Diseño e implementación en FPGA de un filtro Kalman Unscented para aplicaciones biomédicas

Trabajo Fin de Grado
Grado en Ingeniería de Computadores

Pablo Lammers Corral
Manuel Pascual López
Daniel del Pino Sánchez

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

Junio 2017

Documento maquetado con TEXIS v.1.0+.

Diseño e implementación en FPGA de un filtro Kalman Unscented para aplicaciones biomédicas

Trabajo fin de grado

Grado en Ingeniería de Computadores

Autores

Pablo Lammers Corral
Manuel Pascual López
Daniel del Pino Sánchez

Dirigida por el Doctor

Oscar Garnica Alcázar
Juan Lanchares Dávila

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

Junio 2017

Agradecimientos

Este proyecto ha supuesto un duro trabajo y sacrificio para todos, por tanto, no podemos empezar de otra forma sino dando gracias a nuestros directores Óscar Garnica Alcázar y Juan Lanchares Dávila por habernos propuesto este proyecto y por invertir su tiempo y dedicación en ayudarnos a desarrollarlo. Gracias a ellos hemos aprendido todo lo relativo al Filtro Kalman y demás entresijos.

Gracias a la Universidad Complutense de Madrid y a la Facultad de Informática por permitirnos haber estudiado este magnífico grado con el que tanto hemos disfrutado y en especial a todos los profesores que nos han acompañado durante estos cuatro años y han contribuido a nuestra formación para afrontar este trabajo.

Por último y no menos importante, dedicamos este trabajo a todos nuestros familiares, amigos y compañeros que nos han soportado durante estos años y en especial este último.

Para todos vosotros este trabajo.

Resumen

Actualmente, la vida de los pacientes con Diabetes Mellitus está muy controlada. Su enfermedad no les permite mantener un nivel estándar de glucosa en sangre y necesitan de insulina externa para bajarla, ya que este tipo de diabetes suele generar altos niveles de glucosa en sangre o hiperglucemia, que es peligroso para su vida. Otra causa de muerte más peligrosa todavía es el caso contrario, la falta de glucosa en sangre o hipoglucemia, que se puede dar por no llevar una buena dieta o por pasarse en la dosis de insulina.

Para mejorar la calidad de vida de los diabéticos se han desarrollado durante años Monitores Continuos de Glucosa (MCG). Estos miden la Glucosa Intersticial (GI) entre las capas de la piel, derivada de la Glucosa en Sangre (GS). La GI está relacionada con la GS, pero no son el mismo valor. Debido al retardo provocado por todas las capas de piel, hasta llegar a la zona intersticial la GS se va difundiendo y bajando. Además, la medición de la GI es ruidosa y degenerada por la degradación del sensor llamada ganancia del sensor.

Un gran cambio en los MCG fue el inicio de la aplicación de Filtros Kalman para estimar el valor de la GS. La capacidad de este filtro reside en que, con medidas indirectas de una variable que se quiera estimar, es capaz de pronosticar el estado oculto, que es el valor de la variable que nos interesa.

El objetivo de este Trabajo de Fin de Grado es usar un Filtro Kalman Unscented para la estimación de la GS (estado oculto) a través de las medidas de un sensor de glucosa situado debajo de la piel del paciente y que mide la GI (medida rápida, cada cinco minutos). Además se dispondrá de otra medida muy fiable, ya que se medirá la glucosa en sangre directamente mediante un pinchazo en el dedo, para la calibración y corrección del error del filtro (medida lenta, cada ocho horas).

El filtro está implementado en una FPGA por sus características de robustez, seguridad y velocidad de cómputo. El resultado del uso del filtro es un sistema capaz de aproximar la GS y la ganancia del sensor a partir de una medida ruidosa fuertemente no lineal.

Palabras clave:

Filtro Kalman, Filtro Kalman Unscented, VHDL, punto fijo, operaciones matriciales.

Abstract

Actually, Diabetes Mellitus patient's live is so difficult. This disease don't able themselves to manage an standard blood glucose level and need extern source insulin to decrease it because this kind of diabetes uses to raise blood glucose level or hyperglycemia, dangerous for them. Another more dangerous cause of death is the opposite, the low level blood glucose or hypoglycemia, that could be present because of a bad diet or take so much insulin than the normal dose by an accident.

To improve the diabetics quality of live Continous Glucose Monitors (CGM) has been developed for years. This devices sample Interstice Glucose (IG) between skin layers, that came from the Blood Glucose (BG). The IG is related to BG, but isn't the same values. Because of all skin layers delay until came into interstice area the BG spreads himself and decrease. In addition, the IG sampling is noisy and degenerate, because of sensor degradation called sensor gain.

A grate CGM change was the Kalman Filtering using to BG estimation. The feature of this filter besides in the ability of variable hidden state estimation, that it's what we want, using indirect variable sampling.

The goal of this final year dissertation is to use an Unscented Kalman Filter to BG estimation (our hidden state) using IG under skin glucose sensor sampling (fast measure, five minutes frequency). In addition we are able to use an BG sample by an fingerstick device, a very reliable sample, used to filter calibration and sintonization (slow measure, eight hour frequency).

The filter is implemented on a FPGA because of it's good features of strenght, security and compute speed. The result is a device that can estimate BG and sensor gain using hard non-lineal measure.

Keywords:
Kalman Filter, Unscented Kalman Filter, VHDL, fixed point, matrix operations.

Copyright

Autores:
Pablo Lammers Corral
Manuel Pascual López
Daniel del Pino Sánchez

Índice

Índice de figuras

Índice de Tablas

Parte I

Motivaciones y objetivos

Capítulo 1

Motivaciones

La Diabetes Mellitus del tipo 1 es una enfermedad por la cual el páncreas no es capaz de controlar la glucosa en sangre mediante la segregación de insulina propia. Quienes sufren esta enfermedad necesitan insulina externa para hacer entrar la glucosa desde la sangre hasta las células y evitar la hiperglucemia, que es exceso de glucosa en sangre. También puede ocurrir el caso contrario, la hipoglucemia, que es la falta de glucosa en sangre y que se da por fallos al administrar la insulina o por una mala dieta del paciente. La hiperglucemia y la hipoglucemia pueden provocar desde daños menores, pasando por problemas cerebrales, hasta la muerte de los pacientes si no se provee de un mecanismo de control y de segregación de insulina al paciente.

Para mejorar la calidad de vida de los diabéticos se han desarrollado durante años Monitores Continuos de Glucosa (MCG). Estos aparatos miden la Glucosa Intersticial (GI) entre las capas de la piel, derivada de la Glucosa en Sangre (GS). La GI está relacionada con la GS, pero no son el mismo valor, sino que tienen dinámicas distintas:

- La GS no se distribuye uniformemente por todas las células del organismo, sino que va fluctuando y alimentando a todas ellas mientras se mueve. Por eso, al medir la glucosa cuando llega al tejido intersticial con un sensor, no se consigue nunca una medida fiable. Siempre contiene un ruido impredecible que está provocado por muchas variables.

- El sensor de glucosa está expuesto a muchos agentes distintos, tales como los anticuerpos del paciente o la oxidación de la glucosa en el sensor que hacen que cuanto más tiempo lleve dentro del organismo del paciente, menos fiable sea la medida. El tiempo normal de vida de uno de estos sensores es de entre seis y doce días.

- El sensor de glucosa se sitúa en el tejido intersticial, de donde se mide la GI, que tiene su propia dinámica y velocidad de cambio. Está relacionada con la GS pero no tanto como para depender solamente de ella. Esto hace que la GS tenga unos valores distintos a la GI.

- Existe un retardo provocado por todas las capas de piel que se encuentran entre la arteria y donde se encuentra el sensor, debido a que la glucosa fluctúa más rápido en el torrente sanguíneo que en la piel.

Esta dinámica de la GI afecta a la fiabilidad de los MCG y por esto se inició el uso de los Filtros Kalman, para estimar el valor de la GS a partir de la GI. El Filtro Kalman es un tipo de filtro bayesiano cuya base es un modelo del sistema. Gracias a este modelo consigue estimar la variable que se quiere conocer o estado oculto del sistema a través de mediciones indirectas de la misma variable.

La solución que propone este Trabajo de Fin de Grado para evitar los errores en el cálculo de la GS es usar un Filtro Kalman Unscented para la estimación de la GS a través de las medidas de un sensor de glucosa situado debajo de la piel del paciente que mide la GI y una observación fiable de un medidor de glucosa externo que se usará a modo de calibración. La estimación que queremos hacer es sobre una variable aleatoria fuertemente no lineal con ruidos no gaussianos ni blancos.

Capítulo 2

Objetivos

Nuestro objetivo en este Trabajo de Fin de Grado es estimar mediante un Filtro Kalman Unscented la GS del paciente usando medidas indirectas de un sensor de GI cada cinco minutos, afectado por el ruido y la degradación del funcionamiento del propio sensor, y una observación directa de la GS de un medidor de glucosa externo cada ocho horas.

Para cumplirlo, necesitamos establecer los siguientes objetivos intermedios:

- Estudiar los Filtros Kalman, muy populares y usados actualmente. En concreto estudiaremos el Filtro Kalman Unscented capaz de estimar los valores de un sistema fuertemente no lineal afectado por ruidos no necesariamente gaussianos ni blancos.

- Modelar del Filtro Kalman Unscented en Matlab y comprobar la viabilidad del filtro en nuestro caso concreto en punto flotante y en punto fijo, calculando el ancho de palabra y de fracción mínimos que luego usaremos en la implementación hardware.

- Describir el hardware en VHDL para luego implementar el diseño sobre una FPGA. La síntesis del diseño mostrará los recursos usados por el filtro y la lógica necesaria para hacerlo funcionar.

- Comprobar el correcto funcionamiento del hardware con simulaciones funcionales de cada módulo individualmente y en conjunto de manera que se asegure la sincronización y cooperación del sistema.

- Verificar con los datos de las simulaciones del Matlab y los datos de las simulaciones del hardware el correcto comportamiento del sistema. Medir la velocidad de cómputo y el error.

Para conseguir cumplir con todos los objetivos hemos usado distintas aplicaciones:

- GitHub: Es uno de los repositorios más grandes del mundo con más de 52 millones de proyectos privados, públicos y de código abierto, todos ellos equipados con herramientas para ayudarte a almacenar, controlar, versionar y proporcionar código. Lo hemos organizado y preparado para poder mantener la sincronía con todos los equipos. Página principal de GitHub (2017).

- Podio: fomenta la concentración y la claridad que los trabajadores necesitan para volcarse por completo en su trabajo reuniendo las conversaciones y los procesos estructurados en una herramienta. La estabilidad de Podio (con un 99,99 % de tiempo activo el año pasado) y su interfaz intuitiva hacen que su despliegue resulte sencillo. Página principal de Podio (2015 – 2017).

- Matlab: La plataforma de MATLAB está optimizada para resolver problemas de ingeniería y científicos. El lenguaje de MATLAB, basado en matrices, es la forma más natural del mundo para expresar las matemáticas computacionales. Los gráficos integrados facilitan la visualización de los datos y la obtención de información a partir de ellos. Página principal de Matlab (1994 – 2017).

- HDL Designer: Combina capacidades de análisis profundo de diseños hardware, editores de creación avanzados y un control completo del proyecto para formar un entorno de diseño HDL que aumenta la productividad de los ingenieros que trabajan de forma individual tanto como en grupo, en local o en remoto. Consigue comunicar el diseño generado con las herramientas de síntesis, que en nuestro caso es el ISE de Xilinx, y de simulación, que en nuestro caso es Questa Sim. Página principal de HDL Desinger (2014 – 2017).

- Questa Sim: El simulador avanzado de Questa combina alto rendimiento y capacidades de simulación con debug avanzado unificado con el HDL Designer, además del más completo soporte nativo de lo lenguajes Verliog, System Verilog, VHDL, SystemC, SVA, UPF y UVM. Página principal de Questa (2014 – 2017).

- LaTeX: Es un sistema de escritura de alta calidad que incluye características para la producción de documentación científica y técnica. Es el estándar para la comunicación y publicación científica. También TeXiS, la aplicación de la UCM para facilitar el proceso de maquetado del trabajo y el uso del lenguaje. Página principal de LaTeX (2005 – 2017).

Parte II

Estado del arte

Capítulo 3

Teoría del Filtro Kalman Unscented

3.1. Origen del filtro Kalman Unscented

Los Filtros Kalman son muy populares y están dando muy buenos resultados en muchos ámbitos: filtros de audio, vídeo, sensores, etc.. Es un filtro discreto y su funcionamiento es relativamente sencillo, ya que una de sus grandes cualidades es que es posible implementarlo en cualquier sistema gracias a sus operaciones sencillas.

Rudolf Emil Kalman, Doctor en Ingeniería Eléctrica desde 1957, nació en Budapest, Hungría, en 1930, como podemos ver en la Biografía de R. E. Kalman (2014 – 2016). En 1960 desarrolló un filtro iterativo heurístico capaz de solucionar tres problemas fundamentales de los sistemas de comunicación y de control, tal y como aparece en Kalman (1960):

- La predicción de señales aleatorias (regresión de funciones no lineales).
- Separación de señales aleatorias de un ruido aleatorio.
- Detección de señales conocidas en presencia de ruido.

Aun así este filtro tiene dos inconvenientes: sólo funciona óptimamente cuando la señal que queremos reconstruir contiene Ruido Blanco Gaussiano y cuando el sistema es lineal. Por eso, Kalman, después de este trabajo siguió adelante desarrollando la extensión del filtro.

Esta segunda versión mejorada es capaz de trabajar con Ruido Blanco Gaussiano y con sistemas débilmente no lineales. Fue llamado Filtro Kalman Extendido (EFK, por sus siglas en Inglés) aunque también se conoce por Filtro Kalman-Schmidt, debido a su desarrollo conjunto para un proyecto de la NASA en el ámbito del Apollo (Enlace a Apollo Moon Program and Lunar Prospector Mission (March 29, 2008)). Su funcionamiento se divide en

una aproximación de Taylor y Jacobianos para la linealización de un sistema no lineal y además consigue adaptarse a la señal de manera óptima en cada iteración mediante la continua afinación de unas covarianzas ligadas a la variable.

El Filtro Kalman Unscented (UKF, por sus siglas en Inglés), basado en el EKF, fue visto por primera vez en las conferencias de S.J. Julier, J.K. Uhlmann, and H. Durrant-Whyt (1995), S.J. Julier and J.K. Uhlmann (November 1996) y S.J. Julier and J.K. Uhlmann (1997) quienes lo propusieron. Eric A. Wan y Rudolph van der Merwe (Artículo editado por Haykin (2001)) lo implementaron. Es una alternativa al EKF. La diferencia entre estos dos filtros es que el EKF usa una única variable aleatoria gaussiana que modela la señal no lineal aleatoria linealizándola. Esto hace que cometa errores en la aproximación de las covarianzas de la variable que genera un mal comportamiento e incluso divergencia. El UKF desarrolla un set de puntos que caracterizan la media y la covarianza de la señal aleatoria y que son usados para estimar una nueva media y covarianza después de ser propagados a través de unas ecuaciones de transición. A esto se le conoce como Transformación Unscented, lo que da la mitad del nombre a este filtro.

El UKF es una extensión directa recursiva de la Transformación Unscented que entra dentro del grupo de los Filtros Bayesianos basados en modelos, lo que significa que usa un modelo del sistema para realizar estimaciones y después las corrige usando una medida. El modelo del sistema se divide en dos modelos:

- El modelo de proceso: su acción se centra en la media del estado oculto, que es una variable que se quiere conocer pero que no se puede medir. Se encargará de realizar la transición del estado que se representa con una función $F(x_k)$ en la iteración k donde $w_{v,k}$ es el ruido del proceso con covarianza R^v. Este modelo se usa en la fase de predicción y no hace falta que sea muy exacto ya que este filtro usa la covarianza del ruido del proceso R^v para determinar cómo de fiable es el modelo. Cuanto menor es, mayor confianza tenemos de nuestro modelo. Las ecuaciones del modelo de proceso se resumen en la siguiente ecuación:

$$x_{k+1} = F(x_k) + w_k \tag{3.1}$$

- El modelo de medida: está centrado en la medida indirecta de la variable que conforma el estado oculto. Es indirecta porque el valor que medimos tiene relación con el estado oculto pero no tiene porqué tener el mismo rango de valores o la misma dinámica. Realiza la transición de la medida mediante una función $H(x_k)$ en la iteración k usando la media del estado oculto actual y donde $w_{n,k}$ es el ruido de la medida con covarianza R^n. Este modelo se usa en la fase de corrección y, al igual que el modelo de proceso, tiene asociado una covarianza de

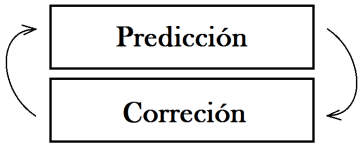

Figura 3.1: Fases del Filtro Kalman

ruido de medida R^n. El modelo de medida se muestra en la siguiente ecuación

$$y_k = H(x_k) + v_k \tag{3.2}$$

Con estos dos modelos podemos ver claramente que el filtro cumple con las etapas generales de los Filtros Kalman (Figura 3.1): predicción y corrección. Estudiamos modelo del sistema en el capítulo 4.

3.2. Transformación Unscented

Aplicamos la Transformación Unscented a la media del estado oculto y a la media de la medida por separado usando las ecuaciones pertinentes de cada modelo asociado. En esta sección solo explicaremos de forma detallada la transformación de la media del estado oculto ya que en las dos aproximaciones usamos las mismas ecuaciones excepto las covarianzas, medias y funciones de transición. El objetivo de esta transformación es estimar la media del estado oculto siguiente usando el modelo de proceso del filtro y la media del estado oculto anterior.

La base de esta transformación es generar un vector de estados ocultos dispersados a una cierta distancia de la media del estado oculto. Cada uno de estos puntos mantienen la misma media y covarianza que el estado oculto, pero su distancia con respecto a la media forma una posible variación del estado. Evaluamos estas variaciones en una media y una covarianza pesada de tal forma que generan una nueva estimación del estado. La explicación detallada muestra como generamos los puntos dispersados que conoceremos como puntos sigma χ_k en la iteración k.

Consideramos propagar una variable aleatoria x, a la que llamaremos estado oculto, también conocida como vector de estado, a través de una función no lineal $F(x)$, asumiendo que x_k tiene como media $\overline{x}_k$ y covarianza P_k, en la iteración k. $F(x)$ es la función que determina la transición de estado según el modelo de proceso del filtro. Formamos una matriz χ_k de $2 \cdot L + 1$ vectores columna $\chi_{i,k}$, conocidos también como vectores sigma o

puntos sigma, según las siguientes ecuaciones:

$$\chi_{0,k} = \overline{x}_k \tag{3.3}$$

$$\chi_{i,k} = \overline{x}_k + (\sqrt{(L+\lambda)\cdot P_k})_i \qquad cuando \quad i = 1, ..., L \tag{3.4}$$

$$\chi_{i,k} = \overline{x}_k - (\sqrt{(L+\lambda)\cdot P_k})_{i-L} \quad cuando \quad i = L+1, ..., 2\cdot L+1 \tag{3.5}$$

donde k es la iteración actual del filtro, L es la dimensión del vector estado x, ya que puede estar compuesto de varios valores, y $\lambda = \alpha^2(L+\kappa) - L$, que es un parámetro de escalado. La constante α determina la dispersión de los puntos sigma alrededor de $\overline{x}_k$, y es un valor de entre $10^{-4} \leq \alpha \leq 1$. La constante κ es un parámetro secundario de escalado que normalmente vale $3 - L$, y la constante β sirve para incorporar información adicional sobre la distribución de x. Para distribuciones gaussianas es óptimo $\beta = 2$.

Es importante destacar que la matriz de puntos sigma cuando $1 \leq i \leq 2 \cdot L + 1$ la calculamos como la suma o resta de $\overline{x}_k$ con respecto a cada una de las columnas de la matriz raíz cuadrada de P_k multiplicada por el desplazamiento característico de la transformación Unscented $\gamma = \sqrt{(L+\lambda)}$. Para implementar la matriz raíz cuadrada se aconseja en el artículo de Haykin (2001) a usar la factorización triangular de Cholesky (descrita en el artículo de Nicholas J. Higham (September / October 2009)).

El siguiente paso es propagar los puntos sigma a través de la función de transición de estado $F(x)$, como pueden ver en la ecuación (3.6) y estimar $\overline{x}_k^-$ y P_k^- usando una media y una covarianza pesada de los puntos sigma propagados, cuyas ecuaciones son (3.7) y (3.8). Para ello, usamos los pesos dados por las ecuaciones (3.9) a (3.11).

$$\chi^*_{k|k-1} = F(\chi_{k-1}) \qquad cuando \quad i = 0, ..., 2\cdot L \tag{3.6}$$

$$\overline{x}_k^- = \sum_{i=0}^{2L} W_i^{(m)} \chi^*_{i,k|k-1} \tag{3.7}$$

$$P_k^- = \sum_{i=0}^{2L} W_i^{(c)} \left(\chi^*_{i,k|k-1} - \overline{x}_k^-\right)\left(\chi^*_{i,k|k-1} - \overline{x}_k^-\right)^T + R^v \tag{3.8}$$

$$W_0^{(m)} = \frac{\lambda}{L+\lambda} \tag{3.9}$$

$$W_0^{(c)} = \frac{\lambda}{L+\lambda} + 1 - \alpha^2 + \beta \tag{3.10}$$

$$W_i^{(m)} = W_i^{(c)} = \frac{1}{2\cdot(L+\lambda)} \quad donde \quad i = 1, ..., 2\cdot L \tag{3.11}$$

Si afinamos bien los valores de α y β esta transformación es comparable a la aproximación polinómica de tercer orden para variables aleatorias gaussianas o de segundo orden o muy parecido al tercero en variables aleatorias no gaussianas.

3.3. Etapas del filtro

Terminado el concepto de transformación Unscented queda por explicar el funcionamiento del filtro y el uso del modelo del sistema. Expandiendo las fases del filtro y entrando en los detalles matemáticos y estadísticos estos son los pasos a dar:

- Inicialización (Ecuaciones (3.29) a (3.33)): nos encargamos de asignar valores iniciales a α, β y κ con los que calcularemos λ y γ. También daremos valores iniciales a la media y a la covarianza del estado. Además calcularemos los pesos para la media y la covarianza, todo esto conforme a lo dicho en la sección 3.2.

$$\overline{x}_0 = E[x_0] \tag{3.12}$$

$$P_0 = E\left[(x_0 - \overline{x}_0)(x_0 - \overline{x}_0)^T\right] \tag{3.13}$$

$$W_0^{(m)} = \frac{\lambda}{L+\lambda} \tag{3.14}$$

$$W_0^{(c)} = \frac{\lambda}{L+\lambda} + 1 - \alpha^2 + \beta \tag{3.15}$$

$$W_i^{(m)} = W_i^{(c)} = \frac{1}{2\cdot(L+\lambda)} \quad donde \quad i = 1, ..., 2\cdot L \tag{3.16}$$

- Predicción:

  - Cálculo de puntos sigma (Ecuaciones (3.34)): formamos la matriz de puntos sigma (χ_{k-1}) con el estado de la iteración pasada y la dispersión propia de la Transformación Unscented. Está explicado con detalle en la sección 3.2. Si es la primera iteración no usamos el estado de la anterior iteración sino el estado inicial.

$$\chi_{k-1} = \left[\overline{x}_{k-1} \quad \overline{x}_{k-1} + \gamma\cdot\sqrt{P_{k-1}} \quad \overline{x}_{k-1} - \gamma\cdot\sqrt{P_{k-1}}\right] \tag{3.17}$$

  - Propagación de puntos sigma (Ecuaciones (3.35)): propagamos uno por uno los puntos sigma a través de las ecuaciones de transición del modelo de proceso (función no lineal $F(x)$) para generar los puntos sigma transformados por las ecuaciones del modelo de proceso ($\chi^*_{k|k-1}$).

$$\chi^*_{k|k-1} = F(\chi_{k-1}) \tag{3.18}$$

- Cálculo de la media a priori del estado (Ecuaciones (3.36)): calculamos la media pesada de los puntos sigma propagados para hallar la media a priori del estado $\overline{x}_k^-$ (es una estimación del estado que el filtro usará para conocer la aproximación de la media del estado a la señal). Indicamos que en la literatura se encuentra en esta ecuación una entrada de control u_{k-1} que nostros no usaremos, por tanto no aparecerá en este trabajo.

$$\overline{x}_k^- = \sum_{i=0}^{2L} W_i^{(m)} \chi_{i,k|k-1}^* \tag{3.19}$$

- Cálculo de la covarianza a priori del estado (Ecuaciones (3.37)): calculamos la covarianza de los puntos sigma propagados añadiéndole la covarianza del ruido del proceso R^v para hallar la covarianza a priori del estado P_k^- (con el mismo fin que la media del estado a priori).

$$P_k^- = \sum_{i=0}^{2L} W_i^{(c)} \left(\chi_{i,k|k-1}^* - \overline{x}_k^-\right) \left(\chi_{i,k|k-1}^* - \overline{x}_k^-\right)^T + R^v \tag{3.20}$$

- Correción:

  - Cálculo de puntos sigma extendidos (Ecuaciones (3.38)): formamos la nueva matriz de puntos sigma $\chi_{k|k-1}$ con el estado a priori del sistema y la dispersión propia de la Transformación Unscented.

$$\chi_{k|k-1} = \left[\overline{x}_k^- \quad \overline{x}_k^- + \gamma \cdot \sqrt{P_k^-} \quad \overline{x}_k^- - \gamma \cdot \sqrt{P_k^-}\right] \tag{3.21}$$

  - Propagación de puntos sigma expandidos (Ecuaciones (3.39)): propagamos los puntos sigma expandidos a través de las ecuaciones de transición de la medida (función no lineal $H(x)$) para generar los puntos sigma transformados por las ecuaciones del modelo de la medida $\Upsilon_{k|k-1}$.

$$\Upsilon_{k|k-1} = H(\chi_{k|k-1}) \tag{3.22}$$

  - Cálculo de la media a priori de la medida (Ecuaciones (3.40)): calculamos la media pesada de los puntos sigma expandidos propagados para hallar la media a priori de la medida $\overline{y_k}^-$ (es una estimación que el filtro usará para conocer la aproximación de la media de la medida a la medida real).

$$\overline{y}_k^- = \sum_{i=0}^{2L} W_i^{(m)} \Upsilon_{i,k|k-1} \tag{3.23}$$

- Cálculo de la covarianza de la medida (Ecuaciones (3.41)): calculamos la covarianza de los puntos sigma expandidos propagados añadiéndole la covarianza del ruido de la medida R^n para hallar la covarianza de la medida $P_{\tilde{y_k}\tilde{y_k}}$.

$$P_{\tilde{y_k}\tilde{y_k}} = \sum_{i=0}^{2L} W_i^{(c)} \left(\Upsilon_{i,k|k-1} - \overline{y}_k^-\right)\left(\Upsilon_{i,k|k-1} - \overline{y}_k^-\right)^T + R^n \quad (3.24)$$

- Cálculo de la covarianza cruzada (Ecuaciones (3.42)): usamos los puntos sigma propagados y los puntos sigma expandidos propagados para hallar su correlación en la covarianza cruzada $P_{x_k y_k}$.

$$P_{x_k y_k} = \sum_{i=0}^{2L} W_i^{(c)} \left(\chi_{i,k|k-1} - \overline{x}_k^-\right)\left(\Upsilon_{i,k|k-1} - \overline{y}_k^-\right)^T \quad (3.25)$$

- Cálculo de la Ganancia de Kalman (Ecuaciones (3.43)): usamos la división de la covarianza cruzada entre la covarianza de la medida para hallar la Ganancia de Kalman K_k que nos servirá para rectificar las estimaciones del filtro.

$$K_k = P_{x_k y_k} \cdot P_{\tilde{y_k}\tilde{y_k}}^{-1} \quad (3.26)$$

- Correción de la media del estado (Ecuaciones (3.44)): usamos la Ganancia de Kalman, junto con el residual de la medida $y_k - \overline{y}_k^-$, para corregir la media del estado a priori y hallar la media del estado corregida o a posteriori $\overline{x}_k$.

$$\overline{x}_k = \overline{x}_k^- + K_k \cdot (y_k - \overline{y}_k^-) \quad (3.27)$$

- Correción de la covarianza del estado (Ecuaciones (3.45)): usamos la Ganancia de Kalman junto con la covarianza de la medida para decrementar la covarianza a priori del estado y hallar la covarianza del estado corregida o a posteriori P_k.

$$P_k = P_k^- - K_k \cdot P_{\tilde{y_k}\tilde{y_k}} \cdot K_k^T \quad (3.28)$$

Finalmente queda añadir que después de la etapa de Corrección, el filtro itera, volviendo a empezar desde la etapa de Predicción. No necesitamos realizar ningún paso de la inicialización del filtro en ese momento.

3.4. Ecuaciones del filtro

A continuación mostramos las ecuaciones del filtro que hemos estado estudiando en una única página. En nuestro caso hemos usado las ecuaciones

del Filto Kalman Unscented para el caso particular de ruido aditivo de media cero.

Inicio:

$$\overline{x}_0 = E[x_0] \tag{3.29}$$

$$P_0 = E\left[(x_0 - \overline{x}_0)(x_0 - \overline{x}_0)^T\right] \tag{3.30}$$

$$W_0^{(m)} = \frac{\lambda}{L+\lambda} \tag{3.31}$$

$$W_0^{(c)} = \frac{\lambda}{L+\lambda} + 1 - \alpha^2 + \beta \tag{3.32}$$

$$W_i^{(m)} = W_i^{(c)} = \frac{1}{2 \cdot (L+\lambda)} \quad donde \quad i = 1, ..., 2 \cdot L \tag{3.33}$$

Predicción:

$$\chi_{k-1} = \left[\overline{x}_{k-1} \quad \overline{x}_{k-1} + \gamma \cdot \sqrt{P_{k-1}} \quad \overline{x}_{k-1} - \gamma \cdot \sqrt{P_{k-1}}\right] \tag{3.34}$$

$$\chi^*_{k|k-1} = F(\chi_{k-1}) \tag{3.35}$$

$$\overline{x}_k^- = \sum_{i=0}^{2L} W_i^{(m)} \chi^*_{i,k|k-1} \tag{3.36}$$

$$P_k^- = \sum_{i=0}^{2L} W_i^{(c)} \left(\chi^*_{i,k|k-1} - \overline{x}_k^-\right)\left(\chi^*_{i,k|k-1} - \overline{x}_k^-\right)^T + R^v \tag{3.37}$$

Corrección:

$$\chi_{k|k-1} = \left[\overline{x}_k^- \quad \overline{x}_k^- + \gamma \cdot \sqrt{P_k^-} \quad \overline{x}_k^- - \gamma \cdot \sqrt{P_k^-}\right] \tag{3.38}$$

$$\Upsilon_{k|k-1} = H(\chi_{k|k-1}) \tag{3.39}$$

$$\overline{y}_k^- = \sum_{i=0}^{2L} W_i^{(m)} \Upsilon_{i,k|k-1} \tag{3.40}$$

$$P_{\tilde{y}_k \tilde{y}_k} = \sum_{i=0}^{2L} W_i^{(c)} \left(\Upsilon_{i,k|k-1} - \overline{y}_k^-\right)\left(\Upsilon_{i,k|k-1} - \overline{y}_k^-\right)^T + R^n \tag{3.41}$$

$$P_{x_k y_k} = \sum_{i=0}^{2L} W_i^{(c)} \left(\chi_{i,k|k-1} - \overline{x}_k^-\right)\left(\Upsilon_{i,k|k-1} - \overline{y}_k^-\right)^T \tag{3.42}$$

$$K_k = P_{x_k y_k} \cdot P^{-1}_{\tilde{y}_k \tilde{y}_k} \tag{3.43}$$

$$\overline{x}_k = \overline{x}_k^- + K_k \cdot (y_k - \overline{y}_k^-) \tag{3.44}$$

$$P_k = P_k^- - K_k \cdot P_{\tilde{y}_k \tilde{y}_k} \cdot K_k^T \tag{3.45}$$

Capítulo 4

Modelos de proceso y de medición

4.1. Modelo

Como hemos visto en el capítulo 3 usamos un filtro basado en modelos, por ello, necesitamos un modelo del sistema que describa fielmente la dinámica del mismo. El modelo se divide en dos modelos: el de proceso o estado y el de medida.

Usamos en este trabajo el modelo propuesto por Kuure-Kinsey, M., Palerm, C. C., & Bequette, B. W. (2006) que describe la dinámica de la glucosa en sangre y que utiliza una medida indirecta y otra directa conmucha menor frecuencia para medir el valor de dicha GS.

Este modelo tiene la capacidad de modelar la glucosa en sangre y la degradación del sensor mediante una ganancia, por tanto, nuestro filtro no será inconsciente de dicha degradación. Además nos interesa porque nos permite conocer la glucosa en sangre a partir de la glucosa intersticial, la cual es una medida indirecta de la glucosa en sangre. El modelo divide estas dos medidas en medida rápida y lenta:

- La medida rápida es el valor obtenido mediante las repetitivas medidas del sensor de glucosa en una zona intersticial. Realizamos una medición cada cinco minutos y si el sensor no ha sido inutilizado por parte de los anticuerpos, la oxidación de la glucosa o el tiempo de uso, la lectura que recibe es ruidosa y desfasada con respecto a la glucosa en sangre real.

- La medida lenta es el valor obtenido mediante la punción de un dedo y extracción de una gota de sangre que servirá para alimentar un medidor de glucosa externo. Esta medida es fiable y sirve para calibrar el filtro pero se da cada ocho horas.

Como el modelo de Kuure-Kinsey, M., Palerm, C. C., & Bequette, B. W. (2006) está ya preparado para su uso en Filtros Kalman, es muy sencilla su inclusión. Simplemente tenemos que sustituir en el filtro las ecuaciones de transición del modelo de proceso y de transición del modelo de medida por las respectivas de cada modelo.

4.2. Modelo de proceso

Usamos el modelo de proceso para evolucionar el estado oculto. En el capítulo 3 estaba denotado por la ecuación (3.35). Su representación es esta:

$$x_{k+1} = F(x_k) + w_k \tag{4.1}$$

La función de transición de estado $F(x)$ en el modelo de Kinsey es una transformación dada por cuatro ecuaciones que podemos ver desde (4.2) hasta (4.5). Son cuatro ecuaciones porque el vector de estado se compone de cuatro valores: la glucosa en sangre g_k, el incremento de glucosa en sangre Δg_k, la ganancia del sensor a_k y el incremento de la ganancia del sensor Δa_k. Incluimos en el modelo de proceso los ruidos que introducen un posible error (no hay modelos exactos). Además ayudarán al filtro a fiarse más del modelo o de la medida dependiendo de los valores de la covarianza R^v. Como nuestro modelo de proceso tiene cuatro componentes $L = 4$ para las ecuaciones del filtro.

$$\begin{aligned} g_{k+1} &= g_k + \Delta g_k && (4.2)\\ \Delta g_{k+1} &= \Delta g_k + w_{g,k} && (4.3)\\ a_{k+1} &= a_k + \Delta a_k && (4.4)\\ \Delta a_{k+1} &= \Delta a_k + w_{a,k} && (4.5) \end{aligned}$$

La ecuación 4.6 muestra de forma matricial las ecuaciones de transición de estado donde podemos apreciar mejor la relación de los valores:

$$\underbrace{\begin{bmatrix} g_{k+1} \\ \Delta g_{k+1} \\ a_{k+1} \\ \Delta a_{k+1} \end{bmatrix}}_{X_{k+1}} = \underbrace{\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} g_k \\ \Delta g_k \\ a_k \\ \Delta a_k \end{bmatrix}}_{X_k} + \underbrace{\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}}_{\Gamma^w} \underbrace{\begin{bmatrix} w_{g,k} \\ w_{a,k} \end{bmatrix}}_{w_k} \tag{4.6}$$

Podemos retocar la matriz A con ciertos valores para conseguir un modelo más parecido al real, como por ejemplo, la matriz (4.7) modela una glucosa

lineal y una ganancia exponencial de la forma $y = e^{td}$ para todo $d \in \Re$.

$$A_{gan\ exp} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^d & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

Estas correcciones del modelo ayudan al perfecto funcionamiento del filtro y a su estabilidad, así como su convergencia hacia la señal real y facilita el encontrar las covarianzas de los ruidos del proceso y del estado inicial.

4.3. Modelo de medida

Para que el filtro pueda realizar estimaciones también sobre la medida y diagnosticar su desviación con respecto a la medida real usamos un modelo de medida. En el capítulo 3 lo mostrábamos mediante la ecuación 3.35. Su representación es esta:

$$y_k = H(x_k) + v_k \tag{4.8}$$

El modelo de medida de Kinsey formula las ecuaciones de transición de la medida rápida que podemos ver en (4.9). Ahí podemos observar perfectamente cómo gracias a la estimación de la ganancia y de la glucosa del sensor conseguimos estimar la medida rápida $y_{s,k}$. En el caso de la medida rápida $y_{s,k}$ y lenta $y_{f,k}$, las ecuaciones que componen la transición de la medida son desde 4.10 hasta 4.11. En este caso, la única diferencia es que la medida lenta toma directamente el valor de la glucosa en sangre y aquí reside la potencia de este modelo. En este momento el sistema se calibra al conocer de forma fiable su desviación con respecto a la glucosa real y rectifica.

$$y_{s,k} = g_k \cdot a_k + v_{s,k} \tag{4.9}$$
$$y_{s,k} = g_k \cdot a_k + v_{s,k} \tag{4.10}$$
$$y_{f,k} = g_k + v_{f,k} \tag{4.11}$$

La forma compacta de las ecuaciones ayuda a su comprensión. Estas operaciones no las realizamos de forma matricial en la fase de corrección. En este caso, no modificamos el modelo de medida. El filtro se encargará de afinar su comportamiento mediante las covarianzas de la medida $P_{\tilde{y_k}\tilde{y_k}}$ y

cruzada $P_{x_k y_k}$.

$$y_{s,k} = \begin{bmatrix} 0{,}5a_k & 0 & 0{,}5g_k & 0 \end{bmatrix} \begin{bmatrix} g_k \\ \Delta g_k \\ a_k \\ \Delta a_k \end{bmatrix} + v_{s,k} \tag{4.12}$$

$$\begin{bmatrix} y_{s,k} \\ y_{f,k} \end{bmatrix} = \begin{bmatrix} 0{,}5a_k & 0 & 0{,}5g_k & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} g_k \\ \Delta g_k \\ a_k \\ \Delta a_k \end{bmatrix} + \begin{bmatrix} v_{s,k} \\ v_{f,k} \end{bmatrix} \tag{4.13}$$

4.4. Ecuaciones finales del filtro

Finalmente, añadimos el modelo del sistema al filtro en la ecuación (4.20) y desde la ecuación (4.24) a la (4.28). Usamos j para denotar la columna en las ecuaciones (4.24) y (4.26). En la tabla 4.1 podemos ver desglosadas todas las variables y constantes del filtro con sus dimensiones.

Inicio:

$$\overline{x}_0 = E[x_0] \tag{4.14}$$

$$P_0 = E\left[(x_0 - \overline{x}_0)(x_0 - \overline{x}_0)^T\right] \tag{4.15}$$

$$W_0^{(m)} = \frac{\lambda}{L+\lambda} \tag{4.16}$$

$$W_0^{(c)} = \frac{\lambda}{L+\lambda} + 1 - \alpha^2 + \beta \tag{4.17}$$

$$W_i^{(m)} = W_i^{(c)} = \frac{1}{2 \cdot (L+\lambda)} \quad donde \quad i = 1, ..., 2 \cdot L \tag{4.18}$$

Predicción:

$$\chi_{k-1} = \left[\overline{x}_{k-1} \quad \overline{x}_{k-1} + \gamma \cdot \sqrt{P_{k-1}} \quad \overline{x}_{k-1} - \gamma \cdot \sqrt{P_{k-1}}\right] \tag{4.19}$$

$$\chi^*_{k|k-1} = A \cdot \chi_{k-1} \tag{4.20}$$

$$\overline{x}_k^- = \sum_{i=0}^{2L} W_i^{(m)} \chi^*_{i,k|k-1} \tag{4.21}$$

$$P_k^- = \sum_{i=0}^{2L} W_i^{(c)} \left(\chi^*_{i,k|k-1} - \overline{x}_k^-\right)\left(\chi^*_{i,k|k-1} - \overline{x}_k^-\right)^T + R^v \tag{4.22}$$

Corrección:

$$\chi_{k|k-1} = \left[\overline{x}_k^- \quad \overline{x}_k^- + \gamma \cdot \sqrt{P_k^-} \quad \overline{x}_k^- - \gamma \cdot \sqrt{P_k^-}\right] \tag{4.23}$$

Si no hay medida lenta

$$\Upsilon_{s,k|k-1} = \chi_{0,j,k|k-1} \cdot \chi_{2,j,k|k-1} \quad \text{con } 0 \leq j < 2 \cdot L + 1 \tag{4.24}$$

$$\Upsilon_{k|k-1} = \Upsilon_{s,k|k-1} \tag{4.25}$$

Si hay medida lenta

$$\Upsilon_{j,s,k|k-1} = \chi_{0,j,k|k-1} \cdot \chi_{2,j,k|k-1} \quad \text{con } 0 \leq j < 2 \cdot L + 1 \tag{4.26}$$

$$\Upsilon_{j,f,k|k-1} = \chi_{0,j,k|k-1} \tag{4.27}$$

$$\Upsilon_{k|k-1} = \begin{bmatrix} \Upsilon_{s,k|k-1} \\ \Upsilon_{f,k|k-1} \end{bmatrix} \tag{4.28}$$

$$\overline{y}_k^- = \sum_{i=0}^{2L} W_i^{(m)} \Upsilon_{i,k|k-1} \tag{4.29}$$

$$P_{\tilde{y}_k \tilde{y}_k} = \sum_{i=0}^{2L} W_i^{(c)} \left(\Upsilon_{i,k|k-1} - \overline{y}_k^-\right) \left(\Upsilon_{i,k|k-1} - \overline{y}_k^-\right)^T + R^n \tag{4.30}$$

$$P_{x_k y_k} = \sum_{i=0}^{2L} W_i^{(c)} \left(\chi_{i,k|k-1} - \overline{x}_k^-\right) \left(\Upsilon_{i,k|k-1} - \overline{y}_k^-\right)^T \tag{4.31}$$

$$K_k = P_{x_k y_k} \cdot P_{\tilde{y}_k \tilde{y}_k}^{-1} \tag{4.32}$$

$$\overline{x}_k = \overline{x}_k^- + K_k \cdot (y_k - \overline{y}_k^-) \tag{4.33}$$

$$P_k = P_k^- - K_k \cdot P_{\tilde{y}_k \tilde{y}_k} \cdot K_k^T \tag{4.34}$$

Símbolo	Dimensiones	Tipo	Definición
$\overline{x}_0$	4x1	Constante	Media inicial del estado oculto
P_0	4x4	Constante	Covarianza a priori inicial del estado oculto
$W_i^{(m)}$	1x9	Constante	Pesos para la media
$W_i^{(c)}$	1x9	Constante	Pesos para la covarianza
R^v	4x4	Constante	Ruido del proceso
R^n	2x2	Constante	Ruido de la medida
χ_{k-1}	4x9	Variable	Puntos sigma (predicción)
$\chi^*_{k\|k-1}$	4x9	Variable	Puntos sigma trasformados por las ecuaciones de transición del estado
$\overline{x}_k^-$	4x1	Variable	Media a priori del estado
P_k^-	4x4	Variable	Covarianza a priori del estado
$\chi_{k\|k-1}$	4x9	Variable	Puntos sigma expandidos
$\Upsilon_{s,k\|k-1}$	1x9	Variable	Puntos sigma rápidos expandidos transformados por las ecuaciones de transición de la medida
$\Upsilon_{f,k\|k-1}$	2x9	Variable	Puntos sigma lentos expandidos transformados por las ecuaciones de transición de la medida
$\Upsilon_{k\|k-1}$	1x9 o 2x9	Variable	Puntos sigma expandidos transformados (1x9 si no hay medida lenta, 2x9 si la hay)
$\overline{y}_k^-$	2x1	Variable	Media a priori de la medida
$P_{\tilde{y}_k\tilde{y}_k}$	1x1 o 2x2	Variable	Covarianza de la medida (1x1 si no hay medida lenta, 2x2 si la hay)
$P_{x_k y_k}$	4x1 o 4x2	Variable	Covarianza cruzada (4x1 si no hay medida lenta, 4x2 si la hay)
K_k	4x1 o 4x2	Variable	Ganancia de Kalman (4x1 si no hay medida lenta, 4x2 si la hay)
$\overline{x}_k$	4x1	Variable	Media del estado corregido (o media del estado a posteriori)
P_k	4x4	Variable	Covarianza del estado corregido (o covarianza del estado a posteriori)

Tabla 4.1: Variables y constantes de las ecuaciones del filtro

Capítulo 5

Teoría del Punto fijo

En este proyecto, es necesario operar con números fraccionarios. Una representación binaria simple no nos ofrece la posibilidad de operar con valores decimales y una representación en coma flotante complica demasiado los cálculos realizados, por ello trabajamos con números binarios representados en punto fijo.

5.1. Que es el punto fijo

El punto fijo es una notación científica para representar números decimales que consiste en destinar una cantidad fija de dígitos a la parte entera del número y otra a la parte fraccionaria.

La conversión de un número decimal a su representación en punto fijo es sencilla, sólo es necesario multiplicar el número por un factor de escala con el objetivo de hacer desaparecer la parte decimal y convertir, por tanto, el número decimal en un valor natural con el que es sencillo operar.
Para recuperar el valor decimal se debe realizar el proceso contrario, dividir el número natural por el mismo factor de escala, de forma que fijamos la coma en la posición indicada por la escala.

Supongamos un factor de 1 000, la representación en punto fijo del valor 0,0034 será 0,0034*1000 = 3. Si el factor fuese de 10 000 la representación del número anterior sería 0,0034*10000 = 34. Como se puede observar, la precisión del número natural que representa al valor decimal viene determinada directamente por el factor de escala utilizado.

La ventaja de utilizar esta representación es que las conversiones y las operaciones con estos valores son casi tan rápidas como trabajar directamente con aritmética entera y pudiendo representar valores inferiores a 1.0, siendo necesario para esto disponer de una cantidad mímina de dígitos que proporcionen tal precisión. Puesto que se basa en aritmética entera, es muy

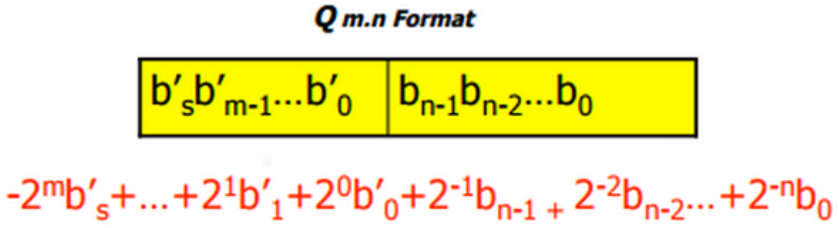

Figura 5.1: Formato $Q_{m,n}$

eficiente, siempre que los datos no varíen mucho en magnitud.

5.2. Operaciones en punto fijo

Dado que el valor numérico lo hemos multiplicado por una constante, debemos tenerlo en cuenta a la hora de realizar las operaciones. Supongamos a y b valores reales y f un factor de precisión, obtenemos $A = a \cdot f$ y $B = b \cdot f$. Podemos describir las operaciones aritméticas básicas de la siguiente forma:

- Suma: $A + B = a \cdot f + b \cdot f = (a + b) \cdot f \rightarrow a + b \sim A + B$
- Multiplicación: $A \cdot B = a \cdot f \cdot b \cdot f = (a \cdot b) \cdot f^2 \rightarrow a \cdot b \sim (A \cdot B)/f$
- División: $A/B = a \cdot f/b \cdot f = a/b \rightarrow a/b \sim (A \cdot f)/B$

Si queremos obtener el valor en punto fijo A a partir de la variable real a debemos hacer $A = a \cdot f$. Sin embargo, si queremos obtener el valor real a' a partir de la variable en punto fijo A debemos hacer $a' = A/f$. Finalmente, el error existente entre las variables a y a' dependerá del redondeo cometidos.

5.3. Notación Q

La *notación Q* se utiliza para especificar el número de dígitos destinados a representar la parte entera y la parte fraccionaria del número real. Una notación $Q_{m,n}$ indica que m bits representan la parte entera y n bits representan la fracción. Es necesario un bit extra que represente el signo, luego, se utilizan m + n + 1 bits.

Para calcular el valor decimal de un número binario de n bits con notación $Q_{m,n}$ se realiza la operación descrita en la figura 5.1:

h

$$\frac{N_1}{d} + \frac{N_2}{d} = \frac{N_1 + N_2}{d}$$
$$\frac{N_1}{d} - \frac{N_2}{d} = \frac{N_1 - N_2}{d}$$
$$\left(\frac{N_1}{d} \times \frac{N_2}{d}\right) \times d = \frac{N_1 \times N_2}{d}$$
$$\left(\frac{N_1}{d} / \frac{N_2}{d}\right) / d = \frac{N_1 / N_2}{d}$$

Figura 5.2: Fórmulas para operar con notación Q manteniendo el denominador constante

Dado un número en notación $Q_{m,n}$, su rango representable es $[-2^m, 2^m - 2^{-n}]$, siendo 2^{-n} la precisión. Como ejemplo, supongamos número representado en complemento a dos cuya notación es $Q_{3,8}$, su rango representable es $[-(2^3), 2^3 - 2^{-8}]$ y su precisión $2^{-8} = 0{,}00391$

5.4. Aritmética en notación Q

Supongamos un número en punto fijo con notación $Q_{m,n}$, podemos interpretarlo como la división entre dos enteros donde el numerador es la representación entera del número y el denominador es el factor de escala utilizado, que en nuestro caso es 2^n por ser n los bits dedicados a representar la fracción del número. Al operar con tal número es necesario mantener la posición original del punto y para ello hay que mantener el denominador constante. La figura **??** muestra las fórmulas de cómo se deben ejecutar las operaciones matemáticas para mantener el denominador constante.

h

Dado que el denominador es una potencia de dos, la multiplicación se implementa mediante un desplazamiento a la izquierda y la división con un desplazamiento a la derecha. Para mantener la precisión, los resultados intermedios de multiplicaciones y divisiones deben ser redondeados antes de convertirlos al número Q deseado.

5.5. Paquete de punto fijo

En el proyecto se ha usado el paquete fixed_pkg.vhdl que define los tipos de datos necesarios para trabajar con punto fijo. En concreto hemos usado el tipo sfixed definido: type sfixed is array (INTEGER range <>) of

STD_LOGIC.

Este tipo indica el valor en punto fijo con signo y utiliza el complemento a dos para representar el número negativo. En este tipo de datos se muestra la localización del punto decimal usando un índice negativo. Así por ejemplo la siguiente señal: signal y : sfixed (5 downto −5) tiene una anchura total de 11 bits de los cuales 5 son la parte decimal, 5 la fracción y el bit más significativo indica el signo.

Tenemos el número 6.5 = 000110.10000. Debido a que tiene 5 bits para representar la parte decimal, el factor de escala es $2^5 = 32$ La representación entera del número binario 00011010000 es 208 y para saber el número al que está representando basta con hacer $\frac{208}{32} = 6{,}5$.

La tabla 5.1 muestra como se numeran los índices en el ejemplo anterior. El primer bit después del punto es el índice 0 y el bit más significativo es el 5, el primer bit después del punto es -1 y el menos significativo es -5.

índice	5	4	3	2	1	0	-1	-2	-3	-4	-5
dígito	0	0	0	1	1	0.	1	0	0	0	0

Tabla 5.1: Numeración de índices en una variable sfixed

Parte III

Implementación

Capítulo 6

Modelado en Matlab

Para diseñar y comprobar que el filtro y el modelo actúan como queremos hemos hecho varios script en Matlab. Unos modelan el filtro en punto flotante y otros en punto fijo. Además, hemos modelado los algoritmos externos al filtro como la división en punto fijo, la matriz raíz cuadrada y Cholesky.

Esta es una parte muy importante del diseño y comprende dos pasos de diseño cruciales que determinarán el diseño final en VHDL:

- Primer diseño en Matlab: Lo usamos para controlar cómo de bueno es el filtro usado, el modelo, las covarianzas, los datos, etc. El resultado de hacer este paso es uno o varios script en Matlab que indican cómo sería la salida ideal del sistema y el error ideal que sería interesante no sobrepasar. Esta es la fase de diseño más rápida y de más alto nivel.
- Segundo diseño en Matlab: Lo usamos para estrechar la barrera entre el alto nivel del software y el bajo nivel del hardware. El resultado de hacer este paso es uno o varios script en Matlab capaces de modelar el filtro en bajo nivel, facilitar el diseño en VHDL y que permiten estimar el error del hardware. Esta implementación también lo usamos para generar datos para las pruebas del hardware y comparar las respuestas que obtenemos con el modelo del filtro y el VHDL.

6.1. Distintas implementaciones

Al ver por encima el filtro observamos una característica especial y es que dependiendo de la medida que tenemos, el filtro realiza unas operaciones u otras. Esto tiene que solucionarse en esta fase ya que, si no lo hacemos, el problema será demasiado grande para solucionarlo a bajo nivel. En este caso se nos ocurrieron dos soluciones:

- Una primera solución es usar dos caminos de datos: uno para la fase rápida del filtro y otro para la fase lenta del filtro. Esta implementación

la denominamos como de dos vías, por la doble ruta de datos.

- La segunda solución es usar un único camino de datos que los llevase todos juntos pero sin mezclarlos. Si nos fijamos en las ecuaciones y escogemos sólo las dimensiones de las matrices cuando hay medida lenta podremos comprobar que las dimensiones de las matrices por la vía rápida son un subconjunto. En efecto, esto debemos esto a que la medida lenta no es exclusiva a la medida rápida, sino que las calculamos conjuntamente. El quid de la cuestión en es encontrar y demostrar la forma de evitar juntar los datos y que puedan coexistir en una única ruta de datos.

Debemos fijamos en la ecuación (4.24), que es la primera en cambiar dependiendo de si hay medida lenta o no. En esa ecuación empiezan a cambiar las dimensiones de las matrices si hay medida lenta o no. La idea que se nos ocurrió fue la de, en vez de multiplexar al final de la iteración, multiplexar en ese momento.

Usando la multiplexación en la etapa de propagación de puntos sigma expandidos nos evitamos replicar hardware debido a que no tenemos que cambiar las dimensiones de las matrices (y por tanto el hardware que habrá después) sino que en su lugar introduciremos ceros donde deberían de ir los valores extraídos de la medida lenta. Este cambio genera un error del orden de 10^{-6} que no apreciamos en el punto fijo.

Para cuantificar el coste hardware del diseño antes de codificarlo y para tener una referencia de qué operaciones necesitamos implementar para la ruta de datos hicimos un análisis de las ecuaciones infiriendo el hardware necesario. Después de una simplificación las operaciones básicas necesarias en la implementación conseguimos la cuantía de 20 unidades funcionales complejas y 8 operaciones sencillas para la implementación de una vía. En la implementación de dos vías las unidades funcionales totales llegaban a ser 35. Podemos ver esto más en detalle en el apéndice A.

6.2. Dimensiones del punto fijo

En el diseño coexisten variables con valores que se diferencian en varios ordenes de magnitud. Esto se lo debemos a los pequeños valores de las covarianzas de los ruidos, que llegan a ser de 0,001 sobre todo y las multiplicaciones por los pesos de la media y la covarianza, que alcanzan resultados del millón de unidades al encadenar hasta tres multiplicaciones seguidas.

Además, para aumentar la efectividad de la división hemos tenido que aumentar la parte fraccional debido al algoritmo escogido. Cuanto mayor sea, menor error da, aunque con un ancho de fracción de 24 bits el error está entre la tercera y la cuarta cifra decimal.

Con todo esto, después de estudiar los datos del filtro usando el Matlab y el VHDL hemos llegado a la conclusión de que la mejor representación en punto fijo del sistema es Q 24, 24.

6.3. Cambio en las fases del filtro

Para organizar mejor el trabajo y la comprensión del filtro incluimos una nueva etapa: la etapa de Ganancia de Kalman. El filtro queda por tanto dividido en tres fases:

- La fase de Predicción comprendida entre las ecuaciones (4.19) y (4.22).
- Delimitamos la fase de Corrección entre las ecuaciones (4.23) y (4.29).
- La fase de Ganancia de Kalman está contenida ente las ecuaciones (4.30) y (10.4).

Esto ayuda ya que ahora la etapa de Corrección comprende puntos parecidos a la etapa de Predicción y todas las fases tienen cuatro ecuaciones, lo que nos ayuda a distribuir el trabajo uniformemente.

6.4. Terminado el diseño en Matlab

Finalmente seleccionamos como válido el diseño de una vía porque, de lo contrario habría que replicar a más o menos el 50 % del hardware debido a que tendríamos que distinguir entre los dos caminos a partir de la ecuación de expansión de los puntos sigma (4.23) para conseguir una mejora ínfima.

Además hemos comprobado que el filtro vacilaba bastante al inicio hasta que llegaba la primera medida lenta. Hemos arreglado esto mediante un llamado pinchazo de calibración inicial que debemos dar una hora después de instalar el sistema en el paciente. Este pinchazo evita excesivas sobreelongaciones de la señal que hacían que el error del filtro se disparase.

Simplemente con estas dos mejoras el hardware del filtro será el menor posible y los resultados lo mejor posible.

Capítulo 7

Implementación Hardware

7.1. Aspectos Generales de la Implementación

Acorde a la sección 3.3, se ha implementado un sistema recursivo que, a partir del estado anterior, consigue estimar el nuevo estado de forma óptima. Según se explicó en la sección 6.3, hemos dividido las ecuaciones del filtro en tres etapas: PREDICTION, CORRECTION y KALMAN_GAIN en lugar de sólo dos etapas, por tanto hemos diseñado 3 módulos principales, nombrados de igual modo que la etapa que implementa. Con esta distribución, cada módulo implementa 4 ecuaciones según se describe a continuación:

- Módulo PREDICTION: Implementa las ecuaciones 3.34, 3.35, 3.36 y 3.37 (Capitulo 8).
- Módulo CORRECTION: Implementa las ecuaciones 3.38, 3.39 , 3.40 , 3.41 (Capitulo 9).
- Módulo KALMAN_GAIN: Implementa las ecuaciones 3.42, **??** , 3.44 , 3.45 (Capitulo 10).

Inherentemente, existe un ciclo de trabajo que realizan todos los módulos del sistema en una iteración dada. Tal ciclo se describe como sigue:

1. Leer un nuevo dato de entrada.
2. Procesar el dato.
3. Proporcionar el resultado al resto de módulos.

El ciclo anterior se repite de forma indefinida mientras el sistema no sea *reseteado*. El sistema funciona con señal de reset síncrono a nivel alto y señal de reloj por flanco de subida. Activar la señal de *reset* = 1 implica poner al sistema en estado de espera y cargar todos los registros con valor 0. Las memorias conservarán su último valor y por tanto serán considerados no válidos.

7.1.1. Inicialización del Sistema

Como cualquier sistema, es necesario inicializarlo antes de comenzar los cálculos. El hecho de tener un sistema iterativo realimentado provoca un problema en la primera iteración tras el *start* del sistema, puesto que no existen valores válidos para realimentar. En nuestro caso, la primera iteración puede describirse como una fase de inicialización en la que asignamos los siguientes valores a las variables:

- Media del estado:

$$\overline{x}_0 = \begin{bmatrix} 0 \\ 0{,}5 \\ 1 \\ -0{,}001 \end{bmatrix}$$

- Covarianza del estado:

$$P_0 = \begin{bmatrix} 0{,}05 & 0 & 0 & 0 \\ 0 & 0{,}5 & 0 & 0 \\ 0 & 0 & 0{,}001 & 0 \\ 0 & 0 & 0 & 0{,}001 \end{bmatrix}$$

- Parámetros para calcular los Puntos Sigma:

$$\alpha = 0{,}1 \qquad \kappa = -1$$

$$\lambda = -3{,}97 \qquad \beta = 2$$

$$\gamma = 0{,}173206$$

- Pesos:

$$W^{(m)} = \begin{bmatrix} -132{,}333 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \end{bmatrix} \qquad W^{(c)} = \begin{bmatrix} -129{,}343 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \\ 16{,}667 \end{bmatrix}$$

$$R^v = \begin{bmatrix} 0{,}001 & 0 & 0 & 0 \\ 0 & 0{,}001 & 0 & 0 \\ 0 & 0 & 0{,}001 & 0 \\ 0 & 0 & 0 & 0{,}001 \end{bmatrix} \qquad R^v = \begin{bmatrix} 0{,}999 & 0 \\ 0 & 0{,}999 \end{bmatrix}$$

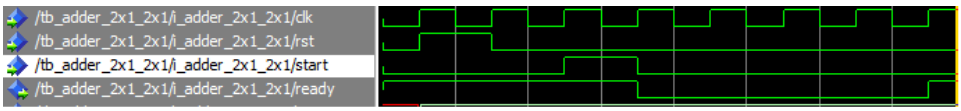

Figura 7.1: Cronograma del protocolo de comunicación

Todos los valores asignados en esta etapa son constantes almacenadas en ROMs. La definición de las constantes puede encontrarse en el paquete *filter_ constants* en la librería *common* (véase sección 7.2). Tras esta inicialización el sistema itera de forma normal.

7.1.2. Protocolo de comunicación

El protocolo de comunicación descrito a continuación es común a todo el proyecto. El hecho de tener unidades de control descentralizadas y jerarquicas implica coordinar cada submódulo con el módulo maestro en el que se instancia, ya que no existe comunicación directa entre submódulos. Para lograr este fin hemos escogido un protocolo de comunicación del tipo start - ready: se basa en enviar la señal *start* al submódulo receptor para indicarle que ya tiene disponible los datos de entrada necesarios y por tanto que comience su ejecución. Tras esto, se espera la recepción de la señal *ready* por parte de ese mismo submódulo, que indica la finalizacion de su ejecución y la disponibilidad de los nuevos datos.

La señal *start* = 1 es enviada por la unidad de control maestra al siguiente módulo esclavo que toque ejecutar durante **un único ciclo de reloj**. Por el contrario, un módulo mantiene la señal *ready* activa (ready = '1') mientras se encuentre preparado para comenzar una nueva ejecución y tras terminar su ejecución y poner a disposición de los demás módulos todos los datos. La señal cambiará su valor a *ready* = '0ál ciclo siguiente de recibir la orden de *start*.

Como ejemplo, se muestra en la figura 7.1 un cronograma donde podemos apreciar el protocolo.

H

Como muestra el cronograma, tras el reset del sistema, enviamos la señal *start* = '1én el cuarto flanco, durante un ciclo, y en el quinto flanco la señal *ready* cambia su valor a 0 indicando el comienzo de ejecución del módulo. En el octavo flanco de reloj la señal *ready* = '1índica el fin de ejecución del módulo, que vuelve a estar disponible para su uso.

7.1.3. Representación y almacenamiento de los datos

El sistema trabaja con vectores de 48 bits. Los datos internamente se representan en formato punto fijo, por ser más sencillo de implementar que

la coma flotante, y además no es necesaria la potencia y precisión de esta última representación. Tras el modelo desarrollado en Matlab en la sección 6.3 son necesarios 24 bits para representar la parte entera y 24 bits para la parte fraccional del número, que en notación Q corresponde a $Q_{48,24}$, por tanto, la palabra consta de 48 bits.

Los datos se almacenan de forma lineal en las memorias. Así una memoria que almacena una matriz de 2x4 elementos dispone de 8 posiciones, cada una con un tamaño de palabra de 48 bits, y los datos se almacenarán secuencialmente entre las posiciones 0 y 7.

La trasmision secuencial de los datos entre módulos implica disponer de memorias *ram_generic* que almacenen los datos a la salida de los módulos productores y que permitan la lectura secuencial a los módulos consumidores. Tambien es necesario el uso de memorias para almacenar completamente las matrices de datos antes de que un módulo consumidor comience a hacer uso de ellas.

El módulo productor es el encargado de generar la dirección válida de escritura *inaddr* y la señal de habilitación de escritura *wea* a la vez que pone el dato a escribir en la entrada *input* de la ram correspondiente para almacenar el valor correctamente.

Un módulo consumidor es el encargado de generar la dirección valida de lectura *outaddr* para obtener el dato correspondiente a dicha posición. Debido a la lectura síncrona de la ram, obtenemos el dato solicitado un ciclo de reloj después por el puerto *output* de la ram correspondiente.

7.2. Jerarquía del diseño

El proyecto se divide en librerias las cuales agrupan componentes que comparten propósitos similares. Una lista detallada de las librerias disponibles puede verse a continuación:

- *top_UKF*: agrupa los módulos que componen el top del diseño. Formado por:
  - el interfaz: *top_UKF* y su controlador *top_UKF_ctrl*
  - el modulo que agrupa las etapas del filtro: *UKF* y su controlador *UKF_ctrl*
- *prediction*: agrupa los módulos que implementan la etapa PREDICTION. Compuesto por:
  - módulo principal de la etapa: *prediction* y su controlador *prediction_ctrl*
  - submódulos, implementan cada una de las ecuaciones: *sigma_pts*, *state_transition*, *state_estimation* y *pcov_estimation*

- *correction*: agrupa los módulos que implementan la etapa CORRECTION. Compuesto por:
  - módulo principal de la etapa: *correction* y su controlador *correction_ctrl*
  - submódulos, implementan cada una de las ecuaciones: *measure_transf*, *measure_estimation* y *pyycov_estimation*
- *kalman_gain*: agrupa los módulos que implementan la etapa KALMAN_GAIN. Compuesto por:
  - módulo principal de la etapa: *kalman_gain* y su controlador *kalman_gain_ctrl*
  - submódulos, implementan cada una de las ecuaciones: *pxycov_estimation*, *kalman_gain_calc*, *state_correction* y *pcov_correction*
- *high_level_ops*: agrupa los módulos comunes que implementan operaciones sobre matrices:
  - sumadores: *adder_2x1_2x1* , *adder_2x2_2x2*, *adder_4x1_4x1*, *adder_4x4_4x4* y *summatory_2x9*
  - restadores: *sub_2x1_2x1* , *sub_2x9_2x1*, *sub_2x9_2x9*, *sub_4x4_4x4*, *sub_4x9_4x1* y *sub_4x9_4x9*
  - multiplicadores: *mult_1x4_4x1* , *mult_1x9_9x1*, *mult_2x9_9x2*, *mult_4x1_1x2*, *mult_4x1_1x4*, *mult_4x2_2x1*, *mult_4x2_2x2*, *mult_4x2_2x4*, *mult_4x9_9x2* y *mult_4x9_9x4*
  - multiplicadores escalares: *mult_scalar_1x9_2x9* , *mult_scalar_1x9_4x9*, *mult_scalar_2x1*, *mult_scalar_4x1* y *mult_scalar_4x4*
- *cholesky*: agrupa los módulos usados para realizar la raiz cuadrada de una matriz mediante el algoritmo de Cholesky:
  - módulo principal del algoritmo: *cholesky4x4* y su controlador *fsm_chol*
  - algoritmo no restoring para calcular la raiz cuadrada: *square_root*
- *common*: agrupa los módulos comunes básicos y genéricos usados en el diseño de otros módulos del filtro, también incluye la definición de diversos paquetes de constantes:
  - contador ascendente: *ascending_counter_generic*
  - elementos de memoria: *reg_scalar* , *ram_generic*
  - maquinas de estado genéricas: *fsm_generic* , *fsm_mult_generic*
  - otros módulos especiales: *det_2x2* , *div_norest*, *round_div*, *transpose*, *fsm_complex_mult*

- paquete *constants*: define el ancho de bits de los buses de datos, así como lo bits utilizados como parte entera y parte decimal en la representación en punto fijo (ver sección Representación de los datos **??**)
- paquete *filter_constants*: define el valor de las constantes utilizadas en el diseño. Dichas constantes se almacenarán en ROMs.

Como se puede comprobar en el listado anterior, cada módulo se compone de submódulos más pequeños y de su propia unidad de control interna. A su vez, los submódulos disponen de una unidad de control propia y específica a demás de otros componentes. Este hecho caracteriza al sistema de disponer de unidades de control descentralizadas y jerarquicas.

7.3. Módulos Comunes

Acorde al diseño jerárquico, las tareas complejas de los módulos principales se pueden descomponer en tareas más simples de operaciones sobre matrices. De esta forma, hemos implementado una serie de módulos sencillos y reutilizables, los cuales efectúan directamente las operaciones sobre los datos. Tal como se comentó en una sección anterior, estos módulos se agrupan en las librerias *high_level_ops* y *common*.

7.3.1. FSMs genéricas

Se trata de controladores genéricos modelados mediante FSMs usados en la implementación de los componentes más básicos, los cuales efectúan directamente las operaciones sobre los datos. Todas utilizan el protocolo de comunicación *start* - *ready* antes descrito. Estos controladores generan internamente las señales de control necesarias, así como las direcciones de acceso a las memorias de datos.

7.3.1.1. Controlador *fsm_generic*

Utilizado para implementar sumadores, restadores y multiplicadores escalares de matrices. Se debe asignar un valor al atributo genérico LENGTH, el cual controla cuantos ciclos internos tiene el controlador antes de notificar que ha terminado su ejecución. Proporciona una interfaz sencilla según se describe en la tabla 7.1.

El diagrama de transición de estados de la figura 7.2 y la tabla 7.2 muestran los estados del controlador.

Puerto	Tamaño	Sentido	Comentario
LENGTH	-	Generic	Especifica el valor máximo de cuenta del contador interno: de 0 hasta LENGTH-1
start	1 bit	Entrada	Comienzo del submódulo (1 ciclo a alta).
ready	1 bit	Salida	Flag de fin del submódulo ('0'= busy) .
wea	1 bit	Entrada	Habilitación de escritura en la memoria de salida
index	$\log_2(LENGTH)$	Salida	Indice con el valor actual de la cuenta, usado como direccion de acceso a las memoria de datos (lectura y escritura)

Tabla 7.1: Interfaz del controlador *fsm_ generic*

Estado	Acción
ready_st	Notifica que se encuentra preparada para comenzar la ejecución (ready $\rightarrow$ 1) y espera a recibir la señal *start* $\leftarrow$ 1. Inicializa el contador interno index_reg $\rightarrow$ 0 y pone en el bus de direcciones de acceso a memorias el valor 0 (el contador indica el valor del bus de direcciones)
wait_ram	Espera un ciclo para que la ram nos de el valor
row_count_st	Incrementa el contador mientras no llegue al maximo valor y pide a las memorias ram el siguiente dato. Habilita la escritura de datos en la ram de salida

Tabla 7.2: Estados de la unidad de control *fsm_ generic*

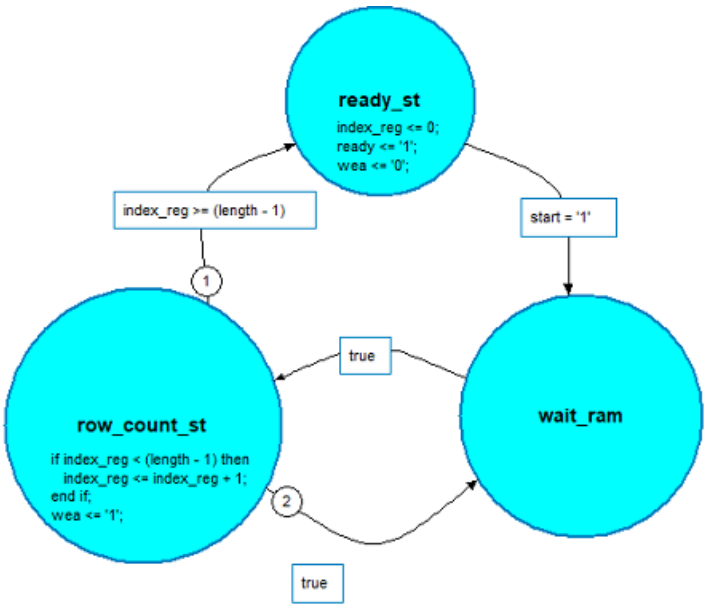

Figura 7.2: Diagrama de estados del controlador *fsm_generic*

7.3.1.2. Controlador *fsm_mult_generic*

Utilizado para implementar multiplicadores de matrices. Este controlador necesita dos contadores externos *ascending_counter_generic* para funcionar correctamente. Uno de los contadores es usado para contar ciclos de operaciones y el otro contador llevará el indice de la columna a procesar de la segunda matriz de entrada. La tabla 7.3 muestra la interfaz que proporciona.

El diagrama de estados muestra detalladamente como se implementa el controlador (figura 7.3). Los estados se describen en la tabla 7.4

7.3.2. Contador ascendente: *ascending_counter_generic*

Contador síncrono ascendente genérico. Permite definir el valor máximo de cuenta mediante el atributo genérico G_MAX_COUNT, por tanto, la cuenta tomará valores entre 0 y G_MAX_COUNT-1

El contador reinicia el valor de cuenta *count* = 0 cuando recibe la señal de *clear*. Mientras reciba la señal de habilitación de cuenta *count_enable* = '1', se incrementará la cuenta una unidad cada ciclo de reloj. Cuando se alcance el valor G_MAX_COUNT lo notificará con la señal *end_count* = '1'.

La tabla 7.5 recoge los detalles de su interfaz.

Puerto	Tamaño	Sentido	Comentario
start	1 bit	Entrada	Comienzo del submódulo (1 ciclo a alta).
ready	1 bit	Salida	Flag de fin del submódulo ('0'= busy) .
clear_cont_col	1 bit	Salida	Señal para limpiar la cuenta del contador de columnas
en_cont_col	1 bit	Salida	Habilitación de cuenta del contador de columnas
clear_cont_m1	1 bit	Salida	Señal para limpiar la cuenta del contador de ciclos de operaciones
en_cont_m1	1 bit	Salida	Habilitación de cuenta del contador de ciclos de operaciones
end_count_m1	1 bit	Entrada	Señal de fin de cuenta del contador de ciclos de operaciones
end_count_col	1 bit	Entrada	Señal de fin de cuenta del contador de columnas

Tabla 7.3: Interfaz del controlador *fsm_mult_generic*

Estado	Acción
ready_st	Notifica que se encuentra preparada para comenzar la ejecución (ready $\rightarrow$ 1) y espera a recibir la señal *start* $\rightarrow$ 1. Limpia la cuenta de ambos contadores (clear_cont_col $\rightarrow$ 1, clear_cont_m1 $\rightarrow$ 1)
working_st	Habilita la cuenta en el contador de ciclos de operaciones y espera hasta que notifique el fin de cuenta (end_count_m1 $\leftarrow$ 1)
col_count_st	Incrementa el contador de columnas (en_cont_col $\rightarrow$ 1) mientras no reciba la señal de fin de cuenta de ambos contadores (end_count_col $\leftarrow$ 1 & end_count_m1 $\leftarrow$ 1). Limpia la cuenta del contador de ciclos de operaciones (clear_cont_m1 $\rightarrow$ 1)

Tabla 7.4: Estados de la unidad de control *fsm_mult_generic*

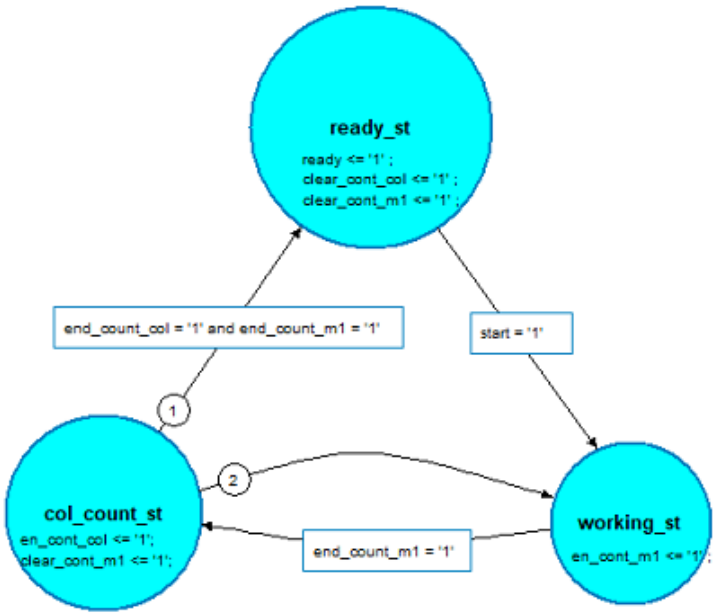

Figura 7.3: Diagrama de estados del controlador *fsm_mult_generic*

7.3.3. Elementos de Almacenamiento

Elementos usados para almacenar los datos.

7.3.3.1. Memoria *ram_generic*

Este módulo describe una memoria ram con doble puerto. Cabe destacar que al sintetizar este componente se infiere en Block_RAM. El puerto de lectura permite la lectura síncrona y el puerto de escritura permite escritura síncrona mediante habilitación. La memoria almacena datos representados en punto fijo con un ancho de palabra de 48 bits. El atributo genérico G_LENGTH permite especificar el numero máximo de posiciones de la memoria. La tabla 7.6 describe la interfaz de la memoria.

No es necesaria una señal de capacitación de lectura, el hecho de proporcionar una dirección de lectura hace que la memoria entregue los datos que contenga en esa posición. El dato estará en la salida *output* al ciclo siguiente de especificar la dirección.

La escritura se realiza mientras la señal *wea* esté activada. Cada ciclo que esta señal esté activa se almacena el dato *input* en la posición indicada por la dirección *inaddr*.

La figura 7.4 muestra el cronograma de escritura y posterior lectura de datos en una memoria *ram_generic* de 4 palabras. Los cinco primeros flancos

Puerto	Tamaño	Sentido	Comentario
G_MAX_COUNT	-	Generic	Permite especificar el valor máximo de cuenta: desde 0 hasta G_MAX_COUNT-1
clear	1 bit	Entrada	Reinicio de cuenta a valor 0
count_enable	1 bit	Entrada	Habilitación de cuenta
end_count	1 bit	Salida	Notificación de fin de cuenta
count	G_MAX_COUNT	Salida	Valor de la cuenta actual

Tabla 7.5: Interfaz del contador *ascending_counter_generic*

Puerto	Tamaño	Sentido	Comentario
G_LENGTH	-	Generic	Permite definir cuántas posiciones tiene la memoria
input	c_sfixed_width	Entrada	Valor a escribir en la memoria
inaddr	G_LENGTH	Entrada	Posición de escritura
wea	1 bit	Entrada	Habilitación de escritura
output	c_sfixed_width	Salida	Valor de la memoria en la posicion indicada por outaddr
outaddr	G_LENGTH	Entrada	Posición de lectura

Tabla 7.6: Interfaz de la memoria *ram_generic*

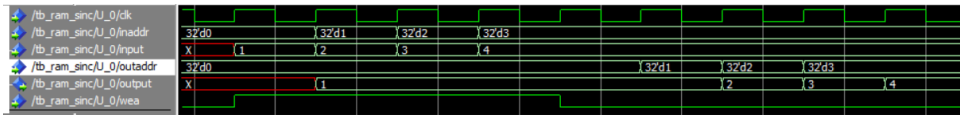

Figura 7.4: Cronograma de lectura/escritura en *ram_ generic*

del cronograma muestran la acción de escritura en la memoria, para ello la señal *inaddr* toma en cada flanco los valores 0, 1, 2 y 3 respectivamente y la señal *input* especifica el valor a guardar en cada posición: el valor 1 en la posición 0, el valor 2 en la posición 1, ... En el quinto ciclo, la señal *wea* cambia, ya se han escrito todos los valores. En los ciclos sexto, septimo y octavo la señal *outaddr* especifica la dirección de las posiciones a leer: 1, 2 y 3 respectivamente y se observa cómo la señal *output* saca los valores pedidos al ciclo de reloj siguiente de especificar la dirección.

7.3.3.2. Registro *reg_ scalar*

Componente usado para almacenar un dato del tipo sfixed con un ancho de palabra de 48 bits. Usado como acumulador en algunas operaciones y como registro de un sólo elemento. Proporciona una interfaz muy sencilla segun se describe en la tabla 7.7.

Puerto	Tamaño	Sentido	Comentario
load	1 bit	Entrada	Señal de carga en el registro el valor *data_ in*
clear	1 bit	Entrada	Señal de borrado del valor almacenado, sobreescribe con 0
data_in	c_sfixed_width	Entrada	Dato a guardar en el registro
data_out	c_sfixed_width	Salida	Valor almacenado en el registro

Tabla 7.7: Interfaz del registro *reg_ scalar*

El dato *data_ in* se almacena en el registro un ciclo después de recibir la señal *load* $\leftarrow$ 1. El puerto *data_ out* mantiene siempre el valor cargado en el registro. La señal *clear* $\leftarrow$ 1 inicializa el valor del registro a 0.

El cronograma de escritura en el registro puede verse en la figura 7.5:

Tras activar la señal *clear* = '1én el primer flanco, la salida *s_ result_ internal* toma el valor 0. En el segundo flanco se activa la señal *load* = '1'y el registro almacenará el dato *mult_ aux* en el siguiente ciclo de reloj.

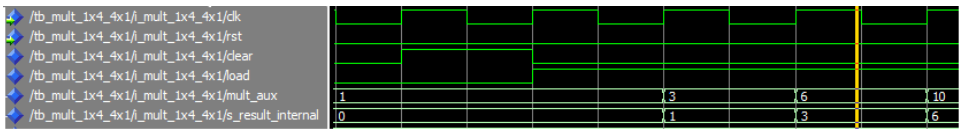

Figura 7.5: Cronograma de escritura en *reg_scalar*

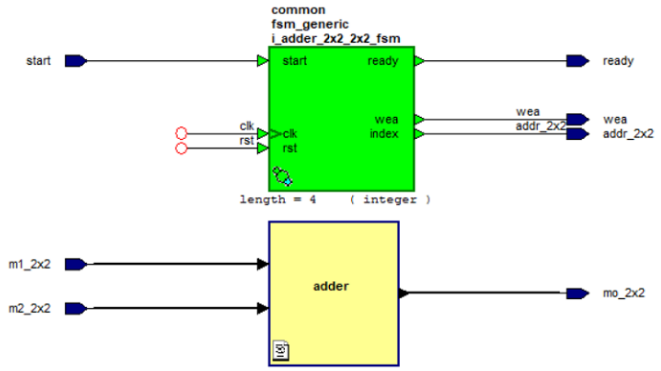

Figura 7.6: Sumador *adder_2x2_2x2*

7.3.4. Sumadores y Restadores

Hemos implementado diversos tipos de sumadores, cada uno encargado de realizar la suma de dos matrices de entrada de dimensiones específicas. Los restadores implementados siguen la misma filosofía. Estos son los sumadores disponibles: *adder_2x1_2x1*, *adder_2x2_2x2*, *adder_4x1_4x1*, *adder_4x4_4x4*. y estos los restadores: *sub_2x1_2x1*, *sub_2x9_2x1*, *sub_2x9_2x9*, *sub_4x4_4x4*, *sub_4x9_4x1* y *sub_4x9_4x9*

Todos los componentes se implementan de igual forma y a modo de ejemplo se va a explicar el sumador: *adder_2x2_2x2* el cual puede verse en la figura 7.6.

Este módulo realiza la suma componente a componente de dos matrices de entrada de dimension 2x2 cada una. Se descomponen en un controlador *i_fsm_generic* y un sumador de dos escalares. El controlador se instancia con un valor de *length* = 4, indicando que se van a realizar 4 sumas en total, una por cada escalar de las matrices de entrada. Como se ha explicado anteriormente, el propio controlador genera las señales de *wea* y la direccion *addr_2x2* usada a la vez para lectura y escritura en las RAM.

7.3.5. Multiplicador de matrices elemento a elemento

Hemos implementado diversos multiplicadores de matrices que realizan la operación componente a componente, si bien, cada uno realiza la operación

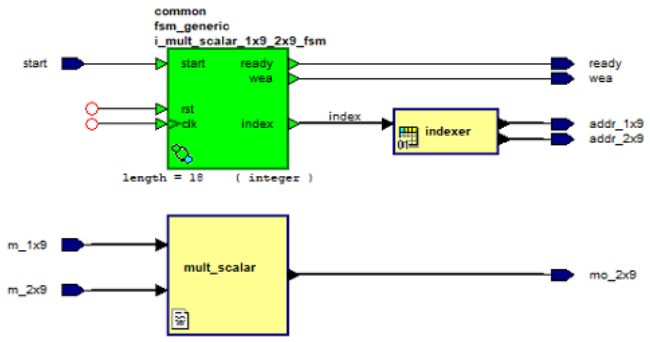

Figura 7.7: Multiplicador *mult_ scalar_ 1x9_ 2x9*

sobre matrices de entrada de una determinada dimensión. Disponemos de los siguientes: *mult_ scalar_ 1x9_ 2x9* , *mult_ scalar_ 1x9_ 4x9*

Cada uno de ellos, multiplica individualmente cada valor de una matriz fila de tamaño 1x9 por cada uno de los valores de cada fila de una matriz 2x9 o 4x9 respectivamente.

Los siguientes multiplicadores realizan una operación similar a los anteriores, la única diferencia existente es que reciben un valor escalar y una matriz como datos de entrada: *mult_ scalar_ 2x1*, *mult_ scalar_ 4x1* y *mult_ scalar_ 4x4*

Todos ellos se implementan de forma similar, por lo que a modo de ejemplo se explica el *mult_ scalar_ 1x9_ 2x9* el cual puede verse en la figura 7.7.

Se compone de un controlador, un multiplicador de dos escalares y una tabla de verdad, la cual genera las direcciones de acceso a las memorias a partir del índice del proporcionado por controlador.

El controlador se instancia con un valor de *length* = 18, indicando que se van a realizar 18 multiplicaciones en total. Llamemos m1 a la matriz de entrada de dimensión 1x9 y m2 a la matriz de 2x9. Las primeras 9 operaciones corresponden a multiplicar cada valor de m1 por el correspondiente valor de m2 de la primera fila, y las 9 finales corresponden a cada valor de m1 de nuevo por el correspondiente valor de m2 de la segunda fila. El propio controlador genera las señales de *wea* y el valor *index* usado para generar, a través de una tabla de verdad, las direcciones de lectura y escritura. La tabla 7.8 muestra resumidamente como se generan las direcciones. *index* toma valores entre 0 y 17, en los primeros 9 valores las dos direcciones indican la misma posición ya que los datos se almacenan secuencialmente en las memorias (direcciones 0 a 8). A partir de *index* = 9 es necesario acceder a la segunda fila de la matriz 2x9 (direcciones 9 a 17) y acceder de nuevo a la matriz 1x9 (direcciones 0 a 8).

index	addr_1x9	addr_2x9
0	0	0
1	1	1
2	2	2
...	...	...
8	8	8
9	0	9
10	1	10
11	2	11
...	...	...
17	8	17

Tabla 7.8: Tabla de verdad de *mult_scalar_1x9_2x9*

7.3.6. Multiplicadores de matrices

Hemos implementado diversos multiplicadores de matrices, cada uno realiza la operación sobre matrices de entrada de una determinada dimensión. Disponemos de los siguientes: *mult_1x4_4x1* , *mult_1x9_9x1*, *mult_2x9_9x2*, *mult_4x1_1x2*, *mult_4x1_1x4*, *mult_4x2_2x1*, *mult_4x2_2x2*, *mult_4x2_2x4*, *mult_4x9_9x2* y *mult_4x9_9x4*

Todos ellos se implementan de forma similar, por lo que a modo de ejemplo se explica el *mult_4x2_2x2*, el cual puede verse en la figura 7.8.

Se compone de un controlador *fsm_mult_generic* y dos contadores, un multiplicador y un sumador de dos escalares para acumular el resultado, un registro que almacenará el valor parcial de la operación y una tabla de verdad, la cual genera las direcciones de acceso a las memorias a partir de los valores de los contadores y las señales de control de *wea*, *load* y *clear*.

El modo de operar es el siguiente:

- Tenemos dos matrices de entrada de dimensiones m_{4x2} y m_{2x2}:

$$m_{4x2} = \begin{bmatrix} a0 & a1 \\ a2 & a3 \\ a4 & a5 \\ a6 & a7 \end{bmatrix} \qquad y \qquad m_{2x2} = \begin{bmatrix} b0 & b1 \\ b2 & b3 \end{bmatrix}$$

- El contador de ciclos de operaciones debe contar 13 ciclos (de 0 a 12) de acuerdo a lo siguiente:
  - Tardamos dos ciclos en obtener los operandos (a0, a1) y (b0, b2) de acuerdo a la forma común de multiplicar matrices.
  - Tardamos un ciclo en escribir el resultado $a0 \times b0 + a1 \times b2$ en la memoria de salida.

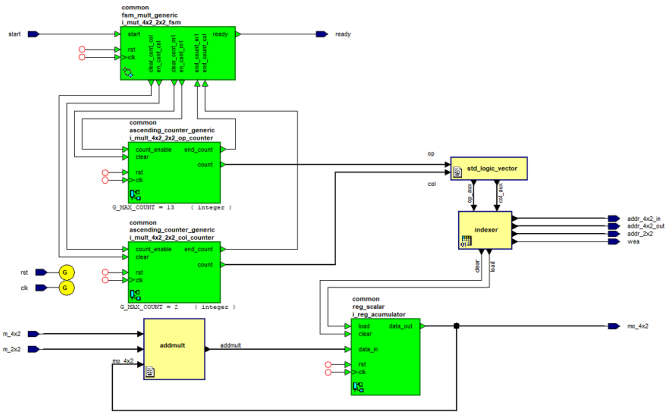

Figura 7.8: Multiplicador *mult_ 4x2_ 2x2*

De esta forma tardamos $3 \times 4 = 12$ ciclos en operar las 4 filas de la matriz m_{4x2} mas un ciclo para activar la señal de *clear* y borrar el valor del registro acumulador, en total 13 ciclos.

- El contador que indica el índice de la columna de la segunda matriz debe contar hasta 2 (de 0 a 1) de acuerdo a que la matriz m_{2x2} dispone de dos columnas y debemos repetir el procedimiento anterior pero esta vez con los operandos (b1, b3) en lugar de (b0, b2)

Así la tabla de verdad contiene $13 \times 2 = 26$ posiciones. La tabla 7.9 muestra como se generan las señales para los primeros trece ciclos mientras col_aux $= 0$, correspondientes a multilpicar las 4 filas de m_{4x2} por la primera columna (b0, b2) de m_{2x2}.

7.3.7. Determinante

El módulo *det_ 2x2* calcula el determinante de una matriz de dimensiones 2x2. El determinante de una matriz de dimensiones 2x2 se describe en la ecuación (7.1). Dividimos este módulo en un submódulo común explicado en la sección 7.3 y cuatro bloques privados. El bloque privado *det_ controller* se encarga de secuenciar el inicio de ejecución de cada submódulo:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = a \cdot d - c \cdot b \tag{7.1}$$

- *i_ det_ 2x2_ reg1* e *i_ det_ 2x2_ reg2* (reg_scalar): Implementan registros en los que se almacenarán, primero en uno y luego en el otro, los datos a multiplicar.

op_aux	col_aux	addr_4x2_in	addr_2x2	addr_4x2_out	wea	load	clear
0000	0	0	0	0	'0'	'0'	'1'
0001	0	1	2	0	'0'	'1'	'0'
0010	0	2	0	0	'0'	'1'	'0'
0011	0	2	0	0	'1'	'0'	'1'
0100	0	3	2	0	'0'	'1'	'0'
0101	0	4	0	2	'0'	'1'	'0'
0110	0	4	0	2	'1'	'0'	'1'
0111	0	5	2	2	'0'	'1'	'0'
1000	0	6	0	4	'0'	'1'	'0'
1001	0	6	0	4	'1'	'0'	'1'
1010	0	7	2	4	'0'	'1'	'0'
1011	0	0	1	6	'0'	'1'	'0'
1100	0	0	1	6	'1'	'0'	'0'
...	...	...	...	...	...	...	...

Tabla 7.9: Tabla de verdad de *mult_4x2_2x2*

- *i_det_2x2_reg* (reg_scalar): Implementa un registro que servirá de acumulador de las multiplicaciones.

La figura 7.9 muestra el diagrama de bloques de este módulo. La matriz a la que accede este módulo debe ser leída en un orden propio del determinante por ello se necesita una lógica de control que sincronice todo el hardware:

- Bloque *mult*: Es el que realiza la multiplicación de los datos guardados en los dos registros *i_det_2x2_reg1* e *i_det_2x2_reg2*.
- Bloque *muxsub*: Está formado por un multiplexor y una resta. Cuando el valor que hemos multiplicado es el primero en la entrada del registro *i_det_2x2_reg* estará la multiplicación de los valores de los dos registros *i_det_2x2_reg1* e *i_det_2x2_reg2*. Si toca realizar la segunda multiplicación el multiplexor actúa y resta al valor del registro la nueva multiplicación de los dos valores de los dos registros *i_det_2x2_reg1* e *i_det_2x2_reg2*.
- Bloque *addr_2x2_in_indexer*: Es una tabla de verdad implementada en LUT para indexar la BRAM de entrada. Su comportamiento se muestra en la tabla 7.10.

Cuando recibe la señal de start el módulo realiza los siguientes pasos secuencialmente, no iniciando el siguiente paso hasta que no haya finalizado el anterior:

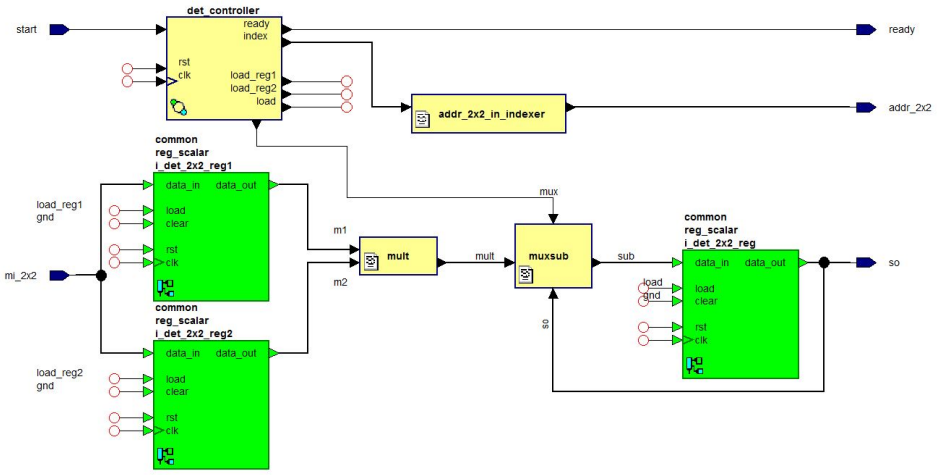

Figura 7.9: Determinante *det_ 2x2*.

Entrada	Salida
0	0
1	3
2	1
3	2

Tabla 7.10: Tabla de verdad para generar la dirección de la BRAM de entrada.

1. Se baja la señal de ready y se espera un ciclo por el retardo de la BRAM externa que tiene almacenada la matriz.

2. El controlador manda cargar el primer dato en el registro *i_ det_ 2x2_ reg1* activando su load. La dirección viene dada por el índice que indexa la LUT, que en este caso es cero.

3. El controlador manda cargar el segundo dato en el registro *i_ det_ 2x2_ reg2* activando su load. La dirección viene dada por el índice que indexa la LUT, que en este caso es uno.

4. El controlador manda cargar el resultado en el registro *i_ det_ 2x2_ reg* activando su load pero sin restarlo a lo acumulado en el registro, es decir, con la señal mux a cero.

5. Se realizan otra vez los pasos uno y dos con los índices dos y tres y después se carga el dato en el registro efectuando la resta con la anterior

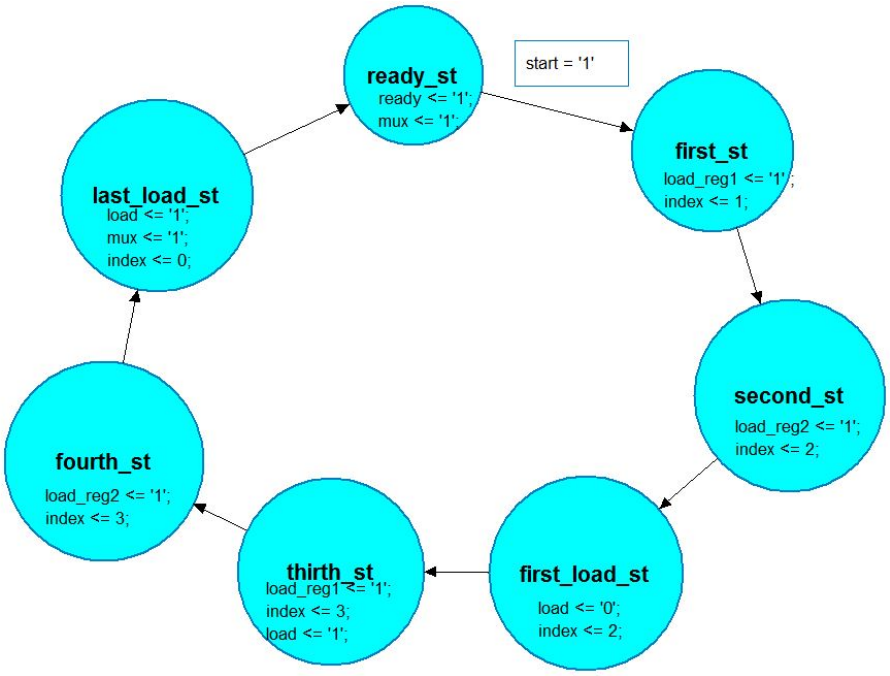

Figura 7.10: Controlador del módulo *det_ 2x2*.

multiplicación que ya estaba almacenada, es decir, con la señal mux a uno.

6. Se marca el final de la operación estableciendo ready a uno.

El bloque *det_ controller* implementa la máquina de estados usada a modo de controlador y tiene los estados que se muestran en la tabla 7.11. Podemos ver el diagrama de máquina de estados en la figura 7.10

7.3.8. Inversa

El módulo *inv_ 2x2* implementa la inversa de una matriz de dimensiones 2x2 descrita en la ecuación (7.2). Para hallar la matriz inversa se necesita primero calcular el determinante que será el divisor de cada elemento de la matriz de adjuntos transpuesta. Este módulo se encontrará localizado entre dos BRAMs: la de entrada, que proporciona los datos de la matriz que se quiere invertir, y la BRAM de salida, que almacenará el resultado. Dividimos este módulo en dos submódulos comunes explicados en la sección 7.3 y cinco bloques privados. El bloque privado *control* se encarga de secuenciar el inicio

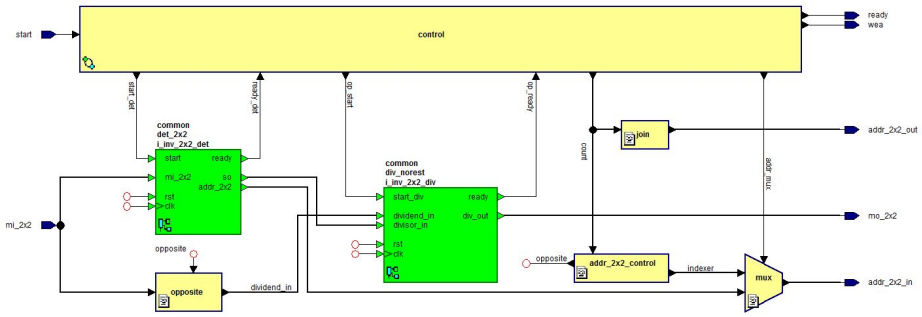

Figura 7.11: Inversa *inv_ 2x2*.

de ejecución de cada submódulo:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{\left(Adj \begin{pmatrix} a & b \\ c & d \end{pmatrix}\right)^{T}}{Det \begin{pmatrix} a & b \\ c & d \end{pmatrix}} \tag{7.2}$$

- *i_ inv_ 2x2_ det* (det_2x2): Implementa el hardware que calcula el determinante que será el divisor de la matriz de adjuntos transpuesta.
- *i_ inv_ 2x2_ div* (div_norest): Implementa la división que se realizará por cada elemento de la matriz de adjuntos transpuesta.

La figura 7.11 muestra el diagrama de bloques de este módulo. La matriz a la que accede este módulo debe ser leída en un orden propio de la transpuesta de los adjuntos de la matriz por ello se necesita una lógica de control que sincronice todo el hardware:

- Bloque *opposite*: Implementa un control de signo del dato que se usa como dividendo. Si la señal opposite está activada a uno, el valor del dividendo es el contrario al del dato de la BRAM de lectura. Es necesario ya que la matriz de adjuntos cambia de signo por cada elemento accedido debido a sus características matemáticas.
- Bloque *addr_ 2x2_ control*: Implementa una tabla de verdad almacenada en una LUT que se encarga de generar las direcciones de lectura de la BRAM de entrada, como podemos ver en la tabla 7.12.
- Bloque *mux*: Implementa un multiplexor controlado por la señal addr_mux que permite al submódulo del determinante tener el control de las direcciones de la RAM de lectura y las que genera *addr_ 2x2_ control*.

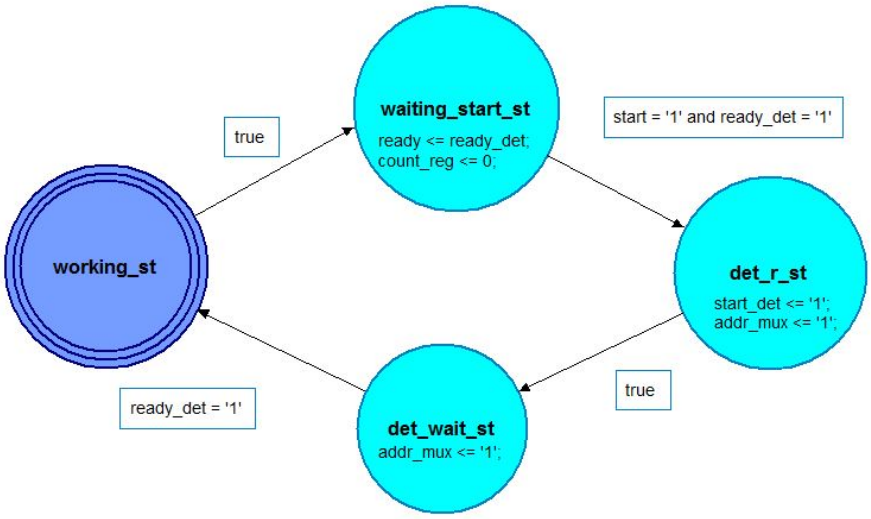

Figura 7.12: Controlador del módulo *inv_ 2x2*.

- Bloque *join*: Implementa una unión de señales para que tengan el mismo valor. Estas señales son count que es el contador que se usará internamente para controlar que valor de la matriz de adjuntos transpuesta estamos computando y addr_2x2_out que es la dirección de la BRAM de escritura.

El bloque *control* implementa la máquina de estados usada a modo de controlador y tiene los estados que se muestran en la tabla 7.13. Podemos ver el diagrama de máquina de estados en la figura 7.12. El estado working_st es un estado jerárquico que engloba los estados de la figura 7.13. Usa un contador como medida para controlar el número de divisiones y para generar las direcciones de las dos BRAMS.

7.4. Interfaz del diseño

El módulo *top_ UKF* representa el nivel mas alto de integración de componentes y realiza la función de interfaz de comunicación del filtro con el exterior. Contiene todos los elementos necesarios para iniciar el sistema, registrar los nuevos datos de entrada al filtro, registrar los resultados de salida y realimentar los valores de salida hacia la entrada creando la conexión necesaria para el funcionamiento continuo del filtro.

La tabla 7.14 recoge detalladamente los puertos del interfaz.

t

Como se puede observar en la figura 7.15, todas las señales de entrada del

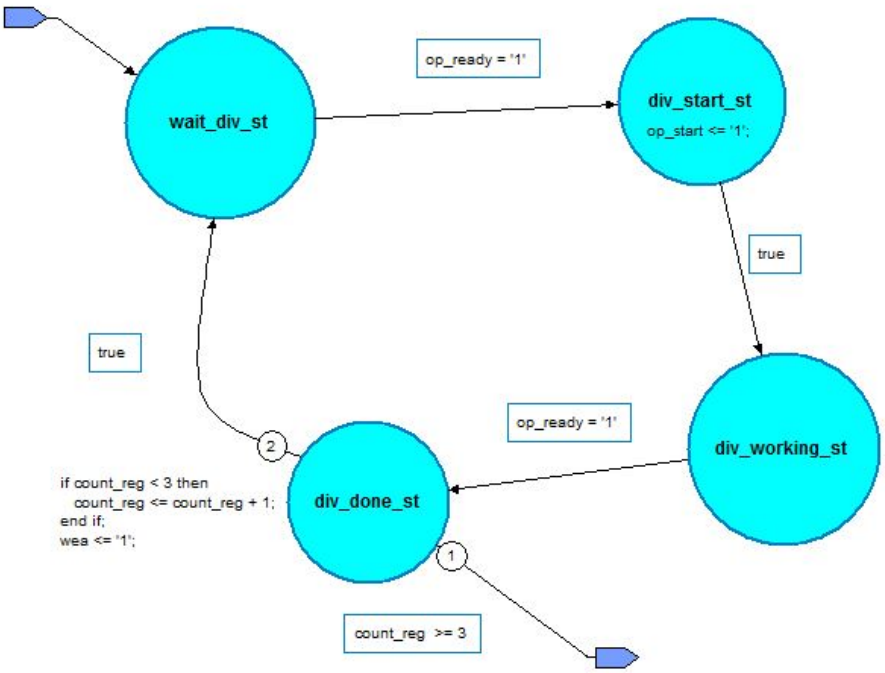

Figura 7.13: Estado jerárquico working_st del módulo *inv_2x2*.

interfaz son registradas dos veces. Por cada registro en las señales de entrada reducimos exponencialmente la probabilidad de que una valor metaestable se propague por el sistema, por ello es muy recomendable no incorporar señales externas al sistema directamente.

El módulo *i_UKF* implementa las ecuaciones propias del Filtro Kalman Unscented y las memorias *i_top_state_ram* e *i_top_pcov_ram* almacenan, respectivamente, los valores de salida del módulo *i_UKF*: estado y covarianza del estado de la iteración *i-1* para realimentarlo de nuevo a la entrada del módulo en la iteración actual *i*. Los registros de salida *i_top_glucose_reg* y *i_top_gain_reg* permiten mantener estable el valor de las salida fuera del interfaz *top_UKF* y sólo se actualizan al nuevo valor tras la señal de *step_done*.

Los multiplexores *init_pcov_mux* e *init_state_mux* permiten asignar valores validos a las entradas del filtro estado y covarianza del estado en la iteración inicial del sistema. Tales valores se toman directamente desde ROMs ya que en la primera iteración no se dispone de datos validos en las memorias RAM *i_top_state_ram* e *i_top_pcov_ram*.

El multiplexor *addr_mux* permite el acceso a la memoria *i_top_state_ram* desde los registro de salida *i_top_glucose_reg* e *i_top_gain_reg* y desde el

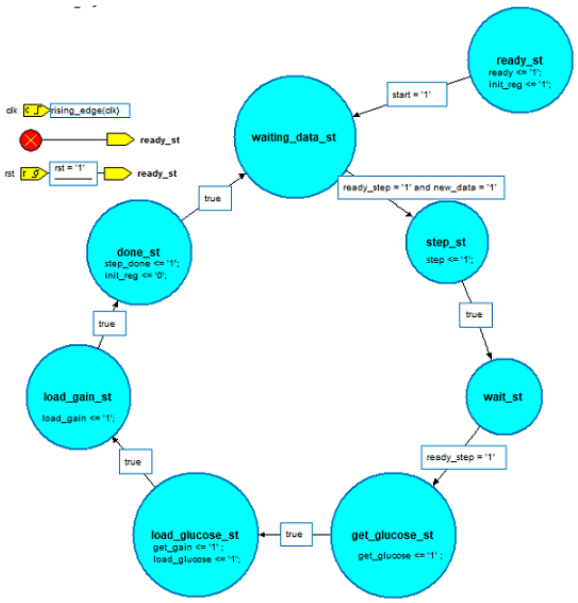

Figura 7.14: Diagrama de estados del controlador *i_top_UKF_ctrl*

módulo *i_UKF* en distintos ciclos.

7.4.1. Controlador *i_top_UKF_ctrl*

Este controlador debe asegurar el correcto funcionamiento cíclico del filtro. Arranca una nueva iteración enviando la señal *step* = '1ál módulo *UKF* cada vez que hay disponible una medida de entrada. La figura 7.14 muestra el diagrama de estados del controlador. Las acciones asociadas a cada estado del controlador se recogen en la tabla 7.15:

7.5. Módulo UKF

El módulo *i_UKF* implementa las ecuaciones propias del Filtro Kalman Unscented. Se compone de tres módulos principales, los cuales implementan cada una de las etapas del filtro según se explicó en la sección 7.1.

La figura 7.16 muestra el diagrama de bloques del módulo, el cual se compone de un controlador *i_UKF_ctrl* encargado de secuenciar las etapas, multiplexar el acceso a las memorias ram que almacenan los datos transmitidos entre módulos y, en general, permite realizar una iteración completa cada vez que recibe nuevos datos de medida.

Su interfaz se describe en la tabla 7.16.

El módulo *i_prediction* es el encargado de implementar las ecuaciones de la fase PREDICTION y a su salida encontramos las memorias *i_prior_state_ram*

y *i_prior_pcov_ram* que almacenan los valores de la media a priori del estado y de la covarianza a priori del estado respectivamente.

El módulo *i_correction* se encarga de implementar las ecuaciones de la fase CORRECTION y tras su ejecución la memoria *i_prior_measure_ram* almacena la media estimada de la medida, la memoria *i_desv_y_ram* almacena la diferencia entre los puntos sigma propagados por el modelo de medida y la medida predicha, *i_pyy_cov_ram* almacena la covarianza de la medida y *i_expanded_chix_ram* almacena el nuevo conjunto de puntos sigma calculados.

El módulo *i_kalman_gain* implementa las ecuaciones de la fase KALMAN GAIN, el cual escribe directamente los resultados en las memorias *i_top_state_ram* y *i_top_pcov_ram*.

Finalmente el multiplexor *i_mux_addr1* permite el acceso a la memoria *i_prior_state_ram* a los módulos *i_correction* y *i_kalman_gain* según se esté ejecutando uno u otro. De la misma forma, el multiplexor *i_mux_addr2* permite el acceso a la memoria *i_prior_pcov_ram* a los citados módulos.

7.5.1. Controlador *i_UKF_ctrl*

Este controlador se encarga de secuenciar el comienzo de los tres módulos principales del filtro además de generar las señales de control necesarias para realizar una iteración completa del filtro. Comienza una nueva iteración cada vez que recibe la señal *step* del controlador *i_top_UKF_ctrl*. Como puede verse en la figura 7.17, el controlador se compone de un estado inicial y 3 estados jerárquicos principales. Cada estado jerárquico se divide a su vez en en tres estados, los cuales controlan el comienzo y fin de ejecución de cada uno de los módulos. Éste controlador utiliza el mismo protocolo de comunicacación descrito anteriormente.

Las acciones asociadas a cada estado del controlador se recogen en la tabla 7.17:

Estado	Acción
ready_st	Notifica que está preparado (ready → 1) y permanece a la espera de la señal start ← 1. Mientras activa el multiplexor que selecciona qué valor se carga en el registro escalar final *i_det_2x2_reg* es solo el de la multiplicación de los dos registros. Esto se hace para que el último estado del controlador tenga efecto. También, por defecto, se asigna index → 0 para controlar la dirección de lectura de la BRAM externa está apuntando al primer valor.
first_st	Indica la carga del primer dato en el primer registro mediante load_reg1 → 1 y cambia el índice al siguiente valor de la BRAM externa para irlo precargando.
second_st	Indica la carga del segundo dato en el segundo registro mediante load_reg2 → 1 y cambia el índice al siguiente valor de la BRAM externa para irlo precargando.
fisrt_load	Es un ciclo de espera para cargar en el registro *i_det_2x2_reg* el valor de la multiplicación sin la resta de lo acumulado en el registro *i_det_2x2_reg*.
thirth_st	Indica la carga del tercer dato en el primer registro mediante load_reg1 → 1 y cambia el índice al siguiente valor de la BRAM externa para irlo precargando. Además activa el load *to* 1 para cargar el dato de la anterior multiplicación como se ha explicado en el estado anterior.
fourth_st	Indica la carga del cuarto dato en el segundo registro mediante load_reg2 → 1 y resetea el índice.
last_load_st	Carga el nuevo dato en el registro *i_det_2x2_reg* con la resta de lo acumulado en la anterior multiplicación gracias a la activación de mux (mux *to* 1).

Tabla 7.11: Tabla de estados del controlador del determinante.

Entrada	Dirección	Opposite
0	3	0
1	1	1
2	2	1
3	0	0

Tabla 7.12: Tabla de verdad para generar la dirección de la BRAM de entrada y opposite.

Estado	Acción
waiting_start_st	Notifica que está preparado (ready $\rightarrow$ 1) si el determinante está preparado y permanece a la espera de la señal start $\leftarrow$ 1 y el ready del determinante mediante la señal ready_det $\leftarrow$ 1. Este estado también resetea el contador de divisiones.
det_r_st	Inicia el determinante mediante la señal start_det $\rightarrow$ 1 y permite que este submódulo pueda controlar el puerto de la BRAM de entrada mediante la activación de la señal addr_mux $\rightarrow$ 1.
det_wait_st	Espera un ciclo hasta que ready_det $\leftarrow$ 1 para pasar al estado jerárquico working_st que contiene los estados de la figura 7.13. Sigue manteniendo el control de la BRAM de lectura en el submódulo del determinante. Desde este estado el módulo de la inversa es el que tiene el control de la dirección de la BRAM de lectura.
wait_div_st	Espera a que la división está preparada mediante la señal op_ready $\leftarrow$ 1.
div_start_st	Activa la división mandándole el start por la señal op_start $\rightarrow$ 1.
div_working_st	Espera a que la división marque que ha terminado con la señal op_ready $\leftarrow$ 1.
div_done_st	En este estado la división ya ha terminado, por tanto la contamos en lamáquina de estados y comprobamos que no hayamos hecho las cuatro divisiones. Si se han realizado es que el contador $\leftarrow$ 3, por tanto, volvemos al estado inicial.

Tabla 7.13: Tabla de estados del controlador del inversor.

Nombre	Tamaño	Sentido	Comentario
clk	1 bit	Entrada	Señal de reloj
rst	1 bit	Entrada	Reseteo del filtro (reset síncrono a alta)
start	1 bit	Entrada	Señal de comienzo de ejecución del filtro.
new_data	1 bit	Entrada	Indica que hay disponible una nueva medida para incorporar al filtro.
slow	1 bit	Entrada	Indica si la medida actual de entrada corresponde a una medida lenta (slow='1') o no (slow='0').
ready	1 bit	Salida	Indica que el filtro está libre y listo para comenzar
step_done	1 bit	Salida	Indica que el filtro ha acabado la iteración actual y las salidas *glucose* y *gain* están actualizadas
fast_measure	c_sfixed_width	Entrada	Valor externo de medida rápida para incorporar al filtro
slow_measure	c_sfixed_width	Entrada	Valor externo de medida lenta para incorporar al filtro
glucose	c_sfixed_width	Salida	Valor de la glucosa en sangre
gain	c_sfixed_width	Salida	Valor de la ganancia del sensor

Tabla 7.14: Interfaz del módulo TOP_UKF

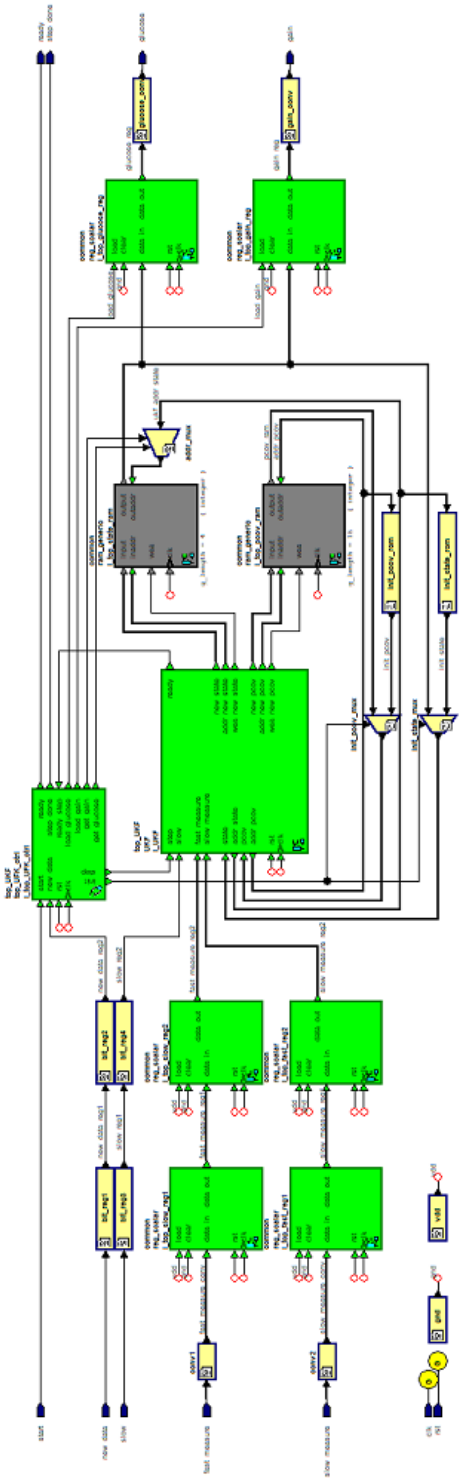

Figura 7.15: Interfaz del filtro, módulo top_UKF

Estado	Acción
ready_st	Notifica que está preparado (ready $\rightarrow$ 1) y permanece a la espera de la señal start $\leftarrow$ 1. Selecciona la entrada 1 en los multiplexores *init_pcov_mux* e *init_state_mux* para realizar la fase de inicialización en la primera iteración
waiting_data_st	Espera a que haya una nueva medida disponible (new_data $\leftarrow$ 1) y a que el filtro esté preparado para comenzar una nueva iteración (ready_step $\leftarrow$ 1)
step_st	Envía la señal *step* $\rightarrow$ 1 al módulo *UKF* para que comience una nueva iteración
wait_st	Espera a que el módulo *UKF* termine la iteración (ready_step $\leftarrow$ 1)
get_glucose_st	Pide a la ram *i_top_state_ram* el valor de la glucosa seleccionando la correspondiente dirección en *addr_mux* mediante la señal *get_gain* $\leftarrow$ 1
load_glucose_st	Guarda el valor de salida de la glucosa en el registro *i_top_glucose_reg* (*load_glucose* $\leftarrow$ 1). Pide a la ram *i_top_state_ram* el valor de la ganancia seleccionando la correspondiente dirección en *addr_mux* mediante la señal *get_gain* $\leftarrow$ 1
load_gain_st	Guarda el valor de salida de la ganancia en el registro *i_top_gain_reg* (*load_gain* $\leftarrow$ 1)
done_st	Notifica al exterior que se ha realizado la iteración (*step_donde* $\rightarrow$ 1). Selecciona la entrada 0 en los multiplexores *init_pcov_mux* e *init_state_mux* para de aquí en adelante realizar las iteraciones con los valores calculados en la anterior iteración

Tabla 7.15: Estados de la unidad de control *i_top_UKF_ctrl*

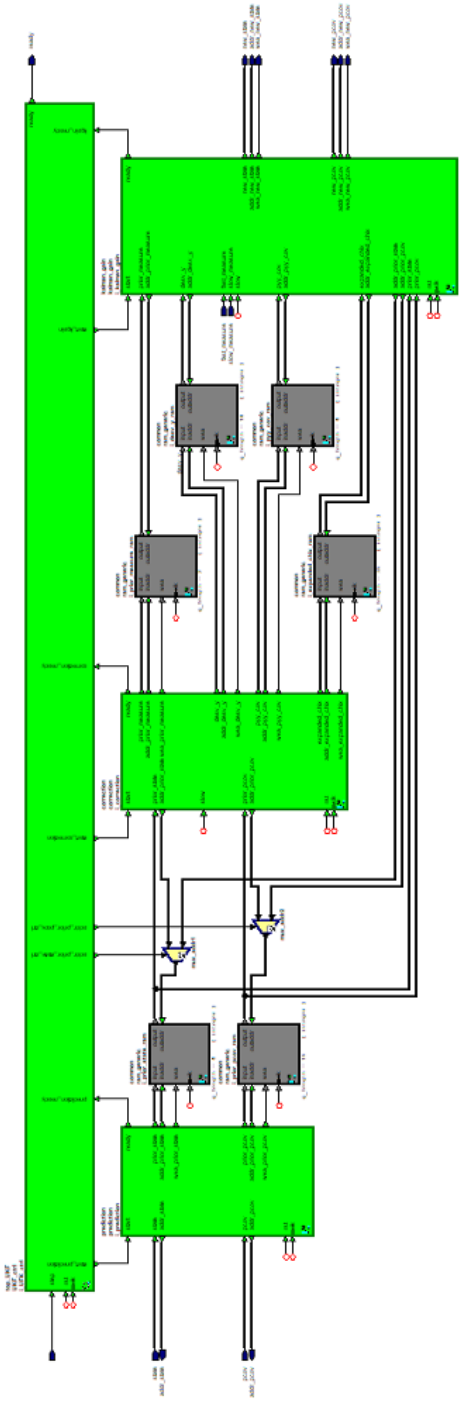

Figura 7.16: Módulo *UKF*

Puerto	Tamaño	Sentido	Comentario
step	1 bit	Entrada	Nueva medida disponible y señal de comienzo del módulo
ready	1 bit	Salida	Indica que el módulo ha terminado de ejecutar una iteración
state	c_sfixed_width	Entrada	Valor del estado calculado en la iteración i-1
addr_state	2 bits	Salida	Direccion para lectura de la RAM que almacena *state*
pcov	c_sfixed_width	Entrada	Valor de la covarianza del estado calculado en la iteración i-1
addr_pcov	4 bits	Salida	Direccion para lectura de la RAM que almacena *pcov*

Tabla 7.16: Interfaz del módulo UKF

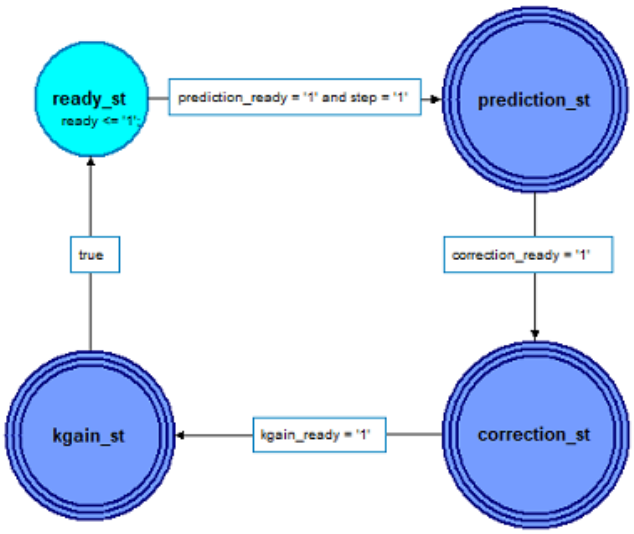

Figura 7.17: Diagrama de estados del controlador *i_UKF_ctrl*

Estado	Acción
ready_st	Notifica que está preparado (ready → 1) y permanece a la espera de la señal step ← 1
prediction_r_st	Envia la señal de comienzo al módulo *Prediction* (start_prediction → 1)
prediction_wait_st	Espera a que el módulo finalice su ejecución (prediction_ready ← 1)
prediction_d_st	Cambia de estado cuando el módulo *Correction* notifica que está preparado (correction_ready ← 1)
correction_r_st	Envia la señal de comienzo al módulo *Correction* (start_correction → 1)
correction_wait_st	Espera a que el módulo finalice su ejecución (correction_ready ← 1)
correction_d_st	Cambia de estado cuando el módulo *Kalman Gain* notifica que está preparado (kgain_ready ← 1)
kgain_r_st	Envia la señal de comienzo al módulo *Kalman Gain* (start_kgain → 1)
kgain_wait_st	Espera a que el módulo finalice su ejecución (correction_ready ← 1)
kgain_d_st	Vuelve al estado *ready*

Tabla 7.17: Estados de la unidad de control *i_UKF_ctrl*

Capítulo 8

Módulo de Predicción

En este módulo se explica la implementación de las ecuaciones correspondientes a la etapa Predicción. Se calculan los puntos sigma de la ecuación (8.1), se aplica el modelo del proceso de la ecuación (8.2), se obtiene la media que representa el estado a priori de la ecuación (8.3) y por último la covarianza a priori de la ecuación (8.4). Este módulo lo hemos dividido en cuatro submódulos, uno por cada ecuación descrita a continuación:

$$\chi_{k-1} = [\hat{x}_{k-1} \qquad \hat{x}_{k-1} + \gamma\sqrt{P_{k-1}} \qquad \hat{x}_{k-1} - \gamma\sqrt{P_{k-1}}] \tag{8.1}$$

$$\chi^*_{k|k-1} = \quad f(\chi_{k-1}, 0) \tag{8.2}$$

$$\hat{x}^-_k = \sum_{i=0}^{2L} W_i^{(m)} \chi^*_{i,k|k-1} \tag{8.3}$$

$$P^-_k = \sum_{i=0}^{2L} W_i^{(c)} (\chi^*_{i,k|k-1} - \hat{x}^-_k)(\chi^*_{i,k|k-1} - \hat{x}^-_k)^T + R^v \tag{8.4}$$

Cada ecuación depende del valor obtenido en la ecuación anterior, por lo que es necesario la implementación de un quinto submódulo, el controlador (*i_prediction_ctrl*) que gestione secuencialmente el inicio de ejecución de cada submódulo mediante el protocolo de *handshake start - ready*. La figura 8.1 representa el diagrama de bloques:

- *i_sigma_pts*: Implementa la ecuación (8.1).
- *i_state_transition*: Implementa la ecuación (8.2).
- *i_state_estimation*: Implementa la ecuación (8.3).
- *i_pcov_estimation*: Implementa la ecuación (8.4).

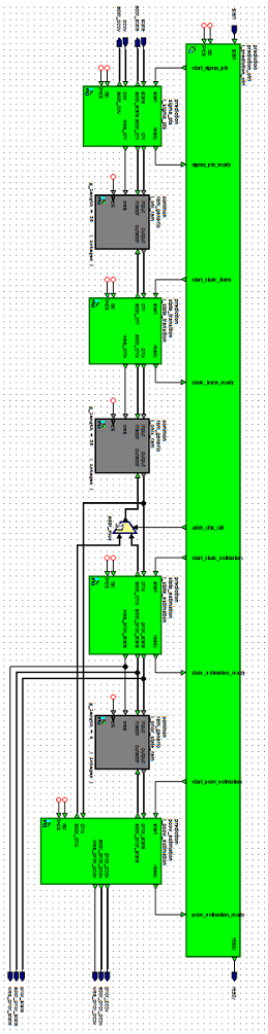

Figura 8.1: *Top-level* del módulo Prediction

La interfaz del módulo *Prediction* está definida en la tabla 8.1.

Nombre	Tamaño	Sentido	Comentario
start	1 bit	Entrada	Indica el comienzo de la etapa predicción (1 ciclo a alta).
state	c_sfixed_width	Entrada	Elemento del estado a posteriori de la anterior iteración.
addr_state	4 pos (2 bits)	Salida	Dirección de la RAM donde se encuentra almacenado el estado a posteriori de la anterior iteración.
pcov	c_sfixed_width	Entrada	Elemento de la convarianza del estado a posteriori de la anterior iteración.
addr_pcov	16 pos (4 bits)	Salida	Dirección de la RAM donde se encuentra almacenado la covarianza del estado a posteriori de la anterior iteración.
ready	1 bit	Salida	Indica si el módulo está disponible para una nueva ejecución.
prior_state	c_sfixed_width	Salida	Elemento de la media que representa el estado a priori.
addr_prior_state	4 pos (2 bits)	Salida	Dirección de la RAM donde se encuentra almacenado el estado a priori.
wea_prior_state	1 bit	Salida	Señal de escritura de la RAM donde se encuentra almacenado el estado a priori.
prior_pcov	c_sfixed_width	Salida	Elemento de la covarianza del estado a priori.
addr_prior_pcov	16 pos (4 bits)	Salida	Dirección de la RAM donde se encuentra almacenado la covarianza del estado a priori.
wea_prior_pcov	1 bit	Salida	Señal de escritura de la RAM donde se encuentra almacenado la covarianza del estado a priori.

Tabla 8.1: Interfaz del módulo PREDICTION

Los submódulos *i_state_estimation* y *i_pcov_estimation* necesitan leer elementos de la ram *i_chix_ram* para realizar sus operaciones, para ello, se añade un multiplexor addr_mux y la señal de control addr_chix_ctrl en la dirección de lectura outaddr de la RAM.

Funcionamiento del módulo

El módulo indica al módulo superior con la señal de *ready* que está listo para una nueva iteración. Cuando recibe la señal de *start*, realiza los siguientes pasos secuencialmente, es decir, el submódulo *i_prediction_ctrl* activa las señales de start de los submódulos y no se inicia el siguiente paso hasta que el submódulo haya indicado la finalización de la operación mediante su señal de ready:

1. Activa la señal de start del submódulo *i_sigma_pts* para el cálculo de los puntos sigma que son almacenados en la RAM *i_chi_ram*. Esta RAM consta de 36 palabras correspondientes a una matriz de dimensiones 4x9.

2. Activa la señal de start del submódulo *i_state_transition*. Este submódulo lee los puntos sigma de la RAM *i_chi_ram*, aplica el modelo de proceso sobre ellos y almacena el resultado en la RAM *i_chix_ram*. Esta RAM consta de 36 posiciones correspondientes a una matriz de dimensiones 4x9.

3. Activa la señal de start del submódulo *i_state_estimation*. Este submódulo lee los puntos sigma calculados en el paso 2 de la RAM *i_chix_ram*, calcula la media que representará el estado a priori y almacena el resultado en la RAM *i_prior_state_ram*. Esta RAM consta de 4 posiciones correspondientes a la matriz de estado de dimensiones 4x1.

4. Activa la señal de start del submódulo *i_pcov_estimation*. Este submódulo necesita los puntos sigma del paso 2 y el estado a priori del paso 3 para relizar el cálculo de la covarianza a priori del estado.

El estado a priori y la covarianza a priori son almacenados en RAMs externas al módulo: *i_prior_state_ram* y *i_prior_pcov_ram* para su uso en el módulo *i_correction*.

8.1. Cálculo de puntos sigma: Módulo i_sigma_pts

Implementa la expresión 8.1, que realiza el cálculo de los puntos sigma a partir del estado y covarianza a posteriori de la iteración anterior y el parámetro Gamma. Este módulo está dividido en varios submódulos comunes explicados en la sección 7.3, además de un sexto submódulo, el controlador *i_fsm_sigma* encargado de secuenciar el inicio de ejecución de cada submódulo:

- *i_cholesky_sigma*(cholesky4x4): Implementa la factorización de Cholesky de una matriz de dimensiones 4x4.

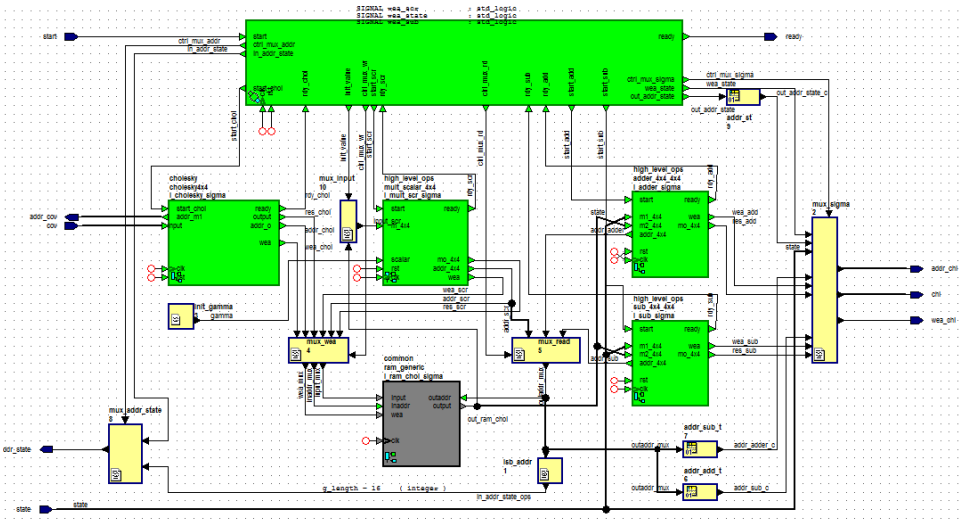

Figura 8.2: *Diagrama de bloques: Módulo i_sigma_pts*

- *i_mult_scr_sigma*(mult_scalar_4x4): Implementa la multiplicación de un escalar por todos los elementos de una matriz de dimensiones 4x4.
- *i_adder_sigma*(adder_4x4_4x4): Implementa la suma de matrices de dimensiones 4x4.
- *i_sub_sigma*(sub_4x4_4x4): Implementa la resta de matrices de dimensiones 4x4.
- *i_ram_chol_sigma*(ram_generic): Implementa una BRAM de 16 posiciones correspondientes una matriz de dimensiones 4x4. Son 16 posiciones para almacenar el resultado del submódulo *i_cholesky_sigma* y *i_mult_scr_sigma*.

La figura 8.2 representa el diagrama de bloques.

Los submódulos *i_cholesky_sigma* y *i_mult_scr_sigma* almacenan elementos en la RAM *i_ram_chol_sigma*. Para que sea posible se añade el multiplexor mux_wea y la señal de control ctrl_mux_wr en la dirección de escritura. Lo mismo sucede para la dirección de lectura en la que acceden los submódulos *i_adder_sigma* y *i_sub_sigma*. En este caso, se añade el multiplexor mux_read y la señal de control ctrl_mux_rd.

El resultado de calcular los puntos sigma es almacenado en una RAM externa *i_chi_ram*. Los puntos sigma son generados desde distintos submódulos (*i_adder_sigma*, *i_sub_sigma* y *i_fsm_sigma*). Para que todos ellos puedan almacenar elementos en la RAM se añade el multiplexor mux_sigma y la señal de control ctrl_mux_sigma en la dirección de escritura.

Cuando recibe la señal de *start*, realiza los siguientes pasos secuencialmente, no iniciando un nuevo paso hasta que no haya finalizado el anterior. Excepto el *paso 1*, sabemos que una operación ha finalizado cuando la señal de ready del submódulo pasa del nivel bajo a alto:

1. El controlador genera las direcciones y señales de escritura para almacenar el primer punto sigma en la RAM *i_chi_ram*. Este punto sigma es el estado a posteriori de la iteración anterior.

2. Almacena ceros en todas las posiciones de la RAM *i_ram_chol* porque el submódulo *i_cholesky_sigma* devuelve una matriz diagonal y no accede a todas las posiciones de la RAM. Para ello, el controlador activa la señal de start del submódulo *i_mult_scr_sigma* con ceros en la entrada para que el resultado de la multiplicación sea ceros.

3. El controlador activa la señal de start del submódulo *i_cholesky_sigma* que lleva a cabo la factorización de Cholesky sobre la covarianza del estado a posteriori. El resultado se almacena en la RAM *i_ram_chol*.

4. El controlador activa la señal de start del submódulo *i_mult_scr_sigma* que realiza la multiplicación de los datos almacenados en la RAM *i_ram_chol* por el parámtro *Gamma*.

5. El controlador activa la señal de start del submódulo *i_adder_sigma* para almacenar cuatro puntos sigma en la RAM *i_chi_ram* creados a partir de sumar lo almacenado en la RAM *i_ram_chol* y el estado a posteriori de la iteración anterior.

6. Almacena los cuatro últimos puntos sigma en la RAM *i_chi_ram* a partir de restar lo almacenado en la RAM *i_ram_chol* y el estado a posteriori de la iteración anterior. Para ello, habilita la señal de start del submódulo *i_sub_sigma*.

i_fsm_sigma

Implementa una máquina de estados encargada de secuenciar las operaciones del módulo. En la tabla 8.2 se detallan los estados correspondientes al diagrama de la figura 8.3.

Los controladores suelen tener estados jerárquicos que agrupan más de un estado, en concreto la mayoría de las veces son dos subestados para cumplir con el protocolo de comunicación entre submódulos 7.1.2:

- Subestado *_ready_st: Activa la señal *start* de un submódulo, esto produce que la señal *ready* $\rightarrow$ 0 y cambia al estado *wait_st*.

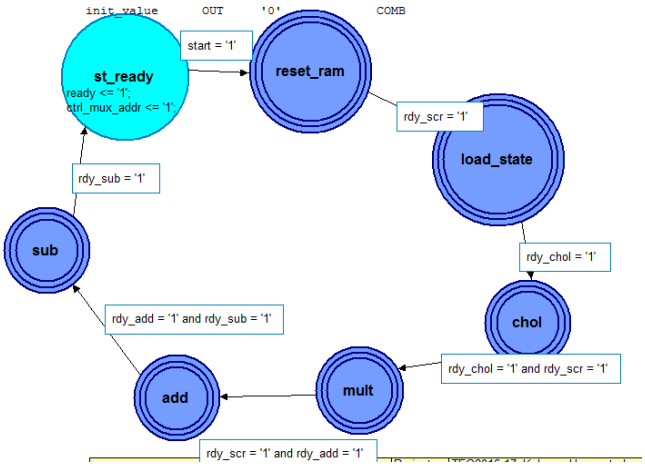

Figura 8.3: *Unidad de control: Módulo i_sigma_pts*

- Subestado *_wait_st: Se mantiene en esta estado hasta que el submódulo finalice la operación, es decir, señal de ready → 1.

Los estados (reset_ram, chol, mult, add, sub) son estados jerárquicos que siguen la misma estructura ya explicada 8.1. El estado jerárquico load_state tiene una estructura diferente y está formado por cuatro estados (*st_1, st_2, st_3, st_4*) encargados de almacenar el primer punto sigma en la RAM externa *i_chi_ram*.

Estado	Acción
st_ready	Mantiene la señal de ready → 1 que indica que el módulo está preparado para un nuevo cálculo y cambia al estado st_rst cuando la señal start ← 1.
st_rst	Activa la señal start_scr → 1 del submódulo i_mult_scr_sigma, la señal de control del multiplexor mux_wea(ctrl_mux_wr → 1), la señal de control del multiplexor mux_input(init_value → 1) para almacenar ceros en la RAM i_ram_chol_sigma y cambia al estado wait_rst.
wait_rst	Se mantiene en este estado hasta que el submódulo i_mult_scr_sigma finalice la operación (rdy_scr ← 1). Una vez finalizada cambia al estado st_1.
st_1	Genera las señales de dirección y capacitación de escritura (out_addr_state → 0 y wea → 1) para almacenar el elemento 1 del estado a posteriori de la iteración anterior en la RAM externa de puntos sigma y cambia al estado st_2.
st_2	Genera las señales de dirección y capacitación de escritura (out_addr_state → 1 y wea → 1) para almacenar el elemento 2 del estado a posteriori de la iteración anterior en la RAM externa de puntos sigma y cambia al estado st_3.
st_3	Genera las señales de dirección y capacitación de escritura (out_addr_state → 2 y wea → 1) para almacenar el elemento 3 del estado a posteriori de la iteración anterior en la RAM externa de puntos sigma y cambia al estado st_4.
st_4	Genera las señales de dirección y capacitación de escritura (out_addr_state → 3 y wea → 1) para almacenar el elemento 4 del estado a posteriori de la iteración anterior en la RAM externa de puntos sigma y cambia al estado st_chol.
st_chol	Activa la señal start_chol → 1 del submódulo i_cholesky_sigma para calcular la factorización de Cholesky y cambia al estado wait_chol.

wait_chol	Se mantiene en este estado hasta que el submódulo i_cholesky_sigma finalice la operación (rdy_chol ← 1). Una vez finalizada cambia al estado st_mult.
st_mult	Activa la señal start_scr → 1 del submódulo i_mult_scr_sigma para realizar la multiplicación de los elementos de la RAM i_ram_chol_sigma por el parámetro *Gamma* y cambia al estado wait_mult.
wait_mult	Se mantiene en este estado hasta que el submódulo i_mult_scr_sigma finalice la operación (rdy_scr ← 1) y selecciona la señal de control (ctrl_mux_wr → 1) del multiplexor i_mux_wea para permitir que el multiplicador tenga acceso a la dirección de escritura de la RAM i_ram_chol_sigma. Una vez finalizada cambia al estado st_add.
st_add	Activa la señal start_add → 1 del submódulo i_adder_sigma para realizar la suma de los elementos almacenados en la RAM i_ram_chol_sigma y el estado a posteriori de la iteración anterior y cambia al estado wait_add.
wait_add	Se mantiene en este estado hasta que el submódulo finalice la operación (rdy_add ← 1), selecciona la señal de control (ctrl_mux_rd → '01') del multiplexor mux_read para permitir que el sumador tenga acceso a la dirección de lectura de la RAM i_ram_chol_sigma y selecciona la señal de control (ctrl_mux_sigma → '01') del multiplexor mux_sigma para almacenar los puntos sigma en la RAM exterior. Una vez finalizada la operación cambia al estado st_sub.
st_sub	Activa la señal start_sub → 1 del submódulo i_sub_sigma para realizar la resta de los elementos almacenados en la RAM i_ram_chol_sigma y el estado a posteriori de la iteración anterior y cambia al estado wait_sub.

wait_sub	Se mantiene en este estado hasta que el submódulo i_sub_sigma finalice la operación (rdy_sub ← 1), selecciona la señal de control (ctrl_mux_rd → '10') del multiplexor mux_read para permitir que el restador tenga acceso a la dirección de lectura a la RAM i_ram_chol_sigma y seleccinoa la señal de control (ctrl_mux_sigma → '10') del multiplexor mux_sigma para almacenar los puntos sigma en la RAM exterior. Una vez finalizada la operación cambia al estado st_ready.

Tabla 8.2: Estados del controlador

8.2. Transición del estado: Módulo i_state_transition

Este módulo implementa la expresión 8.2, que aplica el modelo del proceso sobre los puntos sigma calculados en la sección 8.1. Para ello multiplica la matriz A que representa el modelo del proceso por los puntos sigma mencionados anteriormente. Este módulo se divide en tres submódulos comunes explicados en la sección 7.3, además de un cuarto submódulo, el controlador *i_fsm_st_transition* encargado de secuenciar el inicio de ejecución de cada submódulo:

- *i_counter_st*(ascending_counter_generic): Implementa un contador ascendente módulo 36.
- *i_mult_1x4_4x1_st*(mult_1x4_4x1): Implementa una multiplicación de matrices de dimensiones 1x4 y 4x1 respectivamente.
- *i_ram_sigma_st*(ram_generic): Implementa una BRAM de 4 posiciones correspondientes a una matriz de dimensiones 4x1. Son 4 posiciones porque necesitamos almacenar los elementos que forman un punto sigma.

La figura 8.4 representa el diagrama de bloques de este módulo.

La matriz A del modelo del proceso se almacena como una memoria lineal en una ROM ya que es una constante definida en los parámetros del filtro. Utilizamos un contador para generar las direcciones y acceder a las RAMS que utiliza este módulo (*i_ram_sigma_st* y las RAMs externas *i_chi_ram* y *i_chix_ram*). Pero es necesario añadir lógica extra a la salida del contador *i_counter_st* para modificar la forma de acceder acorde a la RAM que utilicemos. Hay tres bloques que añaden esta lógica:

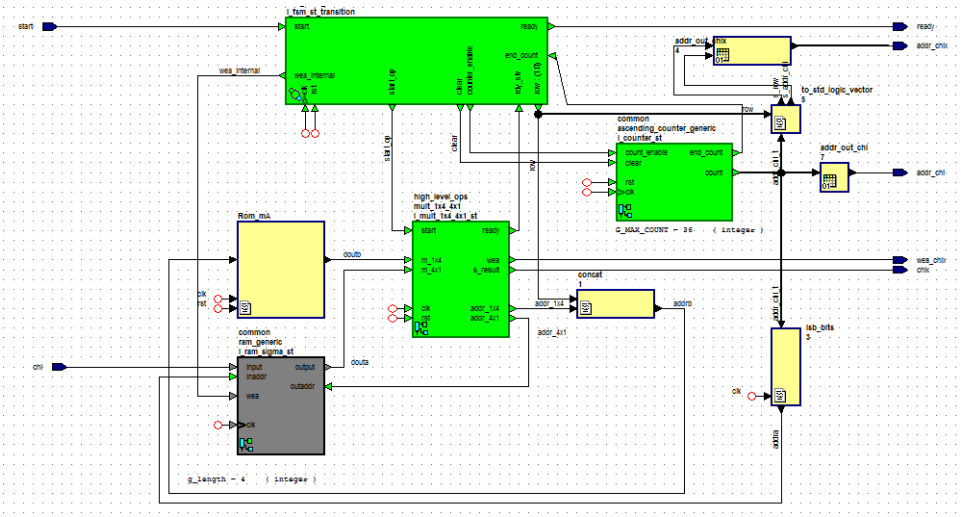

Figura 8.4: *Diagrama de bloques: Módulo i_state_transition*

- Bloque *lsb_bits*: Selecciona los dos bits menos significativos del contador para indexar la RAM *i_ram_sigma_st* de 4 posiciones.

- Bloque *addr_out_chi*: Genera la traspuesta del contador ya que los puntos sigma (almacenados en la RAM externa *i_chi_ram*) se almacenan por columnas. Para ello, implementa una tabla de verdad para realizar la conversión (Figura 8.3).

- Bloque *addr_out_chix*: Este bloque implementa una tabla verdad para saber donde almacenar el resultado de la multiplicación en la RAM externa *i_chix_ram*. Para ello, junto a la salida del contador, utiliza una señal s_row que nos indica que fila de la matriz A estamos procesando. La señal s_row también se utiliza en el bloque *concar* para acceder a la ROM que almacena la matriz A.

Entrada	Salida
0	0
1	9
2	18
3	27
4	1
5	10
6	19
7	28

Tabla 8.3: Conversión

Cuando recibe la señal de *start* realiza los siguientes pasos secuencialmente, no iniciando el siguiente paso hasta que no haya finalizado el anterior:

1. El controlador activa la capacitación de cuenta del contador *i_counter_st* que genera cuatro direcciones consecutivas para leer un punto sigma de la RAM *i_chi_ram*. Los elementos leídos son almacenados en la RAM *i_ram_sigma* habilitando la señal de capacitación de escritura.

2. A continuación realiza cuatro multiplicaciones 1x4-4x1 correspondientes a multiplicar todas las filas de la matriz A (almacenada en la ROM *Rom_mA*) y el punto sigma almacenado en la RAM *i_ram_sigma*. Para ello, el controlador activa la señal de start del multiplicador *i_mult_1x4_4x1_st* que almacena el resultado en la RAM externa *i_chix_ram*.

Repite los pasos anteriores con el resto de puntos sigma o lo que es lo mismo hasta que el contador *i_counter_st* active la señal de fin de cuenta, que significa que ya no hay más direcciones por generar.

i_fsm_st_transition

Implementa una máquina de estados encargada de secuenciar las operaciones del módulo. En la tabla 8.4 se detallan los estados correspondientes al diagrama de la figura 8.5.

El estado jerárquico *load_chi* se encarga de almacenar un punto sigma de cuatro elementos en la RAM *i_ram_sigma_st*. Está formado por un estado *wait* de 3 ciclos y el estado last_load. Los dos estados tienen habilitada la capacitación de escritura de la RAM *i_ram_sigma_st* y se diferencian en que sólo el estado *wait* habilita la capacitación de cuenta del contador *i_counter_st* para generar las direcciones. Como la RAM tiene un ciclo de

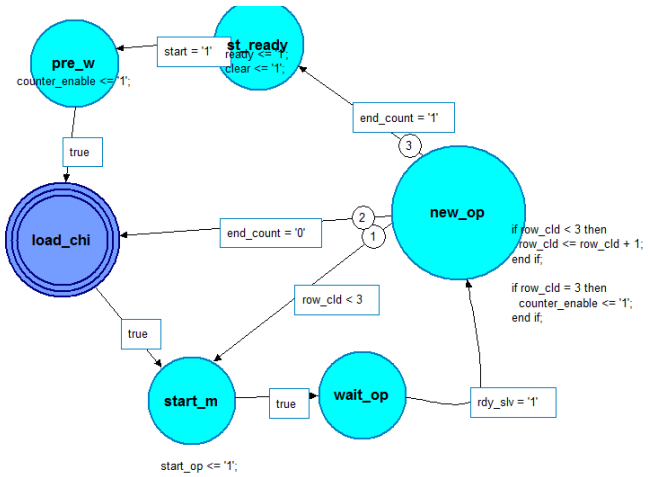

Figura 8.5: *Unidad de control: Módulo i_state_transition*

latencia en la lectura, la capacitación de cuenta del contador se desactiva un ciclo antes para no leer un dato de más.

Estado	Acción
ready_st	Mantiene la señal de ready → 1 que indica que el módulo está preparado para un nuevo cálculo y la señal clear → 1 para resetear el contador i_counter_st. Cambia al estado pre_w cuando la señal start ← 1.
pre_w	Se activa la señal de capacitación de cuenta counter_enable → 1 del contador i_counter_st. Es importante habilitar el contador un estado antes de que se escriban los puntos sigma debido a la latencia de un ciclo en la lectura de la RAM. Cambia al estado load.
load	Estado de espera de tres ciclos en el que se mantiene la señal de capacitación de cuenta activada (counter_enable → 1). Habilita la señal de capacitación de escritura de la RAM i_ram_sigma_st (wea_internal → 1) para almacenar tres elementos de un punto sigma. Cambia al estado last_load una vez transcurridos los tres ciclos.
last_load	Se deshabilita la cuenta del contador (counter_enable → 0) y se realiza la escritura del último elemento del punto sigma (wea_internal → 1). Cambia al estado start_m.
start_m	Se activa la señal de start_op del submódulo i_mult_1x4_4x1 para realizar la multiplicación de una fila de la matriz A y el punto sigma almacenado en la RAM i_ram_sigma_st y cambia al estado wait_op.
wait_op	Se mantiene en este estado hasta que el submódulo i_mult_1x4_4x1 finalice la operación (rdy_slv ← 1). Una vez finalizada, cambia el estado new_op.
new_op	Actualiza la señal s_row para procesar la siguiente fila de la matriz A, y chequea si el contador ha llegado a fin de cuenta (end_count ← 1). Si end_count ← 1 cambia al estado ready_st. Cambia al estado start_m si todavía quedan filas de la matriz A por procesar. Y si ya terminó la multiplicación de todas las filas de A por el punto sigma almacenado en la RAM i_ram_sigma_st, cambia al estado load_chi para almacenar un nuevo punto sigma en la RAM y repetir el mismo proceso.

Tabla 8.4: Estados del controlador

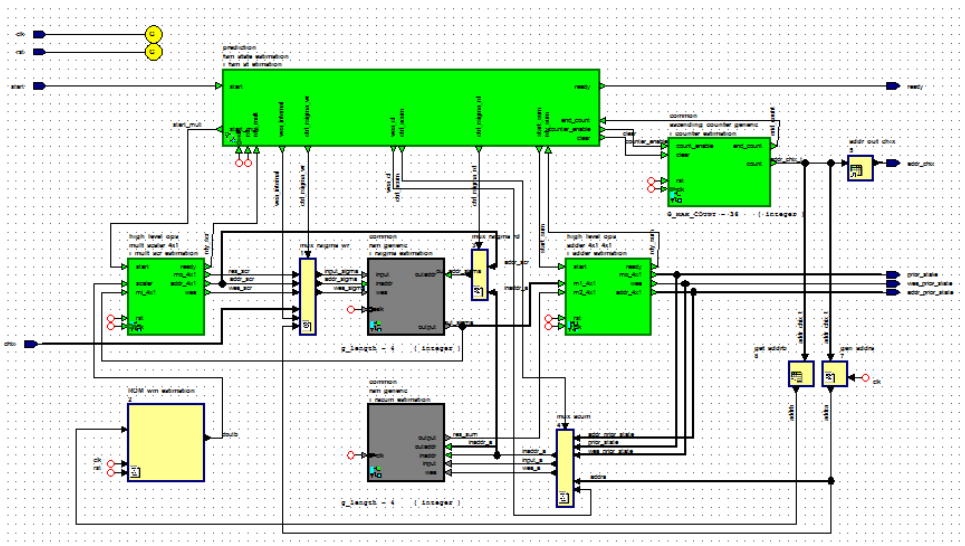

Figura 8.6: *Diagrama de bloques: Módulo i_state_estimation*

8.3. Cálculo del estado a priori: i_state_estimation

Este módulo implementa la expresión 8.3, que calcula la media correspondiente al estado a priori a partir de los puntos sigma calculados en la sección 8.2 y los pesos de la media. Se compone de cinco submódulos comunes 7.3 para realizar las operaciones y un sexto, el controlador *i_fsm_st_estimation* encargado de secuenciar el inicio de ejecución de cada submódulo:

- *i_mult_scr_estimation*(mult_scalar_4x1): Implementa la multiplicación de un escalar por todos los elementos de una matriz de dimensiones 4x1.
- *i_adder_estimation*(adder_4x1_4x1): Implementa la suma de dos matrices de dimensiones 4x1.
- *i_counter_estimation*(ascending_counter_generic): Implementa un contador ascendente módulo 36.
- *i_rsigma_estimation*(ram_generic): Implementa una BRAM de cuatro posiciones.
- *i_racum_estimation*(ram_generic): Implementa una BRAM de cuatro posiciones.

La figura 8.6 representa el diagrama de bloques de este módulo.

Los pesos de la media (matriz 1x9) es una constante que se representa como una memoria lineal en una ROM de nueve posiciones. El contador *i_ counter_ estimation* se utiliza para generar direcciones, las cuales modificadas con bloques de lógica serán utilizadas para acceder a las diferentes RAMs del diseño: la RAM externa *i_ chix_ ram* que almacena los puntos sigma calculados en la sección 8.2, la RAM *i_ rsigma_ estimation* utilizada para almacenar un punto sigma y la ROM *ROM_ wm_ estimation* que almacena los pesos de la media.

Los bloques que modifican la salida del contador *i_ counter_ estimation* son los siguientes:

- Bloque gen_addra: Selecciona los dos bits menos significativos del contador para acceder a la RAM *i_ rsigma_ estimation* de cuatro posiciones.
- Bloque gen_addrb: La salida del contador(36) es más grande que las posiciones de la ROM *ROM_ wm_ estimation* (9). El bloque implementa una tabla de verdad para solucionar esta diferencia, de forma que se modifica la dirección de la ROM cada cuatro direcciones generadas por el contador, en otras palabras, se elige un nuevo peso por cada punto sigma que se procesa, que está formado por cuatro elementos. La tabla 8.5 muestra el contenido de la tabla de verdad.
- Bloque addr_chix: Implementa una tabla de verdad que genera la traspuesta de la salida del contador. Ya explicado en el módulo 8.3.

Entrada	Salida
4	0
8	1
12	2
16	3
20	4
24	5
28	6
32	7
35	8

Tabla 8.5: Conversión

Este módulo ha sido diseñado para minimizar los recursos que utiliza pero tiene la consecuencia que necesita varios multiplexores en la dirección de escritura y lectura de las RAMs. *El porqué los submódulos necesitan tener acceso a estas RAMs se explicará en el funcionamiento del módulo.*

- Multiplexor mux_rsigma_wr: La señal ctrl_rsigma_wr selecciona la dirección de escritura de la RAM *i_rsigma_estimation* entre el controlador *i_fsm_st_estimation* y el submódulo *i_mult_scr_estimation*.
- Multiplexor mux_rsigma_rd: La señal ctrl_rsigma_rd selecciona la dirección de lectura de la RAM *i_rsigma_estimation* entre el submódulo *i_mult_scr_estimation* y el submódulo *i_adder_estimation*.
- Multiplexor mux_acum: La señal ctrl_acum_ctrl selecciona la dirección de escritura de la RAM *i_racum_estimation* entre el submódulo *i_adder_estimation* y el controlador *i_fsm_st_estimation*.

Cuando el módulo recibe la señal de *start* realiza los siguientes pasos secuencialmente, iniciando un nuevo paso una vez haya finalizado el anterior:

1. Almacena ceros en todas las posiciones de la RAM *i_racum_sigma*, RAM utilizada para realizar el sumatorio, que implica inicializarse a cero en cada nueva iteración. El controlador *i_fsm_estimation* activa la señal de capacitación del contador para generar cuatro direcciones y la señal de capacitación de escritura de la RAM.
2. El controlador activa la capacitación de cuenta del contador para generar cuatro nuevas direcciones, necesarias para leer un punto sigma (que está formado por 4 elementos) y almacenarlo en la RAM *i_rsigma_estimation*. Para ello, habilita la señal de capacitación de escritura de la RAM.
3. Multiplica el punto sigma almacenado en la RAM *i_rsigma_estimation* por el peso que le corresponde almacenado en la ROM *ROM_wm_estimation*. El controlador activa la señal de start del submódulo *i_mult_scr_estimation*, que almacena el resultado en la misma RAM.
4. El sumador *i_adder_estimation* almacena en la RAM *i_racum_estimation* el sumatorio de todos los resultados del *paso 3*. El controlado activa la señal de start del submódulo *i_adder_estimation*.

Se repiten los pasos mientras queden puntos sigma por procesar.

i_fsm_st_estimation

Implementa una máquina de estados encargada de secuenciar las operaciones del módulo. En la tabla 8.6 se detallan los estados correspondientes al diagrama de la figura 8.7.

La máquina de estados se divide en varios estados jerárquicos. Los estados *mult* y *sum* siguen la estructura de estado jerárquico explicada anteriormente 8.1. Por otra parte los estados rst_acum y load_s son algo diferenes:

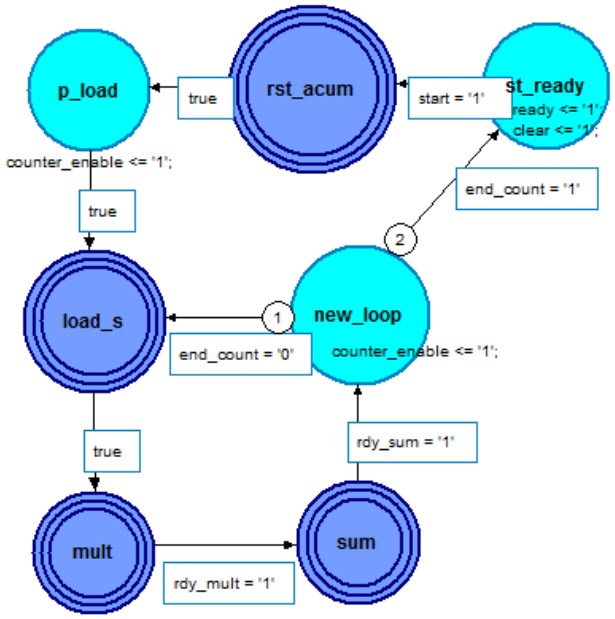

Figura 8.7: *Unidad de control: Módulo i_state_estimation*

- Reset_acum se encarga de almacenar ceros en la RAM i_racum_sigma.
- Load_s almacena un punto sigma en la RAM i_rsigma_estimation.

Estado	Acción
st_ready	Mantiene la señal de ready → 1 que indica que el módulo está preparado para un nuevo cálculo y la señal de clear del contador i_counter_estimation → 1 para mantener la cuenta a cero. Cambia al estado cleanup cuando la señal start ← 1.
cleanup	Selecciona la entrada del multiplexor mux_acum (ctrl_acum → 1) y la señal de capacitación de escritura wea_cl → 1 de la RAM i_racum_estimation para almacenar ceros en todas las posiciones. Para ello, habilita la señal de capacitación del contador counter_enable → 1 para generar las direcciones. Una vez generadas las cuatro direcciones, cambia al estado rst_counter.

rst_counter	Activa la señal clear → 1 del contador i_counter_estimation para reinciar la cuenta. Cambia al estado p_load.
p_load	Activa la señal de capacitación del contador i_counter_estimation (counter_enable → 1) un estado antes de almacenar los elementos de un punto sigma debido al ciclo de latencia con la RAM. Cambia al estado load.
load	Estado de espera de tres ciclos en el que se mantiene la señal de capacitación del contador activada (counter_enable → 1). Durante los tres ciclos se almacenan tres elementos de un punto sigma en la RAM i_rsigma_estimation, para ello, habilita la señal de capacitación de escritura wea_internal → 1 y la señal de control del multiplexor mux_rsigma_wr (ctrl_rsigma_wr → 1). Pasados los tres ciclos, cambia al estado last_load.
last_load	Se deshabilita la capacitación de cuenta del contador (counter_enable → 0) y realiza la escritura del último elemento del punto sigma (wea_internal → 1) en la RAM i_rsigma_estimation. Se mantienen las señales del multiplexor del estado load. Cambia al estado st_mult.
st_mult	Activa la señal start_mult → 1 del submódulo i_mult_scr_estimation y cambia al estado wait_mult.
wait_mult	Se mantiene en este estado hasta que el submódulo i_mult_scr_estimation finalice la operación (rdy_mult ← 1). Selecciona la entrada del multiplexor mux_rsigma_rd (ctrl_rsigma_rd → 1) para que el multiplicador pueda leer de la RAM i_rsigma_estimation. Una vez finalizada la operación, cambia al estado st_sum.
st_sum	Activa la señal start_sum → 1 del submódulo i_adder_estimation y cambia al estado wait_sum.
wait_sum	Se mantiene en este estado hasta que el submódulo finalice la operación (rdy_sum ← 1). Una vez finalizada, cambia al estado new_loop.

new_loop	Prepara una nueva iteración si el fin de cuenta del contador (end_count ← 0) y habilita la señal de capacitación counter_enable → 1 del contador. Si se activó la señal de fin de cuenta, cambia al estado st_ready, por el contrario cambia al estado load.

Tabla 8.6: Estados del controlador

8.4. Cálculo de la covarianza a priori: i_pcov_estimation

Este módulo implementa la expresión 8.4, que realiza el cálculo de la covarianza del estado a priori a partir de los puntos sigma calculados en la sección 8.2, la media correspondiente al estado a priori de la sección 8.3, los pesos de la covarianza y la covarianza del error del proceso. Se compone de seis submódulos comunes 7.3 para realizar las operaciones, además de un séptimo submódulo, el controlador *i_fsm_pcov_est* encargado de secuenciar el inicio de ejecución de cada submódulo:

- *i_sub_pcov_est*(sub_4x9_4x1): Implementa una resta de matrices de dimensiones 4x9 y 4x1 respectivamente.
- *i_desv_x_ram*(ram_generic): Implementa una memoria BRAM de 36 posiciones.
- *i_scalar_pcov_est*(mult_scalar_1x9_4x9): Implementa una multiplicación de un escalar por los elementos de un vector columna de la matriz 4x9. Se repite nueve veces.
- *i_desv_weight_ram*(ram_generic): Implementa una memoria BRAM de 36 posiciones.
- *i_mult_pcov_est*(mult_4x9_9x4): Implementa una multiplicación de matrices de dimensiones 4x9 y 9x4 respectivamente.
- *i_add_pcov_est*(adder_4x4_4x4): Implementa la suma de matrices de dimensiones 4x4.

La figura 8.8 representa el diagrama de bloques de este módulo.

Los pesos de la covarianza y la covarianza del error del proceso son almacenados como una memoria linea en las ROMs *c_weights* y *c_rv* respectivamente. Los submódulos *i_scalar_pcov_est* realiza lecturas de la RAM *i_desv_x_ram*, al igual que el submódulo *i_mult_pcov_est*, por lo tanto es

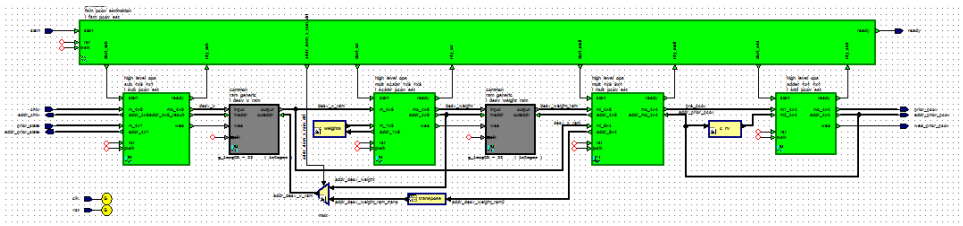

Figura 8.8: *Diagrama de bloques: Módulo i_pcov_estimation*

necesario añadir un multiplexor *mux* y la señal de control addr_desv_x_ram_ctrl en la dirección de lectura de la RAM.

Cuando recibe la señal de *start*, realiza los siguientes pasos de forma secuencial, iniciando un nuevo paso una vez finalizado el anterior:

1. El controlador activa la señal start del submódulo *i_sub_pcov_est* encargado de realizar la resta de los puntos sigma y el estado a priori. El resultado de la resta es almacenado en la RAM *i_desv_x_ram*.

2. El controlador activa la señal de start del submódulo *i_scalar_pcov_est* que realiza la multiplicación de los pesos de la covarianza y los elementos almacenados en la RAM *i_desv_x_ram*. El resultado de la multiplicación es almacenado en la RAM *i_des_weight_ram*.

3. El controlador activa la señal de start del submódulo *i_mult_pcov_est* que multiplica los elementos de la RAM *i_des_weight_ram* por su traspuesta. Para ello utiliza el bloque *transpose* que implementa una tabla de verdad para modificar la forma de acceder a la dirección de lectura de la RAM *i_des_weight_ram* y obtener la traspuesta de los elementos almacenados.

4. Por último, al resultado obtenido en el *paso 3* se le suma la covarianza del ruido del proceso. El controlador activa la señal de start del submódulo *i_add_pcov_est* que genera la dirección y capacitación de escritura para guardar el resultado en la RAM externa *i_prior_pcov_ram*.

i_fsm_pcov_est

Implementa una máquina de estados encargada de secuenciar las operaciones del módulo. En la tabla 8.7 se detallan los estados correspondientes al diagrama de la figura 8.9.

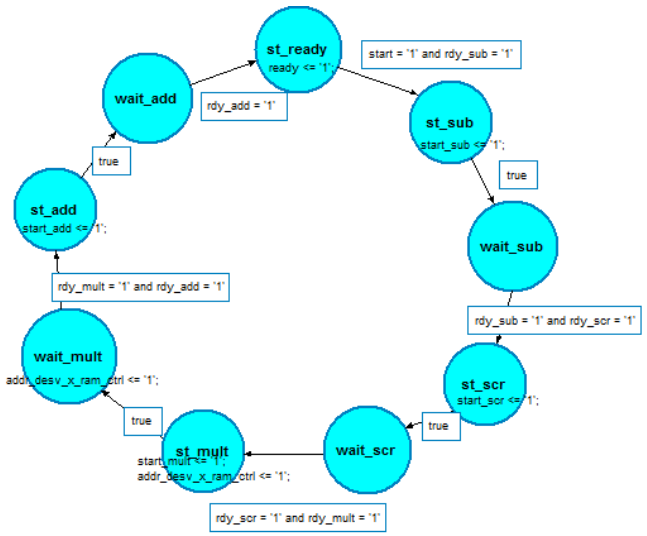

Figura 8.9: *Unidad de control: Módulo i_pcov_estimation*

Estado	Acción
st_ready	Mantiene la señal de ready → 1 que indica que el módulo está preparado para un nuevo cálculo y cambia al estado st_sub cuando la señal start ← 1.
st_sub	Activa la señal start_sub → 1 del submódulo i_sub_pcov_est y cambia al estado wait_sub.
wait_sub	Se mantiene en este estado hasta que el submódulo i_sub_pcov_est finalice la operación (rdy_sub ← 1). Una vez finalizada, cambia al estado st_scr.
st_scr	Activa la señal start_scr → 1 del submódulo i_scalar_pcov_est y cambia al estado wait_scr.
wait_scr	Se mantiene en este estado hasta que el submódulo i_scalar_pcov_est finalice la operación (rdy_scr ← 1). Una vez finalizada, cambia al estado st_mult.
st_mult	Activa la señal start_mult → 1 del submódulo i_mult_pcov_est y cambia al estado wait_mul.
wait_mult	Se mantiene en este estado hasta que el submódulo i_mult_pcov_est finalice la operación (rdy_mult ← 1), selecciona la señal de control del multiplexor *mux*(addr_desv_ram_ctrl → 1) para que el módulo pueda leer de la RAM i_desv_x_ram. Una vez finalizada, cambia al estado st_add.
st_add	Activa la señal start_add → 1 del submódulo i_add_pcov_est y cambia al estado wait_add.
wait_add	Se mantiene en este estado hasta que el submódulo i_add_pcov_est finalice la operación (rdy_sum ← 1). Una vez finalizada, cambia al estado st_ready.

Tabla 8.7: Estados del controlador

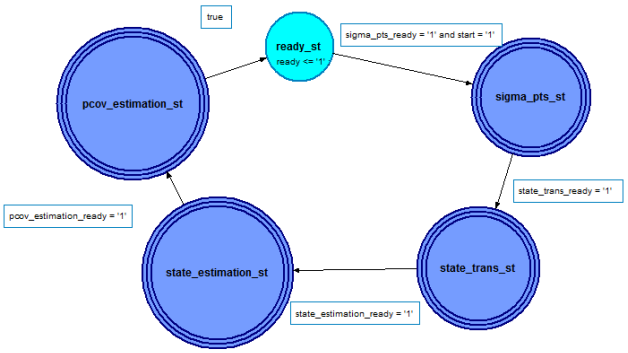

Figura 8.10: *Unidad de Control* del módulo PREDICTION

8.5. Unidad de Control: i_prediction_ctrl

La instancia implementa una máquina de estados para controlar la secuencia de operaciones del módulo Prediction. En la tabla 8.8 se detallan los estados correspondientes al diagrama de la figura 8.10.

Los estados del controlador (sigma_pts_st, state_trans_st, state_estimation_st, pcov_estimation_st) son estados jerárquicos que siguen la misma estructura explicada anteriormente en la unidad de control 8.1.

Estado	Acción
ready_st	Mantiene la señal ready → 1 que indica que el módulo está preparado para una nueva iteración y cambia al estado sigma_pts_ready_st cuando start ← 1.
sigma_pts_ready_st	Activa la señal start_sigma_pts → 1 del submódulo i_sigma_pts y cambia al estado sigma_pts_wait_st.
sigma_pts_wait_st	Se mantiene en este estado hasta que el submódulo i_sigma_pts finalice la operación (sigma_pts_ready ← 1). Una vez finalizada, cambia al estado state_trans_ready_st.
state_trans_ready_st	Activa la señal start_state_trans → 1 del submódulo i_state_transition y cambia al estado state_trans_wait_st.
state_trans_wait_st	Se mantiene en este estado hasta que el submódulo i_state_transition finalice la operación (state_trans_ready ← 1). Una vez finalizada, cambia al estado state_estimation_ready_st.
state_estimation_ready_st	Activa la señal start_state_estimation → 1 del submódulo i_state_estimation y cambia al estado state_estimation_wait_st.
state_estimation_wait_st	Se mantiene en este estado hasta que el submódulo i_state_estimation finalice la operación (state_estimation_ready ← 1). Una vez finalizada, cambia al estado pcov_estimation_ready_st.
pcov_estimation_ready_st	Activa la señal start_pcov_estimation → 1 del submódulo i_pcov_estimation y cambia al estado pcov_estimation_wait_st.
pcov_estimation_wait_st	Se mantiene en este estado hasta que el submódulo i_pcov_estimation finalice la operación (pcov_estimation_ready ← 1) y selecciona la segunda entrada del multiplexor addr_mux (addr_chix_ctrl → 1) para que el módulo i_pcov_estimation pueda leer datos de la RAM i_chix_ram. Una vez finalizada, cambia al estado ready_st.

Tabla 8.8: Estados del controlador

Capítulo 9

Módulo de Corrección

9.1. Descripción general del módulo

Este módulo explica la implementación de las ecuaciones del filtro correspondientes a la etapa CORRECTION. Dicha etapa la hemos dividido en varios submódulos, uno por cada ecuación descrita a continuación:

$$\chi_{k|k-1} = \left[\begin{array}{ccc} \hat{x}_k^- & \hat{x}_k^- + \gamma\sqrt{P_k^-} & \hat{x}_k^- - \gamma\sqrt{P_k^-} \end{array} \right] \tag{9.1}$$

$$\Upsilon_{k|k-1} = H(\chi_{k|k-1}) \tag{9.2}$$

$$\hat{y}_k^- = \sum_{i=0}^{2L} W_i^{(m)} \Upsilon_{i,k|k-1} \tag{9.3}$$

$$P_{\tilde{y}_k\tilde{y}_k} = \sum_{i=0}^{2L} W_i^{(c)} (\Upsilon_{i,k|k-1} - \hat{y}_k^-)(\Upsilon_{i,k|k-1} - \hat{y}_k^-)^T + R^n \tag{9.4}$$

Para gestionar correctamente el camino de datos hemos implementado un controlador, el cual secuenciará el inicio de ejecución de cada submódulo en el orden específico descrito por las ecuaciones. Este controlador se basa en el protocolo de comunicación *start - ready* explicado anteriormente.

La figura 9.1 ilustra el diagrama de bloques del módulo CORRECTION y su interfaz se describe en la tabla 9.1:

A continuación se describe el funcionamiento del módulo y el orden de generación de los datos:

1. Se espera la señal *start* que activa la ejecución del módulo. Esta señal se activa cuando todos los datos previos están disponibles.

2. Se calcula un nuevo conjunto de Puntos Sigma a partir de la media a priori y de la covarianza a priori del proceso actual, según la ecuación9.1.

3. Se aplica el modelo de medición H sobre los puntos sigma anteriores obteniendo un nuevo conjunto de puntos sigma transformados, según la ecuación 9.2.

4. Se estima un valor a priori de la medida a partir de los puntos sigma transformados con el modelo de medida del punto anterior, según la ecuación 9.3.

5. Se calcula la covarianza de la medida, según la ecuación 9.4.

Los datos transmitidos entre dos submódulos son almacenados en memorias *ram_generic* intermedias por el submódulo productor antes de que este finalice su trabajo (active a su FSM maestra la señal ready) y son consumidos por el siguiente submódulo leyendo los valores almacenados en dicha RAM. Utilizamos un total de 3 RAMs para conectar 4 sumódulos tal y como se indica a continuación:

- *i_expanded_chix_ram*: RAM de 36 posiciones (matriz de 4x9) para almacenar el resultado del submódulo i_sigma_pts_expansion.
- *i_chiy_ram*: RAM de 18 posiciones (matriz de 2x9) para almacenar el resultado del submódulo i_measure_transformation.
- *i_prior_measure_ram*: RAM de 2 posiciones (matriz de 2x1) para almacenar el resultado del submódulo i_measure_estimation.

9.1.1. Unidad de control *i_correction_controller*

Se debe asegurar la correcta ejecución de CORRECTION, por este motivo hemos dividido el control de la etapa en dos niveles: una unidad de control maestra *i_correction_controller* encargada de secuenciar el trabajo de los submódulos que implementan cada ecuación y la unidad de control interna de cada submódulo.

La figura 9.2 muestra el diagrama de transición de estados de la unidad y sus estados se describen en detalle en la tabla 9.2:

Todos los estados jerárquicos se modelan de la misma forma. En el caso del primer estado, se basa en activar la señal *start_sigma_pts_expansion* →

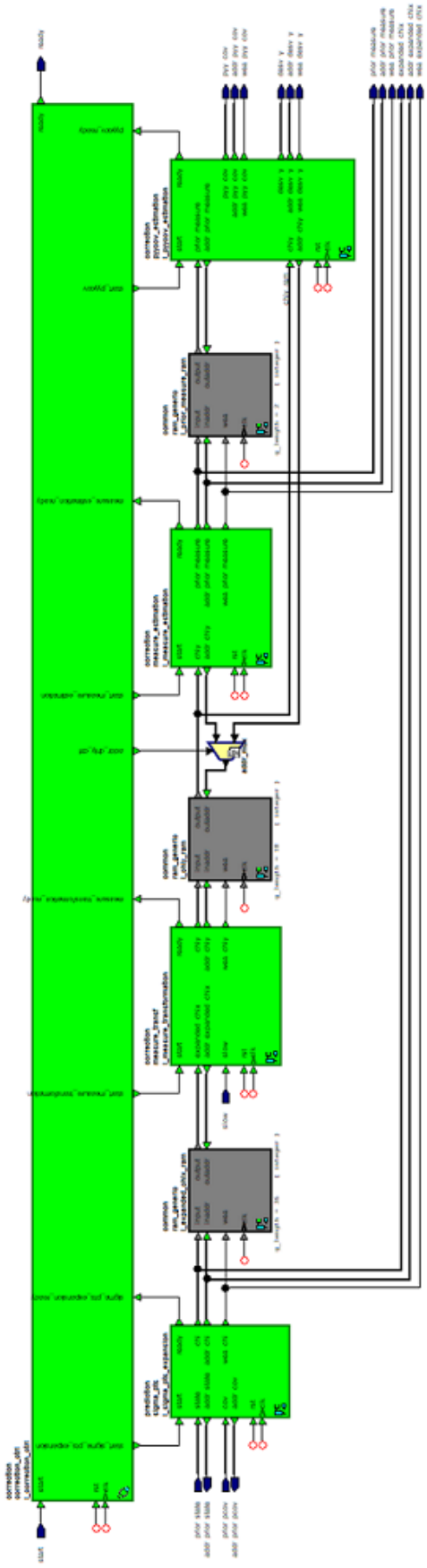

Figura 9.1: *Top-level* del módulo CORRECTION

Nombre	Tamaño	Sentido	Comentario
start	1 bit	Entrada	Comienzo de la etapa corrección (1 ciclo a alta).
slow	1 bit	Entrada	Indica si la medida actual corresponde a una medida lenta (slow='1').
ready	1 bit	Salida	Flag de fin de la etapa corrección ('0'= busy) .
prior_state	c_sfixed_width	Entrada	Estado a priori
addr_prior_state	2 bits	Salida	Direccion para lectura de la RAM que almacena *prior_state*
prior_pcov	c_sfixed_width	Entrada	Covarianza del estado a priori
addr_prior_pcov	4 bits	Salida	Direccion para lectura de la RAM que almacena *prior_pcov*
expanded_chix	c_sfixed_width	Salida	Nuevo conjunto de Puntos Sigma calculados en el submódulo *i_sigm_pts_expansion*
addr_expanded_chix	6 bits	Salida	Direccion para escritura de la RAM que almacena *expanded_chix*
wea_expanded_chix	1 bit	Salida	Habilitación de escritura en la RAM que almacena *expanded_chix*
prior_measure	c_sfixed_width	Salida	Valor estimado para la media de la medida calculado en el submódulo *i_measure_estimation*
addr_prior_measure	1 bits	Salida	Direccion para escritura de la RAM que almacena *prior_measure*
wea_prior_measure	1 bit	Salida	Habilitación de escritura en la RAM que almacena *prior_measure*
desv_y	c_sfixed_width	Salida	Diferencia entre los puntos sigma propagados por el modelo de medida menos *prior_measure*
addr_desv_y	5 bits	Salida	Direccion para escritura de la RAM que almacena *desv_y*
wea_desv_y	1 bit	Salida	Habilitación de escritura en la RAM que almacena *desv_y*
pyy_cov	c_sfixed_width	Salida	Covarianza de la medida
addr_pyy_cov	2 bits	Salida	Direccion para escritura de la RAM que almacena *pyy_cov*
wea_pyy_cov	1 bit	Salida	Habilitación de escritura en la RAM que almacena *pyy_cov*

Tabla 9.1: Interfaz del módulo CORRECTION

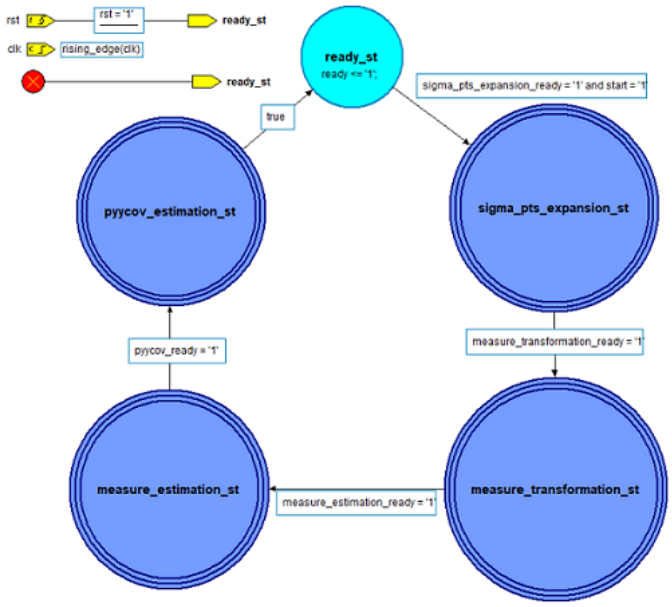

Figura 9.2: CORRECTION controller

1 durante un ciclo y espera a recibir la señal *sigma_pts_expansion_ready* $\leftarrow$ 1 que indica que el submódulo ha acabado su trabajo.

Estado	Acción
ready_st	El sistema notifica que está preparado para iniciar el trabajo de nuevo (ready → 1) y permanece a la espera de las señales start ← 1 y sigma_pts_expansion_ready ← 1
sigma_pts_expansion_st	Inicia el submúdulo que calcula los nuevos Puntos Sigma y espera a que termine
measure_transformation_st	Inicia el submúdulo que transforma los puntos sigma con el modelo de medición y espera a que termine
measure_estimation_st	Inicia el submúdulo que estima un valor a priori de la medida y espera a que termine
pyycov_estimation_st	Inicia el submúdulo que calcula la covarianza de la medida y espera a que termine

Tabla 9.2: Estados de la unidad de control maestra del módulo CORRECTION

9.2. Expansión de puntos sigma: Submódulo i_sigma_pts_expansion

Esta ecuación proceso es una instancia del componente *sigma_pts* utilizado también en la etapa PREDICTION para calcular un primer conjunto de *Puntos Sigma*.

Implementa la ecuación 9.1 que calcula un nuevo conjunto de Puntos Sigma a partir de la media a priori y de la raiz cuadrada de la matriz covarianza a priori del proceso actual.

Las únicas diferencias respecto al submódulo anterior mencionado es la utilización de la media del estado actual predicho *prior_state* y la matriz covarianza del estado predicho *prior_pcov*, datos calculados en las dos última ecuaciones del módulo *i_Prediction*, submódulos *i_state_estimation* y *i_pcov_estimation*)

9.3. Transformación de los puntos sigma con el modelo de medición: Submódulo i_measure_transformation

Implementa la ecuación 9.2 que modifica los puntos sigma expandidos calculados en la fase anterior **??** aplicandoles el modelo de medición H, dependiendo de si se trata de medida lenta o rápida. La función H toma el valor 9.5 en el caso de tener medida rápida, y el valor 9.6 en el caso de

tener medida lenta:

$$\Upsilon_{s,k} = \begin{bmatrix} 0{,}5a_k & 0 & 0{,}5g_k & 0 \end{bmatrix} \begin{bmatrix} g_k \\ \Delta g_k \\ a_k \\ \Delta a_k \end{bmatrix} \tag{9.5}$$

$$\begin{bmatrix} \Upsilon_{s,k} \\ \Upsilon_{f,k} \end{bmatrix} = \begin{bmatrix} 0{,}5a_k & 0 & 0{,}5g_k & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} g_k \\ \Delta g_k \\ a_k \\ \Delta a_k \end{bmatrix} \tag{9.6}$$

En nuestro caso, la funcion de transformación ha sido reducida para simplificar el HW que se ocupará de realizar tal operación:

$$\Upsilon_{s,k} = \begin{bmatrix} g_k * a_k \end{bmatrix}$$

o

$$\begin{bmatrix} \Upsilon_{s,k} \\ \Upsilon_{f,k} \end{bmatrix} = \begin{bmatrix} g_k * a_k \\ g_k \end{bmatrix}$$

La figura 9.3 representa el diagrama de bloques del submódulo y su interfaz se define en la tabla 9.3. El Submódulo se compone de: dos registros que almacenan los valores g_k y a_k necesarios para realizar la trasformación; el bloque *transformation_ equation* que modela la ecuación de transformación; un contador ascendente de 0 a 8 que indexa dos tablas de verdad, las cuales generan las direcciones usadas para lectura de datos y escritura de resultados, multiplexores y demás elementos de interconexion de señales.

El proceso puede describirse en los siguientes pasos:

1. El módulo espera recibir la señal *start* para comenzar. Ésta indica que los datos previos ya están disponibles para su uso en las RAMs correspondientes.

2. Precarga los valores de g_k y a_k necesarios y efectua la operación en la UF *transformation_ equation*

3. La FSM se encarga de actualizar el contador para generar las siguientes direcciones de petición de datos y escritura del resultado, así como generar las señales de control para escribir el resultado en la RAM de salida

Para generar las direcciones de lectura y escritura se usan dos tablas de verdad y el contador *i_ measure_ transf_ counter*, el cual cuenta de 0 a 8 ya que debemos realizar la operación de transformación sobre las 9 columnas de

Puerto	Tamaño	Sentido	Comentario
start	1 bit	Entrada	Comienzo del submódulo *i_measure_transformation* (1 ciclo a alta).
slow	1 bit	Entrada	Indica si la medida actual corresponde a una medida lenta (slow='1').
ready	1 bit	Salida	Flag de fin del submódulo ('0'= busy) .
expanded_chix	c_sfixed_width	Entrada	Nuevo conjunto de Puntos Sigma calculados anteriormente
addr_expanded_chix	6 bits	Salida	Direccion para lectura de la RAM que almacena *expanded_chix*
chiy	c_sfixed_width	Entrada	Puntos Sigma transformados con el modelo de medición
addr_chiy	5 bits	Salida	Direccion para escritura de la RAM que almacena *chiy*
wea_chix	1 bit	Salida	Habilitación de escritura en la RAM que almacena *chiy*

Tabla 9.3: Interfaz del módulo *i_measure_transformation*

la matriz *expanded_chix*. Para generar la dirección de escritura *addr_chiy* primeramente generamos dos posibles valores llamados *addr_chiy_fast* y *addr_chiy_slow* los cuales indican la posicion de almacenamiento en caso de tratarse de una medida rápida o lenta respectivamente. La unidad de control es la encargada de decidir cual de os dos valores escoger en cada caso. La matriz *expanded_chix* se almacena de la siguiente forma:

$$expanded\_chix_{4x9} = \begin{bmatrix} a0 & a1 & a2 & a3 & a4 & a5 & a6 & a7 & a8 \\ a9 & a10 & a11 & a12 & a13 & a14 & a15 & a16 & a17 \\ a18 & a19 & a20 & a21 & a22 & a23 & a24 & a25 & a26 \\ a27 & a28 & a29 & a30 & a31 & a32 & a33 & a34 & a35 \end{bmatrix}$$

y la matriz *chiy* debe guardarse en el orden:

$$chiy_{2x9} = \begin{bmatrix} a0 & a1 & a2 & a3 & a4 & a5 & a6 & a7 & a8 \\ a9 & a10 & a11 & a12 & a13 & a14 & a15 & a16 & a17 \end{bmatrix}$$

La tabla *indexer_fast_measure* genera la dirección de lectura *addr_expanded_chix* y la dirección de escritura *addr_chiy_fast* a partir del valor *index_aux* del contador y de la señal de control *addr_data* que indica a qué dato acceder dentro de la columna, según la tabla 9.4. Así mientras index_aux = 0000 debemos leer los elementos situados en las posiciónes 0 (addr_data = 0) y 18

(addr_data = 1) de *expanded_ chix* y el resultado se almacena en la posición 0 de *addr_ chiy* por tratarse de una medida rápida.

index_aux	addr_data	addr_expanded_chix	addr_chiy_fast
0000	0	0	0
0000	1	18	0
0001	0	1	1
0001	1	19	1
0010	0	2	2
0010	1	20	2
0011	0	3	3
0011	1	21	3
0100	0	4	4
0100	1	22	4
0101	0	5	5
0101	1	23	5
0110	0	6	6
0110	1	24	6
0111	0	7	7
0111	1	25	7
1000	0	8	8
1000	1	26	8

Tabla 9.4: Tabla de verdad *indexer_fast_ measure*

La tabla *indexer_ slow_ measure* genera la dirección de escritura *addr_ chiy_ slow* a partir del valor *index_ aux* del contador según la tabla 9.5. Así cuando index_aux = 0000 debemos guardar el resultado en la posición 9 de *addr_ chiy* por tratarse de una medida lenta.

9.3.1. Unidad de control *i_ measure_ transf_ fsm*

La maquina de estados *i_ measure_ transf_ fsm* es la encargada de generar las salidas de control de los multiplexores, la señal de capacitación del contador y los indices relativos a la dirección del siguiente dato que espera recibir. La figura 9.4 muestra su implementación y la descripcion detallada de sus estados se recoge en la tabla 9.6:

index_aux	addr_chiy_slow
0000	9
0001	10
0010	11
0011	12
0100	13
0101	14
0110	15
0111	16
1000	17

Tabla 9.5: Tabla de verdad *indexer_slow_measure*

9.4. Estimación a priori de la media de la medida actual: Submódulo i_measure_estimation

En este módulo se implementa la ecuación 9.3 que calcula el valor medio por filas de los puntos sigma transformados en la sección **??**.

La figura 9.5 representa el diagrama de bloques del submódulo y la tabla 9.7 describe la interfaz. Se compone de: una memoria *i_ram_chiy_weights* de tamaño 18 que almacena una matriz 2x9 correspondiente a los puntos sigma ya multiplicados por sus pesos correspondientes;

El proceso se describe de la siguiente forma:

1. El modulo espera recibir la señal *start* para comenzar. Ésta indica que los datos previos ya están disponibles para su uso en las RAMs correspondientes.

2. *i_mult_scalar_1x9_2x9*: Multiplicación término a término de la matriz de pesos W^m de tamaño 1x9 con la matriz $\Upsilon_{k|k-1}$ de tamaño 2x9 que representa los puntos sigma transformados, dando como resultado una matriz 2x9. Este módulo debe generar las direcciones de escritura y señales de control correspondientes para almacenar los datos en la memoria *i_ram_chiy_weights*.

3. *i_summatory_rows_2x9*: sumatorio de todos los valores de cada fila pertenecientes a una matriz 2x9 de entrada, dando como resultado una matriz columna de dimensiones 2x1 que representa la media de la medida estimada a priori. Este módulo debe generar las direcciones validas de lectura para obtener los datos de la memoria *i_ram_chiy_weights*. Tras realizar sus operaciones, debe generar las direcciones de escritura y señales de control para escribir el resultado final en la RAM de salida *i_prior_measure_ram*.

Estado	Acción
ready_st	Inicializa el valor del contador y los registro. Pone valor a la señal *addr_data* $\rightarrow 0$. Notifica que está preparado (ready $\rightarrow 1$) y permanece a la espera de la señal start $\leftarrow 1$
wait_st	Ciclo de espera para obtener el primer dato de la ram (addr_data $\rightarrow 0$)
data1_st	Ciclo de espera para obtener el segundo dato de la ram (addr_data $\rightarrow 1$). Carga el primer dato en el registro 1 (load_reg1 $\rightarrow 1$)
data2_st	Carga el segundo dato en el registro 2 (load_reg2 $\rightarrow 1$)
save_st	Habilita la escritura en la ram que almacena el resultado (wea $\rightarrow 1$)
slow_measure_st	Habilita la cuenta si el contador no ha llegado al fin (enable_count $\rightarrow 1$). Si tenemos medida lenta selecciona la salida del multiplexor que contiene el valor de g_k (slow_value_control $\rightarrow 1$) y selecciona los multiplexores apropiados para generar la direccion y el valor a escribir de medida lenta (mux_control $\rightarrow 1$)

Tabla 9.6: Estados de la unidad de control *i_measure_transf_fsm*

Puerto	Tamaño	Sentido	Comentario
start	1 bit	Entrada	Comienzo del submódulo *i_measure_estimation* (1 ciclo a alta).
ready	1 bit	Salida	Flag de fin del submódulo ('0'= busy) .
chiy	c_sfixed_width	Entrada	Puntos Sigma transformados con el modelo de medición
addr_chiy	5 bits	Salida	Direccion para lectura de la RAM que almacena *chiy*
prior_measure	c_sfixed_width	Salida	Media estimada de la medida
addr_prior_measure	1 bit	Salida	Direccion para escritura de la RAM que almacena *prior_measure*
wea_prior_measure	1 bit	Salida	Habilitación de escritura en la RAM que almacena *prior_measure*

Tabla 9.7: Interfaz del módulo *i_measure_estimation*

9.4.1. Unidad de control *i_fsm_measure_estimation*

La figura 9.6 y la tabla 9.8 muestran los estados del controlador *i_fsm_measure_estimation*, el cual se encarga de gestionar las operaciones del módulo generando, para ello, las correspondientes señales de control.

9.5. Cálculo de la covarianza de la medida: Submódulo i_pyycov_estimation

Este módulo implementa la ecuación 9.4 que calcula la matriz covarianza de la medida $P_y y$ a partir de los puntos sigma $\Upsilon_{k|k-1}$ calculados por el submódulo **??** y la media estimada de la medida calculada en el submódulo **??**.

La figura 9.7 muestra el diagrama de bloques y la tabla 9.9 recoge las señales de su interfaz.

El módulo se compone de:

1. *i_sub_2x9_2x1*: Resta a la matriz de puntos sigma $\Upsilon_{k|k-1}$ de tamaño 2x9 el valor media de la medida $\hat{y}_k^-$ de tamaño 2x1, obteniendo como resultado una matriz de dimensiones 2x9. Almacena el resultado en la memoria *i_desv_y_ram*.

2. *i_desv_y_ram*: RAM de 18 posiciones que almacena el resultado del

componente anterior: la diferencia entre los puntos sigma propagados por el modelo de medida menos la media de la medida.

3. *i_mult_scalar_1x9_2x9*: Multiplica término a término la matriz de pesos W^c de tamaño 1x9 con la matriz obtenida en el paso anterior, dando como resultado una matriz de dimensiones 2x9. Almacena el resultado en la memoria *i_desv_y_weights_ram*.

4. *i_desv_y_weights_ram*: RAM de 18 posiciones que almacena el resultado del componente anterior.

5. *i_mult_2x9_9x2*: Multiplica la matriz almacenada en *i_desv_y_weights_ram* por la traspuesta de la matriz almacenada en *i_desv_y_ram*, obteniendo como resultado una matriz de dimensiones 2x2. Almacena el resultado en la memoria *i_desv_y_mult_ram*.

6. *i_desv_y_mult_ram*: RAM de 4 posiciones que almacena el resultado del componente *i_mult_2x9_9x2*.

7. *i_adder_2x2_2x2*: Suma de la matriz 2x2 obtenida en el paso anterior junto a la covarianza del ruido de la medida R^n de dimensión 2x2, dando como resultado una matriz de 2x2 que representa la covarianza de la medida. Almacena el resultado final en la RAM *i_pyy_cov_ram* externa al módulo.

En todos los casos el módulo productor será el encargado de generar las direcciones de escritura y señales de control correspondientes para almacenar los datos correctamente en la RAM. De igual manera, el módulo consumidor será el encargado de generar direcciones validas de lectura para la obtención de los datos una vez los datos estén disponibles.

9.5.1. Unidad de control *i_fsm_pyycov_estimation*

El controlador *i_fsm_pyycov_estimation* se encarga de gestionar las operaciones del submódulo *i_pyycov_estimation*. La figura 9.8 ilustra el diagrama de estados del controlador y la tabla 9.8 describe detalladamente cada estado:

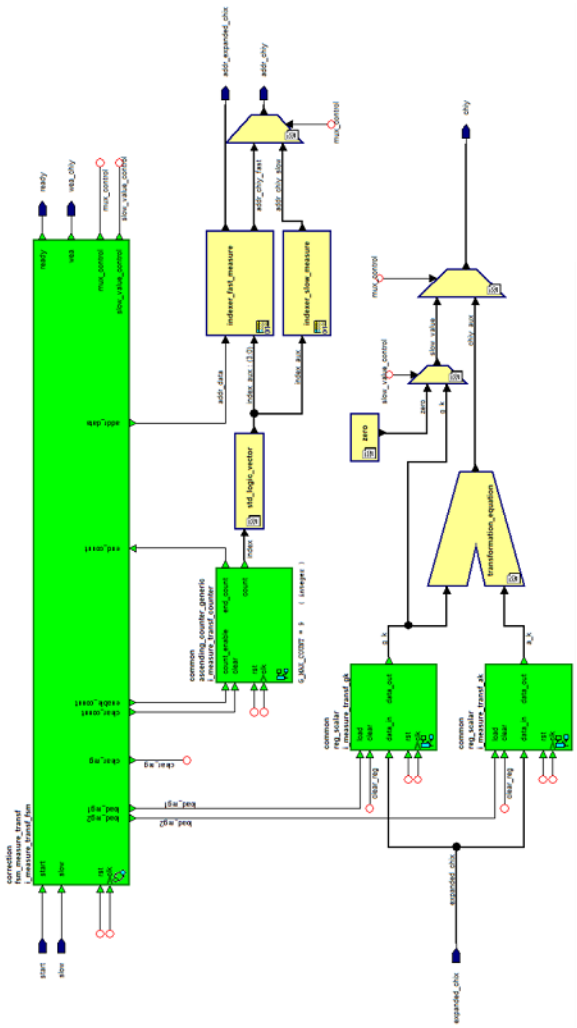

Figura 9.3: Diagrama de bloques del submódulo *i_measure_transformation*

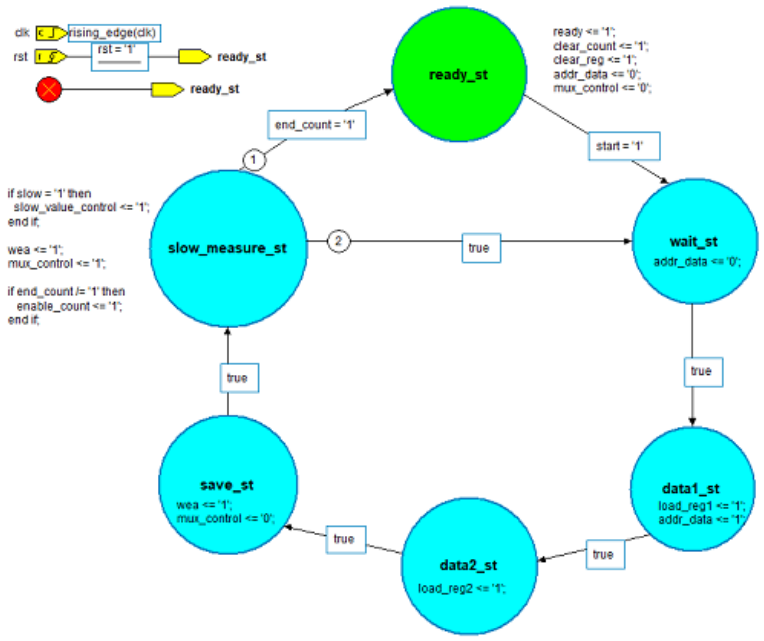

Figura 9.4: Diagrama de estados del controlador *i_measure_transf_fsm*

Estado	Acción
ready_st	Notifica que está preparado (ready → 1) y permanece a la espera de la señal start ← 1 y de que el primer submódulo indique que está preparado (mult_rdy ← 1)
mult_start_st	Envía la señal de comenzar la ejecución al módulo *i_mult_scalar_1x9_2x9* (start_mult → 1)
wait_mult_st	Espera a recibir la señal mult_rdy ← 1 que indica que el submódulo *i_mult_scalar_1x9_2x9* ha terminado su ejecución
add_start_st	Envía la señal de comenzar la ejecución al módulo *i_summatory_rows_2x9* (start_adder → 1)
wait_add_st	Espera a recibir la señal adder_rdy ← 1 que indica que el submódulo *i_summatory_rows_2x9* ha terminado su ejecución
done_st	Este estado indica que todas las operaciones han finalizado y por tanto pasa al estado *ready_st*

Tabla 9.8: Estados de la unidad de control *i_fsm_measure_estimation*

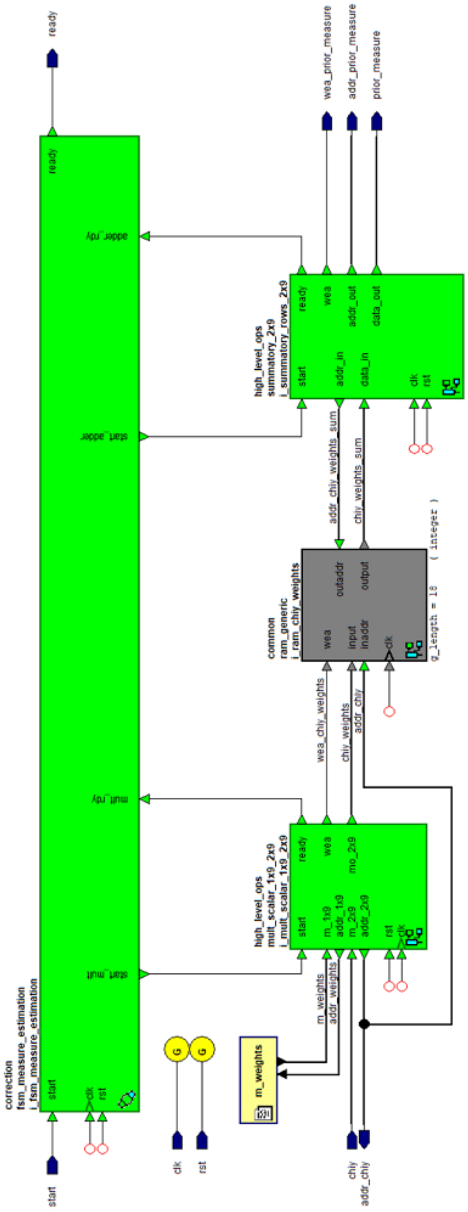

Figura 9.5: Diagrama de bloques del submódulo *i_measure_estimation*

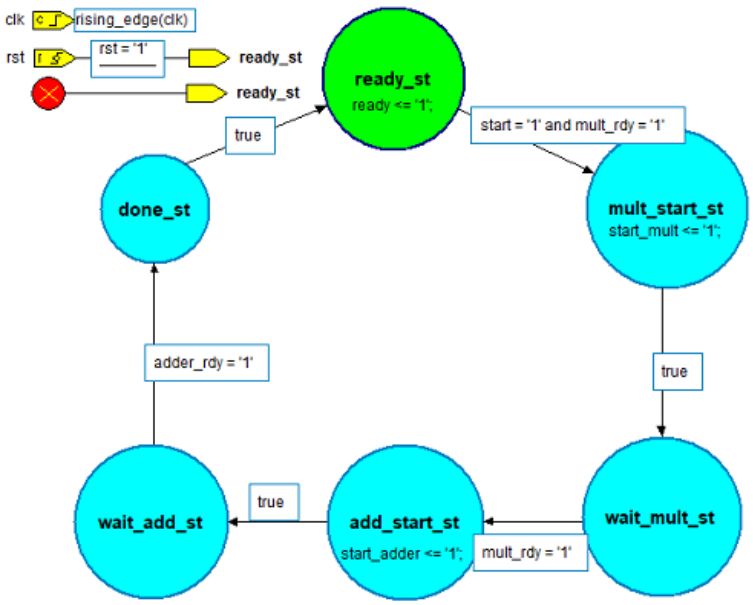

Figura 9.6: Diagrama de estados del controlador *i_fsm_measure_estimation*

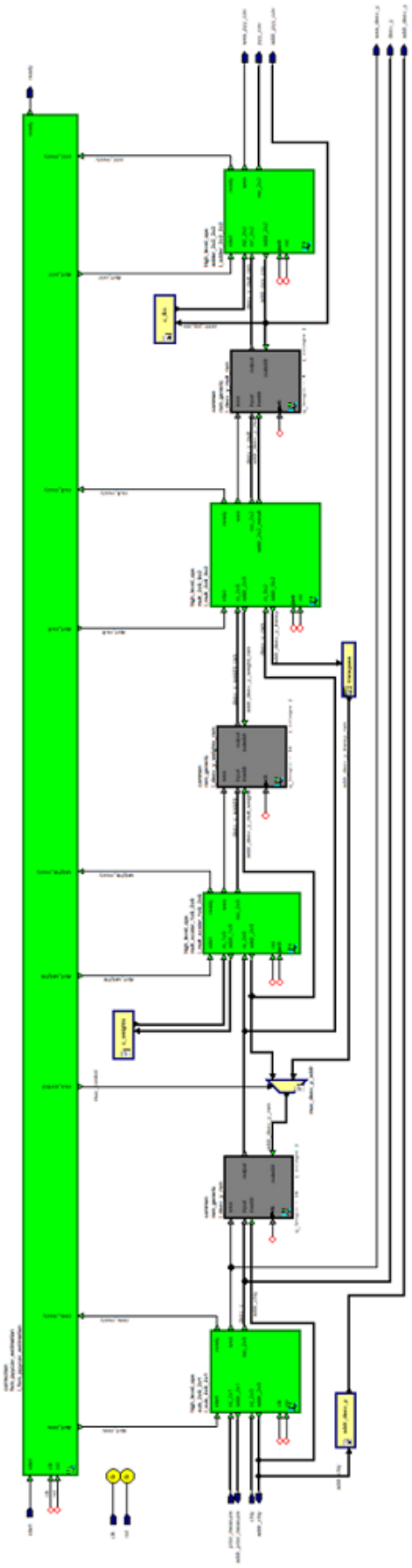

Figura 9.7: Diagrama de bloques del submódulo *i_pyycov_estimation*

Puerto	Tamaño	Sentido	Comentario
start	1 bit	Entrada	Comienzo del submódulo *i_measure_estimation* (1 ciclo a alta).
ready	1 bit	Salida	Flag de fin del submódulo ('0'= busy) .
prior_measure	c_sfixed_width	Entrada	Media estimada de la medida
addr_prior_measure	1 bit	Salida	Direccion para lectura de la RAM que almacena *prior_measure*
chiy	c_sfixed_width	Entrada	Puntos Sigma transformados con el modelo de medición
addr_chiy	5 bits	Salida	Direccion para lectura de la RAM que almacena *chiy*
pyy_cov	c_sfixed_width	Salida	Covarianza de la medida
addr_pyy_cov	2 bit	Salida	Direccion para escritura de la RAM que almacena *pyy_cov*
wea_pyy_cov	1 bit	Salida	Habilitación de escritura en la RAM que almacena *pyy_cov*
desv_y	c_sfixed_width	Salida	Diferencia entre los puntos sigma propagados por el modelo de medida menos *prior_measure*
addr_desv_y	5 bit	Salida	Direccion para escritura de la RAM que almacena *desv_y*
wea_desv_y	1 bit	Salida	Habilitación de escritura en la RAM que almacena *desv_y*

Tabla 9.9: Interfaz del módulo *i_pyycov_estimation*

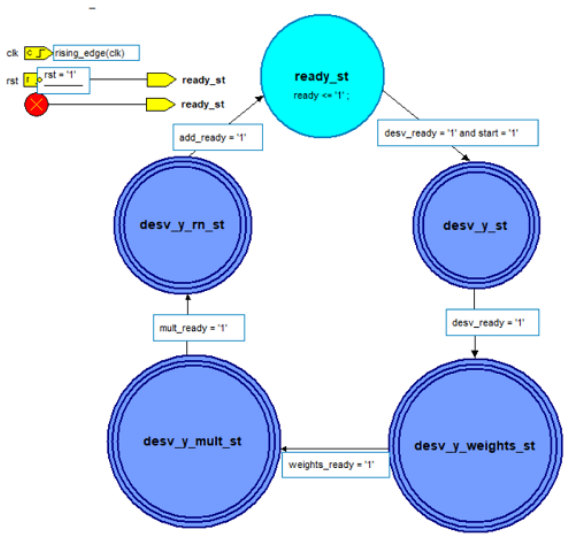

Figura 9.8: Diagrama de estados del controlador *i_fsm_pyycov_estimation*

Estado	Acción
ready_st	Notifica que está preparado (ready → 1) y permanece a la espera de la señal start ← 1 y de que el primer submódulo indique que está preparado (desv_ready ← 1)
desv_y_st	Estado jerárquico que envía la señal de comenzar la ejecución al módulo *i_sub_2x9_2x1* (start_desv → 1) y espera a que éste notifique el fin de su ejecución con la señal desv_ready ← 1
desv_y_weights_st	Estado jerárquico que envía la señal de comenzar la ejecución al módulo *i_mult_scalar_1x9_2x9* (start_weigths → 1) y espera a que éste notifique el fin de su ejecución con la señal weights_ready ← 1. Selecciona la entrada 0 del multiplexor *mux_desv_y_addr* (mux_control → 0)
desv_y_mult_st	Estado jerárquico que envía la señal de comenzar la ejecución al módulo *i_mult_2x9_9x2* (start_mult → 1) y espera a que éste notifique el fin de su ejecución con la señal mult_ready ← 1. Selecciona la entrada 1 del multiplexor *mux_desv_y_addr* (mux_control → 1)
desv_y_rn_st	Estado jerárquico que envía la señal de comenzar la ejecución al módulo *i_adder_2x2_2x2* (start_add → 1) y espera a que éste notifique el fin de su ejecución con la señal add_ready ← 1

Tabla 9.10: Estados de la unidad de control *i_fsm_pyycov_estimation*

Capítulo 10

Módulo de Kalman Gain

En este módulo se explica la implementación de las ecuaciones correspondientes a la etapa Kalman Gain. Se calcula la covarianza cruzada *PXY* (10.1), la ganancia de Kalman *K* (10.2), el estado a posteriori (10.3) y la covarianza del estado a posteriori (10.4). Este módulo lo hemos dividido en cuatro submódulos, uno por cada ecuación descrita a continuación:

$$P_{x_k y_k} = \sum_{i=0}^{2L} W_i^{(c)} \left(\chi_{i,k|k-1} - \overline{x}_k^-\right) \left(\Upsilon_{i,k|k-1} - \overline{y}_k^-\right)^T \quad (10.1)$$

$$K_k = P_{x_k y_k} \cdot P_{\tilde{y_k}\tilde{y_k}}^{-1} \quad (10.2)$$

$$\overline{x}_k = \overline{x}_k^- + K_k \cdot (y_k - \overline{y}_k^-) \quad (10.3)$$

$$P_k = P_k^- - K_k \cdot P_{\tilde{y_k}\tilde{y_k}} \cdot K_k^T \quad (10.4)$$

Es necesario la implementación de un quinto submódulo, el controlador *i_ kalman_ gain_ ctrl* que secuencia el inicio de ejecución de cada submódulo mediante el protocolo de comunicación *start* - *ready* 7.1.2. La figura 10.1 representa el diagrama de bloques:

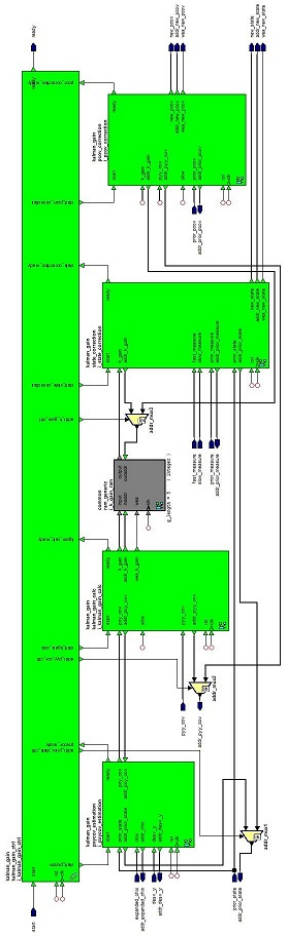

Figura 10.1: *Top-level* del módulo Kalman Gain

- *i_pxycov_estimation*: Implementa la ecuación (10.1).
- *i_kalman_gain_calc*: Implementa la ecuación (10.2).
- *i_state_correction*: Implementa la ecuación (10.3).
- *i_pcov_correction*: Implementa la ecuación (10.4).

La interfaz del módulo *Kalman Gain* está definida en la tabla 10.1:

Nombre	Tamaño	Sentido	Comentario
start	1 bit	Entrada	Indica el comienzo de la etapa Kalman_Gain (1 ciclo a alta).
ready	1 bit	Salida	Indica si el módulo está disponible para una nueva ejecución.
slow	1 bit	Entrada	Tipo de medición (lenta o rápida).
expanded_chix	c_sfixed_width	Entrada	Elemento de los puntos sigma expandidos.
addr_expanded_chix	36 pos (6 bits)	Salida	Dirección de la RAM *i_expanded_chix_ram* que almacena los puntos sigma expandidos.
desv_y	c_sfixed_width	Entrada	Elemento de la desviación de los puntos sigma con respecto a la media a priori.
addr_desv_y	18 pos (5 bits)	Salida	Dirección de la RAM *i_desv_y_ram* que almacena la desviación de los puntos sigma con respecto a la media a priori.
prior_state	c_sfixed_width	Entrada	Elemento del estado a priori.
addr_prior_state	4 pos (2 bits)	Salida	Dirección de la RAM *i_prior_state_ram* que almacena el estado a priori.
prior_pcov	c_sfixed_width	Entrada	Elemento de la covarianza del estado a priori.
addr_prior_pcov	16 pos (4 bits)	Salida	Dirección de la RAM *i_prior_pcov_ram* que almacena la covarianza del estado a priori.
prior_measure	c_sfixed_width	Entrada	Elemento de la medida a priori.
addr_prior_measure	2 pos (1 bit)	Salida	Dirección de la RAM *i_prior_measure_ram* que almacena la medida a priori.
fast_measure	c_sfixed_width	Entrada	Medida rápida de entrada al filtro.
slow_measure	c_sfixed_width	Entrada	Medida lenta de entrada al filtro.
pyy_cov	c_sfixed_width	Entrada	Elemento de la covarianza de la medida.
addr_pyy_cov	4 pos (2 bits)	Salida	Dirección de la RAM *i_pyy_cov_ram* que almacena la covarianza de la medida.
new_pcov	c_sfixed_width	Salida	Elemento de la covarianza del estado a posteriori.
addr_new_pcov	16 pos (4 bits)	Salida	Dirección de la RAM *i_top_pcov_ram* que almacena la covarianza del estado a posteriori.
wea_new_pcov	1 bit	Salida	Señal de escritura de la RAM *i_top_pcov_ram*.
new_state	c_sfixed_width	Salida	Elemento del estado a posteriori.
addr_new_state	4 pos (2 bits)	Salida	Dirección de la RAM *i_top_prior_ram* que almacena el estado a posteriori.
wea_new_state	1 bit	Salida	Señal de escritura de la RAM *i_top_prior_ram*.

Tabla 10.1: Interfaz del módulo KALMAN GAIN

Funcionamiento del módulo Kalman Gain

El módulo indica con la señal de ready que está listo para una nueva iteración. Cuando recibe la señal *start*, realiza los siguientes pasos y no se inicia un paso hasta que no haya finalizado el anterior:

1. El controlador activa la señal de start del submódulo *i_pxycov_estimation* para realizar el cálculo de la covarianza cruzada *PXY*. Este submódulo lee los puntos sigmas expandidos almacenados en la RAM externa *i_expanded_chix_ram*, la desviación de estos con respecto a la medida de la RAM externa *i_desv_y_ram* y el estado a priori de la RAM externa *i_prior_state_ram*. El resultado es almacenado en una RAM interna del submódulo *i_pxycov_estimation*.

2. El controlador activa la señal de start del submódulo *i_kalman_gain_calc* para el cálculo de la ganancia de Kalman. Este submódulo necesita varios datos de entrada: la covarianza cruzada PXY almacenada en la RAM interna del submódulo *i_pxycov_estimation*; la entrada *Slow*; la covarianza de la medida *PYY* almacenada en la RAM externa *i_pyy_cov_ram*. El resultado de la ganancia de Kalman es almacenado en la RAM i_k_gain_ram de 8 posiciones correspondiente a un matriz de dimensiones 4x2.

3. El controlador activa la señal de start del submódulo *i_state_correction* para calcular el estado a posteriori. Este submódulo necesita la medida del sensor (fast_measure y slow_measure), la ganancia de Kalman almacenada en la RAM *i_k_gain_ram*, el estado a priori almacenado en la RAM externa *i_prior_state_ram* y la medida a priori almacenada en la RAM externa *i_prior_measure_ram*. El estado a posteriori es almacenado en la RAM externa *i_top_state_ram*.

4. El controlador activa la señal de start del submódulo *i_pcov_correction* para calcular la covarianza del estado a posteriori. El submódulo necesita: la covarianza del estado a priori almacenada en la RAM externa *i_prior_pcov_ram*; la entrada *Slow*; la ganancia de Kalman almacenada en la RAM *i_k_gain_ram*; la covarianza de la medida *PYY* almacenada en la RAM externa *i_pyy_pcov_ram*. La covarianza del estado a posteriori se almacena en la RAM externa *i_top_pcov_ram*.

Se añaden varios multiplexores para que los submódulos puedan leer de la misma RAM:

- addr_mux1: Multiplexor en la dirección de lectura de la RAM externa *i_prior_state_ram*. La señal de control addr_prior_state_ctrl selecciona el submódulo que tiene acceso (*i_pxycov_estimation* o *i_state_correction*).

- addr_mux2: Multiplexor en la dirección de lectura de la RAM externa *i_pyy_pcov_ram*. La señal de control addr_pyy_cov_ctrl selecciona el submódulo tiene acceso (*i_kalman_gain_calc* o *i_pcov_correction*).

- addr_mux3: Multiplexor en la dirección de lectura de la RAM *i_k_gain_ram*. La señal addr_k_gain_ctrl selecciona el submódulo tiene acceso (*i_state_correction* o *i_pcov_correction*).

10.1. Cálculo de la covarianza PXY: Módulo i_pxycov_estimation

Implementa la expresión 10.1, que calcula la covarianza cruzada PXY a partir de los puntos sigma expandidos, la desviación de estos con respecto a la medida, los pesos de la covarianza y el estado a priori. Este módulo se divide en cinco submódulos comunes explicados en la sección 7.3 para realizar las operaciones y un sexto submódulo (controlador *i_pxy_estimation_ctrl* encargado de secuenciar el inicio de ejecución de cada submódulo:

- *i_sub_4x9_4x1*(sub_4x9_4x1): Implementa la resta de matrices de dimensiones 4x9 y 4x1 respectivamente.

- *i_mult_scalar_1x9_4x9*(mult_scalar_1x9_4x9): Implementa la multiplicación de nueve escalares por cada vector columna de la matriz de dimensiones 4x9.

- *i_mult_4x9_9x2*(mult_4x9_9x2): Implementa una multiplicación de matrices de dimensiones 4x9 y 9x2 respectivamente.

- *i_desv_x_ram*(ram_generic): Implementa una BRAM de 36 posiciones.

- *i_chix_weight_ram*(ram_generic): Implementa una BRAM de 36 posiciones.

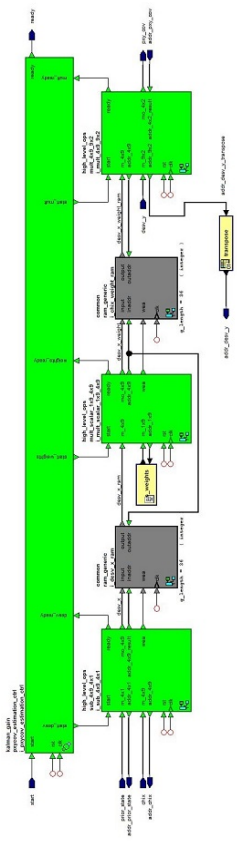

Figura 10.2: *Diagrama de bloques: Módulo i_pxycov_estimation*

La figura 10.2 representa el diagrama de bloques de este módulo.

Los pesos de la covarianza están almacenados en la ROM *c_weights* de nueve posiciones. El submódulo *i_mult_4x9_9x2* genera las direcciones para acceder a las RAMs *i_chix_weight_ram* y *i_desv_y_ram*. Para leer los elementos correctamente de la última RAM, añadimos el bloque transpose que genera la traspuesta para modificar la forma en la que accedemos a la RAM *i_desv_y_ram*.

Cuando recibe la señal de *start* realiza los siguientes pasos de forma secuencial, esperando la finalización del paso anterior para comenzar el siguiente:

1. El controlador activa la señal de start del submódulo *i_sub_4x9_4x1* encargado de realizar la resta de los puntos sigma expandidos y el estado a priori. El resultado es almacenado en la RAM *i_desv_x_ram*.

2. El controlador activa la señal de start del submódulo *i_mult_scalar_1x9_4x9* que multiplica los pesos de la covarianza y los elementos almacenados

en la RAM *i_desv_x_ram*. El resultado es almacenado en la RAM *i_chix_weight_ram*.

3. Por último, el controlador activa la señal de start del submódulo *i_mult_4x9_9x2* que multiplica los elementos almacenados en la RAM *i_chix_weight_ram* y la desviación de los puntos sigma almacenado en la RAM externa *i_desv_y_ram*. El resultado es almacenado en un RAM interna del submódulo *i_mult_4x9_9x2*.

i_pxycov_estimation_ctrl

Implementa una máquina de estados encargada de secuenciar las operaciones del módulo. Está formada por tres estados jerárquicos que siguen la misma estructura mencionada en la sección 8.1, En la tabla 10.2 se detallan los estados correspondientes al diagrama de la figura 10.3.

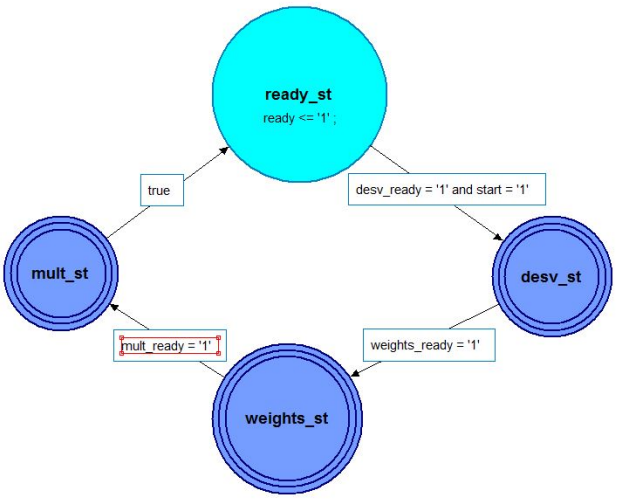

Figura 10.3: *Unidad de control: Módulo i_pxycov_estimation*

Estado	Acción
ready_st	Mantiene la señal de ready → 1 que indica que el módulo está preparado para un nuevo cálculo y cambia al estado desv_ready_st cuando la señal start ← 1.
desv_ready_st	Activa la señal de start_desv → 1 del submódulo i_sub_4x9_4x1 y cambia al estado desv_wait_st.
desv_wait_st	Se mantiene en este estado hasta que el submódulo i_sub_4x9_4x1 finalice la operación (desv_ready ← 1). Una vez finalizada, cambia al estado weights_ready_st.
weights_ready_st	Activa la señal de start_weights → 1 del submódulo i_mult_scalar_1x9_4x9 y cambia al estado weights_wait_st.
weights_wait_st	Se mantiene en este estado hasta que el submódulo i_mult_scalar_1x9_4x9 finalice la operación (weights_ready ← 1). Una vez finalizada, cambia al estado mult_ready_st.
mult_ready_st	Activa al señal start_mult → 1 del submódulo i_mult_4x9_9x2 y cambia al estado mult_wait_st.
mult_wait_st	Se mantiene en este estado hasta que el submódulo i_mult_4x9_9x2 finalice la operación (mult_ready ← 1). Una vez finalizada, cambia al estado ready_st.

Tabla 10.2: Estados del controlador

10.2. Cálculo de la ganancia de Kalman: Módulo i_kalman_gain_calc

Este módulo implementa la expresión 10.2, que calcula la ganancia de Kalman K a partir de la covarianza cruzada PXY calculada en la sección 10.1, la entrada *Slow* (es una medida lenta o rápida) y la covarianza de la medida PYY. Se compone de cinco submódulos comunes explicados en la sección 7.3, además de un sexto submódulo (el controlador *i_kalman_gain_calc_ctrl*) encargado de secuenciar el inicio de ejecución de cada submódulo:

- *i_inv_2x2*(inv_2x2): Implementa la inversa de una matriz de dimensiones 2x2.
- *i_pxy_addr_counter*(ascending_counter_generic): Implementa un contador ascendente módulo 4.
- *i_mult_4x2_2x2*(mult_4x2_2x2): Implementa una multiplicación de matrices de dimensiones 4x2 y 2x2 respectivamente.
- *i_div_norest*(div_norest): Implementa la división de dos escalares.
- *i_inv_2x2_ram*(ram_generic): Implementa una BRAM de 4 posiciones.

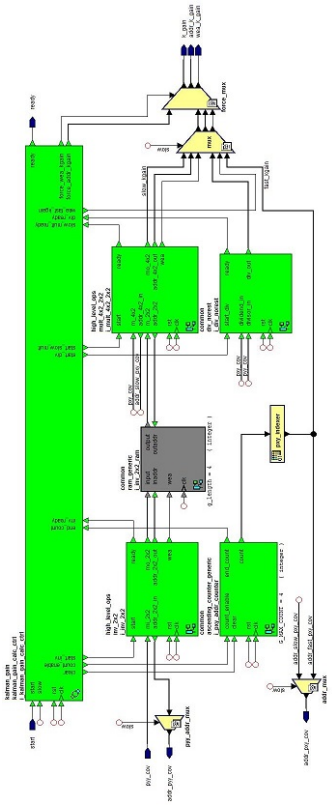

Figura 10.4: *Diagrama de bloques: Módulo i_ kalman_ gain_ calc*

La figura 10.4 representa el diagrama de bloques de este módulo.
Se añaden bloques adicionales:

- Dado que el controlador *i_ kalman_ gain_ calc_ ctrl*, el submódulo *i_ div_ norest* y el submódulo *i_ mult_ 4x2_ 2x2* escriben en la RAM externa *i_ k_ gain_ ram* donde se almacena la ganancia de Kalman, es necesario añadir un multiplexor *mux* controlado por la señal *slow* en la dirección de esctritura de la RAM.
- El submódulo *i_ div_ norest* y el submódulo *i_ inv_ 2x2* necesitan leer elementos de la covarianza de la medida almacenados en la RAM externa *i_ pyy_ cov_ ram*. Para ello, se añade el multiplexor *pyy_ addr_ mux* y la señal de control *slow* en la dirección de lectura de la RAM.
- El submódulo *i_ mult_ 4x2_ 2x2* y submódulo *i_ div_ norest* necesitan leer los elementos de la covarianza cruzada *PXY* almacenados en la RAM externa del submódulo *i_ pxycov_ estimation*. Para ello, se añade el multiplexor *addr_ mux* y la señal de control *slow* en la dirección de lectura de la RAM.

Cuando recibe la señal de *start* realiza diferentes pasos en función del tipo de medición:

Si es una medida lenta (*Señal Slow* $\leftarrow$ 1):

1. El controlador activa la señal de start del submódulo *i_inv_2x2* que calcula la inversa de la covarianza de la medida PYY. El resultado es almacenado en la RAM *i_inv_2x2_ram*.

2. Por último multiplica la covarianza cruzada PXY por el resultado de la inversa del *paso 1* para obtener la ganancia de Kalman. El controlador activa la señal de start del submódulo *i_mult_4x2_2x2* que almacena el resultado en la RAM externa *i_k_gain_ram*.

Si es una medida rápida (*Señal Slow* $\leftarrow$ 0):

1. Divide cada elemento de la primera columna de la matriz de la covarianza cruzada PXY por el primer elemento de la covarianza de la medida PYY. Para ello, el controlador activa la señal de start del submódulo *i_div_norest*. Para acceder a cada elemento de la covarianza cruzada PXY, el controlador activa la capacitación de cuenta del submódulo *i_pxy_addr_counter* para generar las direcciones. El resultado es almacenado en la RAM externa *i_k_gain_ram*.

2. Almacena ceros en el resto de posiciones de la RAM externa *i_k_gain_ram* que no se accede en el *paso 1*.

Para acceder a cada elemento de la primera columna de la matriz de la covarianza cruzada PXY utilizamos el bloque *pxy_indexer* para acceder a la RAM a partir de generar la traspuesta de las direcciones que genera el submódulo *i_pxy_addr_counter*.

i_kalman_gain_calc_ctrl

Implementa una máquina de estados encargada de secuenciar las operaciones del módulo. En la tabla 10.3 se detallan los estados del digrama de la figura 10.5.

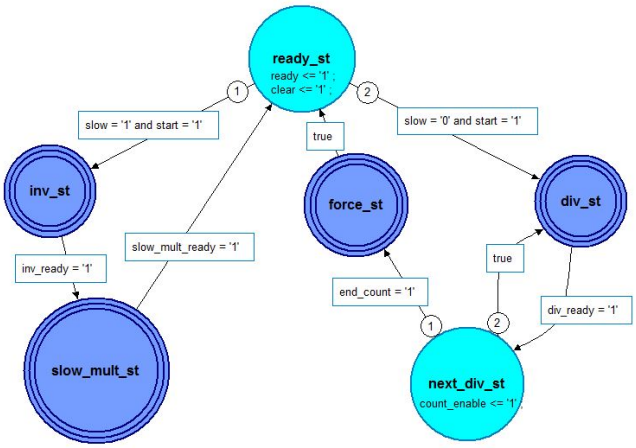

Figura 10.5: *Unidad de control: Módulo i_kalman_gain_calc*

Los estados jerárquicos *inv_st*, *slow_mult*, *div_st* siguen la estructura de estado jerárquico ya explicada anteriormente en la sección 8.1. El estado jerárquico *force_st* es diferente y está compuesto por 4 subestados (force1_st, force2_st, force3_st y force4_st) encargados de generar las direcciones y almacenar ceros en las posiciones de la RAM *i_k_gain_ram* que no se utilizan cuando la señal *Slow* $\leftarrow 0$.

Estado	Acción
ready_st	Mantiene la señal de ready → 1 que indica que el módulo está preparado para un nuevo cálculo y la señal clear → 1 del submódulo i_pxy_addr_counter para mantener la cuenta a cero. Cuando la señal start ← 1, cambia al estado inv_start_st si *slow* ← 1, por el contrario cambia al estado div_start_st.
inv_start_st	Activa la señal de start_inv → 1 del submódulo i_inv_2x2 y cambia al estado inv_wait_st.
inv_wait_st	Se mantiene en este estado hasta que el submódulo i_inv_2x2 finalice la operación (inv_ready ← 1). Una vez finalizada, cambia al estado slow_mult_start_st.
slow_mult_start_st	Activa la señal start_slow_mult → 1 del submódulo i_mult_4x2_2x2 y cambia al estado slow_mult_wait_st.
slow_mult_wait_st	Se mantiene en este estado hasta que el submódulo i_mult_4x2_2x2 finalice la operación (slow_mult_ready ← 1). Una vez finalizada, cambia al estado ready_st.
div_start_st	Activa la señal start_div → 1 del submódulo i_div_norest y cambia al estado div_wait_st.
div_wait_st	Se mantiene en este estado hasta que el submódulo i_div_norest finalice la operación (div_ready ← 1). Habilita la capacitación de escritura (wea_fast_kgain → 1) para almacenar los resultados del submódulo en la RAM externa i_k_gain_ram. Una vez finalizada, cambia al estado next_div_st.
next_div_st	Activa la capacitación de cuenta del submódulo i_pxy_addr_counter (counter_enable → 1) y cambia al estado force1_st si la señal de fin de cuenta del submódulo i_pxy_counter está activada. Por el contrario, cambia al estado div_start_st.
force1_st	Habilita la señal de capacitación de escritura en la RAM i_k_gain_ram (force_wea_kgain → 1) con dirección 1 y cambia al estado force2_st.
force2_st	Habilita la señal de capacitación de escritura en la RAM i_k_gain_ram (force_wea_kgain → 1) con dirección 3 y cambia al estado force3_st.
force3_st	Habilita la señal de capacitación de escritura en la RAM i_k_gain_ram (force_wea_kgain → 1) con dirección 5 y cambia al estado force4_st.
force4_st	Habilita la señal de capacitación de escritura en la RAM i_k_gain_ram (force_wea_kgain → 1) con dirección 7 y cambia al estado ready_st.

Tabla 10.3: Estados del controlador

10.3. Cálculo del estado a posteriori: Módulo i_state_correction

Este módulo implementa la expresión 10.3, que calcula el estado a posteriori a partir de la covarianza del estado a priori, la señal de *slow*, la ganancia de Kalman calculada en la sección 10.2 y la covarianza de la medida *PYY*. Ese módulo se compone de cinco submódulos comunes explicados en la sección 7.3, además de un sexto (el controlador *i_state_correction_ctrl*) encargado de secuenciar el inicio de ejecución de cada submódulo:

- *i_sub_2x1_2x1* (sub_2x1_2x1): Implementa una resta de matrices de dimensiones 2x1.
- *i_mult_4x2_2x1* (mult_4x2_2x1): Implementa una multiplicación de matrices de dimensiones 4x2 y 2x1 respectivamente.
- *i_adder_4x1_4x1* (adder_4x1_4x1): Implementa una suma de matrices de dimensiones 4x1.
- *i_residual_ram* (ram_generic): Implementa una memoria BRAM de 2 posiciones.
- *i_increment_ram* (ram_generic): Implementa una memoria BRAM de 4 posiciones.

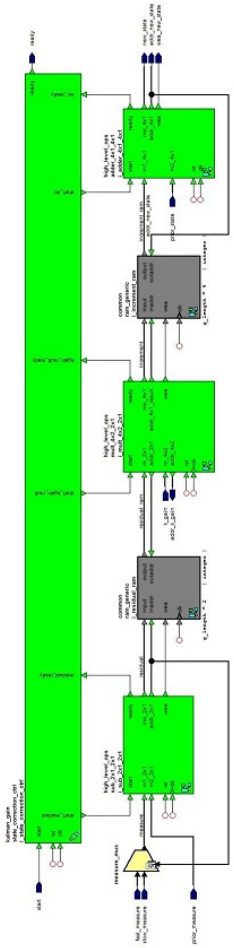

Figura 10.6: *Diagrama de bloques: Módulo i_state_correction*

La figura 10.6 muestra el diagrama de bloques de este módulo.

Cuando recibe la señal de *start* realiza los siguientes pasos secuencialmente, sin iniciar el siguiente hasta que no termine el actual:

1. Cálcula la innovación de Kalman a partir de restar la medición (rápida o lenta) y la medida a priori calculada. El controlador activa la señal de start del submódulo *i_sub_2x1_2x1* que almacena el resultado en la RAM *i_residual_ram*.

2. Multiplica los elementos de la ganancia de Kalman almacenados en al RAM externa *i_k_gain_ram* por los elementos de la innovación almacenados en la RAM *i_residual_ram*. El controlador activa la señal de start del submódulo *i_mult_4x2_2x1* que almacena el resultado en la RAM *i_increment_ram*.

3. Por último, el controlador activa la señal de start del submódulo *i_adder_4x1_4x1* para sumar los elementos de la RAM *i_increment_ram* y los elementos del estado a priori almacenados en al RAM externa *i_prior_state_ram*. El resultado es almacenado en la RAM externa *i_top_state_ram*.

i_state_correction_ctrl

Implementa una máquina de estados para secuenciar las operaciones del módulo. Los estados (*inc_st*, *residual_st* y *kgain_mult_st*) son estados jerárquicos con la misma estructura explicada en la sección 8.1. Los estados de la figura 10.7 se muestran en la tabla 10.7:

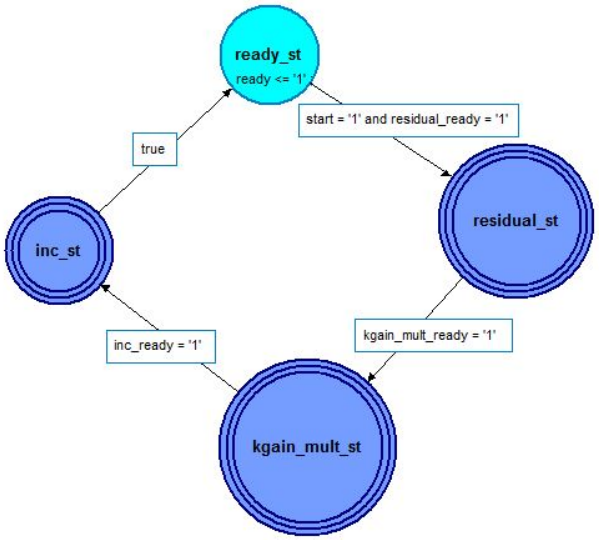

Figura 10.7: *Unidad de control: Módulo i_state_correction*

Estado	Acción
ready_st	Mantiene la señal de ready → 1 que indica que el módulo está preparado para un nuevo cálculo. Cambia al estado residual_start_st cuando la señal start ← 1.
residual_start_st	Activa la señal de start_residual → 1 del submódulo i_sub_2x1_2x1 y cambia al estado residual_wait_st.
residual_wait_st	Se mantiene en este estado hasta que el submódulo i_sub_2x1_2x1 finalice la operación (residual_ready ← 1). Una vez finalizada, cambia al estado kgain_mult_start_st.
kgain_mult_start_st	Activa la señal start_kgain_mult → 1 del submódulo i_mult_4x2_2x1 y cambia al estado kgain_mult_wait_st.
kgain_mult_wait_st	Se mantiene en este estado hasta que el submódulo i_mult_4x2_2x1 finalice la operación (kgain_mult_ready ← 1). Una vez finalizada, cambia al estado inc_start_st.
inc_start_st	Activa la señal start_inc → 1 del submódulo i_adder_4x1_4x1 y cambia al estado inc_wait_st.
inc_wait_st	Se mantiene en este estado hasta que el submódulo i_adder_4x1_4x1 finalice la operación (inc_ready ← 1). Una vez finalizada, cambia al estado ready_st.

Tabla 10.4: Estados del controlador

10.4. Cálculo de la covarianza del estado a posteriori: i_pcov_correction

Este módulo implementa la expresión 10.4 que calcula la covarianza del estado a posteriori. Es el resultado de la resta de la covarianza del estado a priori menos delta, que es la multiplicación de la Ganancia de Kalman por la covarianza de la medida por la Ganancia de Kalman transpuesta. Esta ecuación varía dependiendo de si hay medida lenta o no: Si hay medida lenta la Ganancia de Kalman es una matriz de dimensiones 4x2 y la covarianza de la medida es de dimensiones 2x2, pero si no hay medida lenta la Ganancia de Kalman es una matriz de dimensiones 4x1 y la covarianza de la medida es un único valor. La corrección por tanto queda dividida en corrección lenta, cuando hay medida lenta, y corrección rápida, cuando no hay medida lenta.

Dividimos este módulo en ocho submódulos comunes explicados en la sección 7.3, además de un noveno submódulo, el controlador i_pcov_correction_ctrl encargado de secuenciar el inicio de ejecución de cada submódulo:

- i_mult_4x2_2x2 (mult_4x2_2x2): Implementa una multiplicación de matrices de dimensiones 4x2 y 2x2, respectivamente. Se usa cuando hay medida lenta.
- i_slow_mult_ram (ram_generic): Implementa una BRAM de ocho posiciones correspondientes a una matriz de dimensiones 4x2. Son ocho posiciones porque necesitamos almacenar elementos que forman la multiplicación intermedia de la Ganancia de Kalman por la covarianza de la medida. Se usa cuando hay medida lenta.
- i_mult_4x2_2x4 (mult_4x2_2x4): Implementa una multiplicación de matrices de dimensiones 4x2 y 2x4, respectivamente. Esta multiplicación genera delta_slow, que es la corrección lenta de la covarianza del estado a priori. Se usa cuando hay medida lenta.
- i_mult_scalar_4x1 (mult_scalar_4x1): Implementa una multiplicación de un dato por una matriz de dimensiones 4x1. Se usa cuando no hay medida lenta.
- i_fast_mult_ram (ram_generic): Implementa una BRAM de cuatro posiciones correspondientes a una matriz de dimensiones 4x1. Son cuatro posiciones porque necesitamos almacenar elementos que forman la multiplicación intermedia de la primera columna de la Ganancia de Kalman por el primer valor de la covarianza de la media. Se usa cuando no hay medida lenta.
- i_mult_4x1_1x4 (mult_4x1_1x4): Implementa una multiplicación de matrices de dimensiones 4x1 y 1x4, respectivamente. Esta multiplica-

ción genera delta_fast, que es la correción rápida de la covarianza del estado a priori. Se usa cuando no hay medida lenta.

- i_delta_mult_ram (ram_generic): Implementa una BRAM de 16 posiciones correspondientes a una matriz de dimensiones 4x4. Son 16 posiciones porque necesitamos almacenar elementos que forman delta, que es la correción de la covarianza del estado a priori.

- i_sub_4x4_4x4 (sub_4x4_4x4): Implementa una resta de matrices de dimensiones 4x4.

La figura 10.8 representa el diagrama de bloques de este módulo. Las matrices a las que accede este módulo son las mismas, pero la forma de acceder no. Si hay medida lenta, el módulo recoge todos los datos de las BRAMs externa que tiene los datos de las matrices, pero si no hay medida lenta, el módulo solo lee de las BRAMs la primera columna de la Ganancia de Kalman y el primer elemento de la covarianza de la medida. Además, cuando se llega a la segunda multiplicación, se debe de acceder a la Ganancia de Kalman almacenada en uan BRAM externa de forma transpuesta. Toda esta lógica de control está descrita en los siguientes bloques:

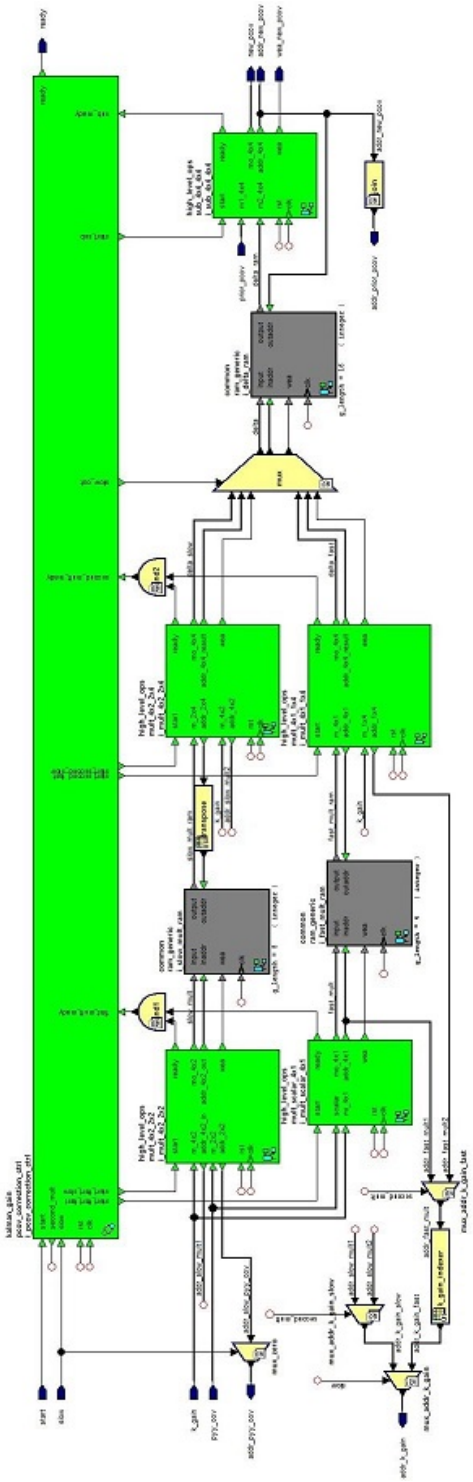

Figura 10.8: *Top-level* del módulo I_PCOV_CORRECTION

- Bloque *mux_zero*: es un multiplexor que controla la dirección de lectura la covarianza de la medida. Si hay medida lenta la dirección de lectura dicha covarianza es la que genere el bloque i_mult_4x2_2x2 que es el encargado de hacer la primera multiplicación de la correción lenta. Si no hay medida lenta entonces la dirección de lectura de la covarianza de la medida es cero para seleccionar el primer dato de dicha covarianza que será usado por i_mult_scalar_4x1 para relizar la primera multiplicación de la corrección lenta.

- Bloques *and1* y *and2*: son dos puertas AND lógicas que se encargan de indicar cuándo ha acabado la primera o segunda multiplicación al bloque controlador, respectivamente, independientemente de si se desarolla la correción lenta o rápida.

- Bloque *mux_addr_k_gain_slow*: es un multiplexor que selecciona *addr_slow_mult1* que es la dirección que genera i_mult_4x2_2x2, la primera multiplicación de la correción lenta, cuando la señal *second_mult* está a tierra o selecciona *addr_slow_mult2*, la dirección que genera i_mult_4x2_2x4, la segunda multiplicación de la correción lenta.

- Bloque *mux_addr_k_gain_fast*: es un multiplexor que selecciona *addr_fast_mult1* que es la dirección que genera i_mult_scalar_4x1, la primera multiplicación de la correción rápida cuando la señal *second_mult* está a tierra o seleccciona *addr_fast_mult2* que es la dirección que genera i_mult_4x1_1x4, la segunda multiplicación de la correción rápida.

- Bloque *k_gain_indexer*: es una tabla de verdad implementada en LUT para convertir las direcciones que generan las multiplicaciones de la corrección rápida. Es necesario porque las direcciones de los módulos i_mult_scalar_4x1 y i_mult_4x1_1x4 van de uno a cuatro mientras que para indexar la columna que necesitamos de la Ganancia de Kalman necesitamos las direcciones cero, dos, cuatro y seis, como se puede ver en la tabla 10.5.

Entrada	Salida
0	0
1	2
2	4
3	6

Tabla 10.5: Comportamiento de la tabla de verdad *k_gain_indexer*

- Bloque *mux_addr_k_gain*: es un multiplexor que selecciona si hay medida lenta la señal *addr_k_gain_slow* para que los bloques de la correción lenta tengan el control del puerto de la dirección de lectura de la BRAM externa en la que contiene la Ganancia de Kalman. Si no hay medida lenta se selecciona la señal *addr_k_gain_fast* para que los bloques de la correción rápida tengan acceso.

- Bloque *transpose*: es una tabla de verdad implementada en LUT para convertir las direcciones que genera el multiplicador i_mult_4x2_2x4. Es necesario porque tenemos que leer a la Ganancia de Kalaman de la BRAM externa de forma transpuesta para la segunda multiplicación de la correción lenta, como se puede ver en la tabla 10.6.

Entrada	Salida
0	0
1	2
2	4
3	6
4	1
5	3
6	5
7	7

Tabla 10.6: Comportamiento de la tabla de verdad *transpose*

- Bloque *mux*: es un multiplexor que selecciona qué delta se escribe en la BRAM i_delta_ram. Si la señal *slow_out* está activa deja escribir al bloque i_mult_4x2_2x4 quién se encarga de hacer la segunda multiplicación de la correción lenta. Si la señal *slow_out* no está activa deja escribir a i_mult_4x1_1x4 que se encarga de la segunda multiplicación de la correción rápida.

- Bloque *join*: es una unión entre la señal *addr_pcov_new* y *addr_prior_pcov* para que siempre tengan el mismo valor. De esta forma el restador i_sub_4x4_4x4 puede indicar la dirección de lectura a la BRAM externa que almacena la covarianza del estado a priori.

Cuando recibe la señal de *start* el módulo realiza los siguientes pasos secuencialmente, no iniciando el siguiente paso hasta que no haya finalizado el anterior:

1. El controlador manda iniciar la primera multiplicación lenta o rápida dependiendo de si hay medida lenta o no, respectivamente, indicado por la activación de la señal *slow* o no, respectivamente. También el

controlador indica que se está realizando la primera multiplicación mediante la puesta a tierra de la señal *second_ mult*.

2. El controlador espera a que haya finalizado la primera multiplicación lenta o rápida. Cuando uno de los dos primeros multiplicadores vuelve a estar ocioso, la *and1* indica al controlador que avance a la segunda multiplicación. Entonces activa la segunda multiplicación con la activación de *second_ mult* y el módulo al que le toque encargarse de hacer la segunda multiplicación, dependiendo de si hay medida lenta o no (activación de la señal de *slow* o no, respectivamente). Además el controlador activa la señal *slow_ out* que se encargará de dejar escribir en la ram a la multiplicación correspondiente.

3. Finalmente el controlador activa la resta final cuando una de las dos multiplicaciones activadas anteriormente vuelve a estar ociosa.

10.5. Unidad de control: i_kalman_gain_ctrl

Implementa una máquina de estados que secuencia las operaciones del módulo. Los estados (*pxypcov_ estimation_ st*, *kgain_ calc_ st*, *state_ correction_ st* y *pcov_ correction_ st*) son estados jerárquicos que siguen la misma estructura explicada en la sección 8.1. En la tabla 10.7 se detallan los estados del diagrama de la figura 10.9.

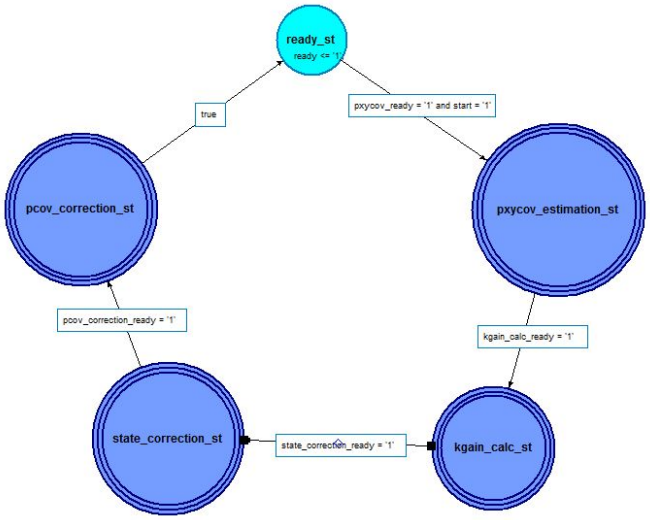

Figura 10.9: *Unidad de control: Módulo Kalman Gain*

Estado	Acción
ready_st	Mantiene la señal de ready → 1 que indica que el módulo está listo para una nueva iteración. Cambia al estado pxycov_estimation_start_st cuando la señal start ← 1.
pxycov_estimation_start_st	Activa la señal de start_pxycov → 1 del submódulo i_pxycov_estimation y cambia al estado pxycov_estimation_wait_st.
pxycov_estimation_wait_st	Se mantiene en este estado hasta que el submódulo i_pxycov_estimation finalice la operación (pxycov_ready ← 1). Una vez finalizada, cambia al estado kgain_calc_start_st.
kgain_calc_start_st	Activa la señal start_kgain_calc → 1 del submódulo i_kalman_gain_calc y cambia al estado kgain_calc_wait_st.
kgain_calc_wait_st	Se mantiene en este estado hasta que el submódulo i_kalman_gain_calc finalice la operación (kgain_calc_ready ← 1). Una vez finalizada, cambia al estado state_correction_start_st.
state_correction_start_st	Activa la señal start_state_correction → 1 del submódulo i_state_correction y cambia al estado state_correction_wait_st.

state_correction_wait_st	Se mantiene en este estado hasta que el submódulo i_state_correction finalice la operación (state_correction_ready ← 1) y selecciona la entrada del multiplexor addr_mux1 (addr_prior_state_ctrl → 1). Una vez finalizada, cambia al estado pcov_correction_start_st.
pcov_correction_start_st	Activa la señal start_pcov_correction → 1 del submódulo i_pcov_correction y cambia al estado pcov_correction_wait_st.
pcov_correction_wait_st	Se mantiene en este estado hasta que el submódulo i_pcov_correction finalice la operación (pcov_correction_ready ← 1), selecciona la entrada del multiplexor adder_mux2 (addr_pyy_cov_ctrl → 1) y selecciona la entrada del multiplexor addr_mux3 (addr_k_gain_ctrl → 1). Una vez finalizada, cambia al estado ready_st.

Tabla 10.7: Estados del controlador

Parte IV

Resultados experimentales, síntesis del hardware y conclusiones

Capítulo 11

Resultados experimentales

Los objetivos de este capítulo es comprobar que el estado oculto del filtro es capaz de estimar la glucosa en sangre y la ganancia del sensor usando medidas indirectas que provienen del sensor de glucosa situado en el tejido intersticial y comprobar que el hardware está correctamente implementado.

Para cumplir estos objetivos hemos explicado y analizado los resultados experimentales obtenidos en el trabajo. Para ello hemos realizado tres pruebas usando Matlab y otra usando el hardware descrito por el VHDL. Hemos generado la glucosa y la ganancia del paciente de forma sintética y hemos medido el error con la media de la diferencia entre la señal sintética y el estado oculto de la prueba software o hardware.

Hemos dividido los experimentos por el tipo de entrada generada. Veremos los resultados de las siguientes pruebas en Matlab:

- Glucosa lineal creciente y ganancia lineal decreciente.
- Glucosa lineal creciente y ganancia exponencial decreciente.
- Glucosa logarítmica creciente y ganancia exponencial decreciente.

En el hardware solo hemos realizado una prueba con entradas sintéticas de glucosa lineal creciente y ganancia exponencial decreciente.

Hemos usando en todas las pruebas un ruido gaussiano de media cero y covarianza 1 (figura 11.1) excepto en el caso de la glucosa logarítmica y la ganancia exponencial cuyo ruido es gaussiano de media cero y covarianza 0.25 (figura 11.2).

La medida rápida que suministramos es la emulación del Monitor Continuo de Glucosa, por tanto es igual a la multiplicación de la ganancia del sensor por la glucosa en ese instante mas el ruido de la medición rápida por ser realizada por un sistema electrónico. Introducimos la medida rápida en el filtro cada cinco minutos. La medida lenta es directamente la glucosa en sangre, aunque solo hacemos una medición cada ocho horas.

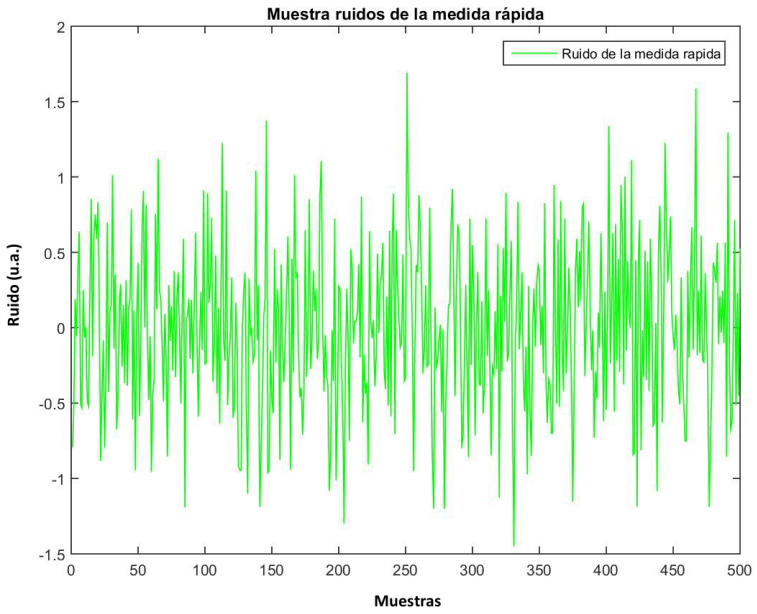

Figura 11.1: Ruido gaussiano de media cero y covarianza 1

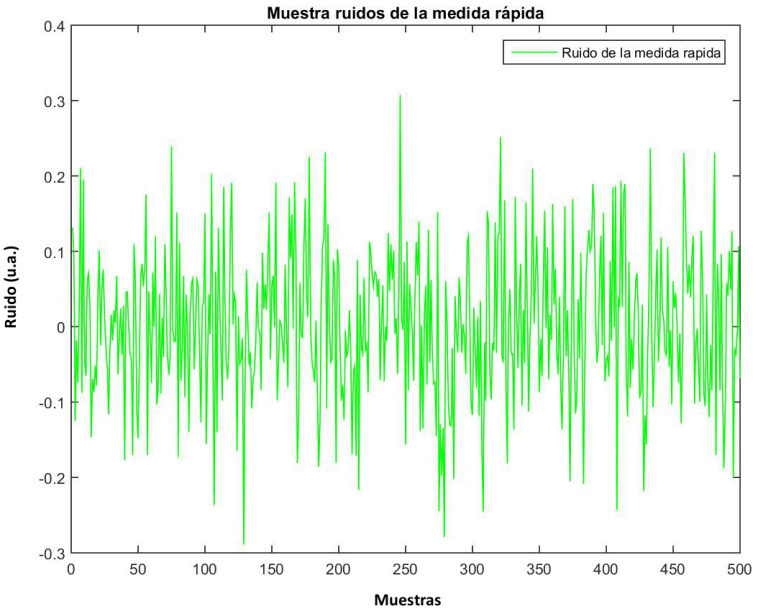

Figura 11.2: Ruido gaussiano de media cero y covarianza 0.25

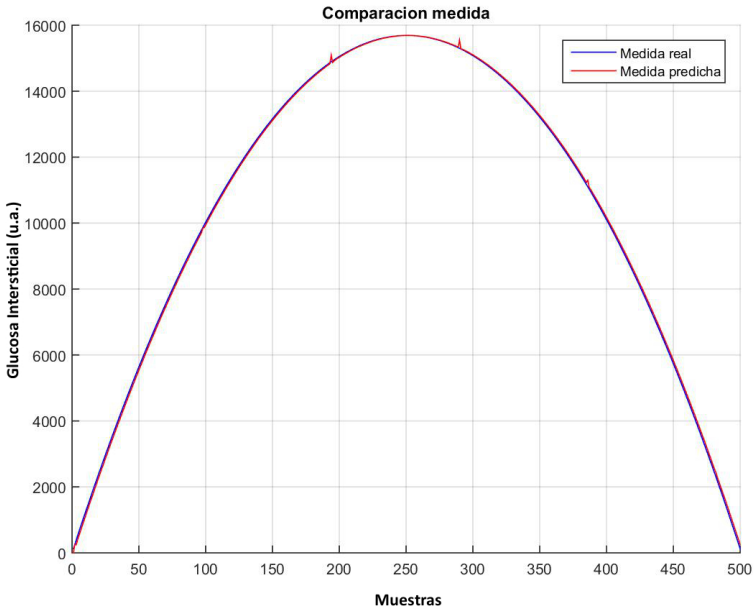

Figura 11.3: Comparación de la glucosa intersticial medida con la predicha por el filtro en la primera prueba del Matlab.

11.1. Primera prueba en Matlab

En esta prueba experimentaremos con glucosa lineal creciente y ganancia lineal decreciente. No suele ser la dinámica normal de la glucosa y de la ganancia pero siguen tendencias parecidas a las reales. La glucosa tiende a aumentar por falta de insulina y la ganancia tiende a disminuir por ser la degradación del sensor.

La medida generada y suministrada al sistema viene dada por la figura 11.3. En ella podemos ver que es el producto de la ganancia y la glucosa (no vemos el ruido porque es pequeño comparado con la señal) y también aparece la medida predicha por el filtro. El filtro es capaz de predecir bien la medida, aunque en algunos casos podemos ver una ligera desviación. Para comprobar el correcto funcionamiento hemos calculado la media del error y la media de la señal y el porcentaje de error medio es 0.825157 %.

La glucosa del paciente y la glucosa aproximada por el filtro la podemos ver en la figura 11.4, en la que casi no apreciamos diferencia entre las dos señales. Aun así, Matlab calcula que hay una diferencia media de 0.676451, lo que es un 0.540081 % de la media de la señal. En la gráfica de la ganancia (figura 11.5) podemos ver una ligera desviación pero no sobrepasa el 0.745592 % de la señal.

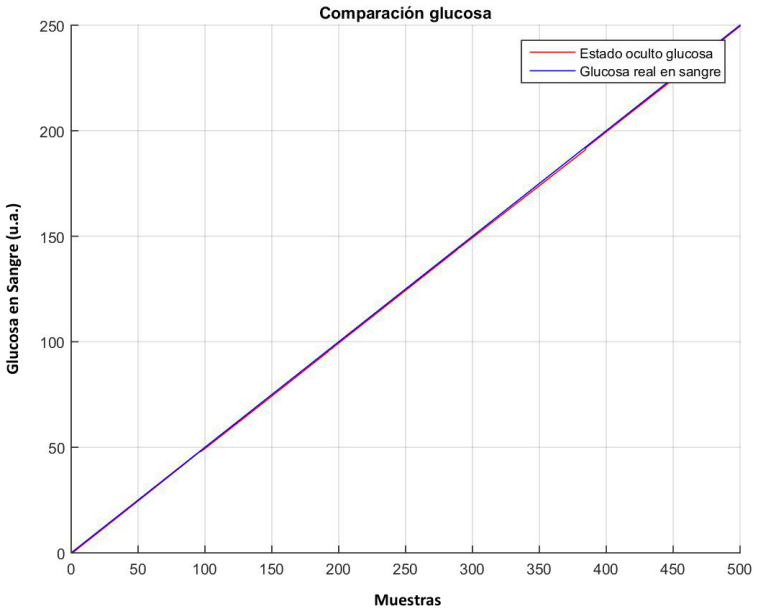

Figura 11.4: Comparación de la glucosa en sangre (estado oculto) con la predicha por el filtro en la primera prueba del Matlab.

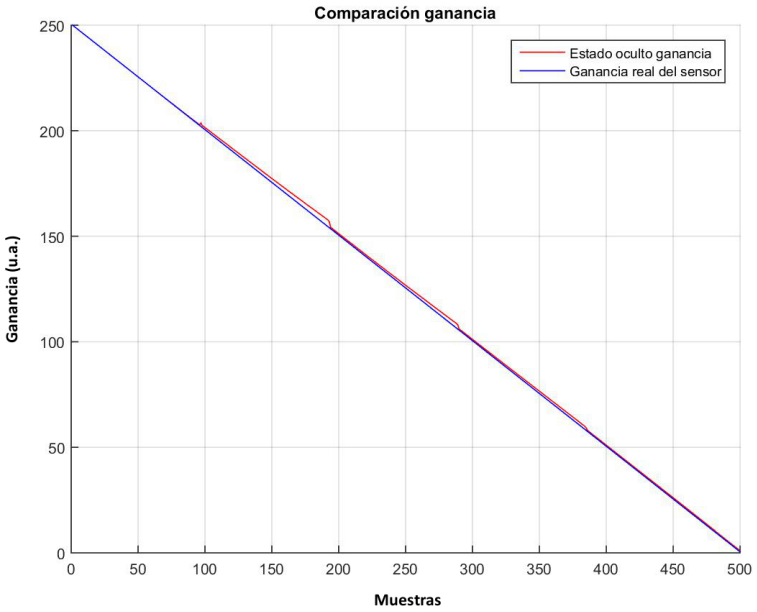

Figura 11.5: Comparación de la ganancia del sensor (estado oculto) con la predicha por el filtro en la primera prueba del Matlab.

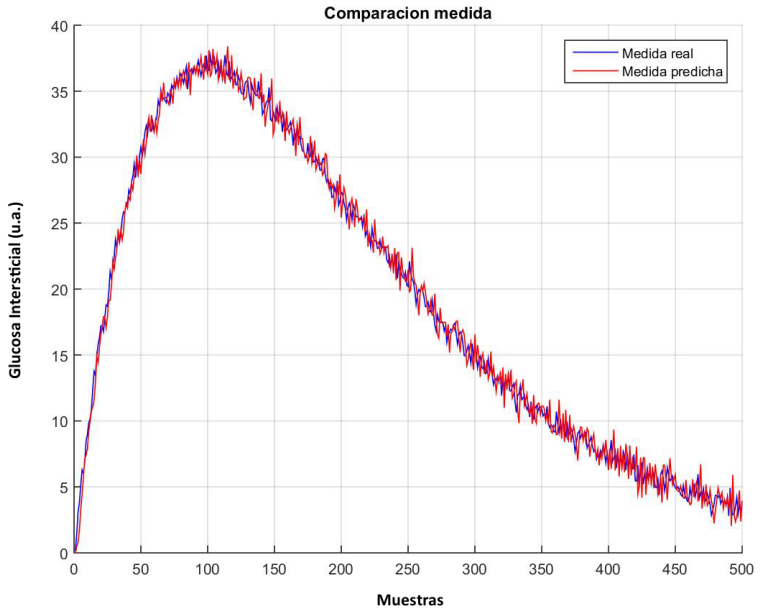

Figura 11.6: Comparación de la glucosa intersticial medida con la predicha por el filtro en la segunda prueba del Matlab.

11.2. Segunda prueba en Matlab

Viendo los anteriores resultados el filtro se comporta muy bien con sistemas lineales. Ahora probaremos con sistemas de distinta dinámica, como puede ser la función exponencial. En este caso, la glucosa es lineal creciente pero la ganancia es exponencial decreciente según $y = e^{-0,01k}$ donde $1 \leq k \leq 500$. Hemos cambiado el modelo de proceso para adecuarlo a la ganancia exponencial usando la ecuación (4.7) que vimos anteriormente.

Podemos observar la comparación de la medida, la glucosa y la ganancia en las gráficas 11.6, 11.7 y 11.8, respectivamente. El porcentaje de error del filtro es 3.376017 %, 0.506838 % y 4.152685 % en la medida, glucosa y ganancia respectivamente. Esta prueba es mucho más interesante que la anterior ya que hemos conseguido aproximar la glucosa lineal y la ganancia exponencial de una señal ruidosa y no lineal.

La Transformación Unscented consigue estos resultados gracias a la generación de los puntos sigma que son intentos de predicción de la señal, usando las covarianzas del proceso a priori, inicial o corregida. Al no afectar a la media ni a la covarianza del estado oculto, cuando realizamos la media pesada conseguimos una predicción que mantiene la dinámica del sistema.

Se puede apreciar un poco de dificultad por parte del filtro para conseguir

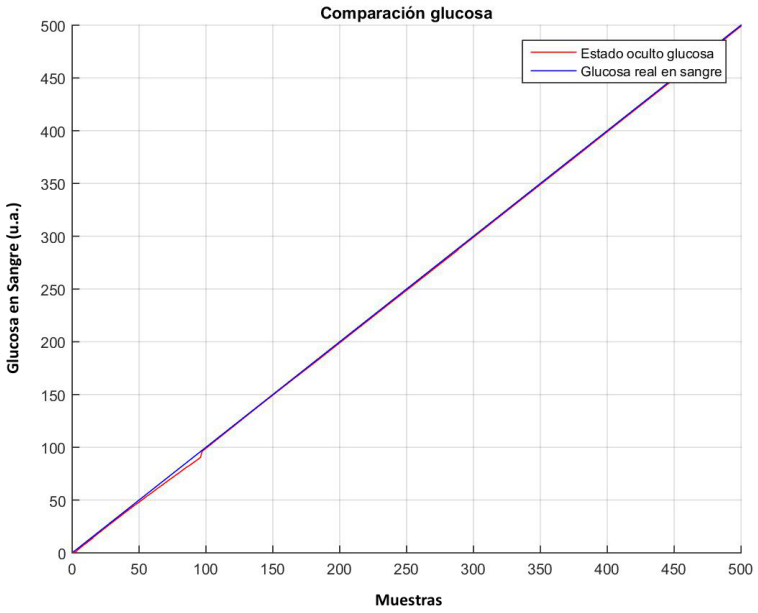

Figura 11.7: Comparación de la glucosa en sangre (estado oculto) con la predicha por el filtro en la segunda prueba del Matlab.

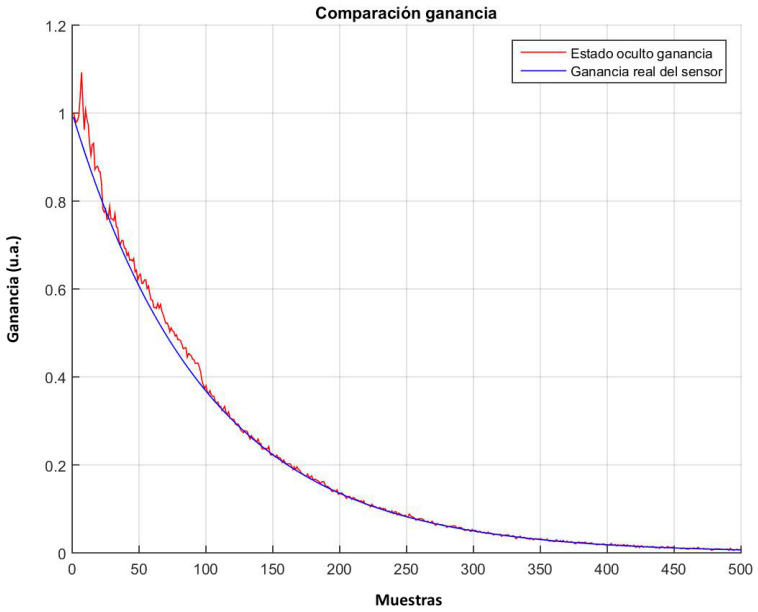

Figura 11.8: Comparación de la ganancia del sensor (estado oculto) con la predicha por el filtro en la segunda prueba del Matlab.

igualar la ganancia, pero es debido a las covarianzas de los ruidos y el estado inicial. Hemos usado valores sencillos ya que esta prueba es la que usaremos para testear el hardware.

11.3. Tercera prueba en Matlab

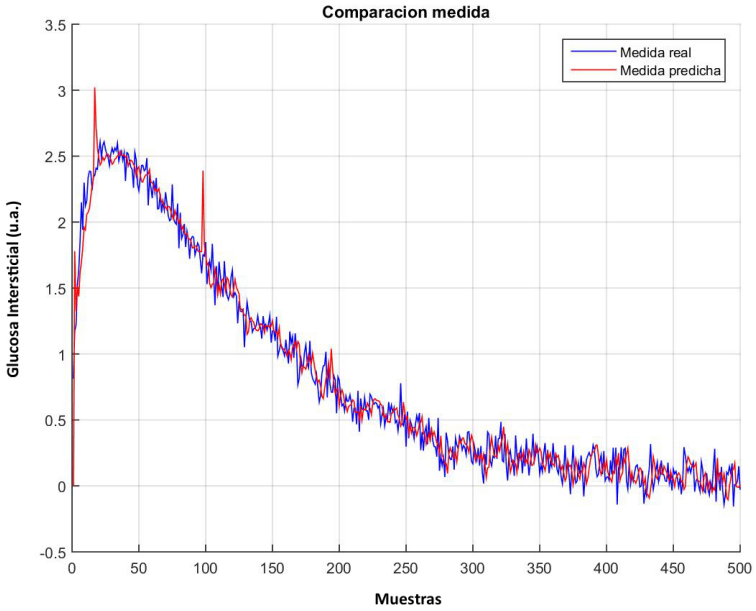

Figura 11.9: Comparación de la glucosa intersticial medida con la predicha por el filtro en la tercera prueba del Matlab.

Realizamos una prueba más para ver el comportamiento del filtro con glucosa logarítmica según $y = log(k)$ donde $2 \leq k \leq 501$ y la ganancia exponencial del caso anterior. Después de la ejecución del filtro podemos observar que la respuesta ya no es tan buena.

La medida predicha por el filtro está desviada como podemos ver en la figura 11.9. En la figura 11.10 la glucosa no consigue emular a la dinámica de la señal real. Podemos ver claramente que en el momento de la medida lenta el filtro se sintoniza, pero rápidamente vuelve a desviarse. La ganancia, como se puede ver en la figura 11.11, también se desvía bastante aunque en las últimas iteraciones se consigue adaptar la covarianza de proceso y llega a estimar bien la salida a la señal real.

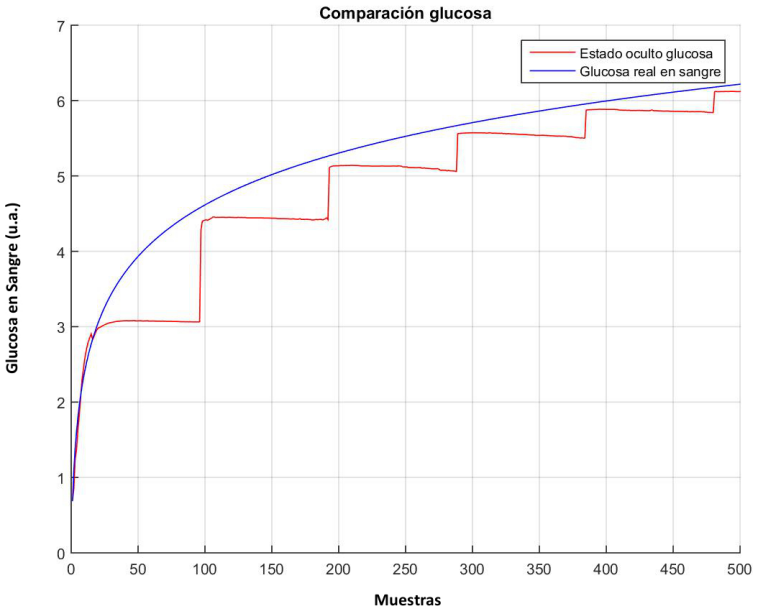

Figura 11.10: Comparación de la glucosa en sangre (estado oculto) con la predicha por el filtro en la tercera prueba del Matlab.

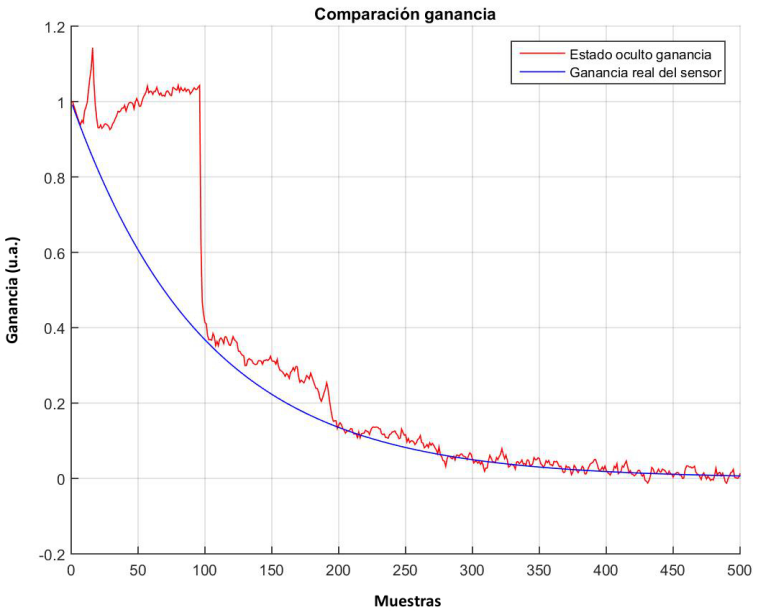

Figura 11.11: Comparación de la ganancia del sensor (estado oculto) con la predicha por el filtro en la tercera prueba del Matlab.

Los resultados de este caso no son buenos. El filtro comete una media de error de 7.893071 %, 45.520496 % y 24.027478 % en glucosa, ganancia y medida predicha. Es inviable esta señal. Hemos tocado las covarianzas y las relación de ruido de proceso y ruido de medida y no ha habido forma de controlar la señal. Probablemente nuestro modelo no sea tan bueno para sistemas de este tipo. También hay que pensar que el ruido es alto y que la señal es pequeña, lo que dificulta su filtrado. Aun así, está claro que no hemos sido capaces de hacer que el filtro funcione para ganancia y glucosa no lineal.

11.4. Prueba en hardware

Terminada esta parte de las pruebas queda comprobar si el hardware funciona como esperamos. Por temas de afinamiento de las covarianzas del ruido del proceso, ruido de la medida, etc. solo hemos realizado en Matlab de punto fijo y en hardware una prueba con glucosa lineal creciente y ganancia exponencial decreciente.

Hemos usado los datos de la segunda prueba en Matlab que alimentarán el testbench en VHDL. La figura 11.12 muestra cómo hemos realizado la prueba. En un proceso leemos de los ficheros generados con el Matlab las entradas del filtro y se las suministramos por los puertos pertinentes. También hemos cambiado las constantes del filtro entre las que encontramos los ruidos de proceso y de medida y la covarianza inicial del estado. Cuando el filtro genera un nuevo dato, el test lo guarda en un nuevo fichero y le suministra la siguiente entrada.

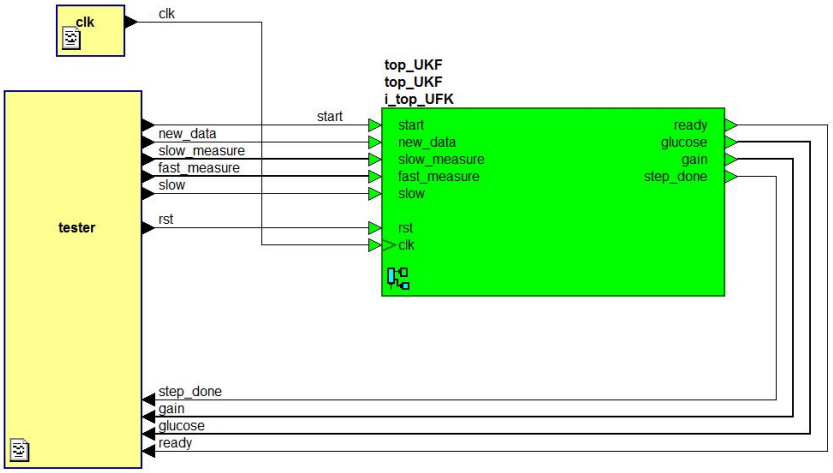

Figura 11.12: Testbench en VHDL del top del diseño.

Usando los archivos generados por el test en VHDL mediante un script en Matlab comprobamos el resultado con la respuesta en punto fijo generada por el Matlab y la respuesta ideal. Las gráficas que muestran la comparación en punto fijo son 11.13 y 11.14. Las gráficas que muestran la comparación con la señal ideal son 11.15 y 11.16.

Los resultados del VHDL aparentan ser exactos pero difieren en un 0.000098 % de la señal en la glucosa y un 0.000448 % de la señal en la ganancia. Esto se debe a que en el Matlab no hemos incluido un script que represente la raíz cuadrada y la división. La división es la operación que más error genera, en torno a la tercera cifra decimal, pero es fácilmente solucionable con un Core de Xilinx llamado Divider Generator v5.1.

Conseguimos paliar el error de la división por la cantidad de operaciones posteriores y la acción del propio filtro, ya que hemos englobado ese error como ruido de proceso. Por ello, hemos modificado la covarianza del proceso un poco a partir de la prueba en punto fijo de Matlab.

Finalmente los resultados del hardware comparados con la señal real es muy acertada. El error medio de la aproximación es 0.001497 % en la glucosa y un 0.034880 % en la ganancia. Comparado con la respuesta de la segunda prueba en Matlab de punto flotante de la sección 11.2 el resultado es mucho mejor. Podemos apreciar que las dos señales están mucho mejor acopladas.

Esto sucede por el segundo afinamiento de las covarianzas y por que el error del hardware ha jugado a nuestro favor. Aun así, podemos asegurar que el error nunca superará el 0.5 % en la glucosa y el 5 % en la ganancia.

Entre los resultados del hardware se tomamos cuenta también el tiempo de ejecución. En nuestro caso, gracias a todo el ahorro de hardware por usar máquinas de estados finitos en operaciones básicas en vez de operaciones en paralelo, el tiempo que toma el filtro para hacer quinientas iteraciones ronda los 160 ms. Como la medida rápida tiene una frecuencia de cinco minutos hay tiempo de sobra.

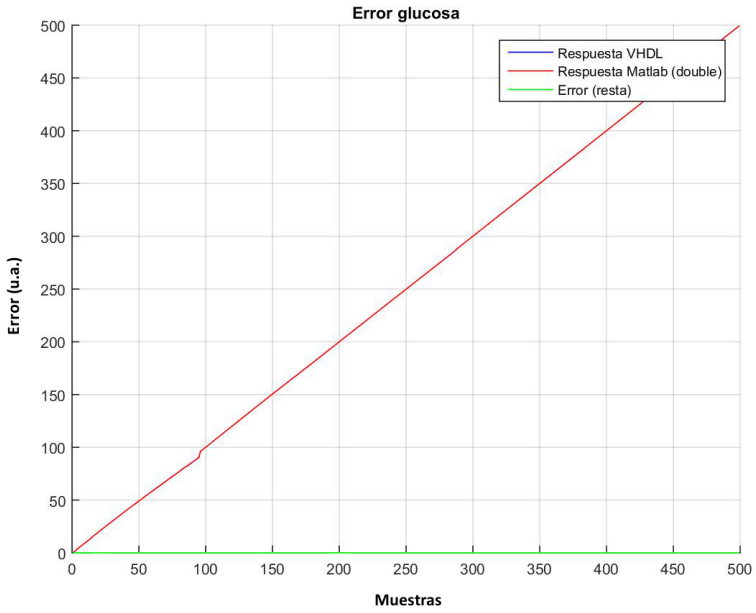

Figura 11.13: Comparación de la glucosa en sangre (estado oculto) del VHDL con la glucosa del Matlab en punto fijo.

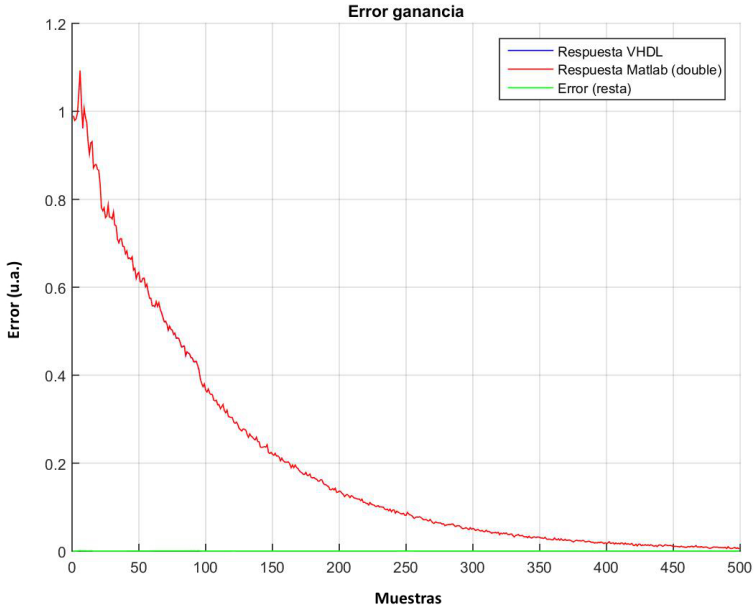

Figura 11.14: Comparación de la ganancia del sensor (estado oculto) del VHDL con la glucosa del Matlab en punto fijo.

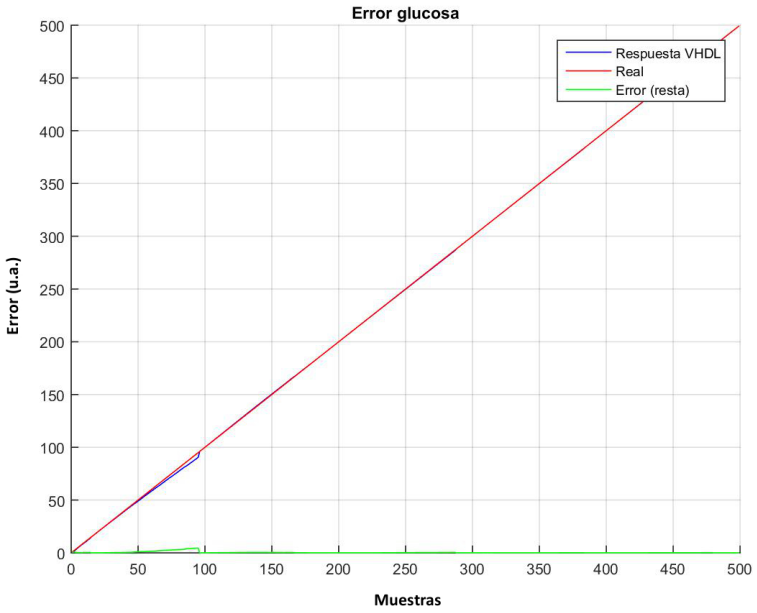

Figura 11.15: Comparación de la glucosa en sangre (estado oculto) del VHDL con la glucosa del Matlab en punto flotante.

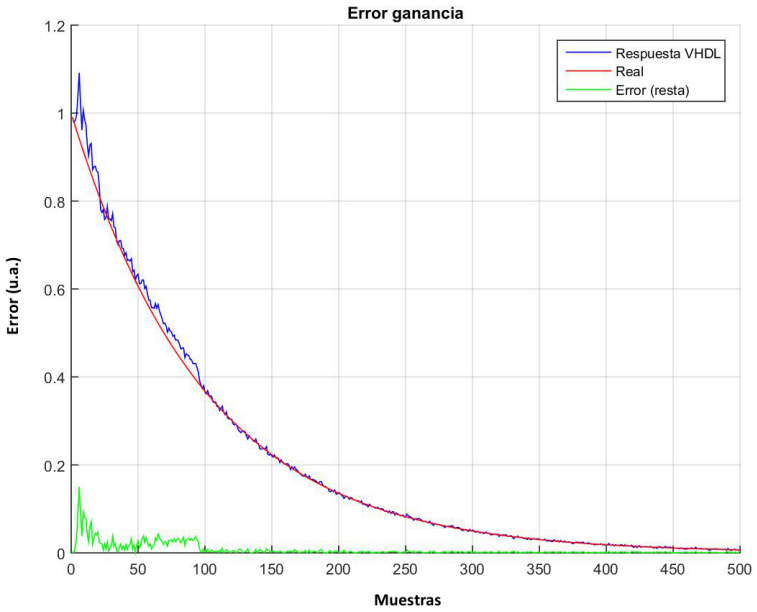

Figura 11.16: Comparación de la ganancia del sensor (estado oculto) del VHDL con la glucosa del Matlab en punto flotante.

Capítulo 12

Síntesis del hardware

La implementación de este trabajo ha sido sintetizada para una Xilinx Virtex 6 6vlx240t ff1156 con velocidad -1. El informe de síntesis muestra los siguientes parámetros del filtro:

- Periodo mínimo: 25.398 ns (frecuencia máxima: 39.373 MHz).
- Tiempo mínimo de llegada de las entradas antes del reloj: 1.866 ns.
- Tiempo máximo necesario para las salidas después del reloj: 1.492 ns.
- Retardo del camino combinacional máximo: No encontrado.

Además de este análisis la síntesis del filtro nos muestra el uso de los distintos circuitos integrados en la FPGA. En la tabla 12.1 se ven detallados antes de la optimización, lo que coincide con lo que se ha descrito. Lo único que puede sorprender es que se usen 2059 registros, pero esto es posible debido a que las rams contienen registros a la salida donde almacenan el dato de la lectura, además de todos los bits necesarios para almacenar los estados de las FSM.

Recurso	**Cantidad**
RAMs	91
Multiplexores	23
Sumadores / restadores	92
Contadores	36
Flip Flops	2059
Comparadores	43
Registros de desplazamiento de 47 bits	8
FSM	60
Xors	215

Tabla 12.1: Recursos de la FPGA usados por el filtro antes de la optimización del circuito.

Slice Logic Utilization:				
Number of Slice Registers:	2526	out of	301440	0 %
Number of Slice LUTs:	15219	out of	150720	10 %
Number used as Logic:	15219	out of	150720	10 %
Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	16784			
Number with an unused Flip Flop:	14258	out of	16784	84 %
Number with an unused LUT:	1565	out of	16784	9 %
Number of fully used LUT-FF pairs:	961	out of	16784	5 %
Number of unique control sets:	227			
IO Utilization:				
Number of IOs:	199			
Number of bonded IOBs:	199	out of	600	33 %
IOB Flip Flops/Latches:	2			
Specific Feature Utilization:				
Number of Block RAM/FIFO:	48	out of	416	11 %
Number using Block RAM only:	48			
Number of BUFG/BUFGCTRLs:	1	out of	32	3 %
Number of DSP48E1s:	243	out of	768	31 %

Tabla 12.2: Recursos de la FPGA a bajo nivel.

Finalmente, los recursos usados por nuestro sistema en la FPGA a bajo nivel son los que mostramos en la tabla 12.2. En este último reporte destacamos que usamos muchas mas LUT que Flip Flops. Podríamos haber balanceado un poco más el diseño usando más contadores y operaciones al formar las direcciones de las RAMs pero el porcentaje de uso es del 10 % lo que no es muy abultado. También destaca que todos los slices han sido usados con lógica, cosa que hemos logrado gracias a la mejora de la descripción en VHDL a lo largo de las versiones del trabajo. El filtro completo necesita de 43.284 bits de RAM, que redondeando la cuenta asciende a los 42.27 KB de memoria ram.

12.1. Camino crítico

El camino crítico es la ruta más lenta entre dos registros contando con el tiempo de set-up y hold de los registros, el skew del reloj, etc.. Llega a ser un cáculo muy complejo. Es una especificación muy importante ya que determina el tiempo de ciclo y la frecuencia de trabajo del sistema. Para hallarlo hemos usado el sintetizador ISE de Xilinx, que además de el reporte se la sección anterior, descubre el camino crítico.

Para la visualización del camino crítico hemos marcado en rojo la ruta a través de los cinco niveles de jerarquía que atraviesa. En la figura 12.1 vemos que el camino crítico se empieza en la ROM que almacena la covarianza inicial del estado oculto que se encuentra en el bloque *init_pcov_rom* y que después se introduce en el módulo *i_UKF* por el puerto de entrada de la covarianza.

Después en la figura 12.2 vemos que directamente la señal se dirige al siguiente bloque, al igual que en la figura 12.3.

En la figura 12.4 el camino crítico entra en el divisor *i_cholesky_sigma* por el puerto *input*. Siguiendo en este nivel se ve el final de la ruta, que es en la BRAM de *i_ram_chol_sigma* después de haber salido del puerto *output* de *i_cholesky_sigma* y atravesado el multiplexor *mux_wea*.

Para finalizar, la ruta del camino crítico que atraviesa el módulo *i_cholesky_sigma* se puede ver en la figura *cc5*. Empieza en el puerto *input* y después de pasar por *op2_in* y *output_mult*, que son afectados por muchos multiplexores y lógica, llega al puerto de salida *output*.

El camino crítico toma la ruta antes descrita porque Cholesky es la operación más lenta que hemos descrito en el filtro. Contiene mucha lógica combinacional llena de multiplexores, lo que realentiza mucho la respuesta del sistema.

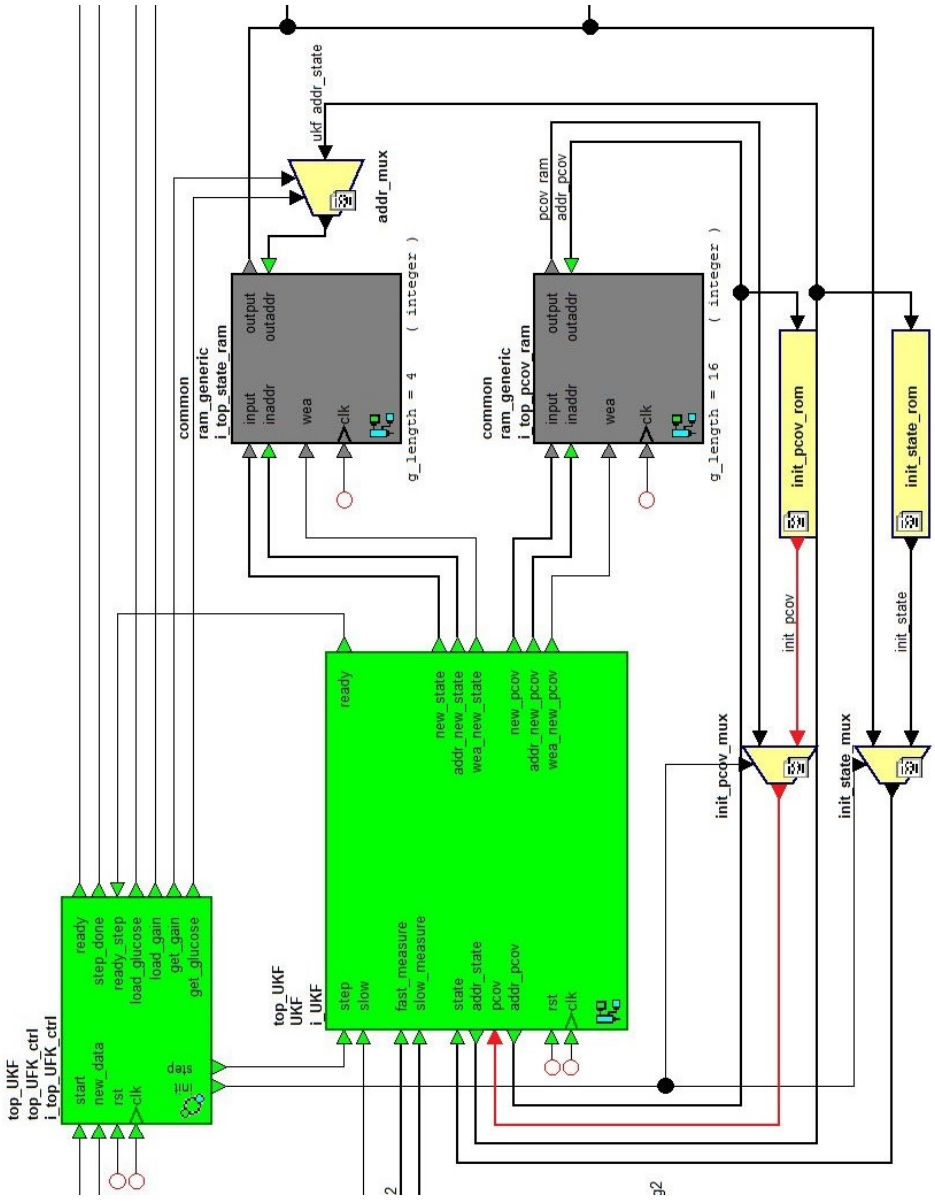

Figura 12.1: Primer nivel del camino crítico.

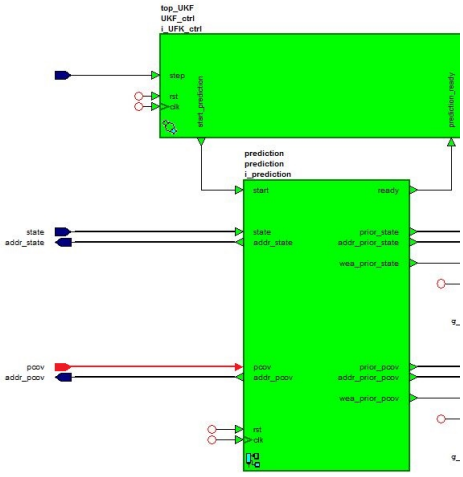

Figura 12.2: Segundo nivel del camino crítico.

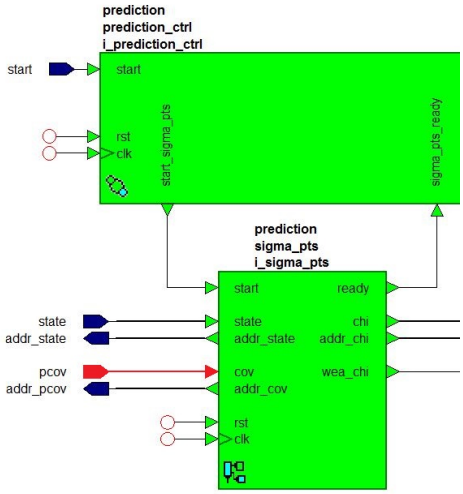

Figura 12.3: Tercer nivel del camino crítico.

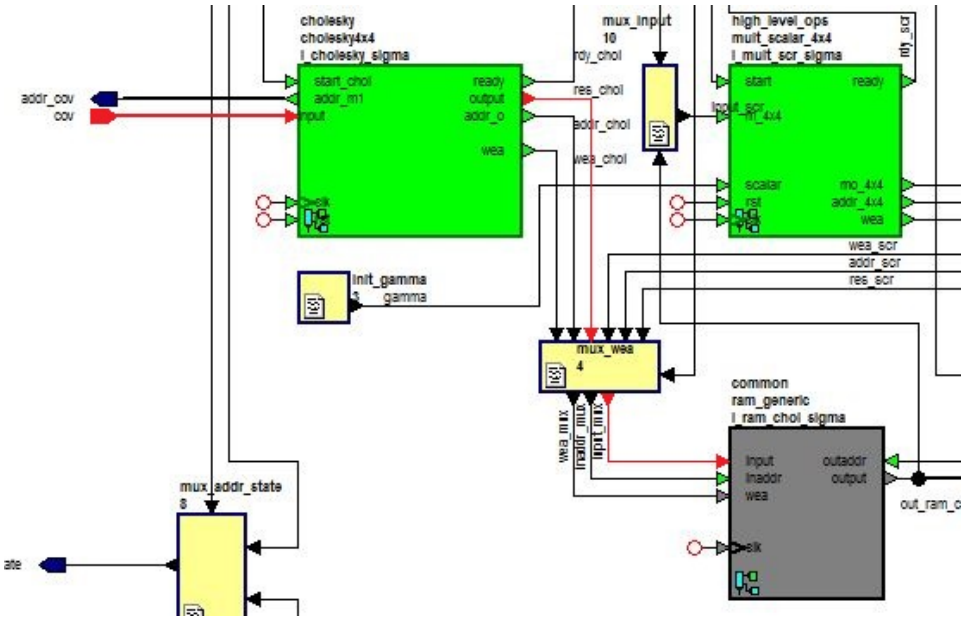

Figura 12.4: Cuarto nivel del camino crítico.

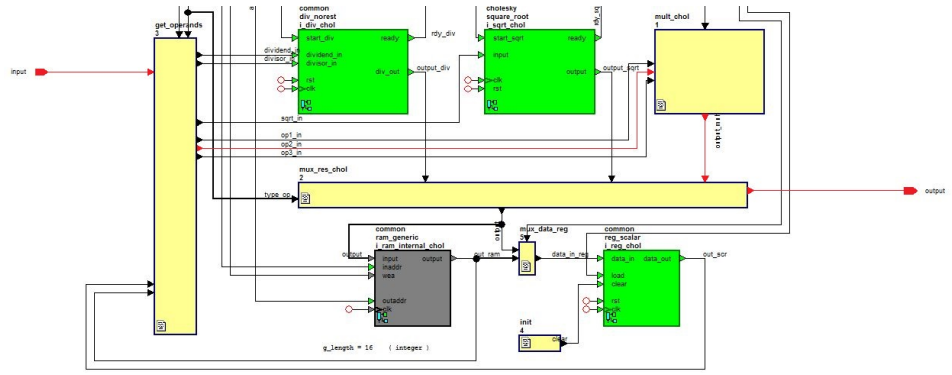

Figura 12.5: Quinto nivel del camino crítico.

Capítulo 13

Conclusiones

13.1. Conclusiones

En este trabajo hemos conseguido varios objetivos:

- Hemos aprendido sobre los Filtros Kalman y en especial sobre el Filtro Kalman Unscented. Hemos buceado en sus ecuaciones y comprendido sus puntos clave más útiles, tales como la aproximación de sistemas fuertemente no lineales y la estimación del estado oculto.
- Hemos estudiado el modelo de Kinsey para el caso de Diabetes Mellitus. Hemos comprendido su funcionamiento, sus ecuaciones, sus trucos para modelar señales no lineales y su facultad especial que le distingue de otros modelos: la doble medida, rápida y lenta.
- Hemos conocido el fundamento del punto fijo, indispensable en sistemas discretos y empotrados. Conocemos que sus características principales son la velocidad de cómputo y la simplicidad comparados con sistemas en punto flotante, más precisos pero más lentos. También sabemos cuánto error es capaz de generar el punto fijo y cómo conseguir que un sistema funcione sin sobrepasar un error máximo.
- Hemos desarrollado scripts en Matlab capaces de simular el filtro y de ayudarnos a comprenderlo. También el Matlab nos ha ayudado a diseñar el hardware del filtro, a generar datos para los test y comprobar los errores entre el punto flotante, el punto fijo y el VHDL.
- Hemos aprendido a desarrollar VHDL en grupo, a controlar las versiones con GitHub, a realizar simulaciones con Questa Sim, a generar scripts de Questa Sim para formar ventanas de visualización de ondas y a generar documentos con LaTeX.
- Y finalmente, hemos conseguido un hardware implementado en una FPGA capaz de realizar 500 iteraciones de un Filtro Kalman Unscen-

ted en menos de 200 ms y con una media de error en glucosa y ganancia menor que el 0.5 % y 5 % de la señal, respectivamente. Además hemos estudiado su comportamiento y es satisfactorio para entradas fuertemente no lineales a excepción de glucosa y ganancia no lineales.

13.2. Conclusions

We have reached several targets in this work:

- We have learnt about Kalman Filtering and about Unscented Kalman Filtering specially. We have explored into it equations and understood it key points, like non linear system approximation and hidden state estimation.
- We have studied the Kinsey Diabetes Mellitus model. We have understood it operation, equations, non linear modeling tricks and it most important feature: it double-measure, fast and slow, capability.
- We have seen the fixed point functionalities, essential for discrete and embedded systems. Now we know that it most important features is the compute speed and simplicity compared to floating point systems, much more accurate but much more slow too. We know how many error fixed point generates and how to get an reliable system too.
- We have developed Matlab scripts for filter simulating and understanding. Matlab had helped us to design the filter hardware, to data test generation and to manage error verification between floating point, fixed point and VHDL too.
- We have learnt to develop VHDL in group, to manage file versions with GitHub, to simulate hardware with Questa Sim, to Questa Sim script generation for manage waveforms windows and to LaTeX documents generation.
- And finally, we have reach an FPGA hardware implementation that is capable of compute 500 Unscented Kalman Filter iterations in less than 200 ms that generates an glucose and gain error media less than the 0.5 % and 5 % of the signal, respectively. We have studied it behavior and we can say that is good for strong non linear entries except non linear glucose and gain.

Parte V

Apéndices

Apéndice A

Análisis del hardware necesario para implementar el filtro

En estas páginas mostramos cómo ha sido la inferencia de operaciones básicas antes de la implementación del filtro en VHDL. De esta forma evitamos riesgos de diseño.

El diagrama usa la siguiente representación:

- Los colores indican la etapa en la que nos encontramos: el color azul representa la etapa de predicción, el color rojo es la etapa de correción lenta y el color verde es la etapa de corrección rápida.
- Los recuadros representan una ecuación de cada etapa. Si el recuadro es de puntitos indica una acción de control.
- Las flechas indican el cambio de una ecuación a otra.

Esta fue una versión preeliminar de las operaciones que nos ayudó a la implementación final. No está actualizada debido a que en versiones posteriores del filtro las ecuaciones no necesitaban más hardware adicional, aunque variaran un poco respecto a la versión anterior.

Podemos ver las inferencias de la implementación de una vía en las figuras A.1, A.2 y A.3. Las de la versión de dos vías las podemos encontrar en las figuras A.4, A.5 y A.6.

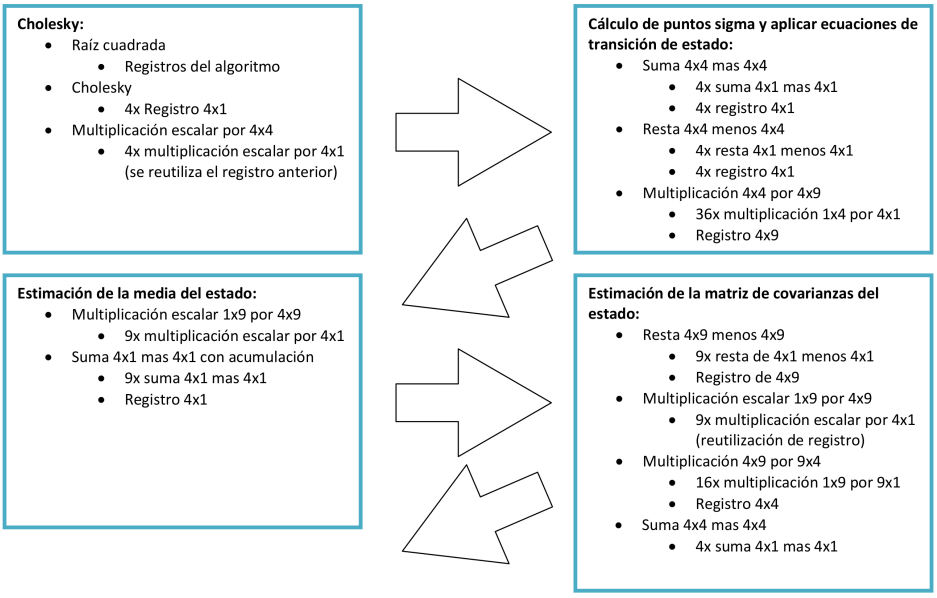

Figura A.1: Inferencia de operaciones básicas del filtro en la implementación de una vía (I).

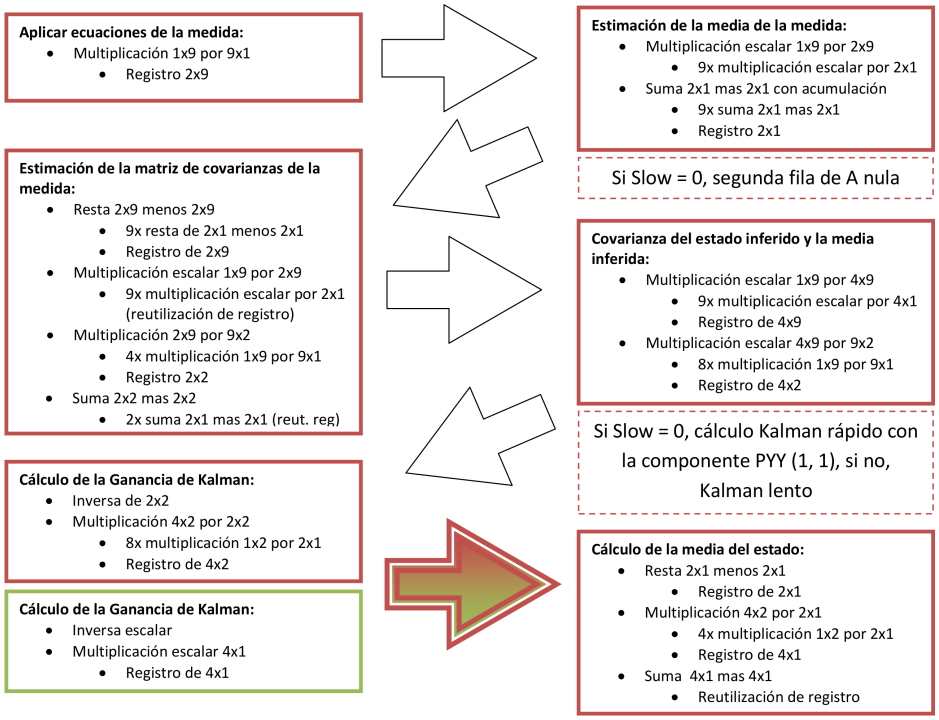

Figura A.2: Inferencia de operaciones básicas del filtro en la implementación de una vía (II).

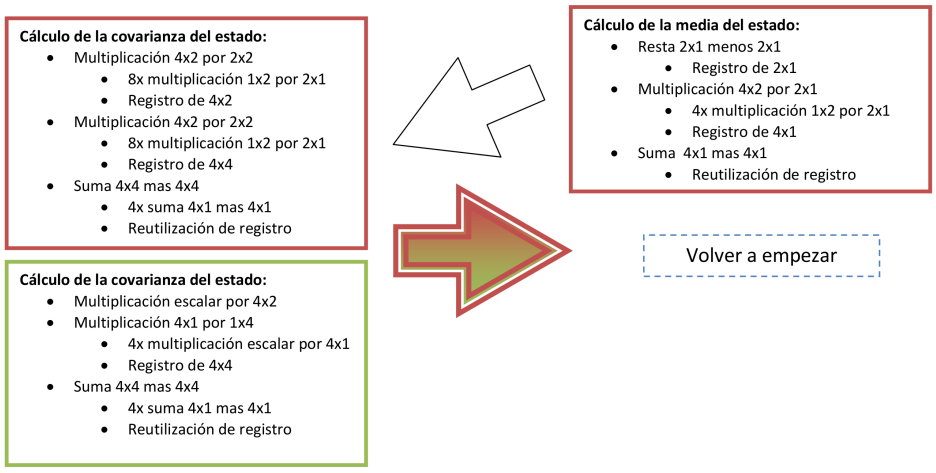

Figura A.3: Inferencia de operaciones básicas del filtro en la implementación de una vía (III).

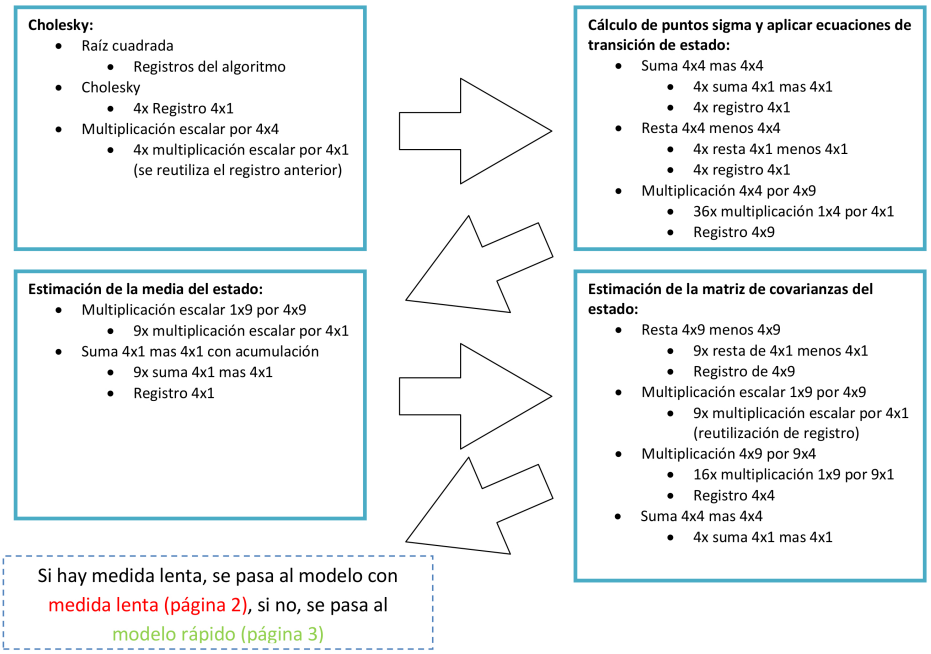

Figura A.4: Inferencia de operaciones básicas del filtro en la implementación de dos vía (I).

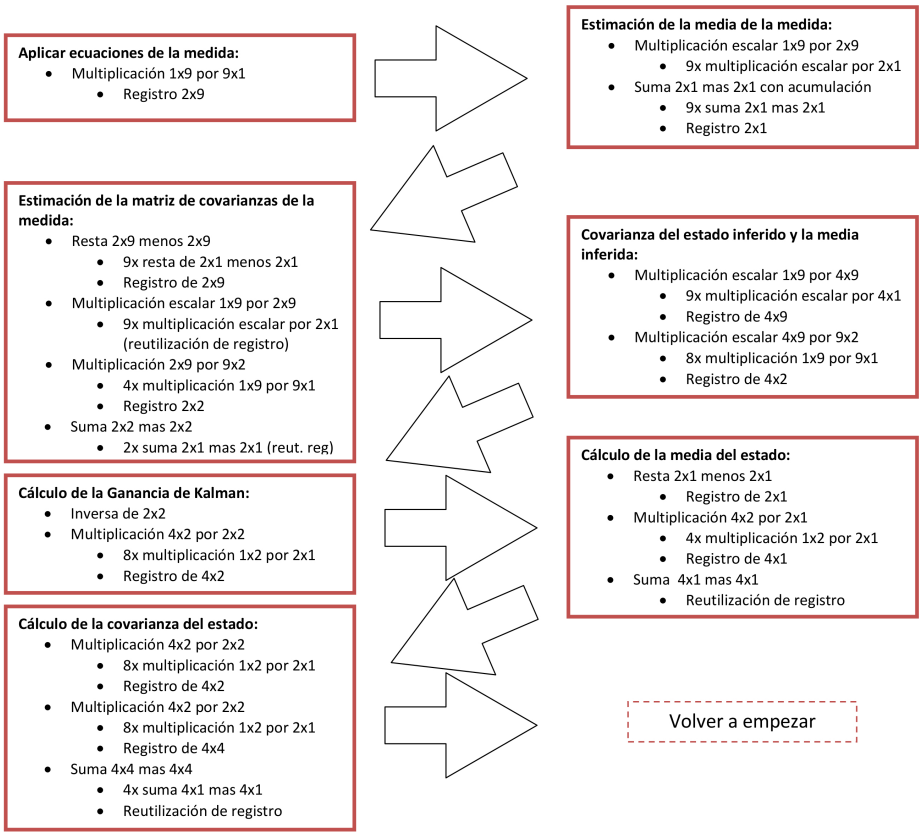

Figura A.5: Inferencia de operaciones básicas del filtro en la implementación de dos vía (II).

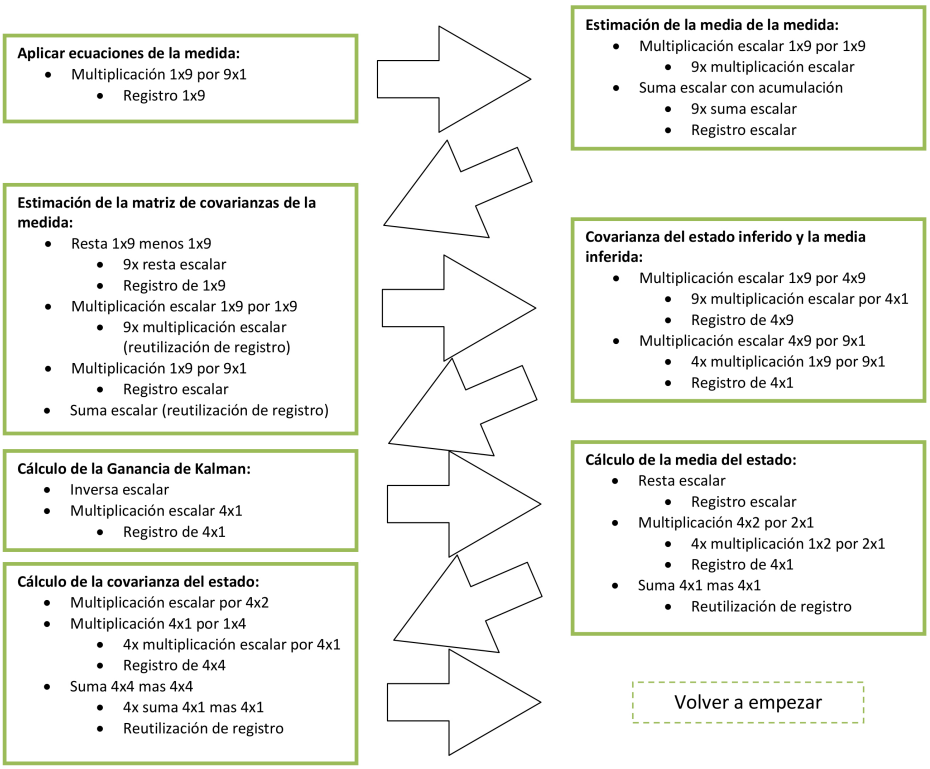

Figura A.6: Inferencia de operaciones básicas del filtro en la implementación de dos vía (III).

Apéndice B

Contribuciones

Aquí se explica la distribución del proyecto entre los integrantes y la contribución de cada uno. Destacar que trabajo de fin de grado ha estado marcado por las fechas. Hemos ido detrás del calendario en todo momento, pero con mucho esfuerzo finalmente el resultado ha sido satisfactorio.

B.1. Contribución de Manuel Pascual

Mi trabajo estuvo enfocado en varios puntos:

- En relación al filtro desarrollado en Matlab, primero generé con ayuda de Pablo el primer script que modelaba el filtro con las ecuaciones del capítulo 3 y del capítulo 4. Esta implementación primera fue la de dos vías, de la que se habla en el capítulo 6. Cuando estuvo terminado vi que usando la implementación de dos vías tendríamos que describir un montón de módulos. Estuve repasando las ecuaciones y se me ocurrió el modelo de una vía con el que se reducía el hardware y quedaba una implementación más homogénea. Terminada esta implementación calculé el error y vi que era factible realizar la implementación de una vía. Desde entonces ha sido el camino que hemos seguido. También conseguí las covarianzas de los ruidos que mejor adaptaban el filtro a las entradas.

- En vistas a la implementación HW, realicé el análisis del hardware necesario que se puede encontrar en el apéndice A. Esto nos sirvió para conseguir una vista rápida de las operaciones básicas que necesitábamos. Dividimos entre todos los módulos y los fuimos implementando.

- También realicé el diseño del *Top-Level* compuesto por los módulos *top_ UFK* y *top_ UFK_ ctrl*, *UFK* y *UFK_ ctrl*, así como la conexión a nivel *Top* de las etapas PREDICTION, CORRECTION y KALMAN_GAIN

con señales de entrada matriciales, esto quiere decir que introducíamos en cada módulo las matrices enteras mediante montones de pines en paralelo. Cambiamos el diseño después a una implementación con vectores unidimensionales de datos y finalmente en puertos serie para introducir los datos de uno en uno, usando BRAMs para almacenarlos. Estos cambios hicieron que el diseño ocupase menos hardware ya que se evitaba la cantidad ingente de multiplicadores, sumadores y multiplexores que generaban los arrays y vectores. Fue una mejora muy acertada por parte de nuestros directores del proyecto. También realicé el diseño de la etapa KALMAN_GAIN que se puede encontrar en el capítulo 10.

- Diseñé los TestBench de los módulos del componentes del *Top-Level* y de la etapa KALMAN_GAIN, además de desarrollar funciones específicas para la comunicación de los datos entre Matlab y el VHDL. Hice un script que calculaba el error del VHDL con respecto a la respuesta del Matlab en punto fijo y punto flotante. Realicé las pruebas del Matlab y del hardware y generé las gráficas de la memoria.

- Redacté los capítulos uno, dos, tres, cuatro, seis, once, doce y trece que introducen el trabajo, tratan los aspectos matemáticos y teóricos del filtro y del modelo, hablan sobre el modelado en Matlab y su aplicación para el diseño del filtro y los resultados de las pruebas, el hardware resultante y las conclusiones.

A mi parecer este trabajo ha sido muy completo y entretenido. Hemos dado lo mejor de nosotros y se ha notado. La falta de tiempo ha hecho que no pudiésemos disfrutarlo más pero también ha ayudado a mejorar nuestra dinámica de trabajo y a aprender a llevar un orden por encima de todo, que es lo que ha hecho posible la consecución de nuestros objetivos.

B.2. Contribución de Pablo Lammers

He contribuido en las siguientes tareas:

- Desarrollo en Matlab

  - Desarrollo en Matlab de una primera versión del filtro que sólo utilizaba una medida rápida: Lamentablemente no conseguí ningún éxito y el filtro no lograba adaptarse a las salidas esperadas.
  - Desarrollo en Matlab de la versión del filtro actual: Solución de errores y optimización del código. Cambie varios fors que teníamos para realizar las operaciones de matrices por una sola instrucción para aprovechar la potencia de Matlab y lograr que el código se ejecutase en menos tiempo.

  - Reorganización del directorio de Matlab en varias subcarpetas: Almacenamiento de gráficas cada vez que se ejecuta el filtro en la carpeta Resultados, scripts generadores de datos (empiezan por make) de la carpeta Generadores_Entrada. Así nos aseguramos que el filtro siempre utiliza las mismas entradas y podemos comparar diferentes configuraciones.
  - Desarrollo de algoritmos especiales en Matlab, implementados para obtener resultados más cercanos respecto a la implementación en hw. Tales algoritmos son *Div_restoring* para realizar la división de dos objetos FI; *Sqrt_norest*que realiza la raíz cuadrada de un objeto FI y *Cholesky_4x4* que realiza la factorización de Cholesky de una matriz 4x4 usando para ello los dos algoritmos anteriores.

- Desarrollo del HW

  - Implementación de todo lo referente a la etapa de Predicción: Módulo top, submódulos que intervienen y testbench que verifican el funcionamiento de cada módulo
  - Implementación de los tres módulos comunes mencionados anteriormente: *Sqrt_norest* que realiza la raíz cuadrada de un elemento; *Div_norest* que realiza la división de dos elementos y *Cholesky_4x4* que realiza la factorización de Cholesky de una matriz 4x4 a partir de los dos módulos anteriores.
  - Ayudas en la simulación de la entity Top del filtro solucionando varios errores como la discrepancia que había entre las constantes que utiliza el filtro en HW y las constantes que se utilizaban en Matlab. El desarrollo fue rápido ya que yo había desarrollado los scripts en Matlab de los algoritmos que se utilizaban y sabían cómo funcionaban exactamente.

- Redacción de la Memoria Por último quedaba redactar la documentación, escribí dos capítulos con bastante contenido ya que explican la implementación de dos etapas en vhdl. Los capítulos son:

  - Capítulo 8: Etapa de Predicción. Yo mismo había realizado todo el vhdl de esa etapa, por lo que fue sencillo documentarlo
  - Capítulo 10: Etapa de Kalman Gain. Un poco más complicado ya que yo no desarrollé el vhdl de esa etapa, por lo que me toco estudiar y examinar que hacía el HW que se implementó
  - Por último destacar problemas que me he encontrado al realizar la documentación, es bastante complicado seguir el mismo criterio que utiliza un compañero en otro capítulo, ya que cada persona escribe a su manera. Sin embargo, la herramienta de *TeXis* fue un

buen descubrimiento ya que nos permitió trabajar por separado y nos ahorró el formateo del texto y la división de la estructura de la memoria. Los principales inconvenientes que me he encontrado al redactar la memoria son ponernos de acuerdo entre todos los compañeros para seguir una misma estructura en los capitulos y revisar lo escrito por otros compañeros ya que no encuentras un momento para ello.

B.3. Contribución de Daniel del Pino

Mi aportación a este proyecto estuvo enfocado en los siguiente puntos:

- En relación al filtro desarrollado en Matlab, revisé el código en busca de errores y ayudé en las primeras pruebas. Mi aportación en esta parte es escasa debido a mi desconocimiento de la plataforma de Matlab.
- En relación a la implementación HW, me centré en implementar todo lo referente a la etapa de CORRECTION: Módulo Top, submódulos que implementan cada una de las ecuaciones y testbenchs que verifican el funcionamiento de cada módulo por separado y de la etapa completa.
- Implementé varios de los componentes comunes usados en el proyecto: sumadores y restadores, multiplicadores de matrices y contador. Cada componente dispone de un testbench para verificar su funcionamiento correcto
- Al redactar la memoria, escribí los siguientes capitulos:
  - Capítulo 5 en relación al Punto Fijo, explicación de que és y su aplicación al proyecto. Para ello fue necesario informarme a fondo sobre sus peculiaridades.
  - Capítulo 7 que trata sobre la implementación HW del proyecto en general, se explican las peculiaridades de la implementación general del filtro, así como una descripción de los componentes comunes usados, la jerarquía de organización del proyecto y los módulos Top-Level del sistema y su funcionamiento. Redactar este capítulo implicó entender profundamente todos los componentes y el funcionamiento paso a paso del filtro.
  - Capítulo 9 que trata sobre la etapa CORRECTION. Yo mismo había diseñado todos los componentes de esta etapa por lo que no supuso un gran esfuerzo documentarlo. Explica profundamente cómo funciona la etapa y los componentes integrados en ella.

Bibliografía

Biografía de R. E. Kalman. http://www.cs.unc.edu/~welch/kalman/kalmanBio.html. 2014 – 2016.

Enlace a Apollo Moon Program and Lunar Prospector Mission. https://www.nasa.gov/centers/ames/news/releases/2004/moon/apollo_ames_atmos.html. March 29, 2008.

Haykin, S. *Kalman Filtering and Neural Networks*. 2001. ISBN 0-471-36998-5. ©2001, John Wiley & Sons, Inc.

Kalman, R. E. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, vol. 82(Series D), páginas 35–45, 1960. http://www.cs.unc.edu/~welch/kalman/media/pdf/Kalman1960.pdf.

Kuure-Kinsey, M., Palerm, C. C., & Bequette, B. W. A dual-rate kalman filter for continuous glucose monitoring. *Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society*, (Conference, 1, 63-6. doi:10.1109/IEMBS.2006.26005), 2006.

Nicholas J. Higham. Cholesky factorization. *WIREs Comp Stat 2009*, vol. 1, páginas 251–254, September / October 2009. ©2009 John Wiley & Sons, Inc. http://www.maths.manchester.ac.uk/~higham/papers/high09c.pdf.

Página principal de GitHub. https://github.com/features. 2017.

Página principal de HDL Desinger. https://www.mentor.com/products/fpga/hdl_design/hdl_designer_series/. 2014 – 2017.

Página principal de LaTeX. https://www.latex-project.org/. 2005 – 2017.

Página principal de Matlab. https://es.mathworks.com/products/matlab.html. 1994 – 2017.

Página principal de Podio. https://podio.com/site/es. 2015 – 2017.

PÁGINA PRINCIPAL DE QUESTA. https://www.mentor.com/products/fv/questa/. 2014 – 2017.

S.J. JULIER AND J.K. UHLMANN. The 11th international symposium on aerospace/defence sensing, simulation and controls. En *Proceedings of AeroSense*. 1997.

S.J. JULIER AND J.K. UHLMANN. A general method for approximating nonlinear transformations of probability distributions. En *Technical Report, RRG*. Department of Engineering Science, University of Oxford, November 1996. http://www.robots.ox.ac.uk/siju/work/publications/letter_size/Unscented.zip.

S.J. JULIER, J.K. UHLMANN, AND H. DURRANT-WHYT. A new approach for filtering nonlinear systems. En *Proceedings of the American Control Conference*, páginas 1628 – 1632. 1995.