gain_fcmp_32ns_32ns_1_1_U18	kalman_gain_fcmp_32ns_32ns_1_1	0	0	66	239
grp_kalman_gain_matrixmul_fu_399	kalman_gain_matrixmul	3	5	616	1010
Total	4	7	14	3297	5360

Figura 2.27. Instancias generadas en Solution 1.

En este informe también es posible ver en qué se han empleado los recursos, como por ejemplo los BRAMS

Memory

Memory	Module	BRAM_18K	FF	LUT	Words	Bits	Banks	W*Bits*Banks
P_local_U	kalman_gain_matrixmul_aux1	1	0	0	100	32	1	3200
R_local_U	kalman_gain_matrixmul_aux1	1	0	0	100	32	1	3200
aux1_U	kalman_gain_matrixmul_aux1	1	0	0	100	32	1	3200
aux2_U	kalman_gain_matrixmul_aux1	1	0	0	100	32	1	3200
aux3_U	kalman_gain_matrixmul_aux1	1	0	0	100	32	1	3200
aux_inv_U	kalman_gain_matrixmul_aux1	1	0	0	100	32	1	3200
Total	6	6	0	0	600	192	6	19200

Figura 2.28. Asignación de BRAMS_18K a las variables de Kalman_gain.

Esta primera solución también se ha realizado para datos en coma fija, tal y como muestra la Figura 2.29.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.73	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
73592	85892	73593	85893	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	433
FIFO	-	-	-	-
Instance	5	32	3082	4782
Memory	6	-	0	0
Multiplexer	-	-	-	278
Register	-	-	270	-
Total	11	32	3352	5493
Available	280	220	106400	53200
Utilization (%)	3	14	3	10

Figura 2.29. Reporte de síntesis (Solution1-fixed, coma fija)

Como se comentó en el apartado anterior, cuando se utilizan datos en coma fija se espera un menor consumo de recursos a costa de perder precisión. Sin embargo, el resultado de la síntesis parece contradecir esta suposición, por lo que es necesario indagar más para encontrar la razón.

Si evaluamos las instancias generadas en esta solución, observamos que efectivamente *"matrixmul"* tiene un menor consumo de recursos y, sin embargo, la implementación *"cholesky_inverse_top"* se vuelve mucho más exigente para la FPGA. Para poder encontrar una respuesta al porqué del aumento de los recursos, debemos ahondar en los reportes de la síntesis.

Desplegando los reportes de la instancia *"cholesky_inverse_top"* en ambas soluciones, podemos analizar en qué ha empleado los recursos.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	2	0	540
FIFO	-	-	-	-
Instance	-	5	1580	2559
Memory	4	-	64	5
Multiplexer	-	-	-	617
Register	-	-	766	-
Total	4	7	2410	3721
Available	280	220	106400	53200
Utilization (%)	1	3	2	6

Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
kalman_gain_faddfsub_32ns_32ns_32_5_full_dsp_U7	kalman_gain_faddfsub_32ns_32ns_32_5_full_dsp	0	2	205	390
kalman_gain_fcmp_32ns_32ns_1_1_U10	kalman_gain_fcmp_32ns_32ns_1_1	0	0	66	239
kalman_gain_fdiv_32ns_32ns_32_16_U9	kalman_gain_fdiv_32ns_32ns_32_16	0	0	761	994
kalman_gain_fmul_32ns_32ns_32_4_max_dsp_U8	kalman_gain_fmul_32ns_32ns_32_4_max_dsp	0	3	143	321
kalman_gain_fsqrt_32ns_32ns_32_12_U11	kalman_gain_fsqrt_32ns_32ns_32_12	0	0	405	615
Total	5	0	5	1580	2559

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	6	0	2672
FIFO	-	-	-	-
Instance	-	24	600	712
Memory	2	-	100	18
Multiplexer	-	-	-	839
Register	-	-	2001	-
Total	2	30	2701	4241
Available	280	220	106400	53200
Utilization (%)	~0	13	2	7

Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
kalman_gain_mul_38s_32s_70_6_U5	kalman_gain_mul_38s_32s_70_6	0	4	0	0
kalman_gain_mul_86s_62s_146_6_U8	kalman_gain_mul_86s_62s_146_6	0	20	0	0
kalman_gain_sdiv_38ns_18s_38_42_seq_U7	kalman_gain_sdiv_38ns_18s_38_42_seq	0	0	304	361
kalman_gain_udiv_37s_24ns_37_41_seq_U6	kalman_gain_udiv_37s_24ns_37_41_seq	0	0	296	351
Total	4	0	24	600	712

Figura 2.30. Reporte de síntesis de cholesky_inverse en coma flotante (arriba) y en coma fija (abajo).

Para implementar este bloque en coma fija, HLS necesita multiplicadores hardware de más bits de los especificados para el formato, y por ello necesita un mayor número de DSP48, así como la lógica que ello conlleva, implementada con FF y LUTs.

Por otro lado, cuando se implementan operaciones en coma flotante, y utilizar el estándar float, HLS utiliza bloque que operan con datos de 32 bits, y no necesita desplegar tal cantidad de DSP48.

Queda patente que los bloques en coma flotante requieren mucha más lógica ara operar, lo que incrementa el consumo de LUTs y FFs, así como los BRAMs.

No obstante, la solución en coma fija es sensiblemente más rápida, y se podría estudiar si una solución en este formato merece la pena, aun si ocupa mayor superficie de la lógica programable de la ZedBoard.

Como se puede observar en la Figura 2.26 y la Figura 2.27, las instancias dedicadas a *"matrixmul"* y *"cholesky_inverse_top"* con las que consumen la mayor parte de los recursos, debido a que son las unidades que realmente procesan los datos. Por ello, son los objetivos principales para la optimización.

Empezaremos por optimizar el bloque *"matrixmul"*. Aplicando la directiva *'pipeline'* a al bucle que recorre las columnas en el producto de matrices (bucle intermedio) conseguimos un bloque segmentado que puede procesar nuevas entradas sin necesidad de haber terminado de procesar las anteriores. La solución en coma fija vienen denotadas con el sufijo *"-fixed"*.

Performance Estimates

Timing (ns)

Clock		solution2	solution2-fixed
default	Target	10.00	10.00
	Estimated	8.73	8.73

Latency (clock cycles)

		solution2	solution2-fixed
Latency	min	19151	14505
	max	38861	26805
Interval	min	19152	14506
	max	38862	26806

Utilization Estimates

	solution2	solution2-fixed
BRAM_18K	15	11
DSP48E	19	50
FF	5222	5096
LUT	7483	8637

Figura 2.31. Reporte de síntesis aplicando pipeline en matrixmul.

Solo esta optimización hace que el sistema sea sensiblemente más rápido que en las soluciones mostradas en las Figura 2.26 y Figura 2.29. De momento, las dos versiones

de Solution 2 son más o menos equivalentes, consumiendo la solución en coma fija un mayor número de DSP48.

Podemos hacer que los datos de entrada a la multiplicación se puedan leer en el mismo ciclo de reloj mediante la directiva *'array_reshape'*, tal y como se ha hecho en Solution 3.

Performance Estimates

Timing (ns)

Clock		solution3	solution3-fixed
default	Target	10.00	10.00
	Estimated	8.73	8.73

Latency (clock cycles)

		solution3	solution3-fixed
Latency	min	17093	9010
	max	36803	21310
Interval	min	17094	9011
	max	36804	21311

Utilization Estimates

	solution3	solution3-fixed
BRAM_18K	38	24
DSP48E	59	50
FF	13003	7902
LUT	61245	24156

Figura 2.32. Reporte de síntesis añadiendo array_reshape.

Esta directiva añade una leve mejora al diseño, pero resulta totalmente prohibitiva, al menos en coma flotante, pues la ZedBoard cuanta con un máximo de 53200 LUTs, y esta solución requeriría 61245. Por lo tanto, debemos remover esta directiva.

Estando el bloque *"matrixmul"* ya optimizado, centramos nuestra atención en el bloque *"cholesky_inverse_top"*. Internamente, este bloque realiza una multiplicación de matrices, que se puede optimizar de la misma manera que el bloque *"matrixmul"*.

Performance Estimates

Timing (ns)

Clock		solution4	solution4-fixed
default	Target	10.00	10.00
	Estimated	8.73	8.73

Latency (clock cycles)

		solution4	solution4-fixed
Latency	min	9981	12091
	max	29691	24391
Interval	min	9982	12092
	max	29692	24392

Utilization Estimates

	solution4	solution4-fixed
BRAM_18K	15	11
DSP48E	19	59
FF	5757	5504
LUT	8025	8967

Figura 2.33. Reporte de síntesis optimizando la multiplicación de matrices en el cálculo de la inversa.

A medida que optimizamos el diseño, la solución en coma flotante empieza a destacar, pues sin ser mucho más lenta que la solución en coma fija, emplea menos recursos hardware de la lógica programable, y además ofrece un alto grado de precisión numérica.

Yendo más allá en la optimización de *"cholesky_inverse_top",* combinando la aplicación de las directivas *'inline'* y *'pipeline'* a la función, obtenemos la solución más potente.

Performance Estimates

Timing (ns)

Clock		solution5	solution5-fixed
default	Target	10.00	10.00
	Estimated	8.63	8.55

Latency (clock cycles)

		solution5	solution5-fixed
Latency	min	8813	11096
	max	8813	11096
Interval	min	8814	11097
	max	8814	11097

Utilization Estimates

	solution5	solution5-fixed
BRAM_18K	13	9
DSP48E	83	1434
FF	37194	129628
LUT	43149	234556

Figura 2.34. Reporte de síntesis aplicando las directivas inline y pipeline en la funcion cholesky inverse.

Esta solución en coma fija es totalmente inviable para la tarjeta ZedBoard. Sin embargo, la solución de coma flotante sí es realizable en la tarjeta, siendo además la más rápida de todas.

Visto que ninguna de las soluciones en coma fija destaca lo suficiente respecto de las de coma flotante, ni en latencia ni en recursos consumido, como para obviar la pérdida de precisión, las descartaremos para una posible implementación en la ZedBoard.

Resumiendo, el conjunto de las soluciones disponibles queda así:

Performance Estimates

Timing (ns)

Clock		solution1	solution2	solution3	solution4	solution5
default	Target	10.00	10.00	10.00	10.00	10.00
	Estimated	8.73	8.73	8.73	8.73	8.63

Latency (clock cycles)

		solution1	solution2	solution3	solution4	solution5
Latency	min	102472	19151	17093	9981	8813
	max	122182	38861	36803	29691	8813
Interval	min	102473	19152	17094	9982	8814
	max	122183	38862	36804	29692	8814

Utilization Estimates

	solution1	solution2	solution3	solution4	solution5
BRAM_18K	13	15	38	15	13
DSP48E	14	19	59	19	83
FF	3683	5222	13003	5757	37194
LUT	6010	7483	61245	8025	43149

Figura 2.35. Resumen de soluciones realizadas.

Si bien la solución 5 es la que más superficie ocupa (81% de LUTs disponibles) es, con diferencia, la solución más potente y, por lo tanto, la primera candidata para exportar a Vivado.

Antes de exportar el diseño, es interesante detenernos en cómo ha implementado la herramienta el periférico de puertas para afuera.

El bloque recibe como entradas las matrices A y C del sistema, P, Q y R, devolviendo el valor actualizado de P y la ganancia K. Por defecto, las matrices las implementa como memorias RAM, y si estas se encuentran como argumentos de la función top, la herramienta interpretará que su contenido se encuentra en una memoria externa, y crea los puertos necesarios para poder acceder a su contenido.

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
A_address0	out	7	ap_memory	A	array
A_ce0	out	1	ap_memory	A	array
A_q0	in	32	ap_memory	A	array
C_address0	out	7	ap_memory	C	array
C_ce0	out	1	ap_memory	C	array
C_q0	in	32	ap_memory	C	array
Q_address0	out	7	ap_memory	Q	array
Q_ce0	out	1	ap_memory	Q	array
Q_q0	in	32	ap_memory	Q	array
R_address0	out	7	ap_memory	R	array
R_ce0	out	1	ap_memory	R	array
R_q0	in	32	ap_memory	R	array
P_address0	out	7	ap_memory	P	array
P_ce0	out	1	ap_memory	P	array
P_we0	out	1	ap_memory	P	array
P_d0	out	32	ap_memory	P	array
P_q0	in	32	ap_memory	P	array
Kgain_address0	out	7	ap_memory	Kgain	array
Kgain_ce0	out	1	ap_memory	Kgain	array
Kgain_we0	out	1	ap_memory	Kgain	array
Kgain_d0	out	32	ap_memory	Kgain	array

Figura 2.36. Interfaz entrada/salida.

Sin contar P y K, el resto de argumentos son de solo lectura, y por ello solo dedica un puerto de 32 bits (entrada), a los datos (A_q0, C_q0, Q_q0, R_q0). Por el contrario, K es un argumento de solo escritura, y por ello crea un puerto de 32 bits (salida) para los datos (Kgain_d0). P es un argumento tanto de escritura como lectura, y por eso consta de los dos puertos citados anteriormente.

Además de los puertos necesarios para implementar los argumentos de la función, el bloque cuenta con un protocolo por defecto, *'ap_cntrl_hs'*, que rige su comportamiento. Este protocolo incluye determinadas señales de control, que permiten gobernar el

diseño y regulan las transacciones de datos. Estas señales, denominadas señales de handshaking, son:

- 'ap_start': controla cuando el diseño debe empezar su ejecución. Para ello, debe fijarse a nivel alto.
- 'ap_ready': indica cuando el sistema está disponible para leer nuevos datos de entrada. Cuando esta situación se da, la señal se fija a nivel alto.
- 'ap_done': indica cuando el sistema ha realizado todas las operaciones y tiene un dato disponible. Cuando esta situación se da, la señal se fija a nivel alto.
- 'ap_idle': indica si el sistema está procesando datos o no. Cuando el sistema está ocupado, la señal se fija a nivel alto.

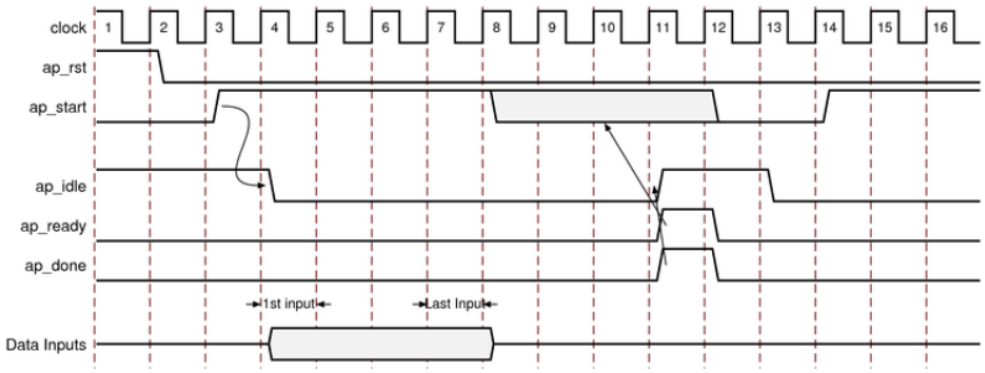


Figura 2.37. Cronograma de las señales de handshaking.

Nuestro objetivo final es conseguir un conjunto de periféricos para Zynq, que implementen la funcionalidad de un filtro de Kalman. Si queremos que el diseño realizado se pueda entender con el microprocesador de la tarjeta, necesitamos que se adapte al protocolo del bus AXI4-LITE. Aplicando la directiva *'interface: s_axilite'* a los argumentos de la función top, HLS interpretara que ahora, en vez de BRAMs, los datos son transferidos a través del bus AXI4_LITE.

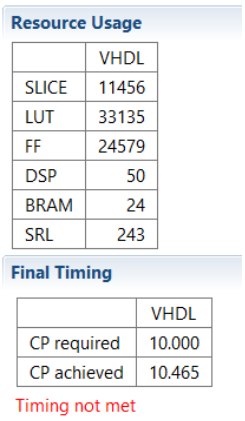
De esta manera, HLS generará un mapa de memoria asociado al bus, reservando el espacio necesario para cada argumento.

Si aplicamos esa directiva a la función top, HLS reserva las primeras posiciones del mapa de memoria para implementar el protocolo *'ap_cntrl_hs'*. De esta manera, conseguimos un periférico esclavo de Zynq fácilmente controlable desde el microprocesador.

Las señales *'ap_ready'* y *'ap_done'* del protocolo *'ap_cntrl_hs'* serán utilizadas como fuente local de interrupción para el microprocesador. Es decir, que es posible lanzar una interrupción cuando el bloque termina de procesar todas las salidas o cuando está listo para recibir nuevas entradas. El mapa de memoria final se muestra en la Figura 2.40.

Una vez definidas las directivas de interfaz deseadas, podemos realizar la exportación del RTL al catálogo de IPs de Vivado, y utilizarlo en un diseño. En este paso, aparte de añadir lógica extra, HLS es capaz de determinar con mayor precisión los recursos y el timing del diseño generado.

Como se comentó anteriormente, la solución 5 es la primera candidata para su exportación a Vivado, y como resultado de hacerlo obtenemos el siguiente reporte:



Resource Usage

	VHDL
SLICE	11456
LUT	33135
FF	24579
DSP	50
BRAM	24
SRL	243

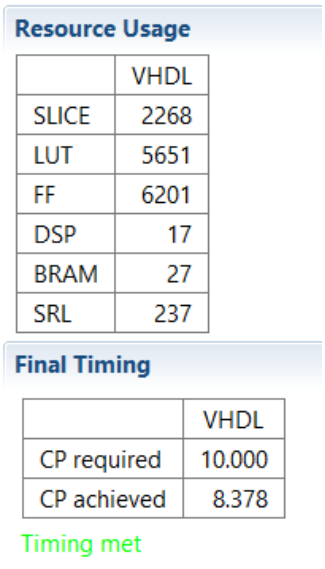
Final Timing

	VHDL
CP required	10.000
CP achieved	10.465

Timing not met

Figura 2.38. Resultado de la exportación RTL para la solución 5.

Por desgracia, HLS no es capaz de exportar esta solución sin violar el periodo del reloj establecido en 10 ns. Por lo tanto, debemos descartar esta solución, pasando a exportar la inmediatamente anterior, la solución 4.



Resource Usage

	VHDL
SLICE	2268
LUT	5651
FF	6201
DSP	17
BRAM	27
SRL	237

Final Timing

	VHDL
CP required	10.000
CP achieved	8.378

Timing met

Figura 2.39. Resultado de la exportación RTL para la solución 4.

En este caso, la exportación ha sido exitosa. Cuando se utilizan las directivas del bus AXI, HLS interpreta que el diseño se acabara utilizando con un microprocesador. Por ello, uno de los productos de la exportación RTL, a parte del IP en sí, con unos drivers en los que

se definen funciones de alto nivel que permiten acceder a las direcciones de memoria mapeadas.

```
// KALMAN_GAIN_PERIPH_BUS
// 0x000 : Control signals
//         bit 0  - ap_start (Read/Write/COH)
//         bit 1  - ap_done (Read/COR)
//         bit 2  - ap_idle (Read)
//         bit 3  - ap_ready (Read)
//         bit 7  - auto_restart (Read/Write)
//         others - reserved
// 0x004 : Global Interrupt Enable Register
//         bit 0  - Global Interrupt Enable (Read/Write)
//         others - reserved
// 0x008 : IP Interrupt Enable Register (Read/Write)
//         bit 0  - Channel 0 (ap_done)
//         bit 1  - Channel 1 (ap_ready)
//         others - reserved
// 0x00c : IP Interrupt Status Register (Read/TOW)
//         bit 0  - Channel 0 (ap_done)
//         bit 1  - Channel 1 (ap_ready)
//         others - reserved
// 0x200 ~
// 0x3ff : Memory 'A' (100 * 32b)
//         Word n : bit [31:0] - A[n]
// 0x400 ~
// 0x5ff : Memory 'C' (100 * 32b)
//         Word n : bit [31:0] - C[n]
// 0x600 ~
// 0x7ff : Memory 'Q' (100 * 32b)
//         Word n : bit [31:0] - Q[n]
// 0x800 ~
// 0x9ff : Memory 'R' (100 * 32b)
//         Word n : bit [31:0] - R[n]
// 0xa00 ~
// 0xbff : Memory 'P' (100 * 32b)
//         Word n : bit [31:0] - P[n]
// 0xc00 ~
// 0xdff : Memory 'Kgain' (100 * 32b)
//         Word n : bit [31:0] - Kgain[n]
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)
```

Figura 2.40. Mapa de memoria del periférico Kalman_gain.

El resultado de la implementación se guarda dentro de la carpeta impl, generada dentro del directorio de la solución 4. Para poder utilizar este diseño en Vivado, basta con añadir este directorio como repositorio en el catálogo de IP.

Con esto, el periférico ya estaría listo para su uso, y podemos pasar al segundo, *"Kalman_filter"*.

Kalman filter

Una vez resuelto el problema del cálculo de la ganancia, solo queda la implementación de la parte del filtro de Kalman encargado de realizar las estimaciones. De eso se va a encargar el periférico *"Kalman_filter"*.

La función recibe como argumentos las matrices A, B, y C del sistema, la entrada al sistema u, las mediciones y y la ganancia del estimador K, y devuelve la estimación de los estados x_est.

En comparación con *"Kalman_gain"*, este periférico es mucho más sencillo, pues en él solo se dan operaciones de multiplicación y suma de matrices.

La función encargada de la multiplicación es la misma que se utilizó en el otro periférico, y por lo tanto ya conocemos todas las optimizaciones posibles que se pueden realizar en ella. La suma de matrices se realiza mediante un lazo con dos bucles, que permite la suma elemento a elemento.

Al haber escogido una solución en coma flotante para *"Kalman_gain"*, solo se van a evaluar soluciones de *"Kalman_filter"* en el mismo formato de datos.

Realizando la síntesis sobre el diseño sin aplicar directivas, obtenemos el siguiente reporte.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.09	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
32477	32477	32478	32478	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	96
FIFO	-	-	-	-
Instance	6	12	1275	2250
Memory	6	-	0	0
Multiplexer	-	-	-	271
Register	-	-	272	-
Total	12	12	1547	2617
Available	280	220	106400	53200
Utilization (%)	4	5	1	4

Figura 2.41. Reporte de síntesis sin directivas de optimización.

Resulta algo llamativo que emplee 12 DSP48 en la implementación de la multiplicación de matrices. Si desplegamos la pestaña de instancias, observamos que HLS a instanciado dos veces el bloque de la multiplicación.

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_kalman_filter_matrixmul_fu_300	kalman_filter_matrixmul	10663	10663	10663	10663	none
grp_kalman_filter_matrixmul_fu_308	kalman_filter_matrixmul	10663	10663	10663	10663	none

Figura 2.42. Instancias generadas en solution 1.

Para encontrar una explicación, debemos fijarnos en cómo se ha codificado el algoritmo. Para implementar la ecuación $\hat{x}_k^- = A\hat{x}_{k-1}^+ + Bu_{k-1}$, nosotros llamamos primero a la

función *"matrixmul"* para realizar la operación $A\hat{x}_{k-1}^{+}$ y luego la volvemos a llamar para realizar Bu_{k-1}, para después sumar los resultados. Puesto que ninguno de los factores depende del resultado de la otra multiplicación, HLS interpreta que esas dos multiplicaciones se pueden realizar en paralelo, y para ello duplica la instancia multiplicadora de matrices.

Esto no ocurría en el otro periférico, pues alguno de los factores de la multiplicación eran producto de la multiplicación anterior.

Como se realizó en el apartado anterior, optimizamos la función de multiplicación de matrices.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.26	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2762	2762	2763	2763	none

Detail

Instance

Loop

Utilization Estimates

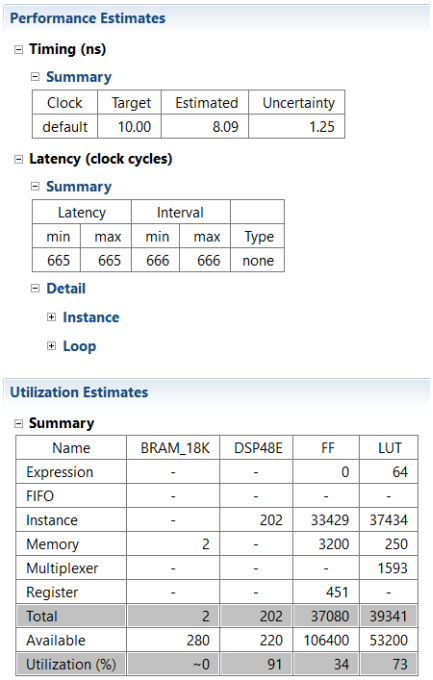
Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	96
FIFO	-	-	-	-
Instance	10	22	3219	5006
Memory	6	-	0	0
Multiplexer	-	-	-	271
Register	-	-	272	-
Total	16	22	3491	5373
Available	280	220	106400	53200
Utilization (%)	5	10	3	10

Figura 2.43. Reporte de síntesis optimizando matrixmul.

En la Figura 2.43. Reporte de síntesis optimizando matrixmul. se parecía que, al optimizar la función de multiplicación de matrices, la latencia disminuye radicalmente y los recursos consumidos prácticamente se duplican, lo recalca el peso de este bloque dentro del diseño.

Podemos ir más allá en la optimización, aplicando la directiva *'pipeline'* a la función completa, y no solo a los bucles internos.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.09	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
665	665	666	666	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	64
FIFO	-	-	-	-
Instance	-	202	33429	37434
Memory	2	-	3200	250
Multiplexer	-	-	-	1593
Register	-	-	451	-
Total	2	202	37080	39341
Available	280	220	106400	53200
Utilization (%)	~0	91	34	73

Figura 2.44. Reporte de síntesis aplicando pipeline a la función matrixmul.

Como resultado, obtenemos un diseño con un alto grado de paralelismo, capaz de procesar todas las salidas en menos de 7 microsegundos, a costa de ocupar prácticamente el 90% de la FPGA.

Esta última solución es inviable si además queremos incluir el otro periférico en la lógica programable. Por lo tanto, la mejor solución que podemos exportar es la solución 2.

Al igual que en el apartado anterior, queremos exportar este diseño como un periférico del bus AXI4-LITE. Para ello, volvemos a aplicar la directiva *'s_axilite'* a los argumentos y a la función top.

Antes de exportar, realizamos la síntesis incluyendo esta directiva, obteniendo:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.26	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2762	2762	2763	2763	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	96
FIFO	-	-	-	-
Instance	24	22	3815	5536
Memory	6	-	0	0
Multiplexer	-	-	-	271
Register	-	-	272	-
Total	30	22	4087	5903
Available	280	220	106400	53200
Utilization (%)	10	10	3	11

Figura 2.45. Reporte de síntesis incluyendo la directiva s_axilite.

Una vez exportado, obtenemos un valor de los recursos empleados y del timing del periférico más cercano a la implementación real.

Resource Usage

	VHDL
SLICE	1567
LUT	3676
FF	4306
DSP	22
BRAM	30
SRL	235

Final Timing

	VHDL
CP required	10.000
CP achieved	7.867

Timing met

Figura 2.46. Reporte de la exportación RTL.

Una vez finalizado el diseño de este último periférico, ya estamos en condiciones de realizar una aplicación del filtro de Kalman basada en un microprocesador.

Implementación del filtro de Kalman en la tarjeta ZedBoard

El objetivo principal de este trabajo era emplear la herramienta Vivado HLS para el diseño de un filtro de Kalman y, como se ha desarrollado en los apartados, se ha logrado satisfactoriamente.

A partir de aquí, la propia herramienta permite exportar esos diseños al catálogo de IPs de Vivado, donde podrán implementarse en una FPGA. En nuestro caso, la aplicación se implementará sobre la tarjeta de evaluación ZedBoard, que incorpora el dispositivo xc7c020clg484-1, perteneciente a la familia Zynq-7000.

Como ejemplo, se propone una aplicación basada en un microprocesador en la que se integren los periféricos diseñados en los apartados 0 y 0. Esta es una ideada para procesar los datos de manera offline, no para embarcarla en un sistema real, y tiene por objetivo evaluar el funcionamiento y comportamiento de los diseños exportados como periféricos de Zynq desde HLS.

Para elaborar dicha aplicación, primero debemos diseñar el sistema hardware que nos proporcione soporte. Lo primero de todo es crear un proyecto nuevo en Vivado. Dentro del proyecto crearemos un diseño por bloques. Cuando se quiere realizar un diseño para Zynq es necesario añadir el bloque *"ZYNQ7 Processos System"* y el resto de periféricos que precise (Vivado los instancia y conecta de forma automática). A parte, se pueden añadir los periféricos deseados. Nosotros incluiremos los periféricos "*Kalman_gain"* y *"Kalman_filter"*

Si queremos incluirlos, debemos añadirlos al repositorio. Podemos añadir el directorio donde se ha guardado la exportación RTL o bien copiar el archivo comprimido existente dentro de la carpeta ip del mismo directorio, pegarlo y descomprimirlo en cualquier otra carpeta, y añadirla.

Una vez añadidos ambos periféricos al repositorio, podemos montar el sistema completo.

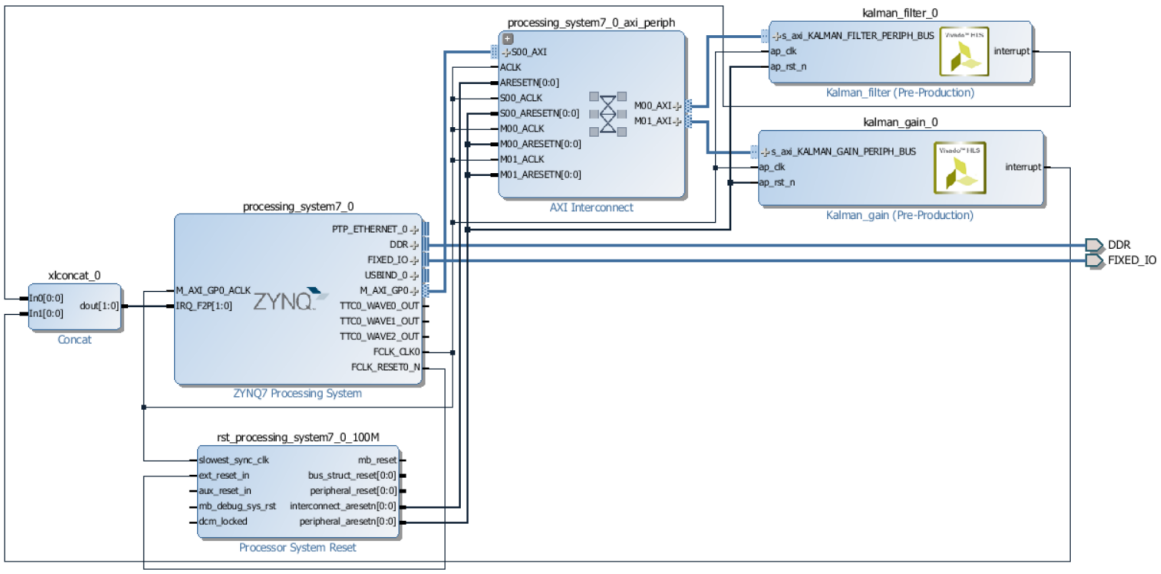


Figura 2.47. Diagrama de bloques generado en Vivado.

Los IPs creados mediante HLS presentan el logotipo del programa en su envoltorio. Una curiosidad, aunque el PS de Zynq soporta hasta 16 fuentes de interrupción desde la PL, el bloque solo admite la conexión de un bus de líneas (IRQ_F2P[n:0]). Por eso es necesario el bloque Concat, que junta las señales de interrupción de los dos periféricos y la conecta la PS.

Una vez realizado el montaje del diseño que introducirá en la tarjeta, creamos un envoltorio HDL, que simplemente traduce el esquema montado a un fichero de texto, sintetizamos, implementamos y generamos el archivo bitstream que programará la tarjeta.

Con estos pasos finalizamos, exportamos el diseño hardware a la plataforma de desarrollo software (SDK) y lanzamos el programa.

La herramienta SDK permite programar la tarjeta con el archivo bitstream, y desarrollar el código de alto nivel (lenguaje C) que ejecutará el microprocesador.

En nuestra aplicación, el microprocesador juega un papel central. Con él gobernamos el funcionamiento de los dos periféricos, siendo el nexo entre ambos y el elemento que nos permite la transmisión de los datos

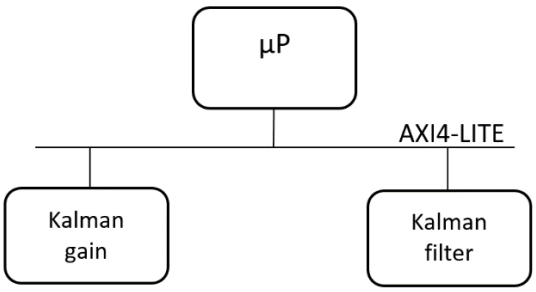


Figura 2.48. Arquitectura empleada.

Al haber separado el filtro de Kalman en dos periféricos, podemos funcionar con los dos por separado. Esto nos permite, por ejemplo, calcular la ganancia del filtro en régimen permanente con *"Kalman_gain"* y, una vez hallada, detener su funcionamiento para luego empezar a realizar las estimaciones con *"Kalman_gain"*; y también es posible implementar el algoritmo de forma ordinaria.

La aplicación de alto nivel ha sido desarrollada para trabajar offline, de manera que únicamente procese un conjunto de datos. El conjunto de datos, que en este ejemplo es el mismo sistema con el que se ha trabajado en el apartado 0, se introduce al microprocesador, que luego es el encargado de escribirlos de manera apropiada en los registros de los periféricos.

Además, al no tener la necesidad de estar transmitiendo datos al exterior, hemos aprovechado las posibilidades que ofrece SDK para introducir y extraer datos del microprocesador. Concretamente, se ha utilizado la consola XMD (Xilinx Microprocessor Debugger) que, durante la depuración, permite la introducción de comandos o scripts, dando el soporte necesario para una verificación exhaustiva del sistema concreto.

Utilizando el comando *'dow -data "input.dat" 0x10000000'* podemos descargar en la memoria de microprocesador el archivo *"input.dat"*. Este archivo binario ha sido generado en Matlab, contiene el conjunto de datos que se van a utilizar en el testeo del filtro. Esos datos son las matrices del sistema y los vectores de entradas al mismo, junto con las mediciones tomadas. El archivo se descargará a partir de la dirección 0x10000000, y para poder leer los datos basta con inicializar un puntero a esa dirección e ir recorriendo las direcciones de memoria consecutivas.

La ZedBoard cuenta con 511 MB hábiles de memoria DDR mapeados entre la dirección 0x00100000 y la0x1FF00000. Las variables propias de la aplicación se ubicarán a partir de la dirección base. Si descargamos el fichero a partir de la dirección 0x10000000, aseguramos que no sobrescribimos ninguna variable.

De la misma forma que existe un comando que permite descargar archivos en la memoria, existe otro con el que se pueden leer el contenido de una dirección de memoria, *'xrmem'*.

Para extraer los datos, utilizamos un script, en el que se define una función para la consola XMD que utiliza ese comando. La función recibe como argumentos el número de posiciones de memoria y la dirección base desde la que empezar a leer. Si almacenamos los datos en una variable global, basta con introducir el número de elementos y la dirección en la que está almacenada.

Se han definido un conjunto de constantes que parametrizan la aplicación:

- N_ITERATIONS: número de iteraciones que se ejecuta el filtro.
- KG_ITERATIONS: número de veces que se itera la ganancia de Kalman para obtener un valor en régimen permanente.
- N_STATES: número de estados del sistema.
- N_INPTS: número de entradas al sistema.
- N_MEASUREMENTS: número de medidas realizadas sobre el sistema.

Para contrastar la validez del sistema elaborado en la Zedboard, volvemos a utilizar la red RLC como sistema de evaluación. En Matlab generamos el archivo binario que contiene los datos necesarios para replicar en la tarjeta la aplicación. Como se ha comentado anteriormente, descargamos los datos al microprocesador mediante la consola XMD; y una vez ejecutadas todas las iteraciones del filtro, exportamos la estimación de los estados realizadas por nuestros periféricos.

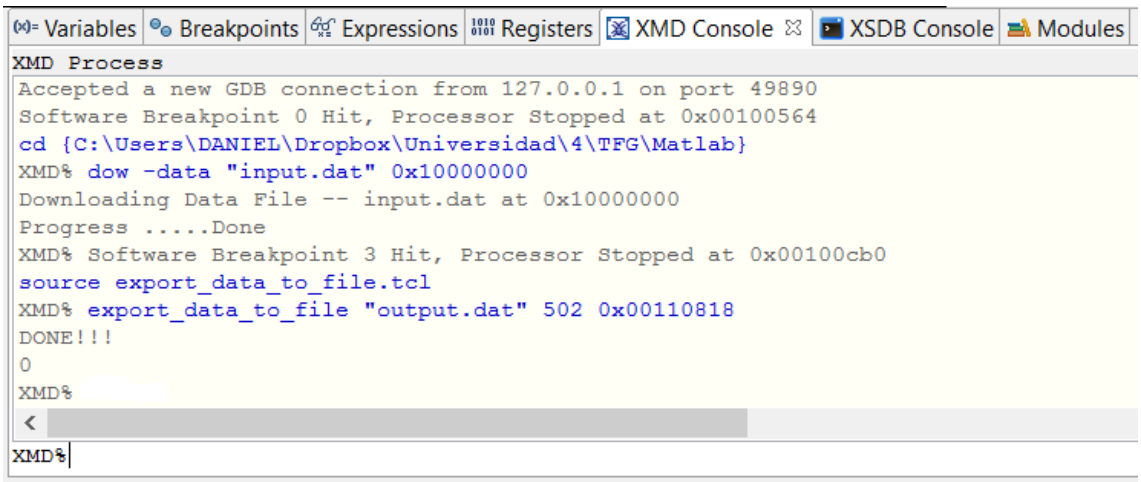


Figura 2.49. Consola XMD después de introducir los comandos de importación y exportación de archivos.

La estimación de los estados se almacena el array global *'XestBuffer[N_STATES][N_ITERATIONS]'*, de manera que cada fila corresponde a un estado y

cada columna a una iteración. En nuestro sistema, se ha simulado durante 0.5 segundos, con un tiempo de muestreo de 2 ms, lo que implica:

$$\frac{0.5\ \mathrm{s}}{2\ ms} + condiciones\ iniciales = 251\ iteraciones \tag{2.26}$$

Si tenemos 2 estados, evaluados durante 251 iteraciones cada uno, hace un total de 502 elementos que debemos exportar. El array *'XestBuffer[N_STATES][N_ITERATIONS]'* está ubicado a partir de la dirección de memoria 0x00110818, y si queremos exportar a un archivo binario su contenido basta con utilizar la función *'export_data_to_file'* y pasarle como argumento el nombre del archivo binario (output.bin), el número de elementos (502) y la dirección de memoria a partir de la que se quiere exportar.

Una vez exportados los datos, basta con reinterpretar el archivo en Matlab. Como resultado, se obtiene:

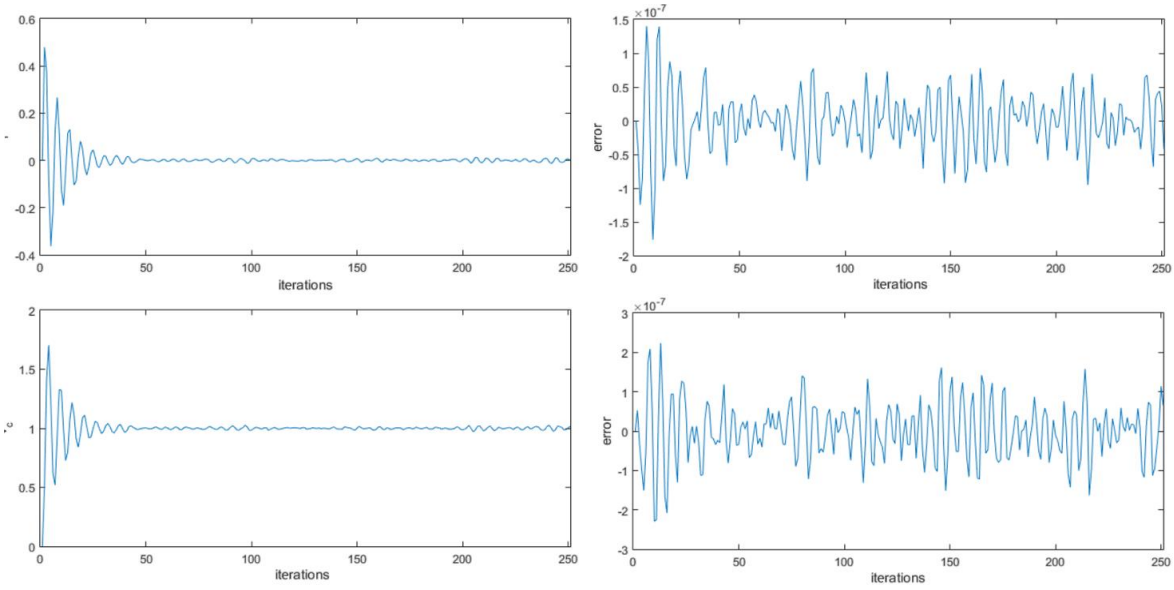


Figura 2.50. Estados estimados por la ZedBoard (izquierda) y error entre estos y los estimados en Matlab (derecha).

En la Figura 2.50 se puede comprobar que el error cometido por el sistema implementado en la ZedBoard es mínimo si lo comparamos con Matlab. Por lo tanto, hemos logrado un conjunto de periféricos que funcionando conjuntamente implementan la funcionalidad del Filtro de Kalman.

En el siguiente apartado se desarrolla un caso más cercano a la realidad, en el que no se cumplen algunas de las condiciones que aseguran una estimación óptima de los estados.

Filtro de Kalman aplicado al robot P3-DX

En este apartado se presenta un caso de aplicación del filtro de Kalman a un sistema real, como es el robot P3-DX. Se trata de un robot móvil de dos ruedas con tracción diferencial. Es un robot de propósito general, orientado principalmente a la investigación y su uso educativo.

Figura 2.51. Robot Pioneer P3-DX.

Para las pruebas con el filtro de Kalman, se ha suministrado el modelo identificado de la planta del robot, en la que se han definido como entradas las consignas de velocidad lineal y angular, los estados son la velocidad lineal, la velocidad angular y las entradas retardadas un periodo de muestreo (4 estados en total), y se han realizado mediciones en la velocidad lineal y angular del robot gracias a los encoders que incorpora en las ruedas. Las entradas retardadas aparecen como estados debido al canal de comunicación utilizado para la transmisión de las consignas.

Junto con el modelo de la planta, también se han suministrado las consignas de una trayectoria, con sus respectivas mediciones.

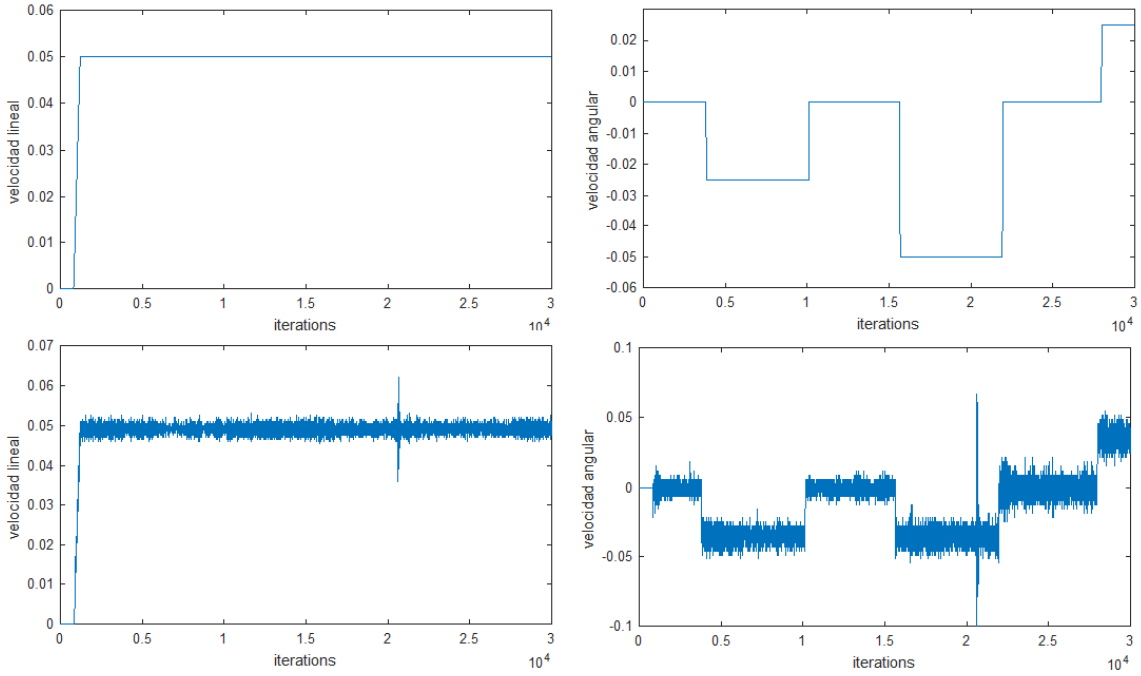


Figura 2.52. Velocidad lineal (izquierda) en consigna (arriba) y medida (abajo) y velodidad angular (derecha) en consigna (arriba) y medida (abajo).

En la Figura 2.52 se muestra el alto contenido ruidoso de las medidas. Utilizaremos el filtro de Kalman para suprimirlo.

Antes de nada, es necesario hacer algunas aclaraciones. El filtro de Kalman asegura que las estimaciones son óptimas para sistemas lineales en presencia de ruido blanco, de media cero gaussiano. En la realidad, estos requisitos son casi imposibles de alcanzar. De entrada, el ruido producido por los encoders es discontinuo en nivel, pues solo toma valores proporcionales a las cuentas realizadas. Por lo tanto, ya no es posible identificarlo como ruido blanco ni de media nula.

El modelo de la planta utilizado es una aproximación lineal fruto de una identificación, que no describe las dinámicas del robot en su totalidad, y por lo tanto ya introduce error.

Con todo, el filtro de Kalman sigue siendo una buena solución lineal al problema del filtrado de las medidas. Para su aplicación, es necesario caracterizar los ruidos presentes.

El ruido presente en las medidas es fruto de los encoders, y para modelar lo tomaremos un conjunto de medidas, y calcularemos su varianza de la siguiente manera.

$$E(x) = \bar{x} = \frac{\sum_{i=0}^{n} x_i}{n}$$
$$var_x = E((x - \bar{x})^2) = \frac{\sum_{i=0}^{n} (x_i - \bar{x})^2}{n} \quad \textbf{(2.27)}$$

Donde x es el conjunto de medidas. De esta manera, obtenemos la siguiente matriz de covarianza del ruido de medida:

$$R = \begin{pmatrix} 1.27 * 10^{-6} & 0 \\ 0 & 7.07 * 10^{-3} \end{pmatrix}$$

El ruido de proceso es, sin embargo, algo que no se puede calcular, al menos en este caso. Ese ruido simboliza el error o la inexactitud de nuestro modelo, y eso es algo que solo se puede aproximar de manera experimental. La matriz de covarianza del ruido de proceso suministrada junto con el modelo es:

$$Q = \begin{pmatrix} 2 * 10^{-4} & 0 & 0 & 0 \\ 0 & 6 * 10^{-4} & 0 & 0 \\ 0 & 0 & 2 * 10^{-4} & 0 \\ 0 & 0 & 0 & 6 * 10^{-4} \end{pmatrix}$$

Conociendo las matrices de covarianza del ruido de medida y de proceso, podemos realizar el filtrado.

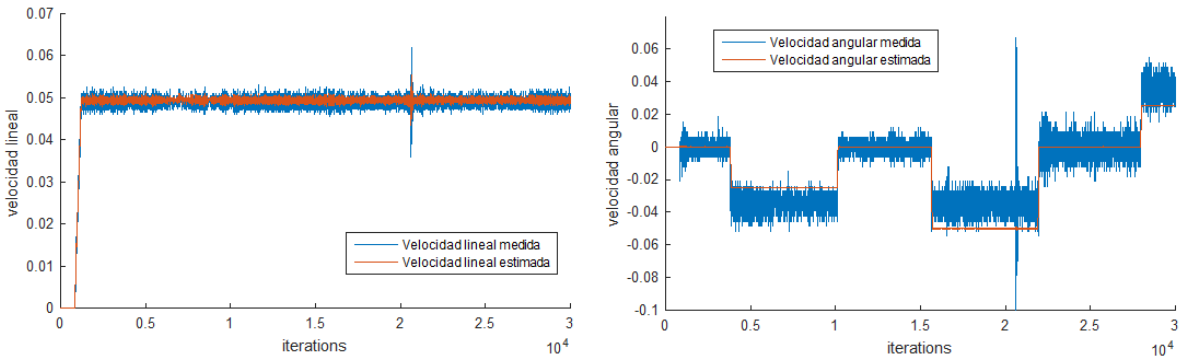


Figura 2.53. Velocidad lineal medida y estimada (izquierda) y velocidad angular medida y estimada (derecha).

En efecto, se consigue reducir el ruido de las medidas, tal y como se muestra en la Figura 2.53. La estimación de la velocidad angular filtra suavemente el ruido debido a que la covarianza de esta medida es mucho menor que la covarianza del estado en cuestión (r11<q11). El filtro considera, entonces, que la medida es mucho más fiable que el modelo, dejando pasar mayar cantidad de información.

Caso contrario ocurre con la velocidad angular. La covarianza de las medidas es mayor, y le da más peso a la información obtenida del modelo.

Al igual que en el apartado anterior, volvemos a repetir el ensayo con el sistema montado en la ZedBoard. La Figura 2.54 muestra los resultados.

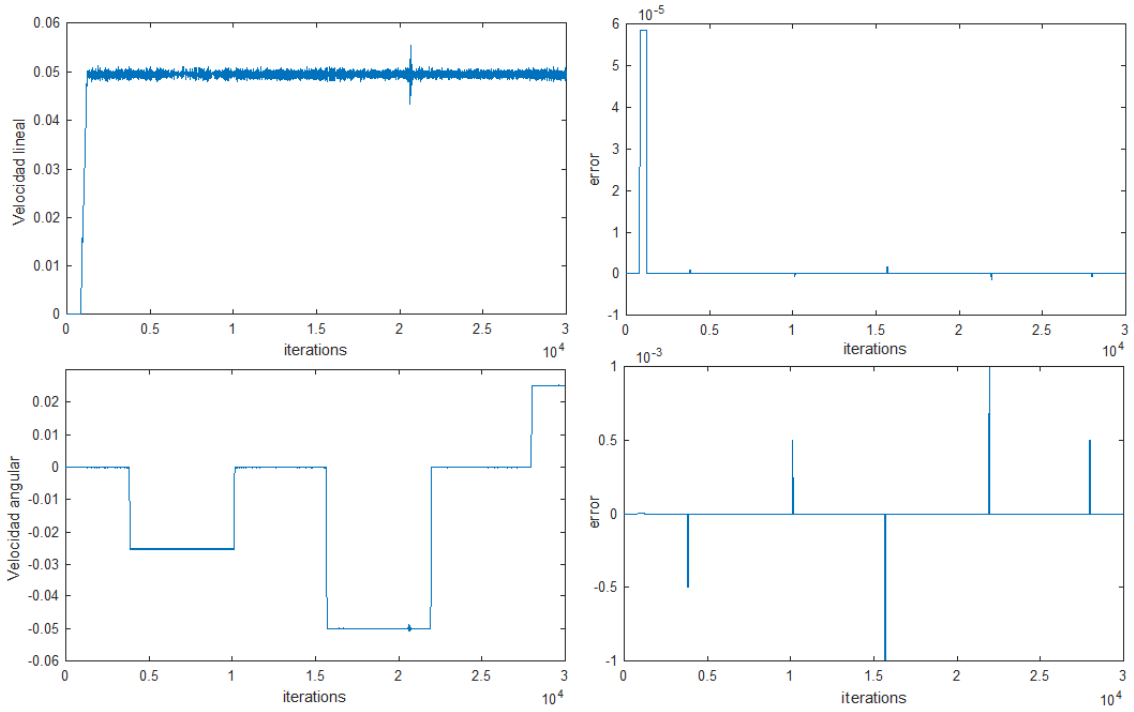


Figura 2.54. Estimaciones dadas por la ZedBoard (izquierda) y error cometido entre Matlab y la ZedBoard.

De nuevo podemos comprobar que el error introducido por el sistema desarrollado en la tarjeta de evaluación es mínimo, salvo momentos puntuales.

Los resultados obtenidos al aplicar el filtro de Kalman al robot denotan la efectividad del algoritmo incluso cuando no se dan las situaciones que aseguran unas estimaciones óptimas. Por tanto, el filtro de Kalman es una solución lineal más que viable al problema de la estimación de los estados en un sistema en ambiente ruidoso.

Resultados y conclusiones

En esta memoria se han desarrollado los procedimientos y metodología empleada para la elaboración de un filtro de Kalman en FPGA utilizando la herramienta de síntesis de alto nivel Vivado HLS. EL objetivo que nos fijamos era conseguir un diseño hardware utilizando esta herramienta que implementase el filtro, y como se ha mostrado en los apartados anteriores, se ha logrado de forma satisfactoria.

El filtro de Kalman es un algoritmo ideal para poner a prueba Vivado HLS, pues es necesario invertir matrices, proceso nada trivial de realizar siguiendo la metodología tradicional de descripción hardware. No solo se ha podido realizar esta operación, sino que además la propia herramienta incorpora librerías que permiten la implementación de esta y otras funciones algebraicas, lo que nos hace intuir el gran trabajo que se realizó

en el desarrollo de esta herramienta para evitar a los diseñadores el tener que idear sistemas fácilmente asequibles en lenguajes de alto nivel, y no tanto en la descripción hardware.

Es obvio que Vivado HLS pretende acercar al mundo hardware metodologías de diseño que hagan más fácil y cómodo el desarrollo de sistemas hardware. Estas nuevas formas de diseñar permiten ahora implementar en hardware fragmentos de código de alto nivel, o aplicaciones enteras, donde antes era una labor ardua y nada agradable. De esta manera, el mercado de las FPGAs se expande, que es el objetivo principal del fabricante, Xilinx en este caso.

Otra de las ventajas de Vivado HLS se la extrema facilidad de integración de los diseños realizados en multitud de dispositivos, tanto FPGAs como SoCs. Por ejemplo, para este trabajo no ha sido necesario desarrollar un software que manejase o accediese a los registros de control de los periféricos desarrollados, simplemente se han utilizado los drivers suministrados en la implementación de los mismos. Este tipo de cosas facilitan la labor de los ingenieros, y seduciéndolos para utilizar (y comprar) estas tecnologías.

En cuanto al propio filtro de Kalman, hemos visto como con una formulación sencilla, somos capaces de resolver el problema de la estimación de los estados dados en un sistema dinámico.

Separar el filtro de Kalman en dos periféricos distintos permiten dividir la tarea. El usuario podrá elegir si quiere realizar el cálculo de la ganancia de Kalman en régimen permanente de manera offline, para después lanzarse a la estimación de los estados, o coordinarlos y utilizar la ganancia óptima para cada estimación. El periférico encargado de la ganancia del filtro es el que soporta la mayor carga computacional y, por lo tanto, es el que más tarda en obtener resultados. Si se escoge la primera opción, todo el tiempo que se tardaría en calcular la ganancia óptima se evita, haciendo más posible el uso del periférico en una aplicación real, en el que existe una restricción de tiempo computo debido al tiempo de muestreo.

En definitiva, como resultado de este trabajo, se han desarrollado dos diseños hardware que combinados implementan un filtro de Kalman, utilizando una metodología de diseño de síntesis de alto nivel, sin haber sido necesario utilizar un lenguaje de descripción hardware

Trabajos futuros

Como se ha comentado en esta memoria, la formulación del filtro de Kalman empleada en el trabajo ha sido aquella que hace referencia a sistemas lineales invariantes en el

tiempo, pues el objetivo no era la estimación de los estados en sí. Pues bien, dando por válida la metodología que ofrece Vivado HLS, se puede profundizar en la materia.

Existen distintas versiones del filtro de Kalman, modificaciones realizadas durante el estudio y la investigación en esta materia, que abordan diferentes aspectos del problema de la estimación de los estados de un sistema, así como algunas ni idealidades que se dan en la realidad. Por ejemplo, para sistemas no lineales, disponemos de una aproximación del filtro, el EKF (Extended Kalman Filter) que permite la estimación de los estados.

La resolución de un problema de estimación de estados implementado un filtro de Kalman, en una variante más elaborada y no la más básica, en una FPGA utilizando Vivado HLS sería el siguiente paso a dar partiendo de este trabajo.

Por otro lado, existen otras herramientas que permiten exportar funciones o aplicaciones como diseños RTL. Matlab, sin ir más lejos, tiene plug-ins que permiten diseñar hardware a partir de scripts. Además, ya cuenta con una función del filtro de Kalman. Evaluar los resultados de un posible diseño empleando Matlab y compararlos con los obtenidos en HLS permitiría evaluar ambas metodologías y encontrar la más óptima.

Por último, por qué no, se podría realizar la implementación del filtro de Kalman utilizando lenguaje de descripción hardware. La elaboración del diseño sería mucho más ardua, de eso estoy seguro, pero haría patente la ventaja de utilizar una herramienta de síntesis de alto nivel tan solo con comparar el tiempo de diseño empleadas en una y otra.

Concluyendo, en este trabajo se aportan las bases de la implementación de un filtro de Kalman utilizando Vivado HLS, pero ésta no es la única herramienta, ni la formulación escogida es la más óptima ni la mejor, simplemente la más sencilla.

3. Pliego de condiciones

En este capítulo se muestran los requisitos hardware y software que han sido necesarios para la realización de este proyecto.

3.1. Requisitos hardware

- Ordenador personal (portátil) Lenovo Y50-70
  - Procesador Intel CORE i7-4170 HQ.
  - 8Gb de memoria RAM.
  - Sistema operativo de 64 bits.
- Tarjeta de evaluación ZedBoard de Digilent.

3.2. Requisitos Software

Requisitos generales:

- Sistema operativo Windows 10.
- Editor de textos NotePad++.

Requisitos específicos:

- Vivado Desing Suite 14.4, incluyendo la herramienta de síntesis de alto nivel Vivado HLS
- Matlab 2015a, incluyendo la toolbox de coma fija.

Para la redacción de la memoria:

- Word 2016.
- Adobe Acrobat Reader DC.

4. Presupuesto

Podemos dividir el coste total en los siguientes grupos:

Recursos software				
Concepto	Coste	Amortización (años)	Tiempo de uso (meses)	Total
Matlab 2015a	2.000€	4	5	208,33€
Vivado System Edition	3.495€	4	5	364,06€
NotePad++	0€	-	5	0€
Word 2016 (Licencia Universitaria)	0€	-	5	0€
Adobe Acrobat Reader DC	0€	-	5	0€
Total				572,39€

Tabla 4.1. Coste de los recursos hardware.

Recursos hardware				
Concepto	Coste	Amortización (años)	Tiempo de uso (meses)	Total
Lenovo Y50-70	1.300€	4	5	135,41€
ZedBoard	444€	4	5	46,25€
Total				181,66€

Tabla 4.2. Coste de los recursos hardware.

Mano de obra			
Concepto	Coste por hora	Horas	Total
Ejecución	30€/h	200	3.000€
Redacción	12€/h	100	120€
Total			3.120€

Tabla 4.3. Coste de la mano de obra.

Material fungible y otros costes	
Concepto	Coste
DVD-RW	1,5€
Impresión y encuadernación de libros	80€
Total	81,5€

Tabla 4.4. Coste del material fungible.

A partir de los datos expuestos, se procede a calcular el coste total del proyecto. El *coste de ejecución por contrata* se calcula a partir de la siguiente ecuación:

$$PEC = CM + b_{\%}CM$$

Donde PEC es el *presupuesto de ejecución por contrata*, CM es el coste de ejecución material del proyecto, tomado a partir de la suma de los totales de los costes de los recursos hardware, software, mano de obra y material fungible, y $b_{\%}$ el porcentaje del coste material del proyecto tomado como beneficio industrial.

El coste total de ejecución resulta:

$$CM = 572{,}39 + 181{,}66 + 3.120 + 81{,}5 = 3955{,}55€$$

Suponiendo que el $b_{\%}$ es un 10%, el presupuesto de ejecución por contrata será:

$$PEC = 3955{,}55 + 0.1 \cdot 3955{,}55 = 4.351{,}10\ €$$

A partir del PEC podemos calcular los Honorarios Facultativos (HF), fijados en un 7% del PEC cuando éste no supera los 30.050,61€:

$$HF = 0.07 \cdot PEC = 304{,}57€$$

El Coste Total (CT) del proyecto será la suma del PEC y los HF, aplicando el impuesto del valor añadido (IVA), fijado en un 21%:

$$CT = 1{,}21(PEC + HF) = 5.633{,}37€$$

El cose final del proyecto asciende a **CINCO MIL SEISCIENTOS TREINTA Y TRES EUROS CON TREINTA Y SIETE CÉNTIMOS.**

5. Anexos

5.1. Anexo 1: Inversión de matrices utilizando la descomposición de Cholesky

La inversión de matrices es una operación compleja y de alto coste computacional. Cuando estas operaciones se implementan en sistemas digitales, es común emplear algoritmos que descompongan las matrices en otras más sencillas, con el fin de reducir el esfuerzo en el cómputo de la matriz inversa. Uno de esos algoritmos es la descomposición de Cholesky.

Sea A una matriz simétrica y positiva definida, puede ser factorizada de manera eficiente por medio de una matrz triangular infereior y una matriz triangular superior. Dadas las condiciones de A, simétrica y definida positiva, la factorización resulta:

$$A = LL^T$$

Donde L (la cual podemos "verla" como la raíz cuadrada de *A*) es una matriz triangular inferior cuyos elementos de la diagonal principal son positivos.

Una variante de la factorización de Cholesky es:

$$A = R^T R$$

Donde R es una matriz triangular superior.

Para encontrar la factorización $A = LL^T$ basta con observar las ecuaciones que se derivan del producto:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix} \cdot \begin{pmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ 0 & l_{22} & \cdots & l_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & l_{nn} \end{pmatrix}$$

Se obtiene:

$$a_{11} = {l_{11}}^2 \rightarrow l_{11} = \sqrt[2]{a_{11}}$$

$$a_{21} = l_{21} l_{11} \rightarrow l_{21} = \frac{a_{21}}{l_{11}} \rightarrow \ l_{n1} = \frac{a_{n1}}{l_{11}}$$

$$a_{22} = {l_{21}}^2 + {l_{22}}^2 \rightarrow l_{22} = \sqrt[2]{a_{22} - {l_{21}}^2}$$

$$a_{32} = l_{31}l_{21} + l_{32}l_{22} \rightarrow l_{21} = \frac{a_{21} - l_{31}l_{21}}{l_{22}}$$

Generalizando para $i = 1, \dots, n$ y $j = i + 1, \dots, n$:

$$l_{ii} = \sqrt[2]{a_{ii} - \sum_{k=1}^{i-1} {l_{ik}}^2}$$

$$l_{ji} = \frac{a_{ji} - \sum_{k=1}^{i-1} l_{jk}l_{ik}}{l_{ii}}$$

Una vez obtenidos los coeficientes de la matriz L, podemos computar A^{-1}:

$$A^{-1} = L^{-1}(L^T)^{-1}$$

Siendo la matriz L^{-1} más fácil de computar por el método tradicional que A^{-1} al ser una matriz triangular, pues:

$$\det(L) = \prod_{i=1}^{n} l_{ii} \quad y \quad (l_{ii})^{-1} = \frac{1}{l_{ii}}$$

5.2. Manual de usuario

Junto con esta memoria se adjuntan los scripts y funciones de Matlab utilizados para comprobar la funcionalidad del filtro de Kalman, así como los archivos fuente para los proyectos de Vivado HLS y Vivado Desing Suite.

Matlab

Dentro del directorio *Matlab*, se encuentran todos los archivos generados en Matlab, que son:

Funciones:

- *Kalman_filter:* calcula la estimación de un estado para un instante. Recibe como argumentos las matrices de la planta, la ganancia del estimador, el valor de los estados en el instante anterior, la entrada a la planta y el valor medido en la salida de la planta.
- *Kalman_filter_fixed:* realiza la misma operación que la función anterior, pero utilizando datos en formato de coma fija. Además del resto de argumentos, recibe el ancho de palabra de los datos y el número de bits decimales.
- *Kalman_gain:* calcula la ganancia de Kalman y la covarianza del error de estimación para un instante. Recibe como argumentos las matrices de la planta, las matrices de covarianza del ruido de proceso y de medida y el valor anterior de la covarianza del error de estimación.
- *Kalman_gain_fixed:* realiza la misma operación que la función anterior, pero utilizando datos en formato de coma fija. Además del resto de argumentos, recibe el ancho de palabra de los datos y el número de bits decimales.
- *Plotear:* esta función recibe como argumentos dos vectores bidimensionales. Se entiende por vector una fila de datos ((1, :) en Matlab). La función dibuja la primera fila de cada vector en una gráfica y la segunda en otra, de manera que se puede comparar gráficamente las dos filas de ambos vectores entre sí.
- *Errorcm:* calcula el error cuadrático medio entre dos vectores de entrada.

Scripts:

- *Evalución_teórica*: en este script se realizan diversas simulaciones sobre un sistema ejemplo para comprobar el funcionamiento del filtro de Kalman, tanto en coma fija como en coma flotante. Los resultados de las simulaciones se almacenan en el fichero *datos_ev_teorica.mat*

- *Kf_p3dx_real*: en este script se realizan simulaciones sobre el robot Pioneer P3-DX utilizando los datos suministrados en los archivos *modelo_p3dx_cont.mat* y *velocidad.mat.* Los resultados de la simulación se almacenan en el fichero *datos_kf_p3dx.mat.*
- *Generar_data_input*: genera un archivo de texto con los datos necesarios para realizar la simulación C en Vivado HLS.
- *Leer_data_output*: lee los datos obtenidos en Vivado HLS.
- *Generar_input_SDK*: genera un archivo binario que contiene los datos necesarios para realizar una simulación del sistema montado en SDK.
- *Leer_output_SDK*: lee el fichero binario fruto de la simulación en SDK.

Vivado HLS

En Vivado HLS se han desarrollado dos periféricos. Para ello, se han seguido los siguientes pasos:

Kalman_gain

Inicialmente se crea un nuevo proyecto de Vivado HLS en el directorio deseado, con nombre *Kalman_gain, y pulsamos next*:

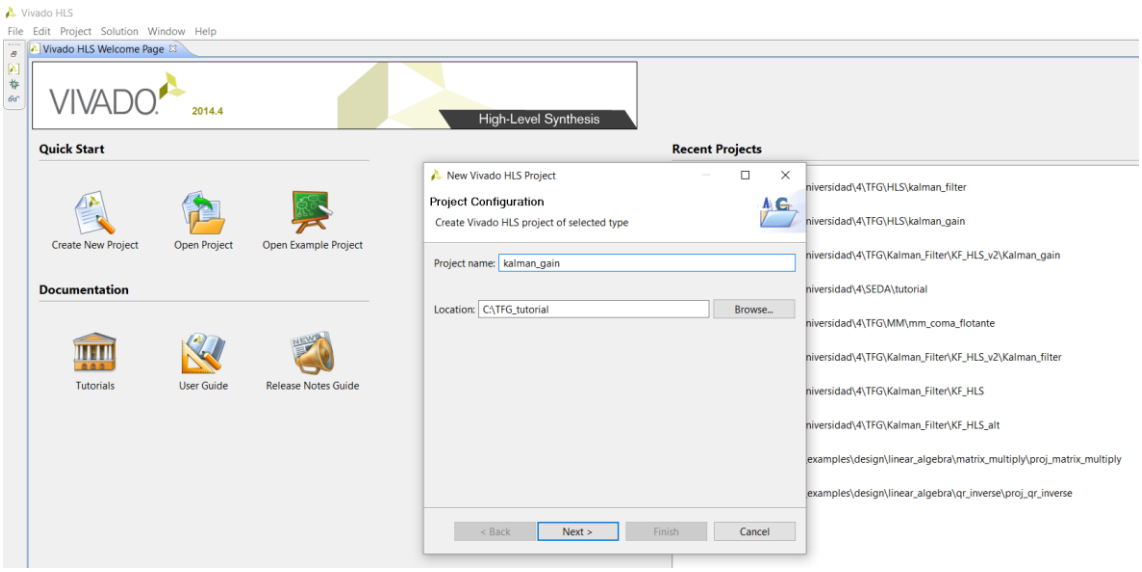


Figura 5.1. Creación del proyecto kalman_gain.

Ahora debemos incluir el código fuente para la síntesis. En el directorio raíz del CD suministrado se encuentra la carpeta fuentes, y en ella una sub-carpeta denominada HLS. Dentro de ella se encuentran los archivos fuente para la síntesis de este proyecto.

Añadimos los archivos mostrados en la Figura 5.2, especificando como top-function *Kalman_gain.*

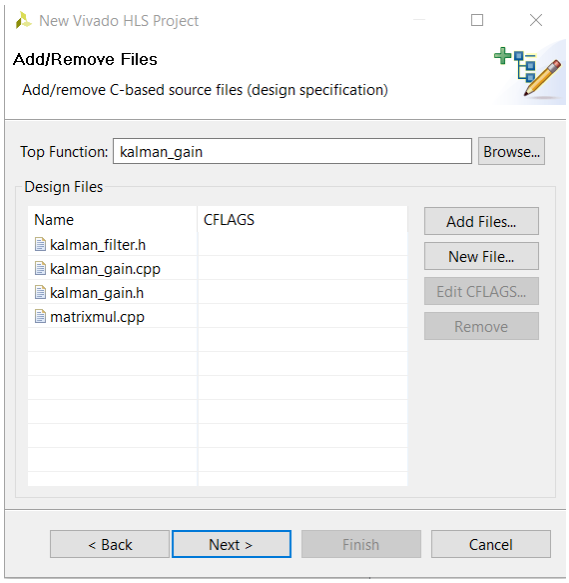


Figura 5.2. Añadimos los archivos fuente.

Posteriormente añadimos el test bench.

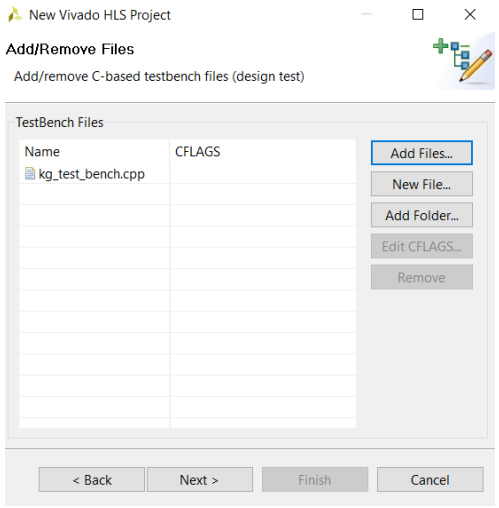


Figura 5.3. Añadimos el test bench.

En la siguiente ventana, fijamos el periodo del reloj a 10 ns y especificamos el dispositivo al que va dirigido el diseño, en este caso, la tarjeta de evaluación ZedBoard.

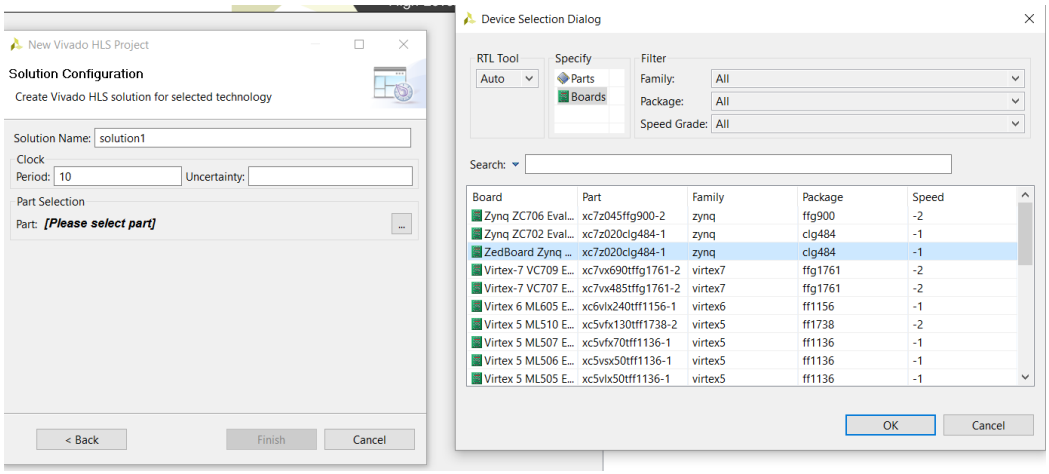


Figura 5.4. Seleccionamos la ZedBoard como dispositivo.

Pinchamos en el botón finish para terminar de crear el proyecto.

Una vez creado el proyecto, ya podemos empezar a sintetizar el diseño. Para asegurar su correcta funcionalidad, podemos depurar la función C antes de sintetizarla pinchando en el botón  o pinchando en *Project-> Run C simulation*. En la simulación C se ejecutará el test bench. El test bench lee el archivo de texto generado con Matlab y extrae los datos necesarios para la simulación, y escribe en otro archivo de texto las salidas de la funcion. Es necesario especificar la dirección donde se encuentran los archivos de texto.

Con la función ya depurada se puede realizar la síntesis. Para ello debemos añadir primero las directivas de optimización. Estas se pueden añadir gráficamente en la pestaña *directives,* situada en la parte derecha.

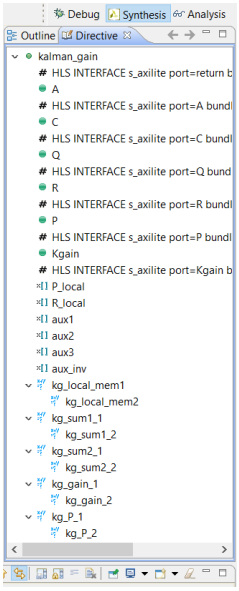


Figura 5.5. Pestaña de directivas.

Lo primero es optimizar la función *matrixmul*. Aplicamos la directiva *pipeline* sobre los bucles *local_mem2*, *col* y *global_mem2* haciendo *click derecho -> insert directive -> pipeline* sobre ellos.

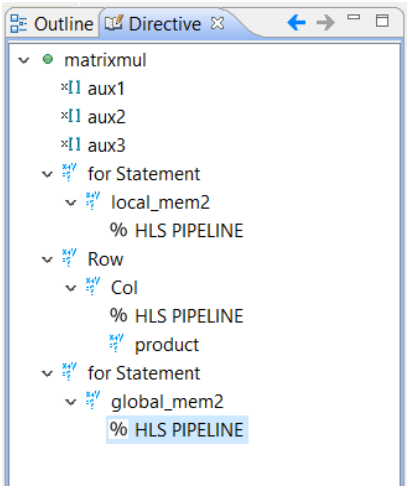


Figura 5.6. Directivas de optimización en matrixmul.

También aplicaremos la directiva *pipeline* a la función *matrix_multiply* utilizada dentro de *Cholesky_*inverse, concretamente al bucle *b_col_loop* de *matrix_multiply_default*.

Por último, optimizamos la función *Kalman_gain.* Aplicamos *pipeline* a los bucles *kg_local_mem2, kg_sum1_2, kg_sum2_2, kg_gain_2* y *kg_P_2.*

También es necesario especificar el protocolo de interfaz que va a seguir el diseño. Para ello, asegurarse que las sentencias pragma no están entre comentarios.

```
5  void kalman_gain(
6          data_fi A[MAX_TAMANO][MAX_TAMANO],
7          data_fi C[MAX_TAMANO][MAX_TAMANO],
8          data_fi Q[MAX_TAMANO][MAX_TAMANO],
9          data_fi R[MAX_TAMANO][MAX_TAMANO],
10         data_fi P[MAX_TAMANO][MAX_TAMANO],
11         data_fi Kgain[MAX_TAMANO][MAX_TAMANO]
12         ){
13 #pragma HLS INTERFACE s_axilite port=return bundle=KALMAN_GAIN_PERIPH_BUS
14 #pragma HLS INTERFACE s_axilite port=A bundle=KALMAN_GAIN_PERIPH_BUS
15 #pragma HLS INTERFACE s_axilite port=C bundle=KALMAN_GAIN_PERIPH_BUS
16 #pragma HLS INTERFACE s_axilite port=Q bundle=KALMAN_GAIN_PERIPH_BUS
17 #pragma HLS INTERFACE s_axilite port=R bundle=KALMAN_GAIN_PERIPH_BUS
18 #pragma HLS INTERFACE s_axilite port=P bundle=KALMAN_GAIN_PERIPH_BUS
19 #pragma HLS INTERFACE s_axilite port=Kgain bundle=KALMAN_GAIN_PERIPH_BUS
```

Figura 5.7. Sentencias pragma.

Con las direcciones de optimización fijadas, lanzamos la síntesis pulsando en el botón v o bien Solution -> Run C Synthesis. Vivado HLS emitirá la información de la síntesis una vez finalice.

Tras la síntesis, ya tenemos un diseño que podemos exportar al IP Catalog de Vivado. Pinchando en el botón o en Solution -> Export RTL realizamos este proceso.

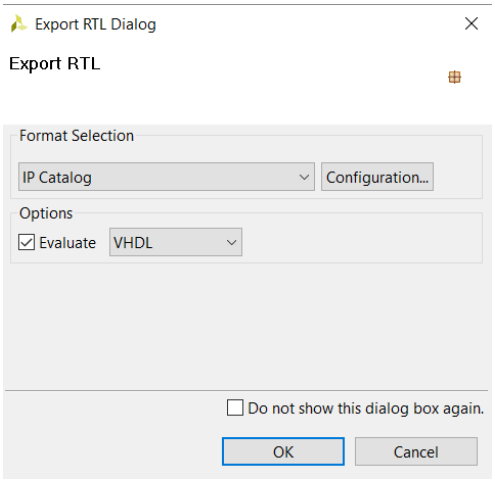


Figura 5.8. Exportar RTL.

Si seleccionamos la opción *Evaluate*, HLS comprobará los recursos hardware que requerirá la implementación del IP, y dará una información más precisa que la de la síntesis.

Una vez exportado el RTL, se creará un archivo comprimido, que contiene la información del IP, dentro de la carpeta *IP* en el directorio *impl* de la solución que se ha exportado. Ese archivo se puede copiar y descomprimir en otro directorio, para después poder añadirlo al repositorio de IPs de Vivado.

Kalman_filter

De la misma forma que en el otro diseño, creamos un nuevo proyecto *kaman_filter* para este periférico. Esta vez, añadimos los siguientes archivos:

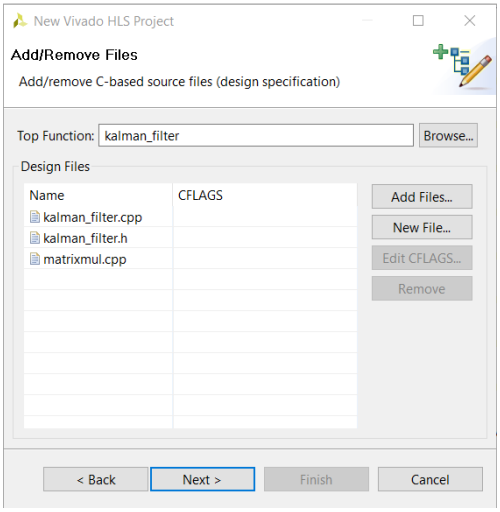


Figura 5.9. Añadimos los archivos fuente a kalman_filter.

Especificamos como top-function *Kalman_filter.* En la pestaña siguiente añadimos el test bench.

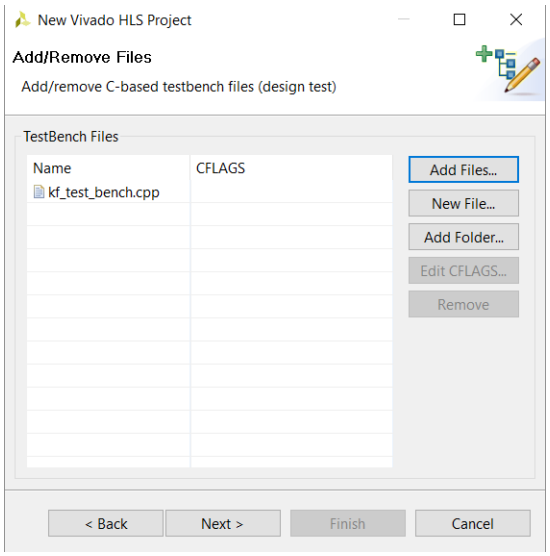


Figura 5.10. Añadimos el test bench.

Posteriormente, volvemos a seleccionar la ZedBoard como dispositivo en el que se va a implementar el diseño.

Una vez está el proyecto creado, podemos volver a lanzar la simulación C para asegurarnos del correcto funcionamiento del código. Tras esto, procedemos a la optimización del diseño.

Es diseño vuelve a utilizar la función *matrixmul*, que optimizaremos de la misma manera que en *Kalman_gain.*

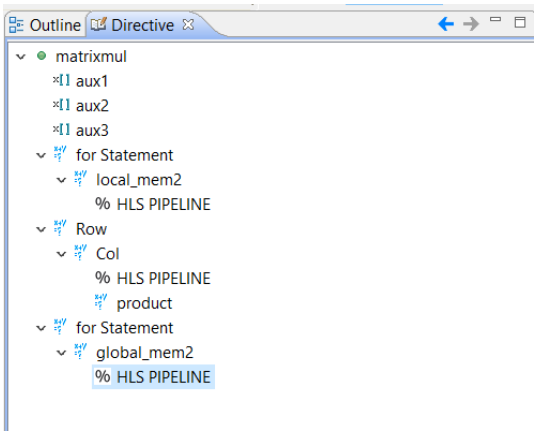


Figura 5.11. Directivas de optimización en matrixmul.

Para la función *Kalman_filter* usaremos *pipeline* en los siguientes bucles: *kf_local_mem2, kalman_filter_label1, kalman_filter_label2* y *kalman_filter_label3.* De nuevo nos debemos asegurar que las sentencias pragma estén fuera de comentarios.

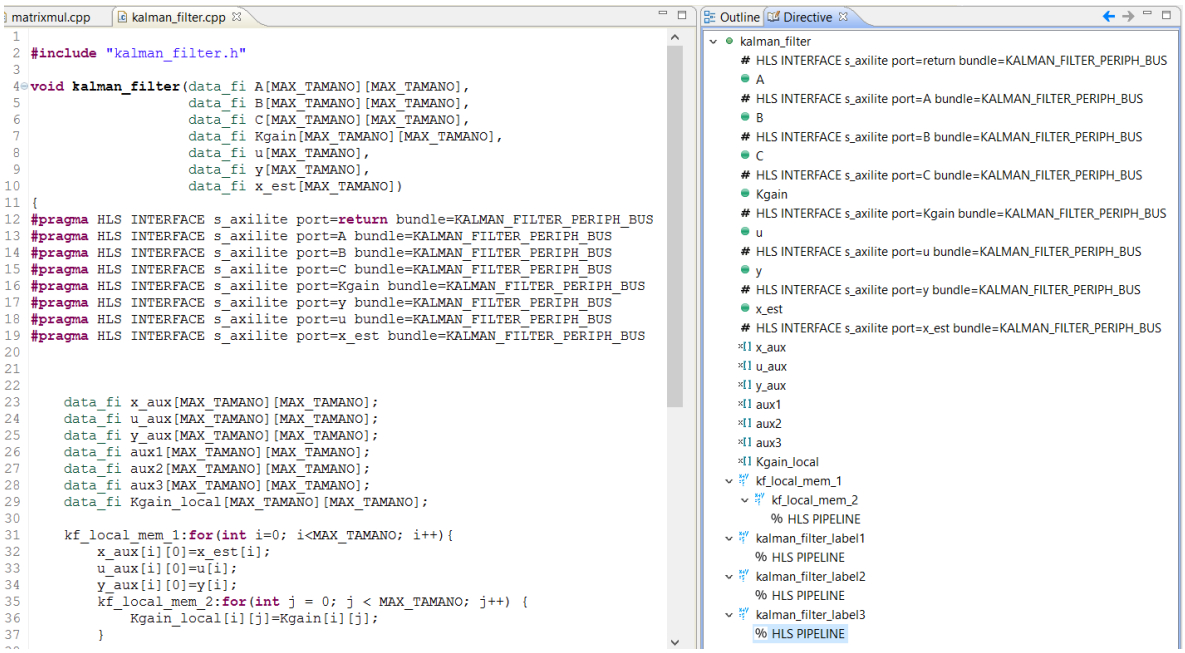


Figura 5.12. Directivas de optimización en Kalman_filter.

Una vez realizada la síntesis, exportamos el RTL para poder utilizarlo en Vivado.

Vivado y SDK.

Una vez terminados los diseños de los dos periféricos, lanzamos la herramienta Vivado y creamos un nuevo proyecto, de nombre *kf_vivado*. Para tener todos los archivos organizados, crearemos el proyecto en el mismo directorio en el que se han creado los proyectos de Vivado HLS.

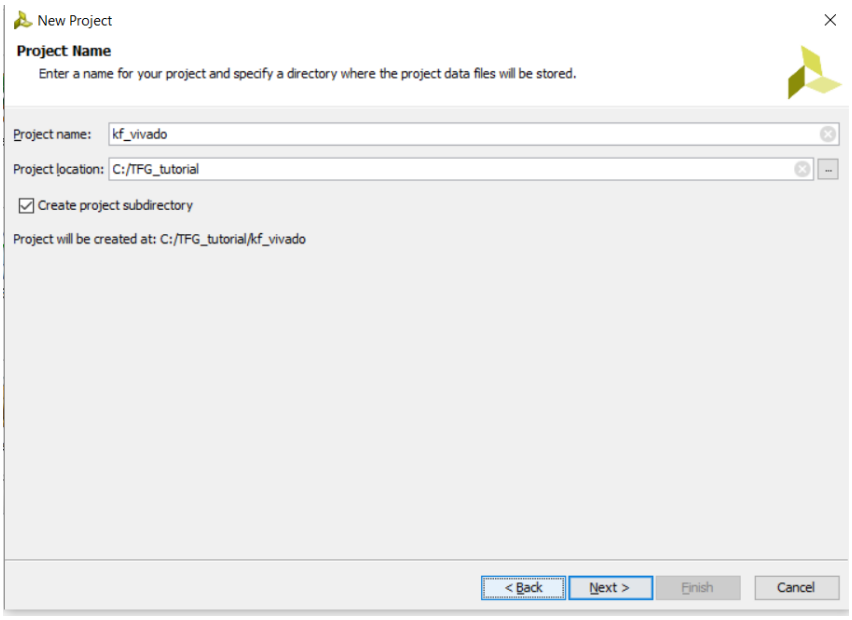


Figura 5.13. Creamos el proyecto de Vivado.

En la siguiente pestaña, seleccionamos *RTL Project* y no especificamos los archivos fuentes. Por último, seleccionamos la ZedBoard como dispositivo.

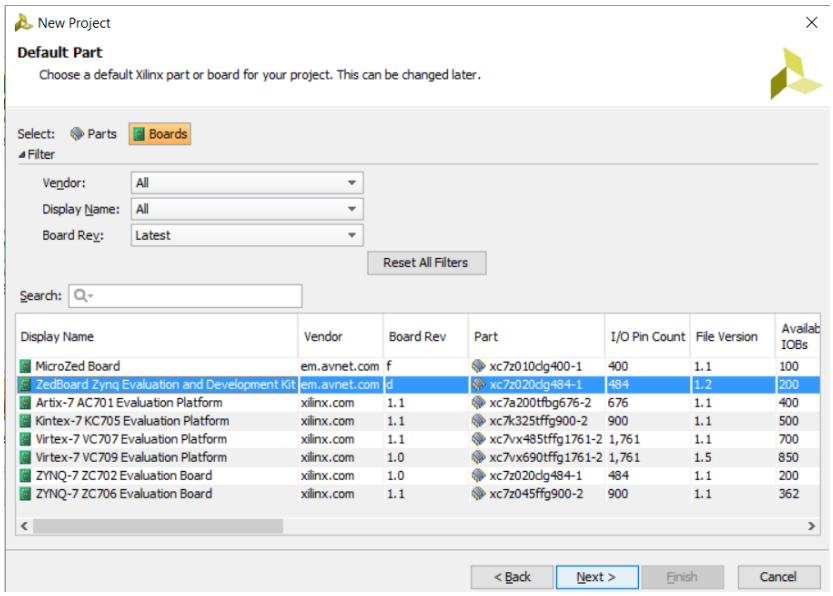


Figura 5.14. Seleccionamos la ZedBoard

Procedemos ahora a la elaboración del sistema completo. El diseño se hará de forma gráfica, por lo tanto, pinchamos en la opción *Create block desing* y creamos un diseño por bloques, de nombre *kf_desing.*

Figura 5.15. Creamos un diseño por bloques.

Para poder disponer de los diseños realizados en HLS, es necesario añadirlos al repositorio de IPs. Para ello, copiaremos los archivos comprimidos generados tras la exportación RTL y los pegaremos en una nueva carpeta, denominada *ip_repo.* Se recomienda que esta carpeta se cree en el mismo directorio que los proyectos de Vivado HLS y Vivado.

Posteriormente, en la pestaña *IP* de la ventana *Project_settings* añadimos la carpeta *ip_repo* al repositorio.

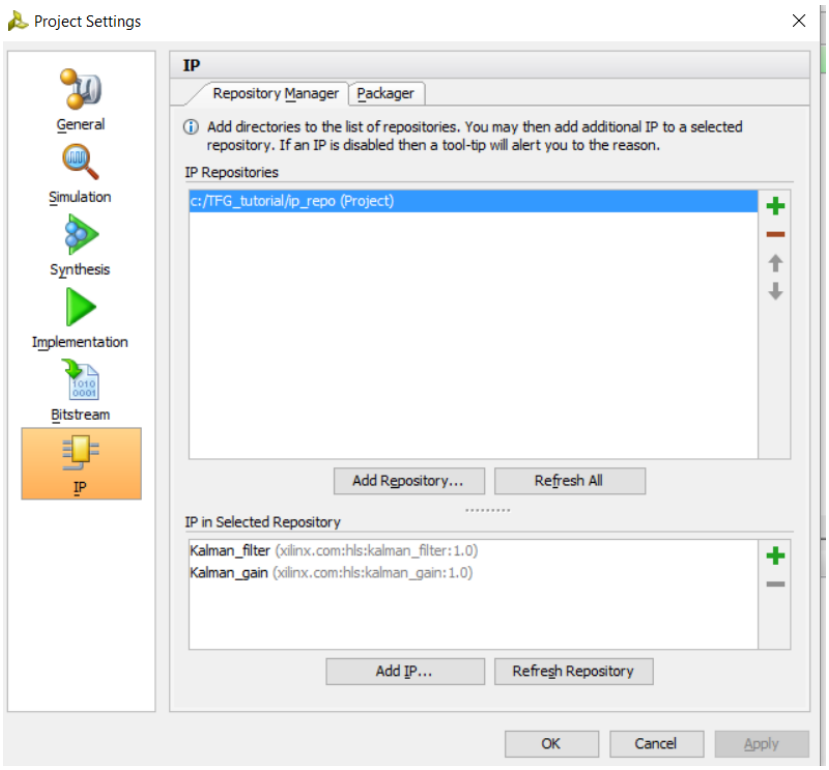


Figura 5.16. Añadimos los IPs al repositorio de Vivado.

Como todo proyecto destinado a Zynq, el primer bloque que debemos añadir al diseño es el *ZYNQ7_Processor_System*. Dejamos que Vivado configure automáticamente el entorno de este bloque pinchando en la opción *Run Block Automation.*

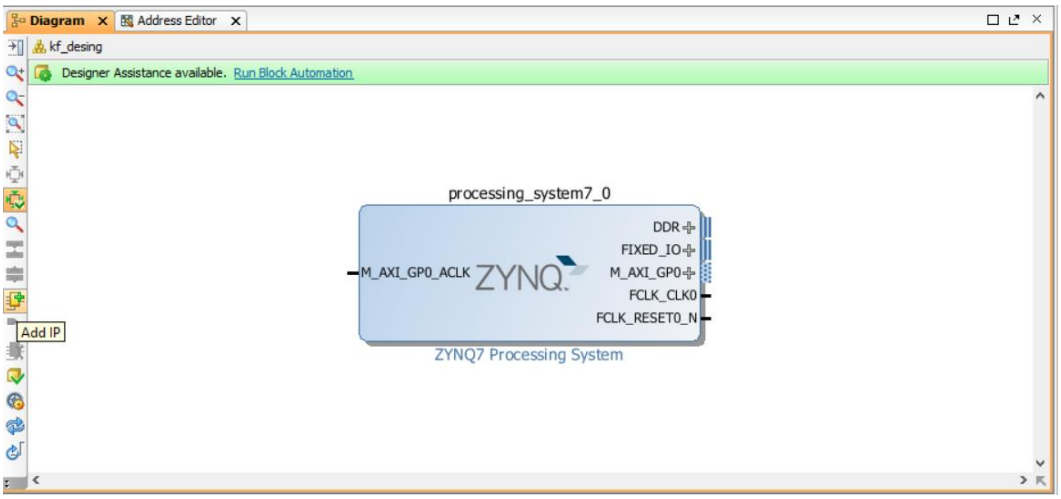


Figura 5.17. añadimos el bloque ZYNQ7_Processor System.

Ahora debemos habilitar la entrada para las señales de interrupción desde la FPGA. Hacemos *click derecho -> cutomize block* y en l pestaña *Interrupts* habilitamos las interrupciones desde la PL al PS.

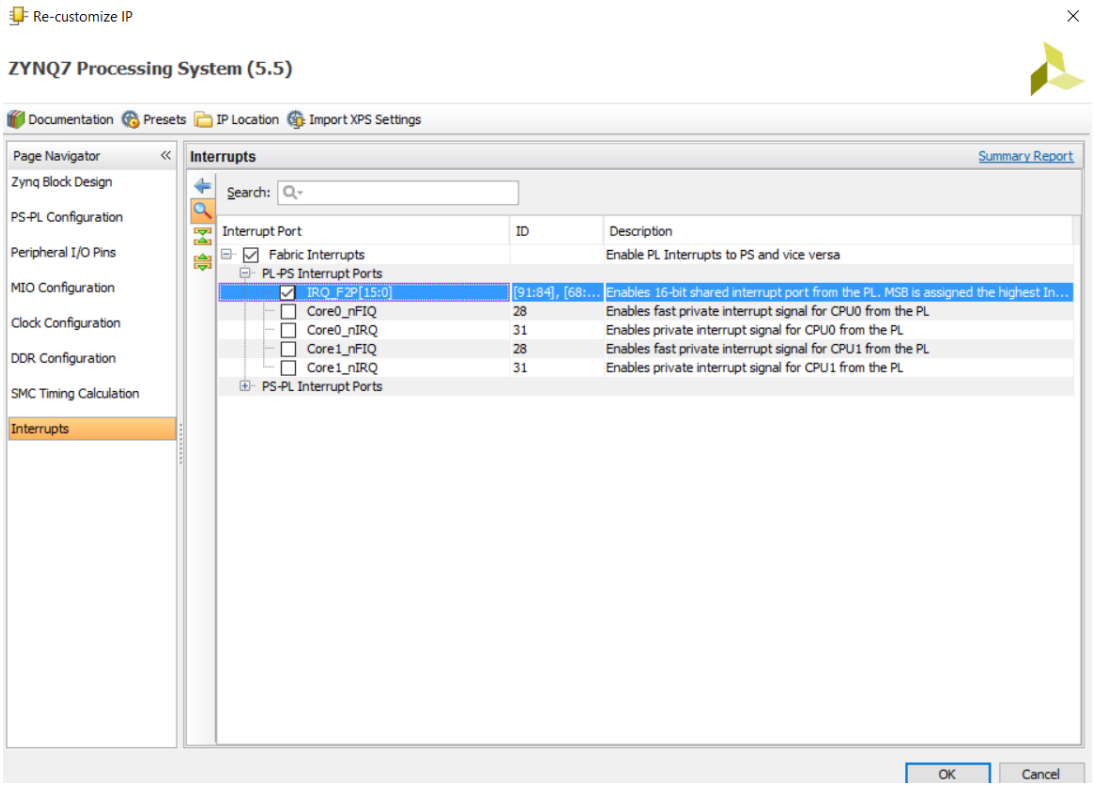


Figura 5.18. Habilitamos la entrada de interrupciones.

Solo falta añadir los diseños realizados en HLS. De nuevo dejamos que Vivado se encargue de la interconexión de los bloques. Para conectar las señales de interrupción

de los periféricos al PS, es necesario utilizar el bloque *Concat.* El diseño final resultaría así:

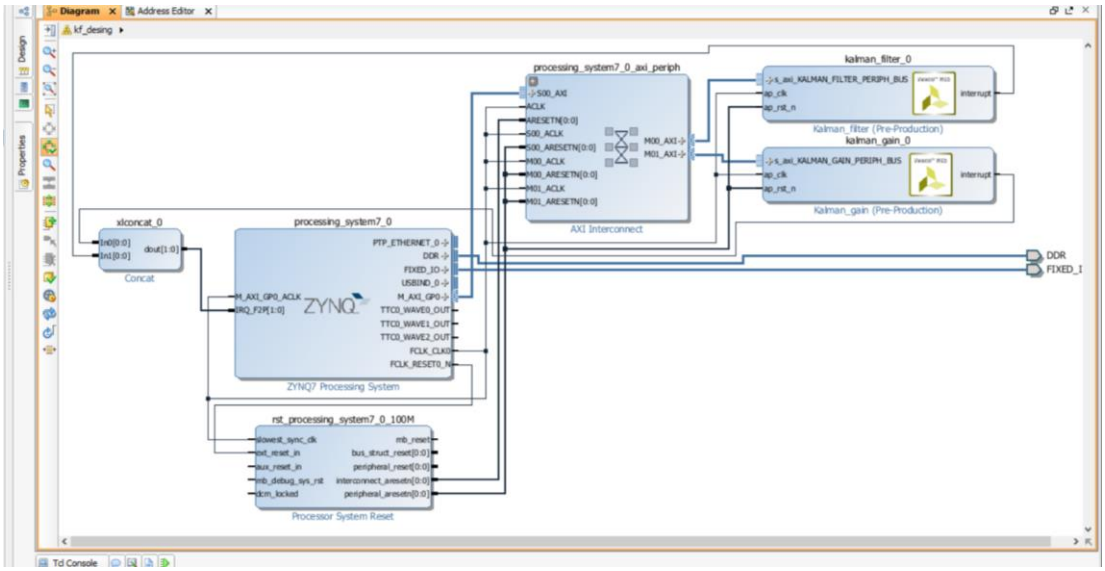


Figura 5.19. Diseño final.

Con el sistema montado, podemos proceder a generar el bitstream y lanzar la herramienta de desarrollo software. Para ello, en la pestaña *Desing -> Sources* hacemos *click derecho -> Create HDL Wrapper,* dejamos que Vivado maneje el proceso y lanzamos la generación del bitstream.

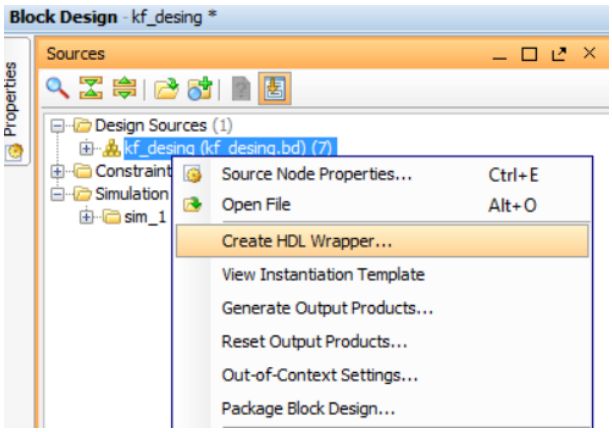


Figura 5.20. Creamos el envoltorio HDL.

Cuando el proceso haya terminado, pinchamos en *File -> export -> export hardware -> include bitstream* y lanzamos el SDK.

En el SDK ya disponemos de toda la información necesario para elaborar una aplicación software que ejecutará el sistema desarrollado en Vivado. Para ello, pinchamos en *file -> new -> application Project -> empty Project,* y llamamos al Proyecto *kf.*

En la carpeta *Fuentes* del CD suministrado existe un directorio llamado SDK, y dentro se encuentran los archivos fuente para la aplicación software. Para incluirlos en SDK, los copiamos en *kf_vivado -> kf_vivado.sdk -> kf -> src.* Una vez incluidos, deberían aparecer en el explorador del proyecto:

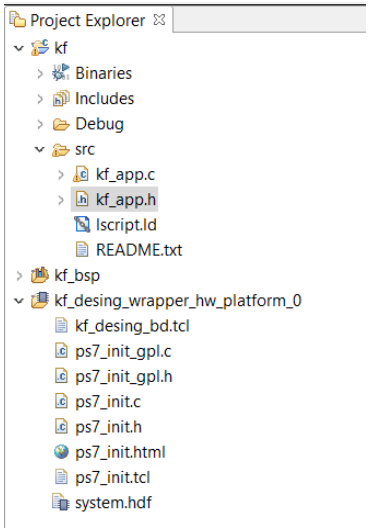


Figura 5.21. Así aparecen los archivos fuentes dentro de la jerarquía del proyecto.

Esta aplicación permite la simulación offline del filtro de Kalman. En el archivo de cabecera *kf_app.h* se encuentran definidos todos los parámetros que permiten adaptar la aplicación a diferentes situaciones. Es offline porque tanto la importación como la exportación de los datos se hace a través de la consola XMD en el modo depuración.

Los comandos necesarios para la ejecución del código correctamente son:

- *cd {<directorio de trabajo>}:* cambia el directorio de trabajo de la consola. Para utilizar archivos y scripts es necesario estar en el mismo directorio en el que se encuentran.
- *dow -data "<nombre del archivo>" <dirección(hex)>:* descarga un archivo binario a la memoria DDR del procesador a partir de la dirección especificada.
- *source export_data_to_file*: permite utilizar la función *export_data_to_file* definida en el script export_data_to_file.tcl. Este script se encuentra en la carpeta Matlab.
- *Export_data_to_file "<nombre del archivo>" <número de elementos> <dirección(hex)>*: escribe (o crea) en un archivo binario el número de elementos situados a partir de la dirección especificada.

Al lanzar el modo depuración, cambiamos el directorio de trabajo a donde se encuentren los archivos que se van a utilizar. Como los archivos binarios se han generado con Matlab, éstos se encuentran el directorio homónimo, al igual que el script export_data_to_file.tcl. Antes de ejecutar ninguna instrucción, descargamos los datos a

la memoria DDR introduciendo el comando “dow -data "input.dat" 0x10000000”. Mantener esa dirección de memoria es importante, pues en el código se especifica que los datos de entradas se encuentran a partir de esa dirección. Cuando se ejecuta toda la simulación, la aplicación se queda parada en un bucle sin fin. En ese momento podemos utilizar la función *export_data_to_file* para extraer los datos que queramos.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá