Implementación en FPGA del *Automatic Target Generation Process* para la detección de objetivos en imágenes hiperespectrales de la superficie terrestre

Sergio Esquembri Martínez
Concepción Núñez Montes de Oca

Proyecto de Sistemas Informáticos

Facultad de Informática
Departamento de Arquitectura de Computadores y Automática
UNIVERSIDAD COMPLUTENSE DE MADRID

Madrid, junio de 2013

Director: Daniel Mozos Muñoz
Codirector: Carlos González Calvo

Implementación en FPGA del *Automatic Target Generation Process* para la detección de objetivos en imágenes hiperespectrales de la superficie terrestre

Sergio Esquembri Martínez
Concepción Núñez Montes de Oca

Proyecto de Sistemas Informáticos

Facultad de Informática
Departamento de Arquitectura de Computadores y Automática
UNIVERSIDAD COMPLUTENSE DE MADRID

Madrid, junio de 2013

Director: Daniel Mozos Muñoz
Codirector: Carlos González Calvo

AUTORIZACIÓN

Los autores de este proyecto Sergio Esquembri Martínez, con DNI 51995369-M y Concepción Núñez Montes de Oca, con DNI 02298000-R, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria como el código de este proyecto.

Sergio Esquembri Martínez

Concepción Núñez Montes de Oca

En Madrid, junio de 2013.

A todos los que han soportado
nuestros nervios y estrés.
A todos los que han colaborado
y nos han ayudado a que sea posible.

AGRADECIMIENTOS

Desde mediados del curso pasado, cuando decidimos que éste sería el proyecto que desarrollaríamos a lo largo de este año, nos percatamos que consumiría una gran parte de nuestro tiempo, lo que no podíamos esperar era que a veces también consumiera parte de nuestras fuerzas.

Durante estos momentos de esfuerzo y energía o de cansancio y flaqueza siempre tuvimos personas a nuestro lado a las que consideramos necesario agradecer todo su apoyo.

A nuestro director Daniel. Por despertar en nosotros el interés por la informática desde nuestro comienzo en la carrera en el año 2008, cuando nos hizo ver los conocimientos que podríamos llegar a adquirir. Por tres años más tarde ser nuestro profesor y edificar la pasión que sentimos por el *hardware* y el interés por los proyectos que tenían en mente cuando dos años antes de lo que nos correspondía acudimos a las charlas informativas para alumnos que al año siguiente tenían que escogerlo. También queremos agradecerle que nos recibiera con los brazos abiertos cuando el año pasado fuimos a preguntarle por un proyecto que nos pudiese interesar y, a pesar de que él no tuviese pensado dirigir uno, nos aceptara, nos acompañase y nos guiase hasta donde hemos llegado hoy.

A nuestro codirector Carlos. Por su infinita paciencia y conocimientos, por hacernos ver que las cosas nunca eran tan negras como las pintábamos, por su apoyo en momentos de estrés y no dormir, y también por hacernos ver que hay cosas más importantes en la vida y no pasa nada por hacer que el proyecto espere una semana. También por su alegría y carácter, porque no hay nada mejor que trabajar en algo que te gusta rodeado de gente que siente la misma emoción por ello y con la que sentirse realmente a gusto, sabiendo que puedes contar con su apoyo y conocimientos incondicionalmente.

A nuestros compañeros y amigos Carlos y Gonzalo. Este proyecto no se habría podido desarrollar sin vosotros. Todos esos días en los que hemos trabajado intensivamente, los momentos de diversión, frustración, enfado o alegría han dejado huella en los resultados de este trabajo y en nosotros. La misión que comenzamos este año está a punto terminar, pero aún quedan aventuras pendientes.

A nuestros compañeros del día a día, en especial a aquellos que han trabajado con nosotros y han tenido que soportar nuestra escasa disponibilidad o nuestra falta de tiempo para dedicarnos de lleno a la tarea que compartíamos. En particular a Fede, que es quien más ha sufrido esta reducida disponibilidad así como esos días en los que todo se nos venía encima y nuestro carácter no era el mejor. Muchas gracias por vuestro apoyo, ánimos y comprensión.

A nuestros amigos, parejas y familias. Porque sabemos que no es fácil compartir nuestro día a día, que desde que comenzamos esta carrera no hemos podido dedicaros todo el tiempo que nos gustaría y merecéis. Habéis sufrido nuestro estrés y nuestros nervios, pero siempre habéis estado disponibles cuando necesitábamos de vuestro apoyo, animándonos a continuar.

En especial, queremos dar las gracias a esas personas que nos formaron en la vida y siempre nos animaron a perseguir nuestros sueños, algunos no pudisteis ver cómo seguimos vuestros consejos y comenzamos una carrera, relaciones y trabajos que nos han hecho realmente felices, otros estuvisteis para vernos comenzar el año y el proyecto pero no habéis podido estar ahora que esto se acaba y comenzamos una nueva etapa. Muchas gracias a ellos y a los que todavía seguís aquí por habernos hecho llegar a donde estamos. Esperamos poder hacer que os sintáis orgullosos de nosotros.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

En la actualidad, la observación remota de la Tierra mediante el análisis de imágenes hiperespectrales constituye una línea de investigación muy activa.

Debido a la forma en que aparecen los materiales en el entorno natural y a la resolución espacial que presentan los sensores de observación remota de la Tierra, los píxeles analizados no siempre están constituidos por la presencia de un único material, sino que están formados por distintos materiales puros a nivel de subpíxel.

Tradicionalmente, se utilizan técnicas de desmezclado espectral para su estudio. Esta situación conlleva a que su análisis comprenda un alto coste computacional debido a que precisa de dos etapas complejas. La primera se basa en la extracción de firmas espectrales puras, es decir, la extracción de *endmembers* como serán denominados a lo largo del documento. La segunda etapa está conformada por la estimación de la abundancia de dichos *endmembers* a nivel de subpíxel.

Esta complejidad computacional supone un problema en la situación de análisis de imágenes hiperespectrales a tiempo real en entornos variables como incendios y otros desastres naturales de estas características.

El desmezclado espectral, al constituir un alto coste computacional, no siempre es adecuado, pudiendo escoger la vía de la clasificación y detección de objetivos o *targets* en su lugar. Esta técnica, de menor coste computacional y de gran utilidad, permite el análisis de imágenes hiperespectrales mediante la obtención y clasificación de elementos en entornos desconocidos que contengan materiales no previstos.

En este proyecto final de carrera se ha llevado a cabo la implementación del algoritmo de detección y clasificación de objetivos conocido como ATGP (*Automatic Target Generation Process*), concretamente la versión del algoritmo que utiliza la proyección sobre el subespacio ortogonal. Para la implementación de dicho algoritmo se ha utilizado el lenguaje de descripción *hardware* VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) para su posterior uso en plataformas de *hardware* reconfigurable tipo FPGA (*Field Programmable Gate Array*).

Palabras clave: ATGP, Detección de objetivos, FPGA, *Hardware* reconfigurable, Imagen hiperespectral, VHDL.

ABSTRACT

In the present day, remote observation of planet Earth through hyperspectral imaging has become a very active research line.

Due to the disposition of materials in a natural environment and the spatial resolution achieved by the sensors in remote observation of the Earth, most of analysed pixels are composed by a mixture of pure elements in a subpixel level, instead of been composed with a single element.

Traditionally, spectral unmixing techniques are used for remotely sensed hyperspectral imagery analysis. These techniques are computationally expensive because they require two complex step process. The first step is the pure spectral signatures (endmember hereinafter) extraction. The second step is the estimation of endmembers fractional abundances at subpixel level.

This computational complexity becomes a serious drawback in applications which require a real-time response in variable environments such a forest fire monitoring or natural disaster tracking.

Target detection and classification can supersede the spectral unmixing techniques at a lower cost in certain applications. This less expensive technique is widely accepted usefulness and allows hyperspectral imaging analysis via obtention and classification of elements in environments without prior knowledgment of the terrain.

In this project, we have designed and implemented a target detection and classification algorithm known as ATGP (Automatic Target Generation Process). In particular, we used the version of this algorithm which uses the concept of orthogonal subspaces projection. This implementation uses VHDL (Very High Speed Integrated Circuit Hardware Description Language) for the subsequent implantation in reconfigurable platforms like FPGA (Field Programmable Gate Array).

Keywords: ATGP, FPGA, Hyperspectral image, Reconfigurable hardware, Target detection, VHDL.

Capítulo 1. Motivaciones y objetivos

En la actualidad los avances tecnológicos en los sensores están revolucionando la obtención, gestión y análisis de los datos recopilados en la observación remota de la Tierra, siendo algunos sensores de última generación, utilizados en satélites y plataformas aéreas, grandes productores de datos de alta dimensionalidad.

Además de recopilar una gran cantidad de información, para un gran número de aplicaciones y situaciones se añade la dificultad de realizar el procesamiento de los datos en tiempo real. Estas situaciones suelen ser sucesos que evolucionan y cuya ubicación se ve alterada a lo largo del tiempo, como pueden ser incendios forestales u otros desastres naturales.

Esta gran cantidad de datos de alta dimensionalidad se hace realmente evidente en el análisis de imágenes hiperespectrales donde un espectrómetro de imagen recoge cientos o incluso miles de imágenes en varios canales de longitudes de ondas para el mismo área de superficie de la Tierra. Debido a su alta dimensionalidad, la información recogida se denomina a menudo como “cubos de datos”.

“El concepto de imagen hiperespectral tiene origen en el Jet Propulsion Laboratory *de la NASA [1] (*National Aeronautics and Space Administration*) que desarrolla instrumentación como el* Airborne Visible Infra-Red Imaging Spectrometer *(AVIRIS). Actualmente este sistema permite la cobertura de un ancho de banda desde 0.4 a 2.5 µm con más de doscientos canales espectrales, con una resolución espectral nominal de 10 nm. El resultado es que cada píxel (considerado como un vector) recogido por un sensor hiperespectral puede ser visto como una firma espectral o '*huella digital' *de los materiales subyacentes en el píxel.”* [2]

Cada píxel perteneciente a estos cubos de datos puede ser de dos tipos. El primer tipo se denominaría píxel puro o *endmember* y es aquel que contiene un único material. El segundo tipo de píxel sería el píxel mezcla, el cual está compuesto por diversos materiales. Para realizar el desmezclado espectral se han diseñado una gran variedad de algoritmos [3], todos ellos con la característica común de tener un gran coste computacional. El comportamiento de estos algoritmos consiste en identificar en la imagen los píxeles con firmas espectrales puras y a continuación expresar cada uno de los píxeles de la imagen como una combinación lineal de estos *endmembers* ponderados por sus fracciones de abundancia.

No obstante, según cuales sean los objetivos que se persigan a la hora de procesar una imagen hiperespectral, en lugar de llevar a cabo un desmezclado espectral se puede optar por dirigir la búsqueda a los elementos anómalos en la imagen u obtener una colección representativa de todos los elementos de la escena. Este último método, conocido como detección de objetivos o *targets* [4] goza de un coste computacional menor que el desmezclado espectral, lo cual facilita el procesamiento de imágenes en tiempo real. Uno de los algoritmos usados para procesar imágenes hiperespectrales en busca de *targets* es el *Automatic Target Generation Process* (ATGP), implementado en este proyecto.

Además del desmezclado espectral y la detección de anomalías u objetivos, el análisis para el procesamiento de imágenes hiperespectrales abarca temas como la reducción de la dimensionalidad [5], la clasificación y la compresión de datos.

La alta demanda computacional y la necesidad de procesamiento en tiempo real en muchas aplicaciones de análisis de imágenes hiperespectrales ha despertado la investigación sobre diversas plataformas o modos de implementación de estos algoritmos. La mayoría de las técnicas utilizadas que han sido recogidas hasta la fecha en la literatura se basan en el uso de *clusters,* sistemas multiprocesador y GPU [6,7].

El uso de *clusters*, a pesar de tener una gran adaptabilidad en el campo del tratamiento de imágenes hiperespectrales en tierra, conlleva un problema de coste, espacio y consumo para soluciones a bordo. Por otra parte, una solución al problema del espacio sería la utilización de GPUs, es decir, las unidades de procesamiento gráfico domésticas que poseen un escaso peso y un tamaño reducido. El problema de estas unidades se presenta con la

gran demanda de energía que poseen, siendo motivo suficiente como para no poder utilizarse en el espacio.

Las FPGAs (*Field Programmable Gate Array*), plataforma donde será implementado nuestro algoritmo, constituyen una opción particularmente interesante para el análisis a bordo de estas imágenes hiperespectrales debido a la posibilidad de reconfiguración, su tamaño compacto y su alta potencia de cálculo. Cabe añadir que existen FPGA endurecidas para la radiación que han sido certificadas para operar en condiciones espaciales.

1.1.- ORGANIZACIÓN DE ESTA MEMORIA

Teniendo en cuenta las motivaciones y objetivos previamente expuestos procederemos a realizar una visión general de los puntos que comprende esta memoria describiendo cada uno de ellos:

- **Análisis hiperespectral**: En este apartado definiremos el concepto de imagen hiperespectral de modo detallado así como sus principales usos en la actualidad.
- ***Hardware* reconfigurable**: El contenido de este apartado abarca una descripción de forma detallada de la estructura interna de un dispositivo FPGA así como un análisis las grandes ventajas que aporta su utilización para el procesamiento de imágenes hiperespectrales a bordo.
- **Detección de *targets***: Exposición sobre las pretensiones del algoritmo de detección de *targets* en que basamos nuestro proyecto final de carrera y los pasos que componen este algoritmo explicando su desarrollo de forma paralela.
- **Implementación en FPGA del algoritmo ATGP**: En este apartado procederemos a explicar primeramente el algoritmo implementado, ya que, a pesar de tratarse del mismo algoritmo, ha sido adaptado para la implementación *hardware*. Posteriormente hablaremos del módulo implementado y los submódulos que lo componen profundizando en el análisis de todos los componentes, siguiendo un patrón de descripción que incluye: un esquema del componente desarrollado, los parámetros de entrada y salida del módulo con su consecuente explicación, la enumeración y descripción de los bloques que lo componen, y, por último, la descripción de la unidad de control así como los estados que componen esta unidad y su funcionamiento e interacción entre ellos.
- **Resultados experimentales**: Este apartado está conformado por los cuatro subapartados siguientes:
  - **Plataforma reconfigurable**: En él se describirá la que hemos utilizado para realizar la síntesis de nuestra implementación.
  - **Conjunto de imágenes hiperespectrales**: En este subapartado detallaremos las imágenes hiperespectrales que hemos utilizado para realizar las pruebas de nuestra implementación.
  - **Evaluación de los *targets***: Exposición y evaluación de los resultados obtenidos y posterior comparación con los de la literatura existente.
  - **Evaluación del rendimiento**: Caracterización del algoritmo según el tiempo de ejecución y los recursos de la plataforma empleados. Comparación de los resultados con otras implementaciones presentes en la literatura.
- **Conclusiones y trabajo futuro**: En este apartado se incluirá un resumen de los logros conseguidos en este proyecto final de carrera y un conjunto de opciones que pueden ser consideradas para trabajos futuros.
- **Bibliografía**: Selección de artículos y referencias que han facilitado el desarrollo de este trabajo además de resultar útiles al lector si desea ampliar su grado de información acerca del análisis de imágenes hiperespectrales, los algoritmos utilizados para ello y sus diversas implementaciones para reducir el tiempo de ejecución.

CAPÍTULO 2. ANÁLISIS HIPERESPECTRAL

En la actualidad el uso de imágenes hiperespectrales está en auge debido al reciente lanzamiento de satélites y sistemas comerciales aerotransportados. El uso de estas imágenes para la observación remota de la tierra puede llegar a convertirse en una herramienta común pudiendo usarse en ámbitos tan diversos como defensa, inteligencia, agricultura de precisión o geología.

2.1.- CONCEPTO DE IMAGEN HIPERESPECTRAL

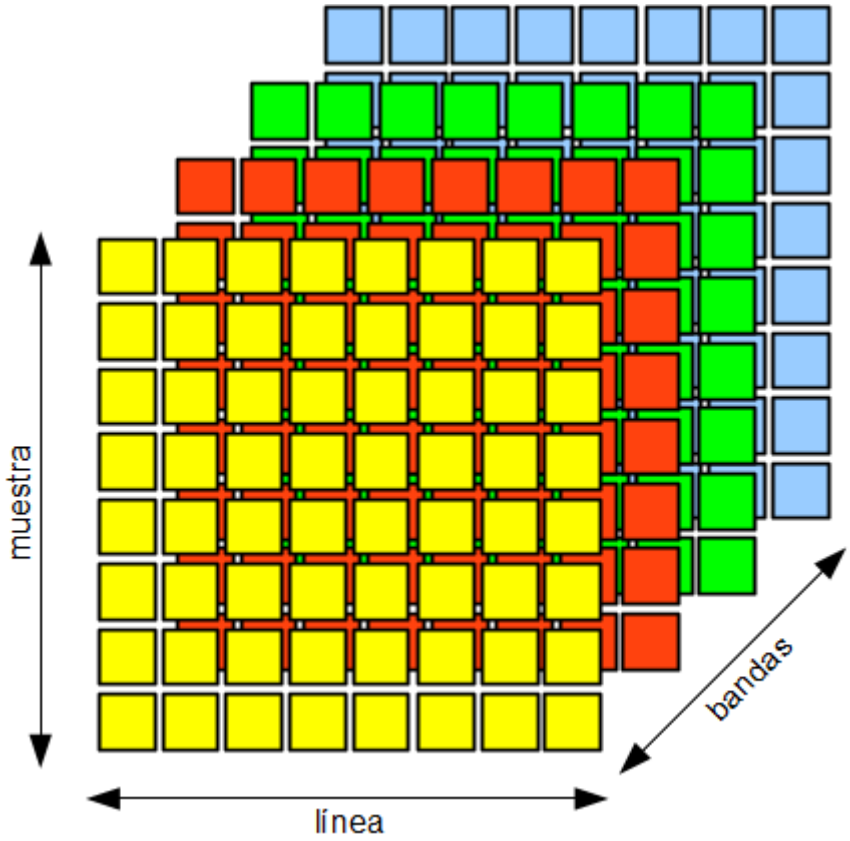


Figura 2.1. Concepto de imagen hiperespectral.

Los sensores hiperespectrales se encargan de obtener imágenes en diversos canales espectrales obteniendo finalmente para cada píxel su firma espectral [8]. Una imagen hiperespectral captada por un sensor puede ser representada en forma de cubo, utilizando dos dimensiones para mostrar la ubicación espacial del píxel en la imagen y una tercera dimensión compuesta por la reflectancia de los píxeles en cada una de las bandas espectrales.

Para facilitar su comprensión véase en la Figura 2.1 como el eje X denominado líneas y el eje Y denominado muestras conforman dos ejes de coordenadas para la ubicación del píxel, mientras que el eje Z, denominado bandas, será el conjunto de reflectancias tomadas en diferentes canales espectrales para cada uno de los píxeles.

De esta forma se podrá obtener la firma espectral de cada píxel analizando la reflectancia recogida por el sensor en diferentes longitudes de onda. Esta firma espectral puede estar constituida por diversos materiales a nivel de subpíxel. A estos píxeles se les denomina píxeles mezcla y serán los que conformen la mayor parte de la imagen hiperespectral, debido a que el hecho de que cohabiten dos materiales diferentes en una porción de la superficie terrestre puede ocurrir incluso a nivel microscópico. Los píxeles que contienen un único tipo de material se denominan píxeles puros o *endmembers*.

2.1.1.- Espectrometría de imágenes

Para obtener las imágenes hiperespectrales se utilizan unos instrumentos llamados espectrómetros de imágenes. Los espectrómetros son capaces de realizar mediciones espectrales de bandas tan próximas como 0.1 µm.

La espectroscopia es el estudio de la luz que es emitida o reflejada por los materiales y su variación de energía con la longitud de onda. Para el estudio de imágenes hiperespectrales la propiedad de la espectroscopia que resulta primordial es la reflectancia espectral, que es el porcentaje de la energía reflejada sobre la energía incidente como una función de la longitud de onda. La reflectancia varía con la longitud de onda para la mayoría de los materiales dado que la energía puede ser absorbida o reflejada en distintos grados.

La forma general de una curva espectral (basada en una gráfica que relaciona longitud de onda y reflectancia) y la posición e intensidad de las bandas de absorción se utilizan frecuentemente para la identificación y discriminación de diversos materiales. En la Figura 2.2 se pueden observar las curvas espectrales de distintos elementos comunes en un terreno natural. También se muestran las bandas espectrales utilizadas por los sensores remotos de satélites multiespectrales SPOT XS y Landasat TM.

La reflectancia en los distintos espectros de luz o distintas longitudes de onda es característica de cada material [8], pudiéndose identificar un material a partir de su *firma espectral.*

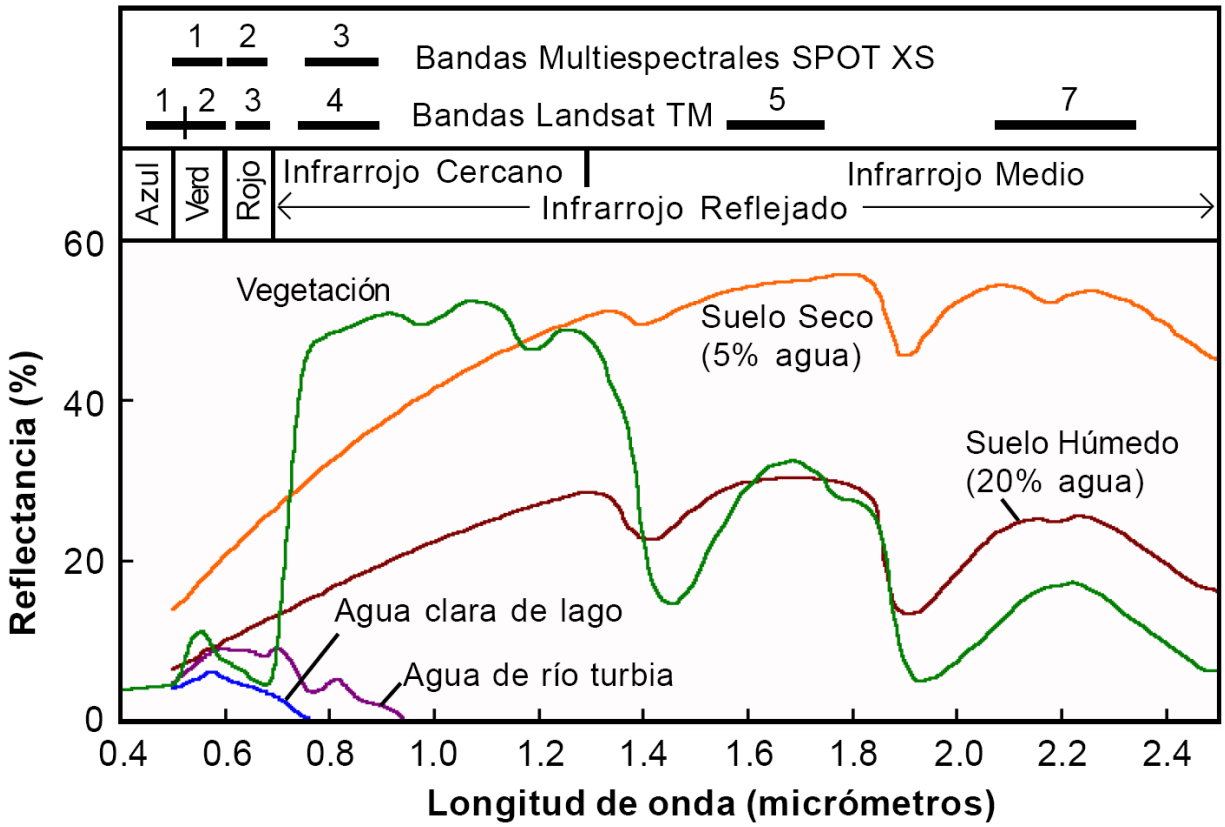


Figura 2.2. Curvas espectrales de distintos materiales.

2.2.- BIBLIOTECAS ESPECTRALES

Para facilitar el trabajo del análisis en imágenes hiperespectrales existen varias bibliotecas espectrales que contienen una gran cantidad de datos de reflectancia espectral ya sea de materiales naturales o desarrollados por el ser humano.

2.2.1.- BIBLIOTECA ESPECTRAL USGS

Desarrollada por el laboratorio de espectrometría *United States Geological Survey* [9] en Colorado. Contiene aproximadamente 500 reflectancias espectrales sobre una longitud de onda entre 0.2 y 3.0 μm. Estos materiales sobre los que almacena su reflectancia espectral son principalmente minerales.

2.2.2.- BIBLIOTECA ESPECTRAL ASTER

Desarrollada gracias a la NASA por el programa *Advanced Spaceborne Thermal Emission and Reflectance Radiometer* [10]. Contiene reflectancias espectrales recogidas por el *Jet Propulsion Laboratory* de la NASA, *Johns Hopkins University* y *United States Geological Survey*. Contiene aproximadamente 2000 reflectancias espectrales sobre una longitud de onda entre 0.4 y 14 μm. Entre los materiales sobre los que se almacenan datos se encuentran minerales, agua, nieve y materiales fabricados por el hombre.

2.3.- PRINCIPALES SENSORES HIPERESPECTRALES

A pesar de que la observación remota de la Tierra lleva realizándose desde hace más de 150 años mediante el uso de cámaras instaladas en globos y dirigibles, la tecnología que desarrolló los sensores hiperespectrales no surgió hasta el inicio de los años 90.

Los sensores hiperespectrales presentaron un gran desarrollo sobre los sensores multiespectrales, los cuales no eran capaces de detectar un gran número de bandas espectrales.

En la actualidad existen misiones comerciales como el *Jet Propulsion Laboratory* de la NASA y el Servicio Geológico de EE. UU, encargados de la adquisición de imágenes hiperespectrales. Lamentablemente estas misiones imponen limitaciones a la hora de la toma de imágenes puesto que está restringida la fecha y hora en que fueron recopiladas, pudiendo verse alteradas por fenómenos atmosféricos. Este problema ha sido solventado mediante la venta de estos sensores por parte de empresas privadas de electro-óptica y su compra por parte de empresas privadas. Estos sensores constan de una gran resolución espacial y son utilizados para la detección de yacimientos minerales, agricultura de precisión y valoración de impactos ambientales.

2.3.1.- SENSOR AVIRIS

Sensor hiperespectral aerotransportado capaz de analizar zonas visibles e infrarrojas del espectro [11-13]. Encontrándose en estado operativo desde el año 1987, este sensor tiene la capacidad de obtener información en 224 canales espectrales contiguos, con un ancho entre las bandas de 10 nm, cubriendo un rango de longitudes de onda entre 0.4 y 2.5 μm.

Este sensor ha realizado tomas de imágenes hiperespectrales en Europa, Estados Unidos y Canadá mediante el uso de dos plataformas distintas, un avión ER-2 perteneciente al *Jet Propulsion Laboratory* de la NASA y otro avión denominado *Twin Otter* desarrollado por la compañía canadiense *Havilland Canada* con velocidades de 730 km/h y 130 km/h respectivamente.

2.3.2.- SENSOR EO-1 HYPERION

Lanzado de forma satisfactoria en noviembre del año 2000 dentro del programa *New Millenium* [14] de la NASA. Este sensor es capaz de obtener información en 220 bandas espectrales cubriendo un rango de longitudes de onda entre 0.4 y 2.5 μm con una resolución espacial de 30 metros. Cada línea de datos en las imágenes tomadas esta constituida por 256 píxeles.

2.4.- PRINCIPALES RETOS EN EL TRATAMIENTO DE IMÁGENES HIPERESPECTRALES

Debido a la gran cantidad de información que registran las imágenes hiperespectrales y a la composición de los píxeles que en su gran mayoría contienen información sobre diversos materiales, el análisis de imágenes hiperespectrales tiene un alto coste computacional.

La alta dimensionalidad de los datos no sólo dificulta la manera en que estos deben ser almacenados y procesados en un ordenador determinado, sino que añade la complicación de que esta información debe ser enviada previamente por el satélite que se ha encargado de la toma de las imágenes.

Las posibles soluciones al problema de la alta dimensionalidad de los datos serían la compresión de datos con pérdidas (mayor compresión pero menos información) o sin pérdidas (menor compresión pero manteniendo la información) [15] o la recogida selectiva de determinadas bandas espectrales en base a los materiales sobre los que se requiere información pudiendo tratarse de minerales y rocas en zonas mineras o vegetación y agua en zonas de selva o bosque.

En relación al desmezclado espectral existen dos aproximaciones para su realización. El desmezclado espectral lineal supone que los espectros recogidos por los espectrómetros pueden ser representados mediante una combinación lineal de *endmembers* representados por su porcentaje de abundancia en el píxel. Esta aproximación asume que los efectos producidos por los reflejos o dispersiones son mínimos. La otra aproximación es el desmezclado espectral no lineal, que a pesar de que puede proporcionar una mejor información de los *endmembers* recogidos en la imagen incluyendo aquellos que se distribuyen al azar a lo largo del campo de visión del sensor, requiere información previa acerca de las propiedades físicas de los materiales que conforman la imagen, no pudiendo ser aplicable por tanto a zonas sobre las que no se posee información *a priori*. En la Figura 2.3 puede observarse una comparativa de ambas aproximaciones.

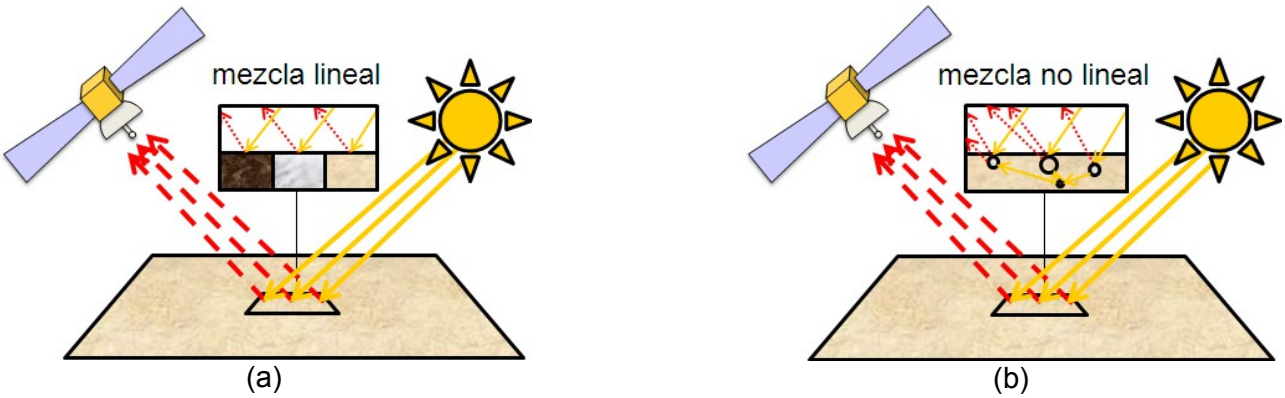


Figura 2.3. Comparación de los modelos lineal (a) y no lineal (b).

CAPÍTULO 3. *HARDWARE* RECONFIGURABLE

Para realizar computación, tradicionalmente se están usando dos enfoques: *hardware* específico y *hardware* de propósito general. El primer método consiste en el diseño de un *hardware* específico mediante interconexión fija de circuitos para realizar los cálculos requeridos en una aplicación. Esta interconexión se puede realizar mediante la integración en un circuito de aplicación específica o ASIC (*Application Specific Integrated Circuit*) o bien con la conexión de los componentes individuales en una placa [16]. El segundo enfoque, también conocido como procesamiento *software*, se basa en usar *hardware* de propósito general capaz de ejecutar un conjunto de instrucciones más o menos amplio.

Si bien la primera alternativa ofrece una mayor eficiencia y velocidad de cálculo para la aplicación para la que ha sido diseñado, el método de diseño implica una gran rigidez y su reutilización para otros propósitos es escasa o nula. Utilizando *hardware* de propósito general, como un microprocesador, la variedad de funcionalidades que puede implementar mediante distintos programas y la facilidad de alterar dicho programa supone una gran flexibilidad, pero esa flexibilidad impone penalizaciones en los tiempos de ejecución debido a las secuencias necesarias de búsqueda, lectura, decodificación y ejecución de instrucciones.

El *hardware* reconfigurable aparece como una tercera alternativa, ofreciendo a la vez eficiencia de procesamiento y una gran flexibilidad. Estas arquitecturas ocupan el lugar intermedio entre el uso de ASIC y el uso de microprocesadores (véase Figura 3.1). Si bien no se obtiene la velocidad y eficiencia de procesamiento del *hardware* específico ni es tan flexible como el *hardware* de propósito general, al beneficiarse de las ventajas de ambos métodos suponen una opción muy atractiva [17].

En la actualidad, el *hardware* reconfigurable sigue cobrando importancia, ya que las características anteriormente descritas resultan muy convenientes para una gran cantidad de campos, algunos tan diversos como la criptografía [18] y la robótica [19]. Hoy en día es una alternativa que se sigue investigando y mejorando, de manera que se pueden encontrar en el mercado alternativas de *hardware* reconfigurable que incorporan microprocesadores y ASIC, así como las nuevas plataformas de *hardware* 3D [20], que suponen un gran aumento en la relación espacio-capacidad de cálculo.

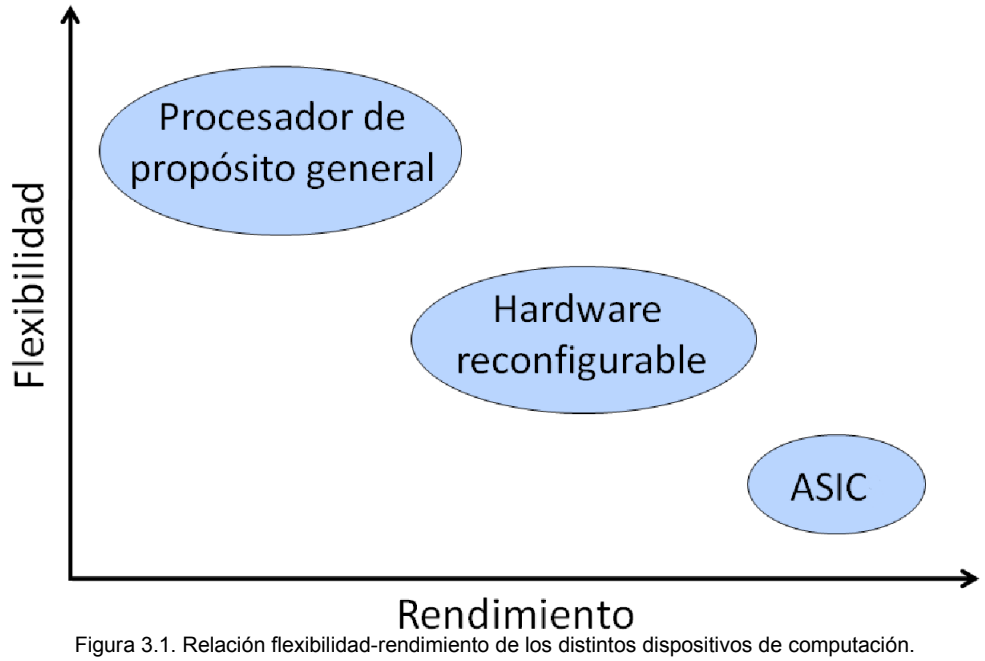


Figura 3.1. Relación flexibilidad-rendimiento de los distintos dispositivos de computación.

3.1.- TECNOLOGÍA RECONFIGURABLE

Existen distintos tipos de arquitecturas de *hardware* reconfigurable, tales como las arquitecturas basadas en filas, *sea-of-gates*, jerárquicas o *arrays* precableados. Esta última será la que explicaremos a continuación, pues es la arquitectura utilizada para la implementación del proyecto.

Las FPGAs (*Field Programmable Gate Array*) consisten en una matriz de bloques lógicos (*Logic Blocks*, LB de aquí en adelante) y una red de interconexión. Mediante la descarga de los bits de configuración *hardware* [21] puede configurarse la funcionalidad de los LBs y de la red de interconexión.

Los LBs suelen contener circuitos combinacionales programables *Look-Up-Table* (LUT), biestables, lógica adicional y bits de memoria SRAM para la configuración de los elementos. Estos bits de memoria SRAM pueden sustituirse por dispositivos *antifuse* [22], a cambio de perder capacidad de reprogramación, mientras que la configuración mediante SRAM [23] admite incluso reconfiguración dinámica o parcial. Las labores de entrada/salida se realizan en la periferia del dispositivo con bloques específicos denominados *Input-Output-Blocks* (IOB) o mediante LBs. Como se ha comentado antes, la tendencia en el *hardware* reconfigurable es integrar otros elementos tales como memoria dedicada [24], unidades aritmético-lógicas complejas o incluso microprocesadores [25] y ASIC.

3.2.- DISEÑO CON FPGAS

El diseño en FPGAs se realiza mediante la descripción del sistema en un lenguaje de alto nivel cuyas instrucciones tengan una fácil relación con elementos *hardware*, de cara a facilitar la traducción a un circuito equivalente. En este caso, se ha hecho uso de los conocidos como lenguajes de descripción *hardware* o HDL (*Hardware Description Language*), cuyos exponentes más conocidos son Verilog [26] y VHDL [27].

El proceso de síntesis de una tarea al *hardware* equivalente se divide en varias etapas. La primera de ellas es la descripción del algoritmo en un lenguaje HDL. Tras esto, se determina el tipo de bloques necesarios y las conexiones entre éstos, para después asignar bloques básicos (en el caso de las FPGA son CLBs, *Configurable Logic Block*) configurables concretos de la plataforma y trazar las rutas de señales entre ellos. El siguiente paso es determinar la configuración necesaria para conseguir el diseño determinado en el paso anterior, es decir, obtener el mapa de bits de configuración que determinan el comportamiento de los LBs necesarios para la implementación del circuito. El último paso consiste en cargar dicho mapa de bits en la memoria de configuración del dispositivo. El proceso completo aparece reflejado en la Figura 3.2.

En este caso se ha hecho uso del lenguaje VHDL (*Very High Speed Integrated Circuit HDL*). Este lenguaje permite la descripción y modelado de la funcionalidad y organización de sistemas *hardware*. VHDL permite el modelado estructural y funcional de circuitos. Mediante el modelado estructural podemos conseguir la simulación de circuitos indicando los componentes y las conexiones entre éstos. No obstante, el modelado funcional resulta una alternativa muy interesante para el diseño de sistemas digitales. Este método se basa en describir un circuito mediante su comportamiento, ignorando cuál es su estructura interna.

El modelado funcional aporta a VHDL la aplicación de síntesis automática de circuitos. El proceso de síntesis no es directo, puesto que el lenguaje no fue inicialmente concebido para ello, pero las herramientas de síntesis utilizadas hoy en día facilitan mucho la tarea, permitiendo la implementación de circuitos descritos a muy alto nivel.

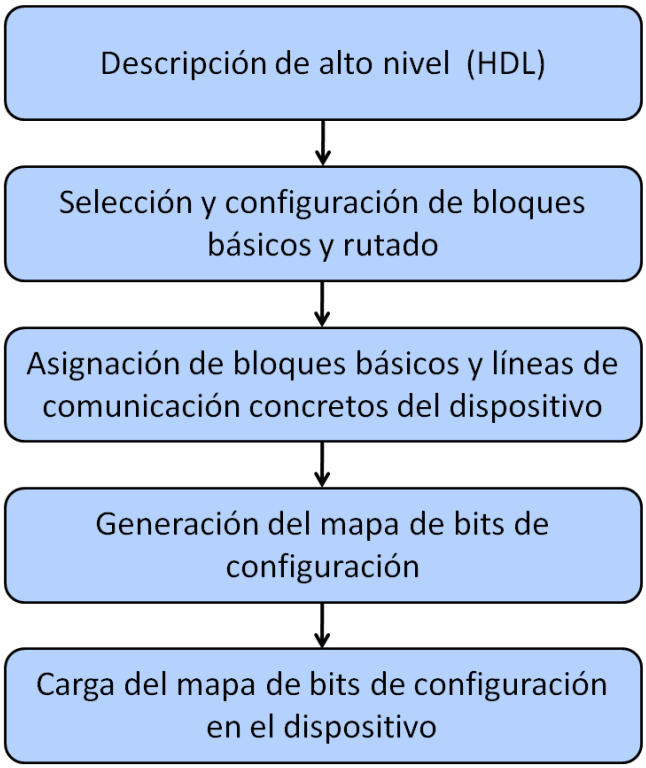


Figura 3.2. Etapas del diseño con FPGA.

VHDL consta de una gran sintaxis y la posibilidad de modularizar los circuitos a implementar, facilitando la labor de diseño. Estas características, añadidas al hecho de que forma parte de un estándar internacional (*IEEE Std. 1076-1987*), lo que reduce los posibles problemas de compatibilidad o comunicación, hacen de VHDL un lenguaje muy usado para el diseño automático.

En el caso de este proyecto, se ha usado como entorno de desarrollo *Xilinx ISE Design Suite*. Este entorno, además de proveer de una amplia biblioteca de módulos, se encarga de todo el proceso de síntesis a partir de la descripción, proporcionando el mapa de bits listo para ser cargado en la memoria de configuración de la FPGA.

3.3.- VENTAJAS DE LAS FPGAS

A continuación procederemos a enumerar algunos de los principales beneficios de usar FPGAs como plataformas para computación de propósito general.

- **Aumento de la velocidad de procesamiento**: Gracias a la capacidad de un diseño *hardware* específico y a la posibilidad de explotar las opciones de paralelismo en una aplicación utilizando los dispositivos básicos disponibles y un diseño modular se incrementa enormemente la velocidad de procesamiento respecto a la ejecución de un código *software* en un procesador de propósito general.
- **Reducción de consumo**: La tecnología usada para fabricar las FPGAs hace gala de unos índices de consumo muy aceptables dentro del mercado de la computación de alto rendimiento [28]. Por si ésto no fuera suficiente, el simple hecho de un procesamiento eficiente de la aplicación a realizar y la posibilidad de activar solo las áreas que están realizando cálculos en cada momento, dejan clara la ventaja de las FPGAs respecto de otras plataformas.
- **Flexibilidad**: El hecho de poder reconfigurar dinámicamente, tanto parcial como totalmente, los diseños implementados sobre FPGA, aporta la flexibilidad necesaria

para adaptarse a un mercado en continuo cambio.

- **Coste**: El incremento en la demanda de FPGAs y el consecuente aumento de la línea de producción han conseguido junto con los avances tecnológicos la disminución de los precios de estos dispositivos hasta niveles muy comerciales.
- **Entornos y herramientas de desarrollo**: Si bien el proceso de síntesis automática de sistemas podía resultar complejo en sus comienzos, las herramientas y entornos de desarrollo de los que se dispone hoy en día facilitan mucho esta tarea, sin contar con la gran cantidad de documentación y ejemplos que pueden encontrarse.

3.4.- INCONVENIENTES DE LAS FPGA

A pesar de sus grandes ventajas, caben destacar algunos factores a tener en cuenta antes de decantarse por un diseño basado en FPGA.

- **Rutado de señales**: Las herramientas proveen de un rutado de las señales a partir de la descripción del circuito que no siempre es óptimo, y la tarea de revisión y retoque manual no es simple. No obstante, los fabricantes trabajan en soluciones a este problema, tales como las nuevas FPGAs con diseño en 3D [20, 29].
- **Tiempo de reconfiguración**: Uno de los argumentos más comunes de los detractores del uso de las FPGAs dinámicamente reconfigurables para la ejecución multitarea *hardware* es el tiempo de reconfiguración total o parcial. Al suponer un cuello de botella en este tipo de sistemas, es una de los principales objetivos a mejorar y numerosos esfuerzos están siendo dirigidos en pos de nuevas arquitecturas o tecnologías que permitan paliar este efecto [30-33].

3.5.- *HARDWARE* RECONFIGURABLE EN MISIONES DE OBSERVACIÓN REMOTA

Ya se han mencionado anteriormente el bajo coste de los dispositivos, así como su alta capacidad de cómputo y el consumo reducido que poseen. Estas características hacen del *hardware* reconfigurable, y en especial de las FPGAs, una plataforma especialmente útil en aplicaciones que requieran una gran capacidad de procesamiento a bordo, tales como las misiones espaciales, donde ya se han utilizado (*Mars Pathfinder* y *Mars Surveyor* entre otras [34, 35]). La capacidad de reconfiguración aporta una gran flexibilidad necesaria en caso de producirse errores o cambios en los objetivos de la misión. Además, los dispositivos empleados en este tipo de misiones necesitan protección frente a los efectos de la radiación, para lo cual ya se fabrican FPGA endurecidas.

Cabe destacar que en las misiones de observación remota se genera un gran volumen de datos que debe ser transmitido desde los satélites hasta las estaciones terrestres. Las FPGAs, como dispositivos de computación de alto rendimiento, podrían desempeñar la función de preprocesamiento de los datos, disminuyendo así el flujo de datos necesario, y debido a su bajo consumo y coste y la resistencia a la radiación, no suponen un aumento excesivo en la carga o *payload* del satélite. Esto aparece ejemplificado en la Figura 3.3.

Como ya se ha comentado, la capacidad de reconfiguración aporta una capa de seguridad extra, pudiendo restablecer la configuración inicial de la FPGA en caso de producirse algún fallo, o cargar una nueva configuración para cambiar la función que desempeña. Para ello no sería necesaria una nueva misión, sino simplemente transmitir el nuevo mapa de bits de configuración y proceder a su reconfiguración. Si lo comparamos con la alternativa del *hardware* específico, el tiempo de desarrollo de un nuevo sistema es mucho menor en FPGA, y el *hardware* necesitaría de una misión encargada de implantar el nuevo sistema.

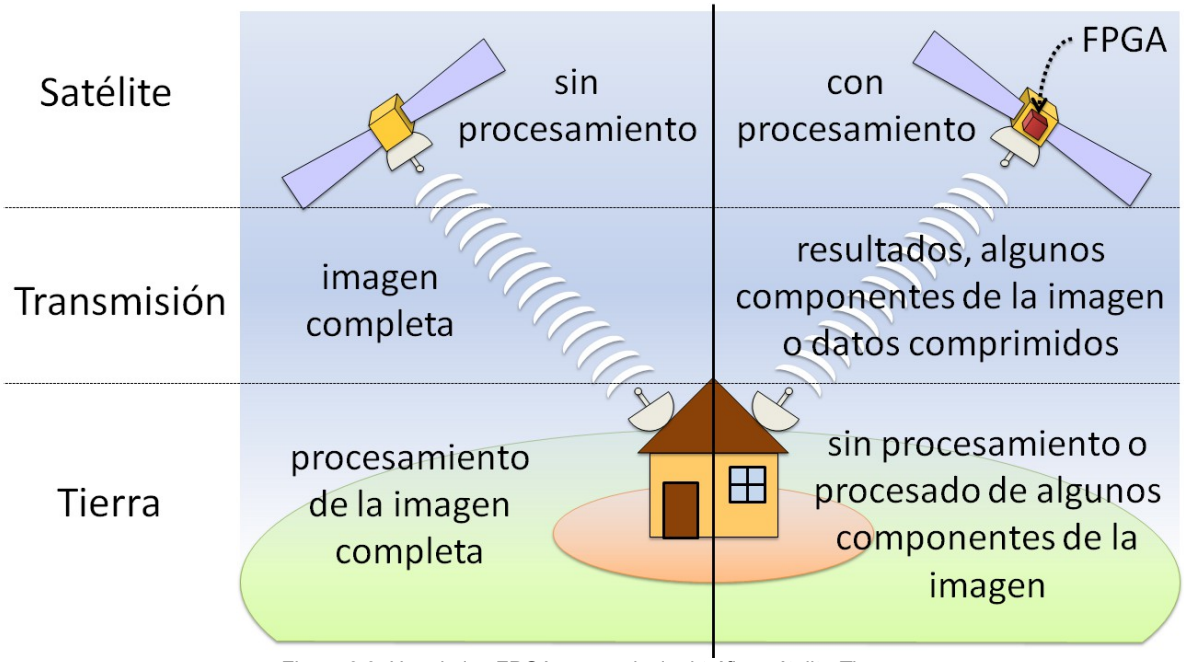


Figura 3.3. Uso de las FPGA para reducir el tráfico sátelite-Tierra.

En la actualidad ya existen diversas misiones de observación remota que cuentan con sistemas basados en microprocesadores y periféricos *hardware* dedicados. Las nuevas FPGAs que incluyen microprocesadores empotrados podrían suponer una mejora significativa respecto de estos sistemas, aportando las principales ventajas de un *hardware* de propósito general y un *hardware* de diseño específico.

CAPÍTULO 4. DETECCIÓN DE *TARGETS*

En los capítulos anteriores ya se ha hablado de la diferencia entre las firmas espectrales puras y mixtas, así como del desmezclado espectral como medio de obtención de los *endmembers* o firmas espectrales puras presentes en la imagen. Este enfoque está orientado a un posterior procesamiento de la imagen con objeto de obtener la presencia cuantitativa de dichos *endmembers* en cada píxel.

En contraste con estos métodos aparecen los procedimientos encaminados a la obtención de *targets*, como es el caso del algoritmo ATGP implementado en este proyecto. Los *targets* son objetivos con características espectrales distintas. Es decir, el conjunto de *targets* de una imagen ofrece una colección de firmas espectrales muy diferentes entre sí representativa de los distintos elementos que se pueden encontrar en la imagen.

Es importante destacar que si bien los conceptos de *endmember* y *target* resultan similares puesto que un conjunto de *endmembers* muestra firmas espectrales muy diferenciadas al igual que un conjunto de *targets*, los primeros están asociados a firmas espectralmente puras, mientras que los segundos simplemente son una muestra representativa de los objetos presentes en la imagen.

Los algoritmos de detección de *targets* suelen funcionar iterativamente, obteniendo cada *target* en una iteración del algoritmo. Como resultado de la ejecución del algoritmo se obtiene el conjunto de *targets* detectados.

Algoritmos como el ATGP requieren de una inicialización del conjunto de *targets*. Una inicialización usada frecuentemente es utilizar como primer *target* el píxel más brillante (con más intensidad) de la imagen, si bien no es la única alternativa viable. No obstante, experimentalmente se demuestra que dicho píxel siempre figura en el conjunto de píxeles de la solución [36], lo que certifica la elección del píxel más brillante como primer *target* como correcta.

4.1.- ALGORITMO ATGP

Este algoritmo usa el concepto de proyección ortogonal de un subespacio para hallar los *targets* en la imagen. Inicializando el subespacio con el píxel más brillante de la imagen, el siguiente píxel será aquel con el valor de proyección más alto. Este píxel es el siguiente objetivo detectado, que se incluye en el subespacio, repitiendo el proceso tantas veces como objetivos se quieran detectar. A continuación se muestra una implementación de este algoritmo en pseudocódigo.

Entradas al algoritmo: La imagen hiperespectral *F* con un número *r* de píxeles y el número *t* de objetivos.

```
U[:,0] = X0

# Siendo X0 el píxel más brillante de F. ":" indica que son referidos todos los
elementos.

para i = 1 hasta t-1 hacer
            P⊥U = I−U(UᵀU)⁻¹Uᵀ

            # Resultando P⊥U el vector ortogonal del subespacio contenido por U.

            V = P⊥UF
            j = argMax{1,..,r}V[:,j]

            # Obtención de la posición que ocupa el mayor valor proyectado.
```

```
            U[:,i] = F[:,j]

            # Se actualiza la matriz de objetivos con el componente

            Xi = F[:,i]
fin para
```

Salida de algoritmo: $U = [X_0, X_1, \ldots X_{t-1}]$.

CAPÍTULO 5. IMPLEMENTACIÓN EN FPGA DEL ALGORITMO ATGP

Como se ha explicado anteriormente, en este proyecto se ha llevado a cabo la implementación del algoritmo ATGP en FPGA. La implementación se ha realizado mediante distintos componentes especificados en un lenguaje de descripción *hardware*, en este caso, VHDL. En este apartado se explica la adaptación del algoritmo realizada para la implementación *hardware* así como los módulos empleados para ello.

5.1.- ALGORITMO IMPLEMENTADO

Para su implementación en FPGA se ha llevado a cabo una adaptación del algoritmo con objetivo de aprovechar las características que ofrece esta plataforma, minimizando en lo posible el *hardware* empleado y el tiempo de ejecución. El algoritmo implementado es el siguiente:

Entradas al algoritmo: La imagen hiperespectral *F* con un número *r* de píxeles, el número *t* de objetivos y el número de bandas *n.*

```
#Cálculo del píxel más brillante

max = 0
pos = 0
para i = 0 hasta r-1 hacer
       brillo = F[:,i]*F[:,i]^T
       si (max<=brillo) entonces
                     max = brillo
                     pos = i
       fsi
fpara
U[:,0] = F[:,pos]
Upos[0] = pos

#Cálculo del resto de píxeles anómalos

para i = 1 hasta t-1 hacer

       #Cálculo de P⊥U

       aux1 = U[:,0..(i-1)]^T*U[:,0..(i-1)]
       aux2 = I[0..(i-1),:]
       inversa(aux1,aux2)        #aux2 = aux1[0..(i-1),:]^-1
       aux1 = aux2[0..(i-1),:]*U[:,0..(i-1)]^T
       aux2 = I[0..(i-1),:]-U[:,0..(i-1)]*aux1       #aux2 = P⊥U

       #Cálculo del píxel cuya proyección respecto de P⊥U tenga mayor módulo

       max = 0
       pos = 0
       para j = 0 hasta r-1 hacer
                     x = aux2[0-(i-1),:]*F[:,j]
                     mod = x*x^T
                     si (max<=mod) entonces
                            max = mod
                            pos = i
                     fsi
       fpara
```

```
      U[:,i] = F[:,pos]
      Upos[i] = pos
fpara
```

Salida del algoritmo: *Upos* = $[i_0,i_1,\ldots,i_{t-1}]$ siendo i_j la posición en *F* de X_j.

La equivalencia de este algoritmo con respecto al descrito en el apartado 4 es evidente, y de esta manera se consigue crear una unidad de control y ruta de datos bien estructuradas, reutilizando el *hardware* disponible y disminuyendo la cantidad de memoria necesaria.

Reducir el almacenamiento de datos en memorias internas es crucial en desarrollos sobre plataformas de *hardware* reconfigurable en general, y en FPGA en particular, ya que la cantidad de memoria disponible en forma de módulos *hardware* es limitada, y aunque pueden utilizarse LB para implementar memorias, se requiere una gran cantidad de recursos que se podrían destinar para aumentar el paralelismo o realizar otros cálculos.

5.2.- VISIÓN GENERAL

Para la implementación *hardware* del algoritmo ATGP se ha llevado a cabo un diseño modular del sistema, siempre siguiendo el esquema tradicional de Unidad de Control y Ruta de Datos. Se han desarrollado módulos específicos para las principales operaciones llevadas a cabo en el algoritmo, véase el cálculo de la inversa de una matriz, la multiplicación de matrices, el cálculo del cuadrado del módulo de un vector, la resta entre la matriz identidad y otra matriz y la inicialización de una matriz con la matriz identidad. Todos estos módulos son lo suficientemente genéricos como para ser reutilizados en otros diseños que lo requieran, siendo parametrizados según las necesidades del sistema.

Todos los módulos se desarrollan bajo el mismo patrón de comportamiento:

- Los módulos se encuentran inicialmente en un estado de espera.
- Los módulos que realizan operaciones de lectura y escritura de memorias comienzan a operar al activarse una señal de comienzo. Estos módulos deben encargarse de direccionar las lecturas y escrituras.
- Los módulos que esperan recibir una secuencia de datos leen cada dato cuando se activa la señal de operar. Una vez leído el último dato esperan a terminar el cálculo.
- Todos los módulos activan la señal de finalización al terminar.

5.3.- COMPONENTES EMPLEADOS

A continuación se detallan los componentes empleados para realizar la implementación del algoritmo.

Cabe destacar que todos los datos usados corresponden a números en punto flotante con precisión simple, según el estándar *IEEE 754*.

La lectura en las memorias empleadas es síncrona, es decir, no se dispone de un dato direccionado hasta el comienzo del ciclo siguiente. La escritura también es síncrona, por lo que al activar la señal de escritura, ésta no es efectiva hasta el ciclo siguiente.

5.3.1.- IP-CORES

Durante el desarrollo de la implementación del algoritmo hemos precisado del uso de algunos componentes básicos para la construcción de módulos o memorias. Estos componentes han podido ser añadidos para su uso sin la necesidad de ser implementados por nuestra parte gracias a una herramienta disponible dentro del software de desarrollo utilizado (*Xilinx ISE Design Suite*), el *IP-Core Generator*.

Los módulos utilizados para la implementación en VHDL del algoritmo ATGP se listan a continuación indicando las características de diseño a establecer en la herramienta de generación de *IP-Cores*.

5.3.1.1.- OPERADORES DE PUNTO FLOTANTE

Estos componentes comparten determinadas características entre las que cabe destacar el uso de la señal *operation_nd*, la cual indica al módulo que debe comenzar, y la presencia de la señal *ready*, indicativo de que el componente ha terminado de operar.

Además es importante añadir que estos módulos trabajan en punto flotante con precisión simple y que admiten una secuencia de operaciones a modo de *pipeline*, es decir, a pesar de que una operación tarde en finalizar 5 ciclos (latencia = 5), si hay un suministro consecutivo de datos (uno cada ciclo) desde el primer ciclo, el primer resultado será en el 5º ciclo, el segundo en el 6º ciclo y de la misma forma para los demás datos introducidos de esta manera.

- Sumador PF(add01)
  Características de diseño: Sumador de punto flotante con precisión simple, latencia 1, con *operation_nd* y *ready*.
- Restador PF(sub01)
  Características de diseño: Restador de punto flotante con precisión simple, latencia 1, con *operation_nd* y *ready*.
- Multiplicador PF(mult01)
  Características de diseño: Multiplicador de punto flotante con precisión simple, latencia 1, con *operation_nd* y r*eady*.
- Divisor PF(div05)
  Características de diseño: Divisor de punto flotante con precisión simple, latencia 5, con *operation_nd* y *ready*.
- Comparador PF(goe_comp01)
  Características de diseño: Comparador mayor o igual de punto flotante con precisión simple, latencia 1, con *operation_nd* y *ready*.

5.3.1.2.- BLOQUES DE MEMORIA

Para optimizar los recursos empleados en el almacenamiento de las matrices necesarias para la realización del algoritmo, es recomendable usar el generador de bloques de memoria incluido en el *IP-Core Generator*. Esta utilidad permite diseñar bloques de memoria del tamaño deseado y con características tales como doble puerto para realizar dos lecturas/escrituras simultáneas, determinar la prioridad entre lectura y escritura o configurar los tamaños y número de filas para cada puerto.

Para la implementación del algoritmo ATGP hemos utilizado dos bloques de memoria distintos:

- Módulo de memoria 1 (memBasica)
  Características de diseño: Memoria, *True Dual Port*, datos de 32 bits y 512 filas en ambos puertos.
- Módulo de memoria 2 (memRowCol)
  Características de diseño: Memoria, *True Dual Port*, datos de 32 bits y 64 filas por el puerto A, datos de 1024 bits y dos filas por el puerto B.

Con objeto de asegurar la correcta lectura de los datos en las memorias creadas con estos bloques, se definen 3 etapas distintas para la lectura, o lo que es lo mismo, la lectura se realiza en 3 ciclos:

- **Preparar lectura**: Para evitar posibles problemas derivados por problemas de propagación y *set-up*, las señales encargadas de direccionar las filas para la lectura deben ser síncronas. Es la manera de implementar el cambio en la señal sincronizado con el ciclo de reloj, la asignación se prepara en el ciclo anterior y pasará a tener el valor asignado con la llegada del flanco de reloj. Éste será pues el ciclo en el que se haga la asignación, no haciéndose efectiva hasta el ciclo siguiente.
- **Lectura del dato:** En esta etapa ya se ha hecho efectiva la asignación de la señal que direcciona la fila de la memoria. No obstante, los datos que está dando la memoria no corresponden a esta lectura.
- **Obtención del dato**: En este ciclo ya se muestra la fila leída por el correspondiente puerto de la memoria.

5.3.2.- MEMORIA SIMPLE (MEM.VHD)

Esta memoria de doble puerto permite dos lecturas/escrituras simultáneas. La memoria está estructurada para guardar las matrices que se utilizan en el algoritmo, es decir, cada dato son 32 bits, los necesarios para almacenar un número en punto flotante, y cada fila de la memoria consta de tantos datos como columnas tenga la matriz, siendo configurable en cada instancia del módulo. En cada lectura se proporciona en un puerto una fila entera y en otro el dato de una columna de esa fila. La escritura puede realizarse también a nivel de fila o de dato, siendo excluyentes ambas escrituras.

La principal característica de esta memoria es que permite intercambiar los datos de dos filas entre sí, funcionalidad muy conveniente para la implementación del algoritmo de Gauss para calcular la inversa (descrito más adelante).

Las señales de entrada de este módulo son:

- *rowA*, *rowB*: Señales para el direccionamiento de filas en cada puerto.
- *colA*, *colB*: Señales para el direccionamiento de datos/columnas en cada puerto.
- *dinRowA*, *dInRowB*: Datos de escritura a nivel de línea en cada puerto.
- *dInA*, *dInB*: Dato de escritura a nivel de dato en cada puerto.
- *wrRowA*, *wrRowB*: Señales de escritura a nivel de línea en cada puerto.
- *wrA*, *wrB*: Señal de escritura a nivel de dato en cada puerto.
- chgRows: Señal para el intercambio de líneas.

Las señales de salida de este módulo son:

- *dOutRowA*, *dOutRowB*: Señal con la fila leída.

- *dOutA*, *dOutB*: Señal con el dato leído.

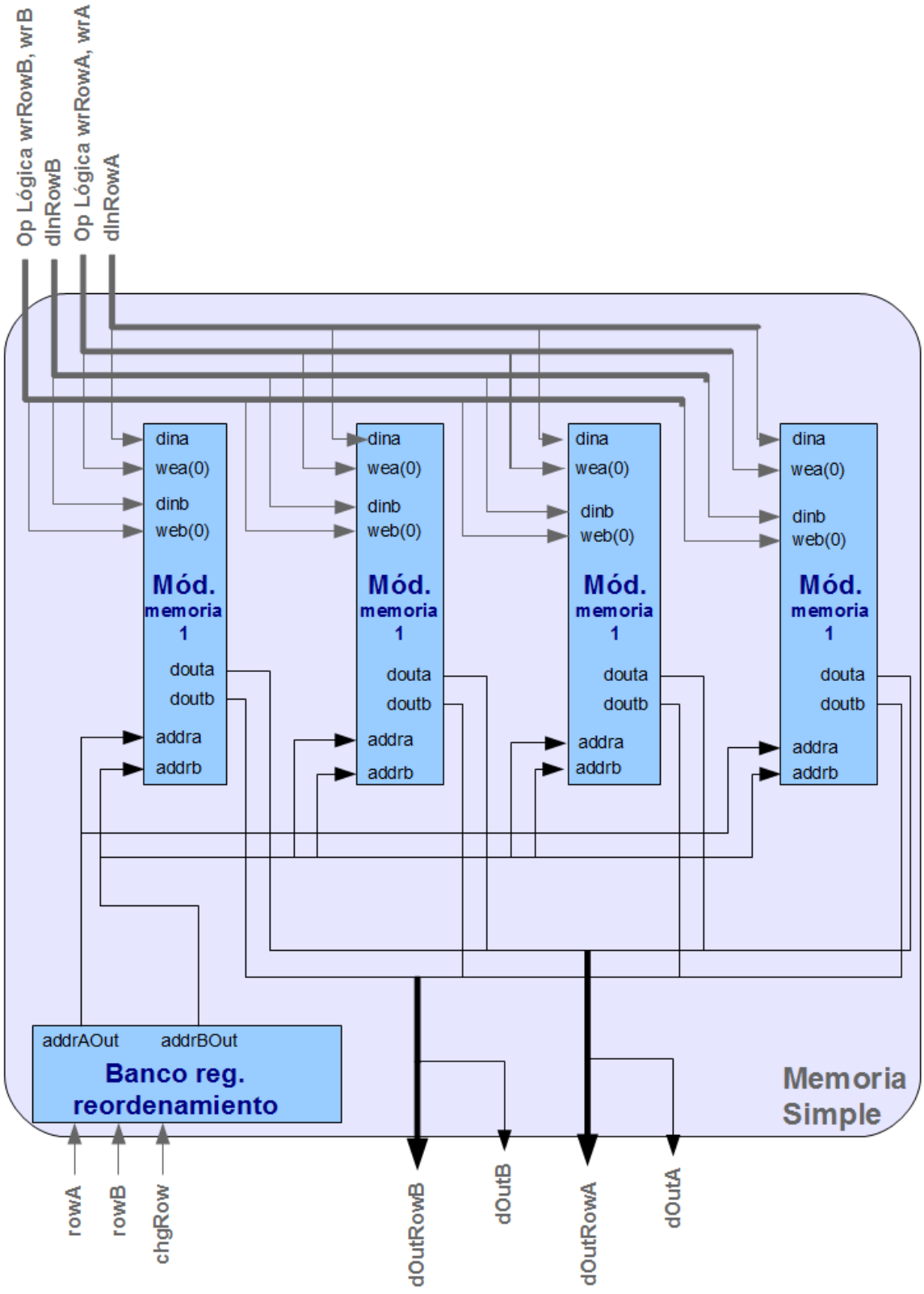


Figura 5.1. Esquema de la memoria simple.

Este módulo está compuesto por:

- *Módulo de memoria 1*: Módulo de memoria de palabras de 32 bits con doble puerto. Cada módulo almacena una columna de la matriz.
- *Banco de registros de reordenamiento*: Tiene tantos registros como filas tenga la matriz. Inicialmente cada registro está cargado con el valor de la fila que tiene asociada y al realizar un cambio de filas se intercambian los valores de esos registros.

La estructura de esta memoria se realiza instanciando tantos módulos de memoria como columnas tenga la matriz más grande que deba almacenar. Todos los bloques comparten el direccionamiento de cada puerto, es decir, la señal de direccionamiento de fila del puerto A es la misma para todos los bloques, y de igual manera con el puerto B. El dato de la fila leída se obtiene concatenando los datos leídos de cada bloque y, para la lectura a nivel de dato, sólo se selecciona aquel obtenido de la memoria que almacena la columna direccionada. La señal de escritura de cada bloque se activa en caso de escritura de fila o en caso de escritura de dato en esa columna.

El direccionamiento a los bloques se realiza mediante el banco de registros de direccionamiento. Este banco permite direccionar dos registros al mismo tiempo e intercambiar los valores de los dos registros direccionados. Este es el mecanismo usado para implementar el cambio de filas en la memoria, ya que es la salida de este módulo la que realmente direcciona las filas leídas en los bloques de memoria. Al estar inicializado cada registro con el número de fila que tiene asociado, la lectura se realiza con normalidad. Al intercambiar el valor de dos registros, por ejemplo, el 3 y el 6, el registro 3 tendría guardado el valor 6 y al revés, de manera que cuando en otro momento se quiera leer la fila 3 de la memoria, el banco de registros de reordenamiento direccionará la fila 6 en los bloques de memoria.

5.3.3.- MEMORIA POR FILAS Y COLUMNAS (MEMU.VHD)

Esta memoria también está estructurada para almacenar las matrices del algoritmo, de manera que nos referimos con filas, columnas y datos como si de una matriz se tratase. Permite la lectura por filas o por dato por uno de los puertos y la lectura por columnas por el otro puerto, lo que resulta idóneo para almacenar matrices de las que se necesite leer una fila o una columna indistintamente, o lo que es lo mismo, se necesiten leer filas de una matriz y de su traspuesta.

Las señales de entrada de este módulo son:

- *rowA*: Señal para el direccionamiento de filas en el puerto A.
- *colA*: Señal para el direccionamiento de datos en el puerto A.
- *colB*: Señal para el direccionamiento de columnas en el puerto B.
- *dInRowA*: Datos de escritura a nivel de palabra en el puerto A.
- *dInColB*: Datos de escritura a nivel de columna en el puerto B.
- *dInA*: Dato de escritura a nivel de dato en el puerto A.
- *wrRowA*: Señal de escritura a nivel de fila en el puerto A.
- *wrColB*: Señal de escritura a nivel de columna en el puerto B.
- *wrA*: Señal de escritura a nivel de dato en el puerto A.

Las señales de salida de este módulo son:

- *dOutRowA*: Señal con la fila leída.
- *dOutColB*: Señal con la columna leída.
- *dOutA*: Señal con el dato leído.

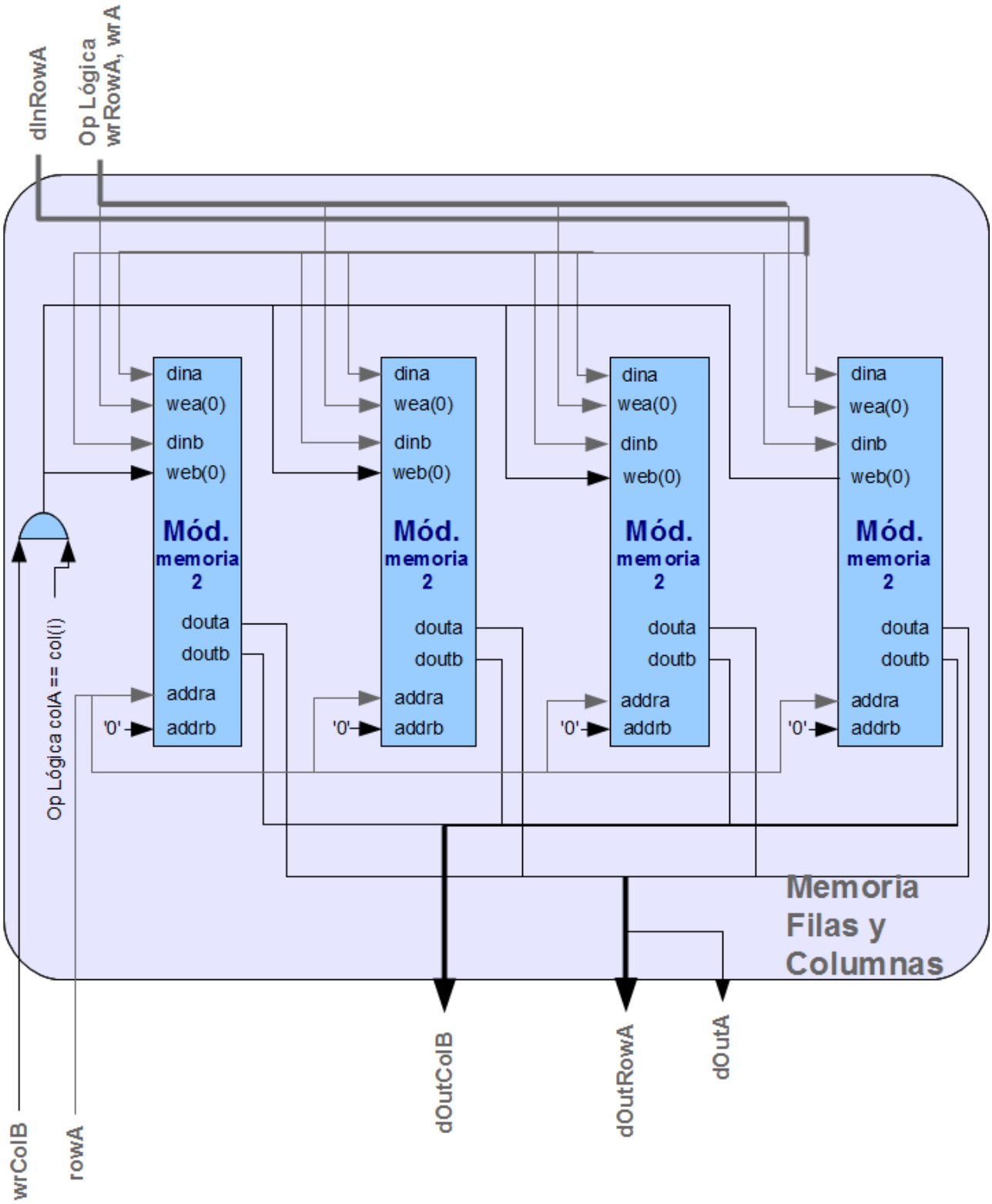


Figura 5.2. Esquema de la memoria por filas y columnas

Este módulo está compuesto por:

- *Módulo de memoria 2*: Módulo de memoria de líneas de 32 bits por el puerto A y 1024 por el puerto B. Cada módulo almacena una columna de la matriz.

La estructura de esta memoria en lo referente a la distribución de los bloques y al puerto A es idéntica a la *Memoria Simple*. En lo que corresponde al puerto B, siempre se esta leyendo la fila 0 por el puerto B, lo que corresponde a una columna entera de la matriz. Al direccionar una columna en la memoria por el puerto B se obtiene el mostrado por el bloque correspondiente a esa columna, funcionando del mismo modo la escritura por columnas. Ya que la lectura por columnas es combinacional, porque los datos de la lectura están disponibles siempre y sólo es necesario multiplexar entre los módulos, se añade un registro a la salida para que el comportamiento temporal de ambas memorias sea el mismo.

5.3.4.- MÓDULO MULTIPLICADOR DE VECTORES (VECTOR_N_MULT.VHD)

Este módulo realiza el producto escalar de dos vectores dados. Está compuesto por una fila de multiplicadores y un árbol de sumadores de manera que para dos vectores A = [a_0, a_1,..., a_n] y B = [b_0, b_1, ,..., b_n] primeramente calcula el producto de sus componentes uno a uno resultando [a_0*b_0, a_1*b_1,..., a_n*b_n].

Cada una de estas multiplicaciones se suman 2 a 2 entre sí, obteniendo [$a_0*b_0+a_1*b_1$, $a_2*b_2+a_3*b_3$,…, $a_{n-1}*b_{n-1}+a_n*b_n$] en el primer nivel. La suma de 2 a 2 se repite a través de los niveles hasta obtener el resultado del producto escalar. Los vectores pueden ser de cualquier dimensión menor o igual al máximo fijado en el componente. Al variar este máximo varían el *hardware* empleado y el tiempo de ejecución.

Este módulo es instanciado dentro de los módulos *Multiplicador de Matrices* y *Máximo Brillo*.

Las señales de entrada de este módulo son:

- *operate*: Señal de operar.
- *dataA* y *dataB*: Datos de ambos vectores.

Las señales de salida de este módulo son:

- *rdy*: Señal de fin.
- *result*: Dato de salida.

Este módulo está compuesto por:

- *Multiplicador PF*: Multiplicador de punto flotante con precisión simple. Realizan la multiplicación de cada a_i*b_i.
- *Sumador PF*: Sumador de punto flotante con precisión simple. Componen el árbol que suma cada resultado de los multiplicadores hasta obtener el escalar resultado.

Dada su sencillez, este módulo carece de unidad de control y el flujo de los datos es sólo cableado:

- La señal de operar del módulo es conectada a la señal de operar de los multiplicadores.
- La señal de operar de cada sumador del árbol de sumadores es la conjunción de las señales de fin de los dos operadores que proporcionan los operandos para ese módulo.
- La señal de fin del módulo es la señal de fin del último sumador del árbol.

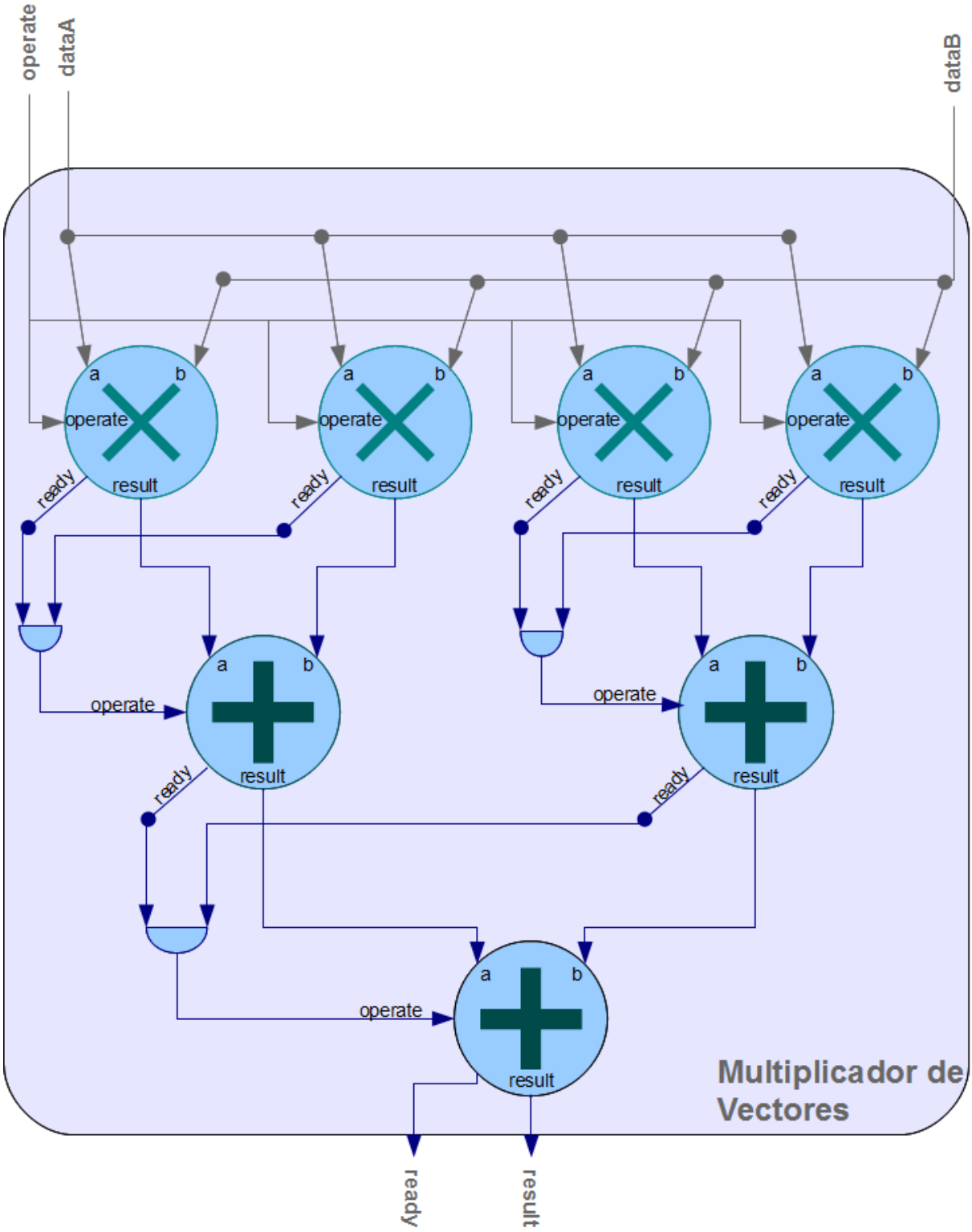


Figura 5.3. Esquema del módulo multiplicador de vectores.

5.3.5.- MÓDULO MULTIPLICADOR DE MATRICES (MATRIXMULT.VHD)

Este módulo puede realizar la multiplicación de dos matrices $A_{(fxn)}*B_{(nxc)}$ o la multiplicación de una matriz por su traspuesta, dependiendo del modo en el que se le indique operar. Estos dos modos operan de una manera diferente, ya que al multiplicar una matriz por su traspuesta sólo utilizamos el puerto del operando A para la lectura de datos. Las diferencias entre estos dos modos serán explicadas de forma más detallada en el estado que los diferencia. En ambos casos, el número de columnas de la matriz A (y en el caso de dos matrices, el número de filas de la matriz B) viene dado por un máximo fijado por el componente, pudiendo operarse cualquier par de matrices A y B con *n* menor o igual a este máximo. Al variar este máximo varían el *hardware* empleado y el tiempo de ejecución.

En esta implementación, el máximo viene dado por el número de bandas de la imagen hiperespectral.

Las matrices A y B deben estar almacenadas en memorias de manera que pueda leerse la matriz A por filas y la matriz B por columnas. El resultado es escrito dato a dato.

Las señales de entrada de este módulo son:

- *start*: Señal de inicio.
- *modeIN*: Modo de multiplicación (si con una sola memoria o con 2).
- *rowsIN*: Número de filas de la primera matriz.
- *colsIN*: Número de columnas de la segunda matriz.
- *dataAIN*: Datos de una fila de la primera matriz.
- *dataBIN*: Datos de una columna de la segunda matriz

Las señales de salida de este módulo son:

- *finish*: Señal de fin.
- *rowA*: Fila direccionada de la matriz A.
- *colB*: Columna direccionada la matriz B.
- *rowC* y *colC*: Fila y columna que direccionan el dato en la memoria en que se escribe la matriz resultado.
- *dataCOUT*: Dato de salida.
- *memWriteC*: Señal de escritura en la memoria.

Su ruta de datos está compuesta por dos módulos:

- *Multiplicador de vectores*: Módulo que calcula el producto escalar de dos vectores. El módulo se instancia con el número de bandas como máximo tamaño de los vectores. Multiplica las filas de la matriz A por las columnas de la matriz B.
- *Controlador de escritura (Multiplicador de matrices)*: Este módulo se encarga de direccionar la fila y columna donde se van a escribir los resultados de las multiplicaciones y de avisar a la unidad de control cuando dichas escrituras hayan terminado.

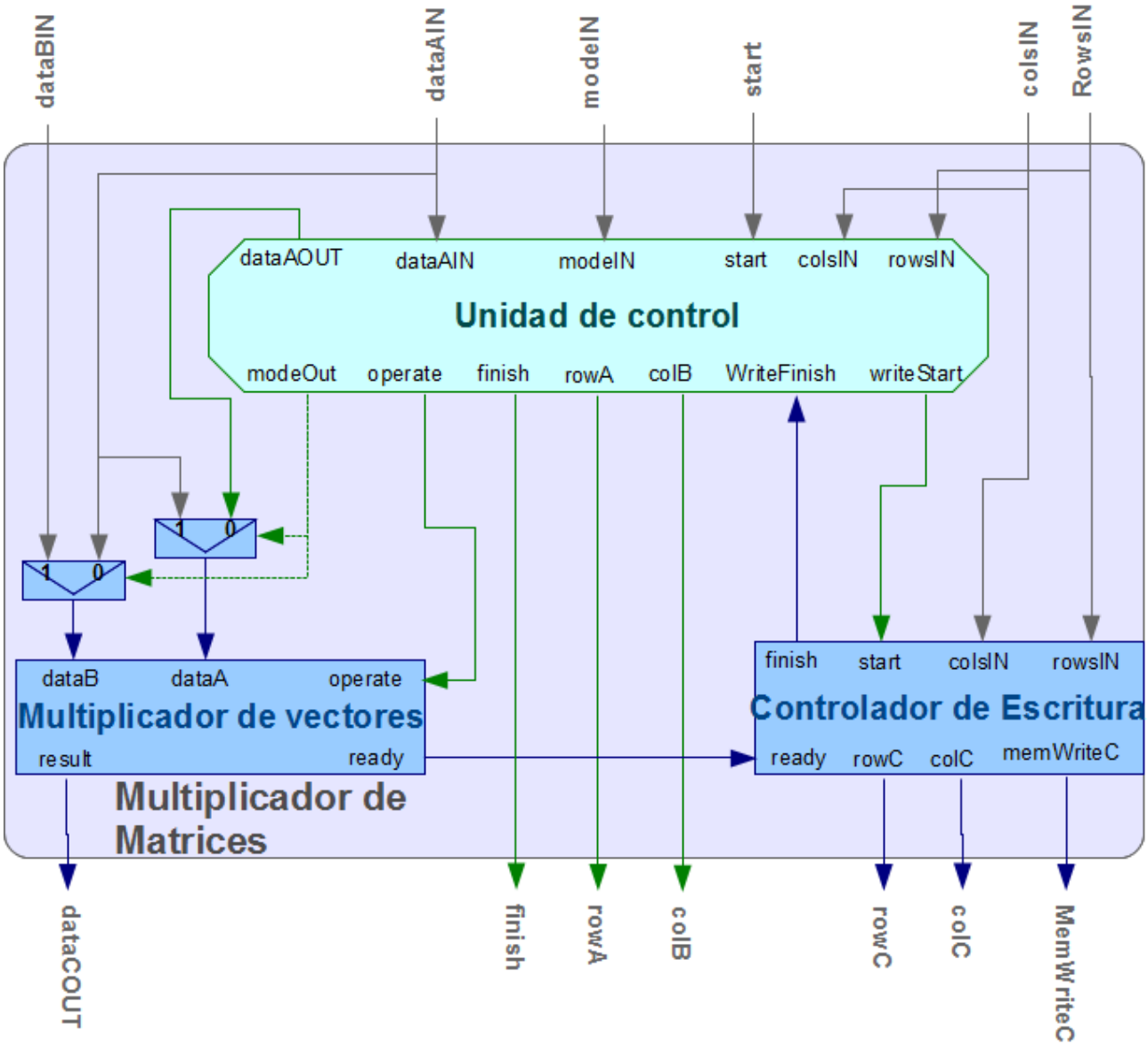


Figura 5.4. Esquema del módulo multiplicador de matrices.

La unidad de control de este módulo se basa en 5 estados. Para facilitar su explicación supondremos 2 memorias, A y B, siendo A la que pueda multiplicarse por su traspuesta (A^T). Denotaremos como i la fila de A con la que se está operando en ese momento, y del mismo modo se usará j para las columnas de B y A^T. Nótese que se habla de recorrer y leer las columnas A^T a pesar de estar siendo direccionados los datos por la misma vía que A. Esto se realiza de esta manera para mantener una cierta similitud con el otro modo y por tanto facilitar su comprensión y se refiere al recorrido de leer A utilizando el dato como segundo operando ya que el primero (las filas de A) se encuentra en el registro. Esto es posible debido a que las filas de A son las columnas de A^T.

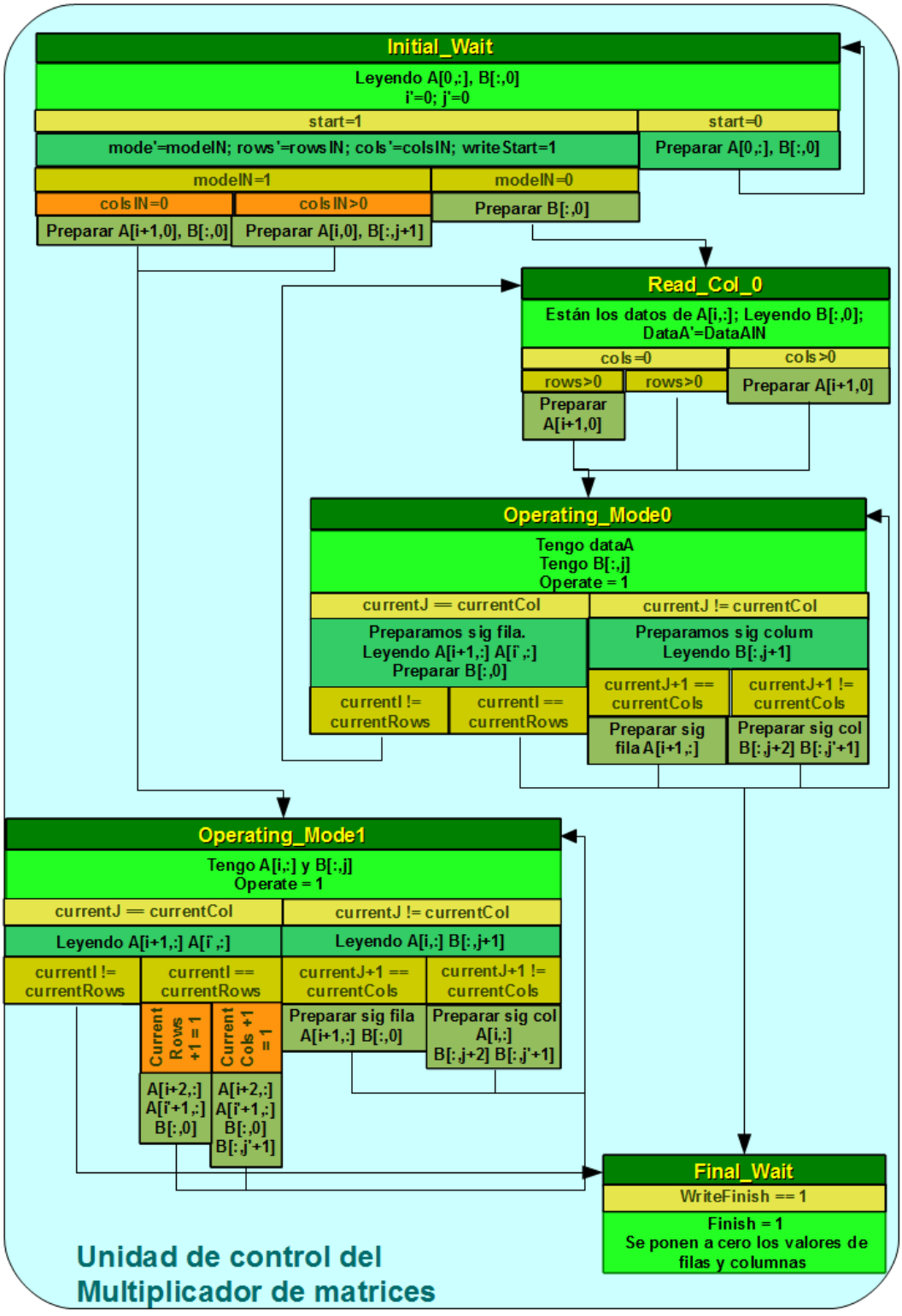


Figura 5.5. Esquema de la unidad de control del módulo multiplicador de matrices.

- **Initial_Wait**: En este estado se realiza la espera inicial al cálculo de la multiplicación. En él se leen A[0,:] y B[:,0] para asegurar su disponibilidad en el siguiente estado. Una vez se activa la señal de inicio se lleva a cabo una distinción de casos:
  - En caso de estar multiplicando una matriz por su traspuesta, preparamos la lectura de A[:,0] y en el siguiente ciclo se pasará al estado *Read_Col_0*.
  - En caso de estar multiplicando dos matrices se considera el caso que B sea una matriz con una única columna para preparar la lectura de la siguiente fila de A (A[i+1,:]) y B[:,0]. En caso contrario, se mantiene la lectura de la fila A y se prepara la siguiente fila de B (B[:,j+1]). En ambos casos se realiza la transición al estado *Operating_Mode1*.
- **Read_Col_0**: En este estado se guarda A[i,:] (leído en el ciclo anterior) en un registro y se lee A[0,:]. Se distingue el caso que el segundo operando tenga sólo una columna para preparar A[i+1,:], que si bien no tiene sentido a la hora de multiplicar una matriz por su traspuesta, añade genericidad. En caso contrario prepara A[j+1,:]. El siguiente estado será siempre *Operating_Mode0*.
- **Operating_Mode0**: Es en este estado cuando se realiza la multiplicación $A[i,:]*(A[j,:]^T)$. El primer operando se obtiene del registro cargado en Read_Col_0, y el segundo del valor que proporciona la memoria. Según los valores de i y j se realizan distintas acciones:
  - Si no se han terminado de recorrer las columnas de A^T al final del ciclo se incrementará la columna actual (j:= j+1). Debe de estar leyéndose A[j+1]. En caso de haberse operado con la penúltima columna, se prepara A[i+1,:] (necesario en *Read_Col_0*), si no, se prepara A[j+2,:] para continuar con el recorrido de las columnas de A^T . No se cambia de estado.
  - Si se ha terminado de recorrer A^T pero no A, al final de ciclo se asigna *j=0* y se incrementa *i*. Se pasa a *Read_Col_0* y se prepara A[1,:]. En este caso se estará leyendo A[i+1,:].
  - Si se han terminado de recorrer tanto A como A^T, se realiza la transición al estado *Final_Wait*.
- **Operating_Mode1**: En este estado se realiza la multiplicación de A[i,:]*B[:,j]. De modo análogo al estado *Operating_Mode0*, se preparan los datos que vayan a ser necesarios dos ciclos más adelante y se actualizan ciclo a ciclo, según corresponda, las filas y columnas. Sólo se cambia de estado una vez recorridas las matrices A y B, pasando al estado *Final_Wait*.
- **Final_Wait**: Este estado espera a que el controlador de escritura notifique de la finalización de las escrituras para activar la línea de fin y pasar a *Initial_Wait*.

Explicado de una forma más sencilla. Si tenemos dos matrices A y B la unidad de control leerá A[i,:] y B[:,j] (siendo en 1 estado si son matrices diferentes y en dos estados si se trata de una matriz y su traspuesta) y los pasará como argumento de entrada al módulo multiplicador de vectores. Cuando este módulo haya finalizado, el valor calculado se almacenará otra matriz en la fila y columna indicadas por el controlador de escritura.

Esta operación se repite para todas las filas y columnas de las matrices de entrada, sin esperar a que una multiplicación acabe para empezar con la siguiente.

5.3.6.- MÓDULO MAYOR BRILLO (MAXSHINE.VHD)

Este módulo calcula la posición del píxel con mayor módulo en una imagen de tamaño fijo, en este caso de la imagen F con *r* píxeles. Para ello, realiza la multiplicación escalar de un vector por sí mismo, lo que resulta en el cuadrado del módulo. Esta operación se lleva a cabo sobre la proyección de F sobre $P_{\perp U}$ y sobre F, equivaliendo en éste último caso a hallar el píxel más brillante, lo que da nombre al módulo.

Las señales de entrada de este módulo son:

- *operate*: Señal de operar.
- *xPixelData*: Dato del píxel.

Las señales de salida de este módulo son:

- *finish*: Señal de fin.
- *posOut*: Posición en la que se encuentra el píxel más brillante/con mayor módulo.

Módulos que componen la ruta de datos:

- *Multiplicador de vectores*: Módulo que calcula el producto vectorial de dos vectores. El módulo se instancia con el número de bandas como máximo tamaño de los vectores. Se encarga de calcular el cuadrado del módulo del vector multiplicando el vector por sí mismo.
- *Comparador PF*: Comparador de mayor o igual para números en punto flotante con precisión simple. Compara el valor máximo actual con el cuadrado del módulo del vector que se acaba de calcular.
- *Dato anterior*: Registro en el que se almacenará el valor de salida del módulo multiplicador de vectores.
- *Dato máximo*: Registro que almacenará el máximo valor de las salidas de multiplicador de vectores, es decir, el píxel más brillante hasta el momento, o aquel con mayor módulo al ser proyectado sobre $P_{\perp U}$.

La unidad de control es muy sencilla, únicamente se encarga de controlar la posición que ocupa el píxel en la matriz F. Se basa en 2 estados:

- **Operating**: Este es el estado inicial del módulo y donde permanece en espera de recibir los valores de los píxeles. Cuando el comparador produce su señal de *ready*, si el resultado del comparador indica que se ha obtenido un valor mayor o igual al máximo que teníamos hasta el momento, actualiza el valor de la posición resultado. De esta manera, la unidad de control siempre tendrá como salida la posición del máximo valor obtenido hasta el momento. Si se ha terminado de recorrer la imagen, se pasa al estado *FinalState*, si no, se incrementa el índice del próximo píxel que se va a comparar.
- **FinalState**: Activa la señal *finish* para indicar que ha terminado. En el caso de que vuelva a recibir la señal de operar pasara de nuevo al estado *Operating* reiniciando los índices y reseteando los registros.

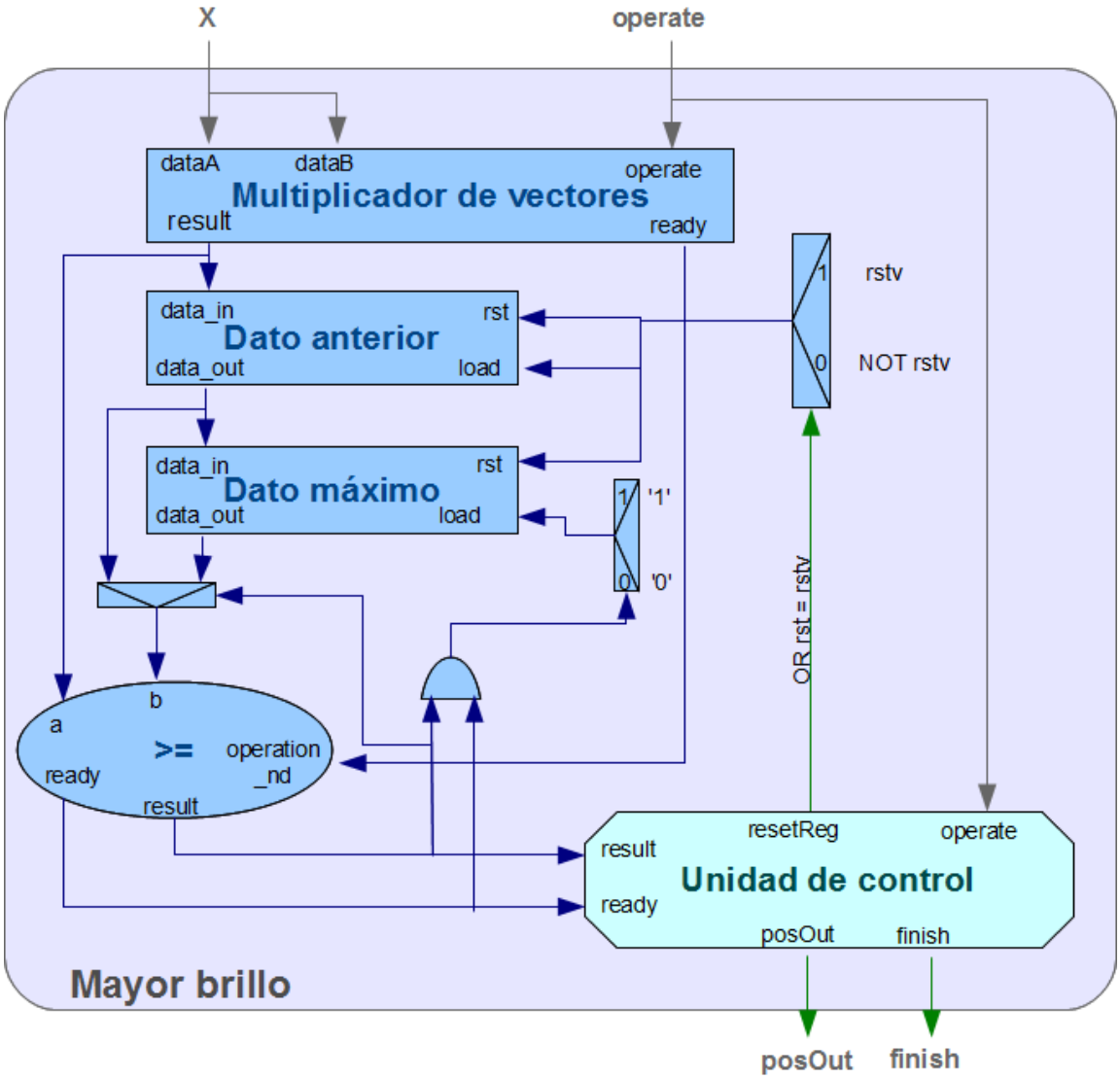


Figura 5.6. Esquema del módulo mayor brillo.

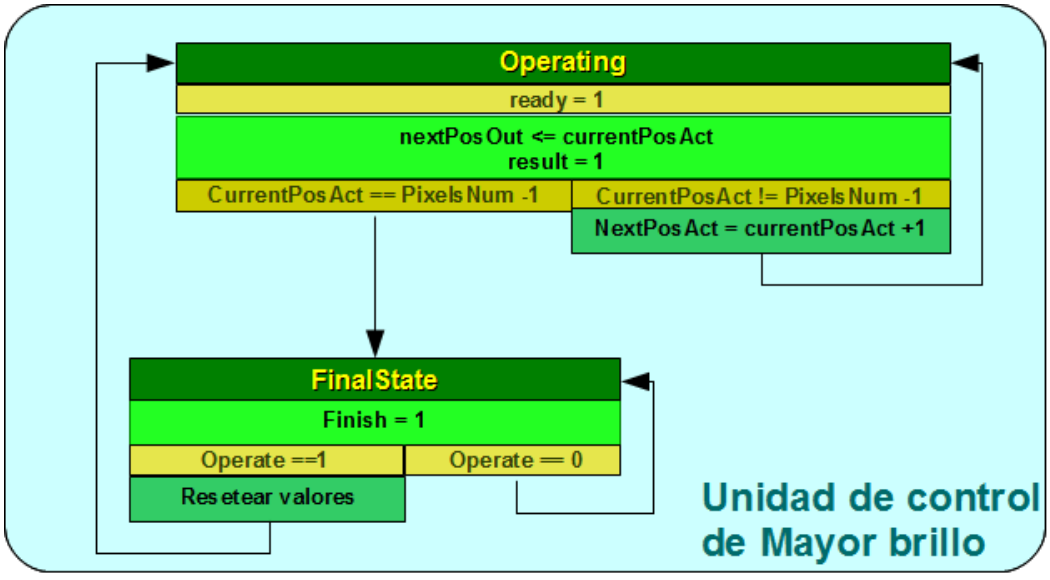


Figura 5.7. Esquema de la máquina de estados del módulo mayor brillo.

Esta unidad de control no explica de una forma detallada el funcionamiento del módulo si no el control sobre la posición del píxel que está evaluando en ese momento. Para entender mejor el funcionamiento de este módulo procedemos a una explicación de su modo de ejecución.

Supongamos que tenemos una imagen con *r* píxeles de entre los que hay que obtener el de mayor módulo y nos encontramos en el momento inicial con el píxel X_0.

El píxel X_0 entra por los dos puertos de *vector_N_mult*, que calcula el cuadrado del módulo de éste, es decir, $X_0 * X_0$. Una vez haya terminado este módulo, simultáneamente tendremos el valor del cuadrado del módulo de X_0 en el puerto A del comparador y a la entrada del registro *Dato anterior*, donde será almacenado. El valor del puerto B del comparador se escoge de entre la salida de *Dato anterior* y *Dato máximo* de manera que si *Dato máximo* debe actualizarse como resultado de la comparación anterior, el dato ese ciclo debe ser el guardado en *Dato anterior*, pues es el nuevo máximo. En ese momento se inicia la comparación.

Al ciclo siguiente, el valor de la comparación está disponible, y puede haber un nuevo dato a la salida del multiplicador de vectores. Si la comparación determina que A es mayor o igual que B, *Dato máximo* se actualizará con el valor de *Dato anterior* y, como se ha dicho antes, será ese nuevo valor el que se utilice en la comparación. Así mismo, se actualiza en la unidad de control la posición del dato que es máximo hasta el momento. La primera comparación siempre actualizará el máximo, puesto que el valor inicial de *Dato máximo* es 0 y el cuadrado de un número siempre es positivo.

Una vez hayamos llegado al último píxel. La unidad de control activará la señal de fin, teniendo en *posOut* la posición del píxel con mayor módulo.

5.3.7.- MÓDULO RESTADOR A LA MATRIZ IDENTIDAD (SUBTOI.VHD)

Este módulo es el encargado de restar una matriz a su identidad. Para evitar el uso de una memoria donde almacenar la identidad y teniendo en cuenta el contexto de su uso, esta resta se calcula dato a dato, de forma que en la diagonal de la matriz realizará '1 – A[i,i]' y a los elementos que no se encuentren en la diagonal calculará '0-A[i,j]'.

Las señales de entrada de este módulo son:

- *start*: Señal de comienzo de la operación.
- *dataIN*, *rowIN* y *colIN*: Dato a restar y posición (fila y columna) que ocupa en la matriz.
- *finishIN*: Señal de finalización del módulo superior.

Las señales de salida de este módulo son:

- *ready*: Señal de finalización de la operación.
- *dataOUT*, *rowOUT*, *colOUT*: Valor resultante de la resta con la posición en fila y columna que ocupa en la matriz .
- *finishOUT*: Señal de finalización del módulo superior retrasada un ciclo.

Los componentes de la ruta de datos son:

- *Reg. Fila*: Registro que almacena la posición de la fila que ha sido restada.
- *Reg. Columna*: Registro que almacena la posición de la columna que ha sido restada.
- *Reg. Fin*: Registro que almacena el valor *finishIN*.
- *Restador PF*: Restador de números en punto flotante de precisión simple. Realiza la resta de 1 ó 0 menos el valor actual según la posición que ocupe el dato en la matriz.

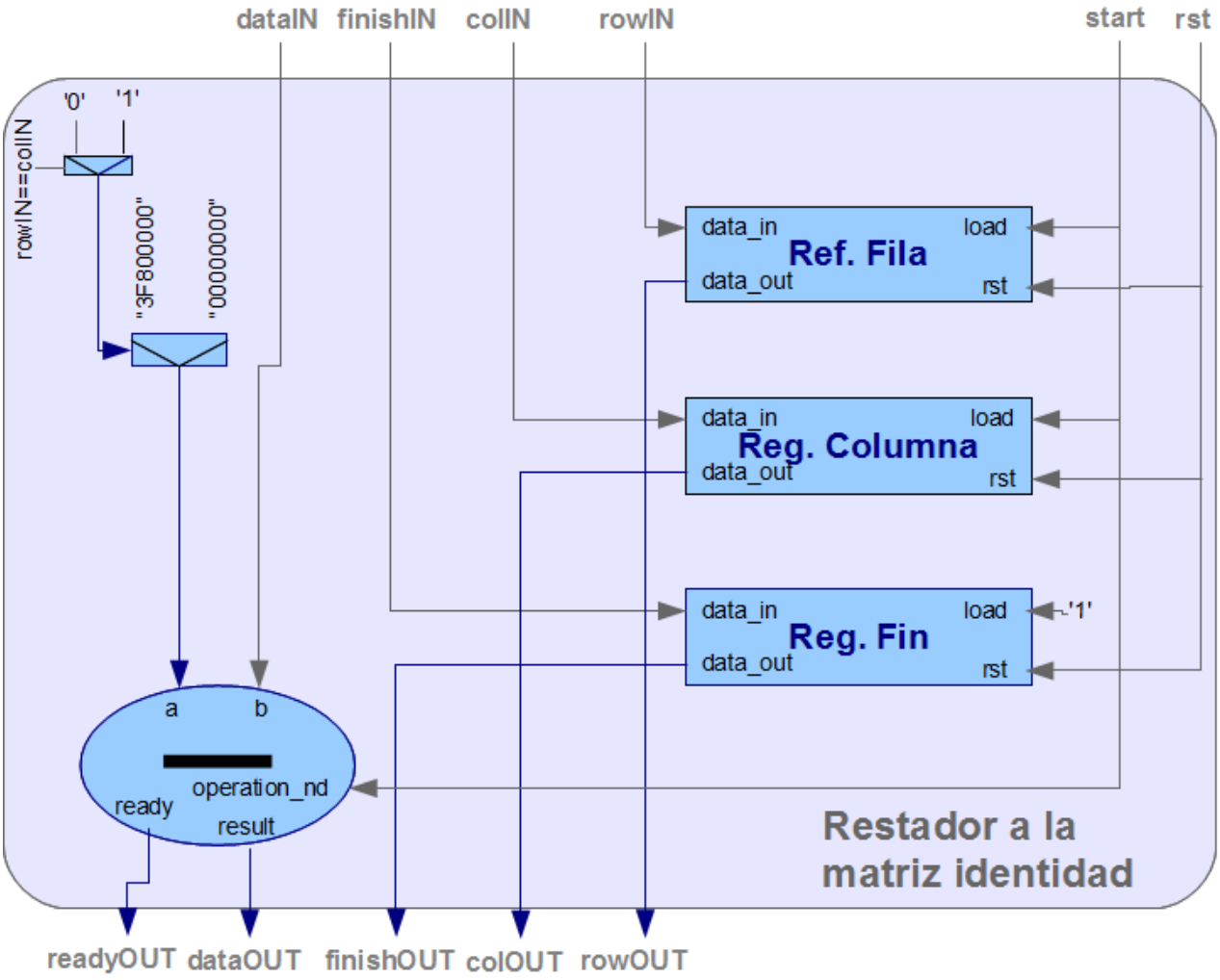


Figura 5.8. Esquema del módulo restador a la matriz identidad.

Este módulo carece de unidad de control ya que su funcionamiento depende únicamente de los parámetros de entrada.

El funcionamiento en ejecución es el siguiente:

Al indicar el inicio de la operación se almacenan las posiciones del dato suministrado y comienza a operar el restador. Si el valor de la fila y la columna es el mismo, el operando *a* del restador es 1, en caso contrario es 0, de este modo se realiza la resta de 1-*dataIN* en caso de estar en la diagonal o 0-*dataIN* si está fuera de la diagonal. La operación tarda un ciclo en completarse, tras lo cual tenemos la posición del dato obtenido a la salida de los registros y puede usarse la señal ready como señal de escritura.

La simplicidad de este módulo se debe a que usará como valores de entrada la salida del módulo multiplicador de matrices, que proporciona la matriz resultado dato a dato. Como dicho módulo proporciona la posición del dato cuando ha terminado, es suficiente con añadir un ciclo de retardo a estos datos para poder realizar la resta, incluyendo la señal de fin para determinar cuando la resta ha terminado.

5.3.8.- MÓDULO INVERSA (INVMODULE2.VHD)

Este módulo calcula la inversa de una matriz cuadrada del tamaño deseado mediante el método de Gauss. Se precisa de dos memorias para el cálculo de la inversa, una memoria para almacenar la matriz sobre la que se quiere calcular su inversa (a partir de ahora la denominaremos como matriz K) y una memoria inicializada con la matriz identidad que será la que se irá modificando para terminar conteniendo los valores de la matriz inversa. A esta última matriz o memoria la denominaremos matriz KInv. Cabe destacar que la memoria que almacene K será sobreescrita con la matriz identidad.

Las memorias usadas para almacenar K y KInv deben tener doble puerto, lectura a nivel de fila y dato de la matriz y escritura a nivel de fila.

El tamaño de las matrices viene dado por un máximo fijado en el componente. A pesar de que un aumento de este máximo no introduce penalización en tiempo, sí que implica un mayor uso del *hardware*.

Las señales de entrada de este módulo son:

- *start*: Señal para inicio de operación del módulo.
- *rowsIN*: Número de filas de las matrices.
- *dataOutRowA_K* y *dataOutA_K*: Datos de la fila y pivote leídos de la matriz K.
- *dataOutRowA_KInv*: Datos de la fila leídos de la matriz KInv.
- *write_rdy*: Señal que indica que se han terminado de inicializar las memorias.
- dataPivot: Valor de un pivote seleccionado.

Las señales de salida de este módulo son:

- *finish*: Señal de fin.
- *error*: Señal de error.
- *chgRows*: Señal que indica que deben intercambiarse dos filas.
- *dataInRowB*: Datos de una fila para la escritura. Común para K y KInv.
- *addrRowB*: Señal para direccionar la escritura del dato. Común para K y KInv.
- *addrRowK* y *addrColK*: Fila y columna que direccionan la lectura de la matriz K.
- *wrRowB_K*: Señal de escritura de fila de la matriz K.
- *addrRowKInv*: Fila que direcciona la lectura de la matriz KInv.
- *wrRowB_KInv*: Señal de escritura de fila de la matriz Kinv.

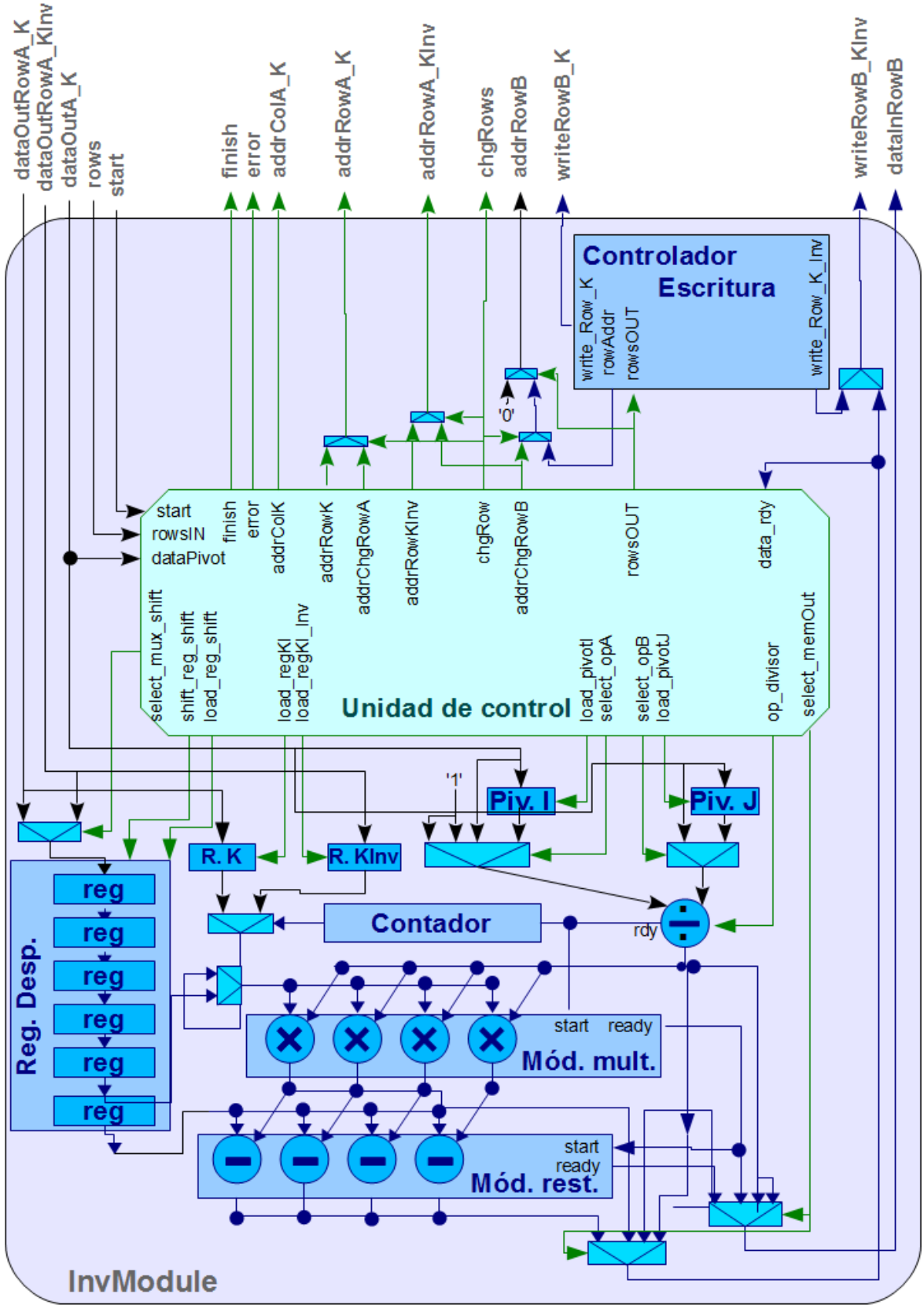


Figura 5.9. Esquema del módulo inversa.

La ruta de datos se compone de los siguientes módulos:

- *Divisor PF*: Divisor de números en punto flotante precisión simple que calcula el factor por el que multiplicar la fila pivote para diagonalizar.
- *Módulo restadores*: Módulo compuesto de restadores en punto flotante de precisión simple encargado de realizar la resta entre elementos de 2 filas de la matriz, es decir, si tenemos $A_i=[a_0,a_1,...,a_n]$ y $B_j=[b_0,b_1,...,b_n]$ el resultado será $C_{i,j}=[a_0-b_0,a_1-b_1,...,a_n-b_n]$.
- *Módulo multiplicadores*: Módulo compuesto de multiplicadores encargado de realizar la multiplicación de los elementos de una fila de la matriz por el factor calculado por el divisor.
- *Registro de desplazamiento*: Registro de desplazamiento de 6 registros del que se pueden obtener los últimos datos del registro o todos. Está pensado para almacenar las filas en las que habrá que hacer ceros, como tiene que ser la entrada del módulo de restadores debe tener tantos registros como retardo provoque el divisor (5 ciclos) y el módulo de multiplicadores (1 ciclo). Para la segunda etapa se utiliza la salida del quinto registro, pues los datos introducidos deben ser entrada para los multiplicadores y por tanto requiere un ciclo menos de espera.
- *Registro KInv*: Registro que almacena el valor de la fila de la memoria KInv que se está tratando.
- *Registro K*: Registro que almacena el valor de la fila de la memoria K que se está tratando.
- *Pivote J*: Registro que almacena el contenido del pivote de la fila donde se van a hacer ceros.
- *Pivote I*: Registro que almacena el contenido del pivote de la fila usada para diagonalizar.
- *Contador*: Contador de 2 bits realizado para cambiar el dato de entrada de los multiplicadores alternativamente entre la fila de K y de Kinv.
- *Controlador de escritura (Inversa)*: Controlador de escritura de las memorias. Se encarga de direccionar correctamente los datos producidos por la ruta de datos para su escritura en las memorias K y Kinv.

De manera similar a lo utilizado para describir el módulo Multiplicador de Matrices, nos referiremos aquí con *i* al índice para recorrer los pivotes de K y KInv y con *j* al índice para recorrer las filas a diagonalizar de ambas matrices.

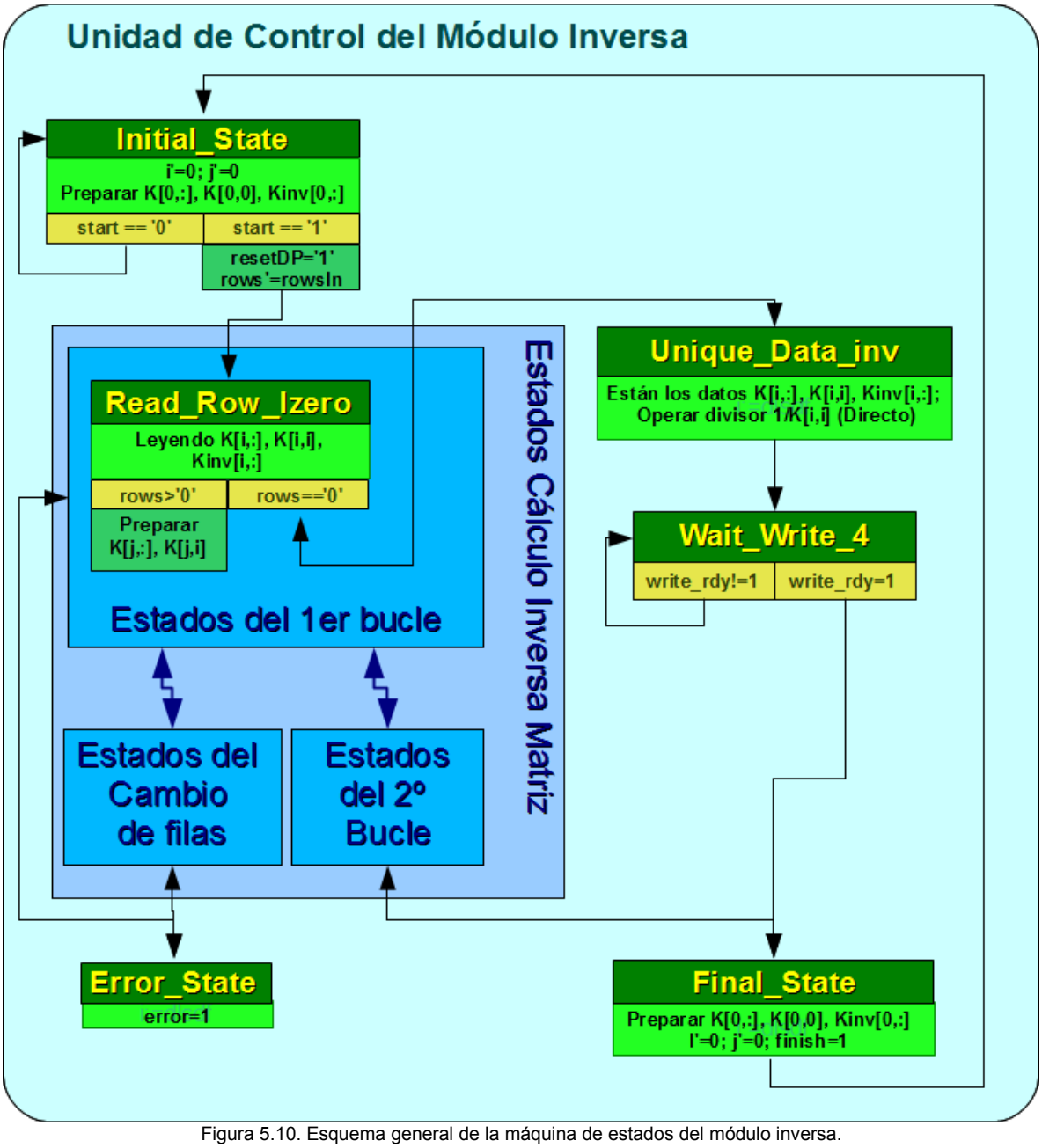


Figura 5.10. Esquema general de la máquina de estados del módulo inversa.

La unidad de control del Módulo Inversa tiene lo siguientes estados:

- **Initial_State**: Este es el estado inicial y de espera al inicio de la operación. en él se preparan las filas K[i,:] y KInv[i,:] y el pivote K[i,i]. Al indicarse el inicio de la operación se resetean los registros de la ruta de datos, se guarda el número de filas de la matriz K y se pasa al estado *Read_Row_Izero*.
- **Read_Row_Izero**: En este estado se leen las filas K[i,:] y KInv[i,:] y el pivote K[i,i]. En caso de estar calculando la inversa de un sólo dato ($K_{(1x1)}$) se pasa al estado *Unique_Data_Inv*, si no, se preparan K[j,:] y K[j,i]
- **Read_Row_J**: Este estado se comprueba el valor del pivote K[i,i] realizándose la siguiente distinción de casos:
  - Si K[i,i] != 0 puede usarse como pivote, por lo que se guardan los datos K[i,:], K[i,i] y Kinv[i,:] en sus correspondientes registros y se prepara Kinv[j,:].

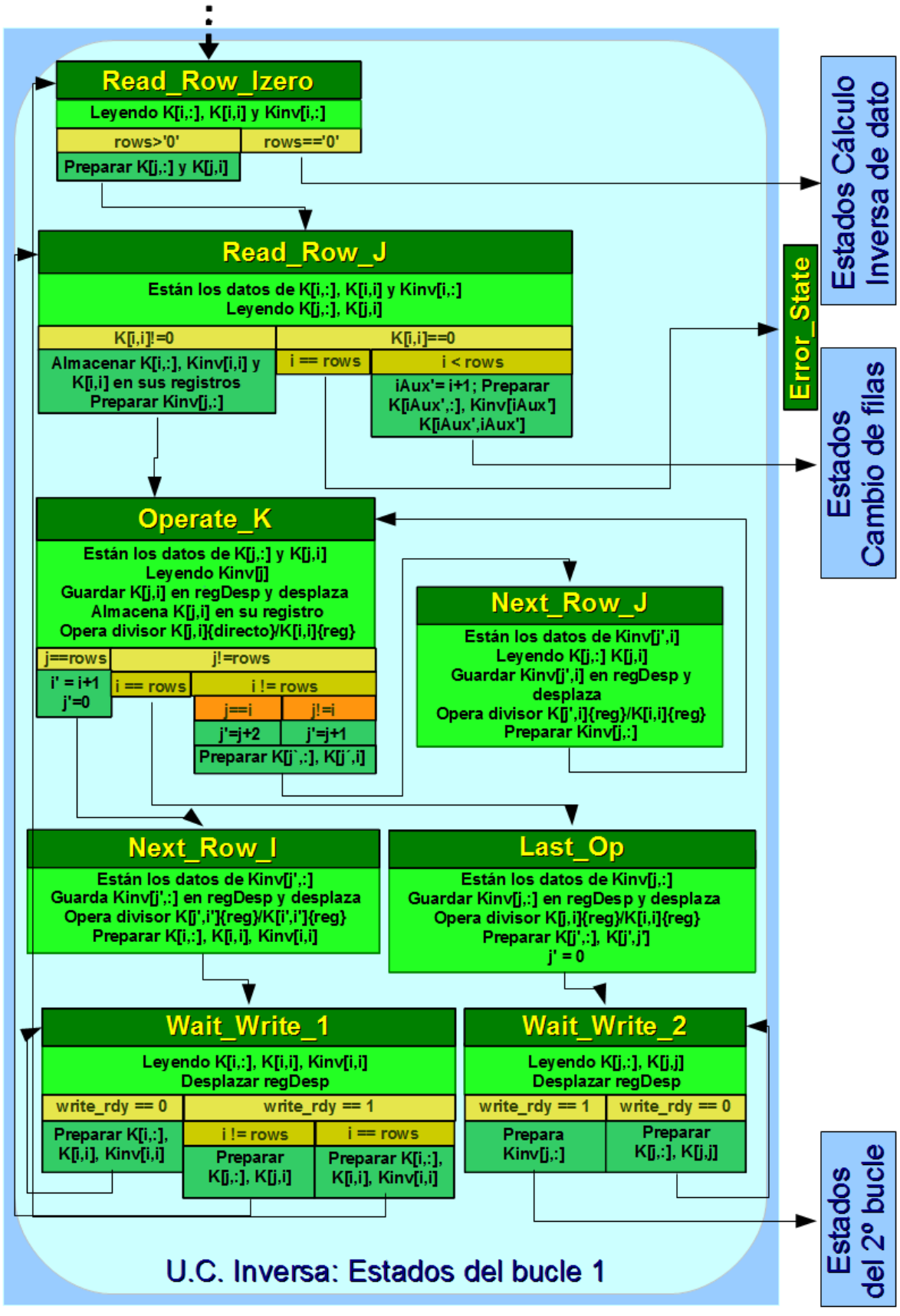


Figura 5.11. Esquema de los estados del primer bucle del módulo inversa.

- Si K[i,i] = 0 no es un pivote válido. Si no estamos pivotando sobre la última fila, cabe la posibilidad de cambiar esta fila por otra y continuar el proceso, por lo que se inicializa un índice *iAux* auxiliar con *i+1* para recorrer la matriz en busca de una fila para cambiar y se preparan K[i+1], K[i+1,i] y KInv[i+1]. En este caso se pasa al estado *Read_Row_Chg*. Si por lo contrario se estuviera pivotando sobre la última fila, la matriz K proporcionada no tiene inversa y se pasa al estado *Error_State*.

- **Operate_K**: En este estado se realiza la división entre el pivote K[i,i] y K[j,i], se almacena K[j,i] en un registro y se introduce K[j,:] en el registro de desplazamiento. Si se ha terminado de diagonalizar, se pasa al estado *Last_Op*, si se ha terminado de pivotar sobre esa fila se pasa al estado *Next_Row_I*, en caso contrario, la transición se realiza al estado *Next_Row_J*. En cualquier caso se actualizan los índices de manera oportuna, teniendo en cuenta que hay que evitar el caso que *i=j*. También se preparan los datos K[j',:], K[j',i] siendo *j'* el índice *j* ya actualizado.
- **Next_Row_J**: En este estado se realizan las mismas operaciones que en el estado *Operate_K*, pero el valor almacenado en el registro de desplazamiento es Kinv[j-1,:] y no se escribe en el registro K[j,i]. Se prepara la fila Kinv[j,:] y el siguiente estado es siempre *Operate_K*.
- **Next_Row_I**: Este estado es similar al estado *Next_Row_J* pero los datos preparados son K[i,:].K[i,i] y Kinv[i,:]. El estado siguiente es *Wait_Write_1*.
- **Wait_Write_1**: En este estado se espera a la finalización de la escritura de los datos de K y KInv. Mientras tanto, se preparan los datos K[i,:].K[i,i] y Kinv[i,:] al tiempo que se sigue desplazando el registro de desplazamiento. Cuando la escritura ha terminado, se preparan K[j,:] y K[j,i], pasando al estado *Read_Row_J*, salvo que la siguiente fila *i* se trate de la última, en cuyo caso se pasa al estado *Read_Row_Izero*.
- **Last_Op**: Este estado es parecido al estado *Next_Row_J* pero los datos preparados son K[0,:] y K[0,0]. El estado siguiente es *Wait_Write_2* y se asigna *j = 0*.
- **Wait_Write_2**: En este estado se espera a la finalización de la escritura de los datos de K y KInv. Mientras tanto, se preparan los datos K[j,:] y K[j,j] al tiempo que se sigue desplazando el registro de desplazamiento. Cuando la escritura ha terminado, se prepara KInv[j,:], pasando al estado *Divide_Row_K*.
- **Read_Row_Chg**: En este estado intermedio del cambio de filas se preparan los datos K[iAux+1,:], K[iAux+1,i] y Kinv[iAux+1,:]. Siempre se pasa al estado *Verificate_ChgRow*.
- **Verificate_ChgRow**: En este estado se permanece incrementando el índice *iAux* y preparando las siguientes filas de K y KInv y el siguiente pivote hasta encontrar un pivote válido o llegar al fin de la matriz. Si se llega al final de la matriz se pasa al estado *Error_State*. En caso de encontrar un pivote válido, se guardan las filas en los registros correspondientes, se cambian las filas *i–iAux* en K y KInv, se preparan K[j,:] y K[j,i] y se pasa al estado *End_ChgRow*.
- **End_ChgRow**: En este estado se prepara Kinv[j,:] como paso intermedio para volver al cauce normal de ejecución en *Operate_K*.
- **Divide_Row_K**: En este estado se cambia el cableado de la salida de datos y la entrada de los multiplicadores, asignándose al primero la salida de los multiplicadores y al segundo el penúltimo dato del registro de desplazamiento. También se divide 1/K[j,j] y se carga K[j,:] en el registro de desplazamiento. Es necesario preparar K[j+1,:] y K[j+1,j+1] y transitar al

estado *Divide_Row_KInv*.

- **Divide_Row_KInv**: En este estado se realiza la misma división y cambio de cableado que en *Divide_Row_K*, se almacena Kinv[j,:] en el registro de desplazamiento y dependiendo de si se ha terminado de recorrer la matriz o no, se actualiza el índice *j*, se prepara K[j+1,:] y se pasa a *Divide_Row_K* o simplemente se pasa al estado *Wait_Write_3*.
- **Wait_Write_3**: Manteniendo el cambio de cableado de los dos estados anteriores y desplazando el registro de desplazamiento, se espera a que finalicen las escrituras para pasar al estado *Final_State*.
- **Unique_Data_Inv**: En este estado sólo se divide 1/K[i,i] y se pasa al estado *Wait_Write_4*.
- **Wait_Write_4**: Este estado sirve para esperar el término de la escritura y transitar al estado *Final_State*.
- **Final_State**: Es el estado de fin. Activa la señal de fin, reinicia los índices y vuelve al estado de inicio *Initial_State*.
- **Error_State**: Es el estado de error. Activa la señal de error y bloquea el módulo.

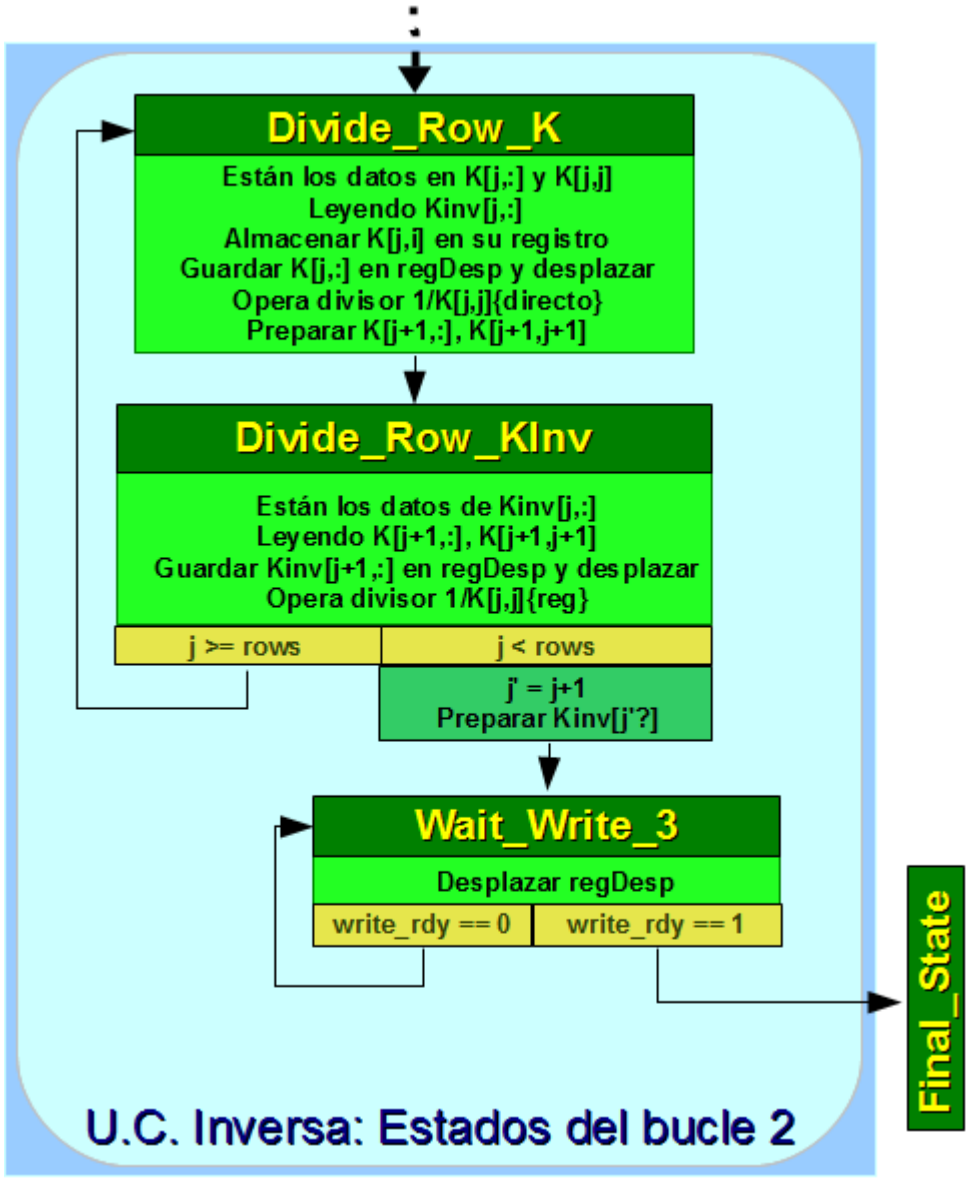


Figura 5.12. Esquema de los estados del segundo bucle del módulo inversa.

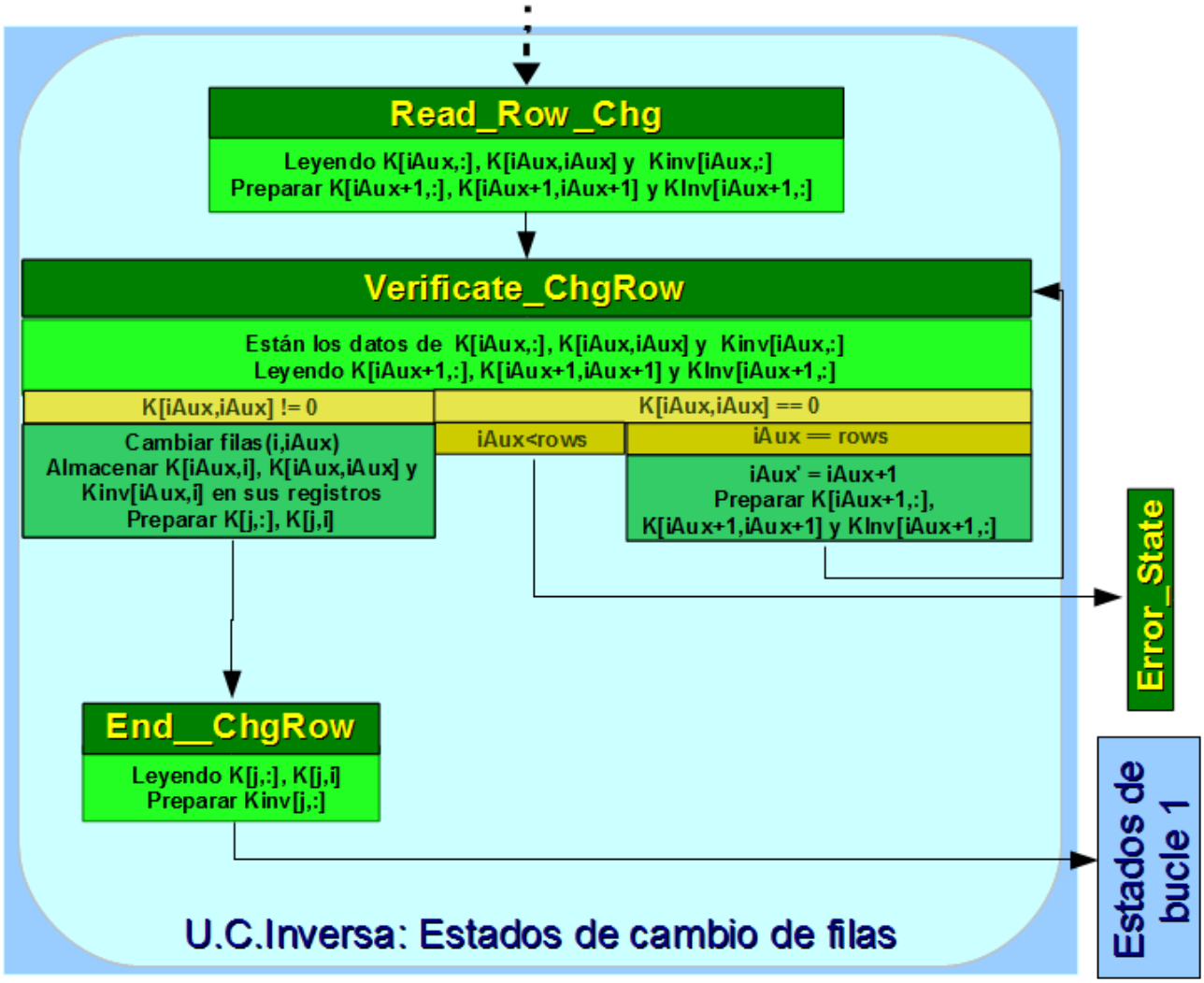


Figura 5.13. Esquema de los estados de cambio de filas del módulo inversa.

Para explicar el funcionamiento de este módulo nos basaremos en un pseudocódigo que calcule la inversa de una matriz dada por el método Gauss, ajustado ligeramente para encajar con el comportamiento del módulo. Dicho código es el siguiente:

Entradas al algoritmo: Matriz *K*, matriz *Kinv* inicializada con *I,* y el número *n* de filas de ambas matrices.

```
si n = 1 entonces
       KInv[0,0] = 1/K[0,0]
sino
       para i = 0 hasta n-1 hacer
                     si K[i,i] = 0 entonces     #Cambio de filas
                            iAux = i+1
                            encontrado = false
                            mientras !encontrado && iAux<n hacer
                                   si K[iAux,i]!=0 entonces
                                          FilaK = K[i,:]
                                          FilaKInv = KInv[i,:]
                                          Pivote = K[i,i]
                                          FilaAux = K[i,:]
                                          K[i,:] = K[iAux,:]
```

```
                                        K[iAux,:] = FilaAux
                                        encontrado = true
                                sino
                                        iAux++
                                fsi
                        fmientras
                        si encontrado = false entonces
                                return error
                        fsi
                sino
                        FilaK = K[i,:]
                        FilaKInv = KInv[i,:]
                        Pivote = K[i,i]
                fsi
                para j = 0 hasta n-1 hacer
                        si j!=i hacer
                                valor = K[j,i]/pivote
                                K[j,:] = K[j,:] - valor*FilaK
                                KInv[j,:] = KInv[j,:] - valor*FilaKInv
                        fsi
                fpara
        fpara
        para i=0 hasta n-1 hacer
                pivote = K[i,i]
                valor = 1/pivote
                K[i,:] = valor*K[i,:]
                KInv[i,:] = valor*KInv[i,:]
        fpara
fsi
```

Salida del algoritmo: $Kinv = K^{-1}$.

Como puede observarse, es simplemente una implementación normal del método Gauss para hallar la inversa de una matriz. A continuación estableceremos la relación entre las líneas de pseudocódigo anteriores y los estados donde se llevan a cabo esas operaciones. Para ello debemos tener en cuenta que la ruta de datos no se encarga de escribir los datos en las memorias, simplemente los calcula en el orden esperado por el módulo *Controlador de escritura*, quien se encarga de direccionarlos y escribirlos en las correspondientes memorias.

El estado *Initial_Wait* no corresponde con ninguna instrucción concreta, pues simplemente se encarga de la espera inicial y de preparar los datos necesarios para el siguiente estado.

```
si n = 1 entonces
        KInv[0,0] = 1/K[0,0]
sino
```

Esta distinción de casos se lleva a cabo en el estado *Read_Row_Izero*, pasando al estado *Unique_Data_Inv* en caso que la matriz K solo tenga una fila. En *Unique_Data_Inv* se calcula 1/K[0,0] y se pasa a *Wait_Write_4* donde se espera a que el módulo *Controlador de escritura* indique la finalización de la escritura.

Para el resto de los casos el siguiente estado a *Read_Row_Izero* será *Read_Row_J*.

```
para i = 0 hasta n-1 hacer
             si K[i,i] = 0 entonces    #Cambio de filas
                    iAux = i+1
```

```
                encontrado = false
                mientras !encontrado && iAux<n hacer
                        si K[iAux,i]!=0 entonces
                                FilaK = K[i,:]
                                FilaKInv = KInv[i,:]
                                Pivote = K[i,i]
                                FilaAux = K[i,:]
                                K[i,:] = K[iAux,:]
                                K[iAux,:] = FilaAux
                                encontrado = true
                        sino
                                iAux++
                        fsi
                fmientras
                si encontrado = false entonces
                        return error
                fsi
```

El proceso de cambio de filas en caso de encontrar un pivote con valor 0 se realiza entre los estados *Read_Row_J*, *Read_Row_Chg*, *Verificate_ChgRow* y *EndChgRow*.

En *Read_Row_J* se comprueba que el valor de K[i,i] leído sea distinto de cero, en caso contrario, se inicializa el índice *iAux* con *i+1* y se preparan las lecturas oportunas para realizar la búsqueda en las filas siguientes en busca de un cambio. *Read_Row_Chg* es un estado intermedio para que se completen las lecturas mencionadas antes y prepare las siguientes filas que se puedan necesitar. Una vez en *Verificate_ChgRow*, se comprueba si la fila leída es válida para el cambio, en cuyo caso se ordena el cambio y se guardan los valores leídos en los registros *Registro K*, *Registro KInv* y *Pivote I*, preparando las lecturas de K[j,:] y K[j,i] y pasando a *End_ChgRow*, el cual prepara los siguientes datos necesarios y transita a *Operate_K*. Si durante la búsqueda de filas realizada en *Verificate_ChgRow* se recorriese toda la matriz sin encontrar una fila sustituta, se pasa al estado *Error_State*, el estado de error.

```
        sino
                FilaK = K[i,:]
                FilaKInv = KInv[i,:]
                Pivote = K[i,i]
        fsi
        para j = 0 hasta n-1 hacer
                si j!=i hacer
                        valor = K[j,i]/pivote
                        K[j,:] = K[j,:] - valor*FilaK
                        KInv[j,:] = KInv[j,:] - valor*FilaKInv
                fsi
        fpara
fpara
```

Si en *Read_Row_J* se detecta que el pivote no es nulo, se guardan los valores de K[i,:] y KInv[i,:] leídos, se preparan las siguientes lecturas y la ejecución sigue en el estado *Operate_K*. Este bucle se realiza entre los estados *Operate_K* y *Next_Row_J*. En el primero se ordena el cálculo de valor y se guarda la fila K[j,:] para que sea restada al llegar los datos al módulo restador, mientras que en el segundo se hace lo mismo para KInv[j,:]. Una vez recorridas todas las filas con el índice *j*, se espera en *Wait_Write_1*, pasando primero por *Next_Row_I* para preparar los datos necesarios y realizar la última operación con KInv[j,:]. Una vez el módulo *MemoryInv2_CU* señalice el fin de las escrituras, se repite el proceso para la siguiente fila de *i*. En el caso de la última fila, es necesario volver a *Read_Row_Izero* para introducir un ciclo más de

espera, pues esa fila acaba de ser escrita y es necesario esperar un ciclo más para su correcta lectura.

```
      para i=0 hasta n-1 hacer
                  pivote = K[i,i]
                  valor = 1/pivote
                  K[i,:] = valor*K[i,:]
                  KInv[i,:] = valor*KInv[i,:]
      fpara
fsi
```

Cuando en *Operate_K* se está operando con el último dato de K[j,:] para la última fila *i* en lugar de lo descrito anteriormente, siguiente estado será *Last_Op*, que realiza la misma función de intermediario que *Next_Row_I* pero enlazando con la espera en *Wait_Write_2* y posteriormente con *Divide_Row_K*. En éste último estado y *Divide_Row_KInv*, se recorren las matrices K y KInv multiplicando cada fila *j* por 1/K[j,j]. Para ello se introducen las filas en el registro de desplazamiento al tiempo que se comienza la división, y cableando la salida del quinto registro del registro de desplazamiento a la entrada de los multiplicadores, así como la salida de los multiplicadores es cableada a la salida. Al terminar el recorrido, una vez más es necesaria una espera para la finalización de las escrituras (*Wait_Write_3*) y termina la ejecución del módulo.

5.3.9.- MÓDULO IDENTIDAD (IDENMODULE.VHD)

Este módulo escribe en una memoria la matriz identidad del tamaño que le sea indicado. Este es un paso necesario para el cálculo de la matriz inversa.

El tamaño de la matriz debe ser menor de un máximo fijado en el componente, pero ésto sólo afecta al tamaño de los vectores utilizados para direccionar y proporcionar el dato de escritura.

La memoria en la que se escriba la matriz identidad debe admitir la escritura de una línea (fila o columna) entera de la matriz.

Las señales de entrada de este módulo son:

- *start*: Señal de comienzo de operación del módulo.
- *rowsIN*: Número de filas de la matriz identidad.

Las señales de salida de este módulo son:

- *finish*: Señal de fin de operación del módulo.
- *memWrite*: Señal de escritura en la memoria.
- *rowOUT*: Fila de la memoria a escribir.
- *dataRowOUT*: Dato a escribir.

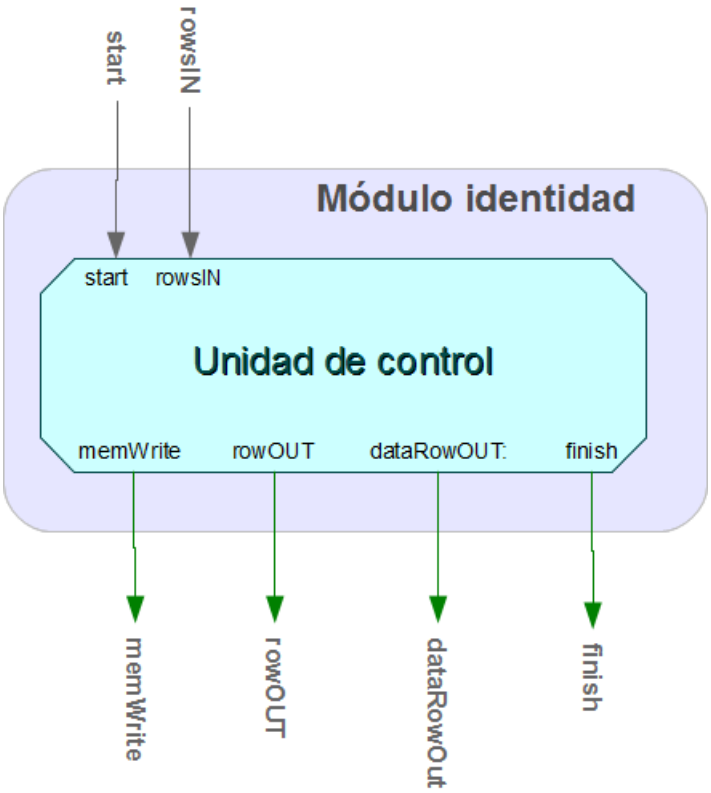


Figura 5.14. Esquema del módulo identidad.

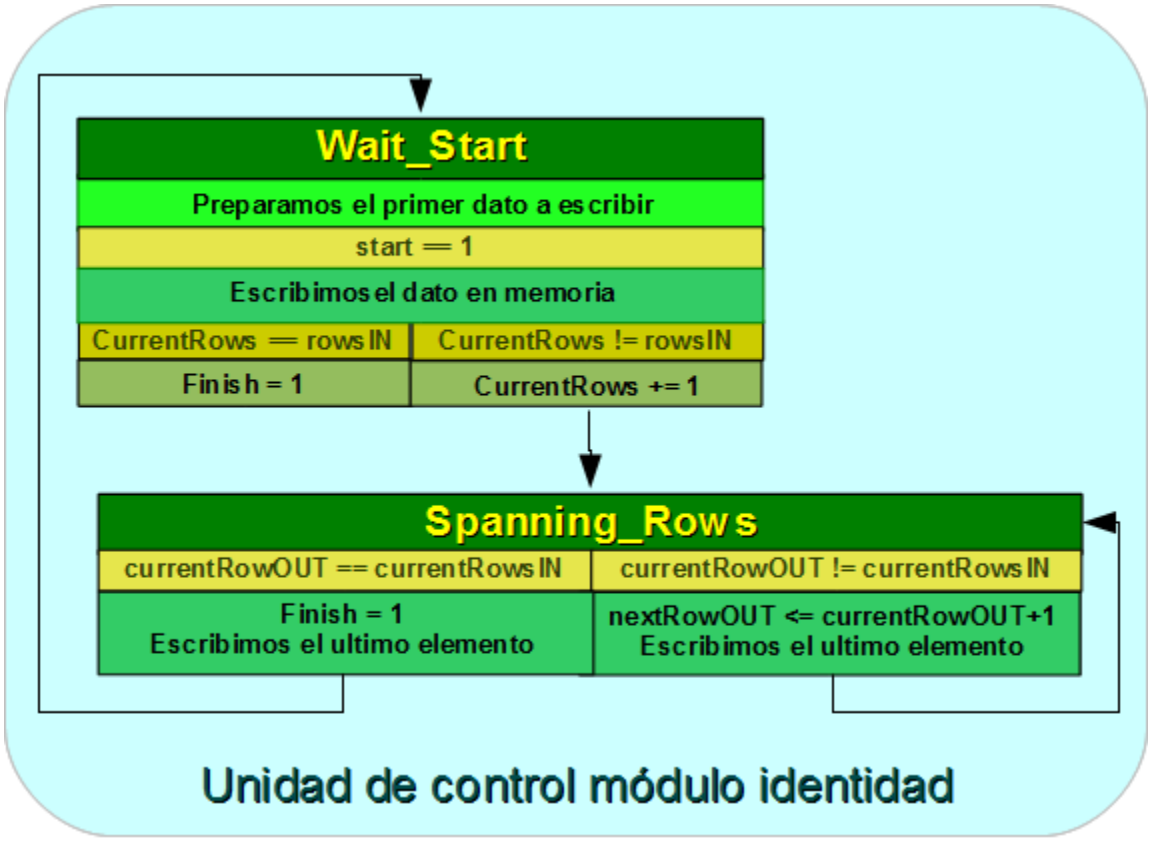


Figura 5.15. Esquema de la máquina de estados del módulo identidad.

Este módulo sólo consta de una pequeña máquina de estados con los siguientes estados:

- **Wait_Start**: Estado de espera inicial. Prepara el primer dato y cuando se indique el comienzo de operación el dato es escrito en la memoria. Si almacena el tamaño de la matriz resultado y se comprueba si hemos llegado a la última fila, activando la línea de fin en ese caso, si no, incrementamos en uno el índice de fila y pasamos al estado *Spanning_Rows*.
- **Spanning_Rows**: En este estado seguimos escribiendo datos y actualizando el índice hasta completar la matriz identidad. Al terminar se activa la línea de fin y se pasa al estado *Wait_Start*.

Su funcionamiento es el siguiente:

Dado un número de filas *n*, al activarse la señal de inicio, se comienzan a escribir en la memoria tantas filas como las indicadas por *rowsIN*. En cada fila se escribe el dato correspondiente de la matriz identidad, es decir, un vector con todo ceros salvo las posiciones correspondientes a I[i,i] cuyo valor debe ser 1 (*x"3f800000"* en PF). Al terminar se activa la señal de fin.

5.3.10.- MÓDULO ATGP (ATDCA.VHD)

Este módulo, que engloba todos los anteriores, es el que lleva a cabo la función de detección de *targets* en una imagen.

Las señales de entrada de este módulo son:

- *start*: Señal de inicio.
- *dataFIn*: Dato del píxel leído de *F* (Matriz imagen).

Las señales de salida de este módulo son:

- *colF*: Señal de direccionamiento de píxeles de *F*.
- *posResult*: Señal de resultado. Muestra las direcciones de los *t* objetivos detectados.
- *error*: Señal de error.
- *finish*: Señal de fin.

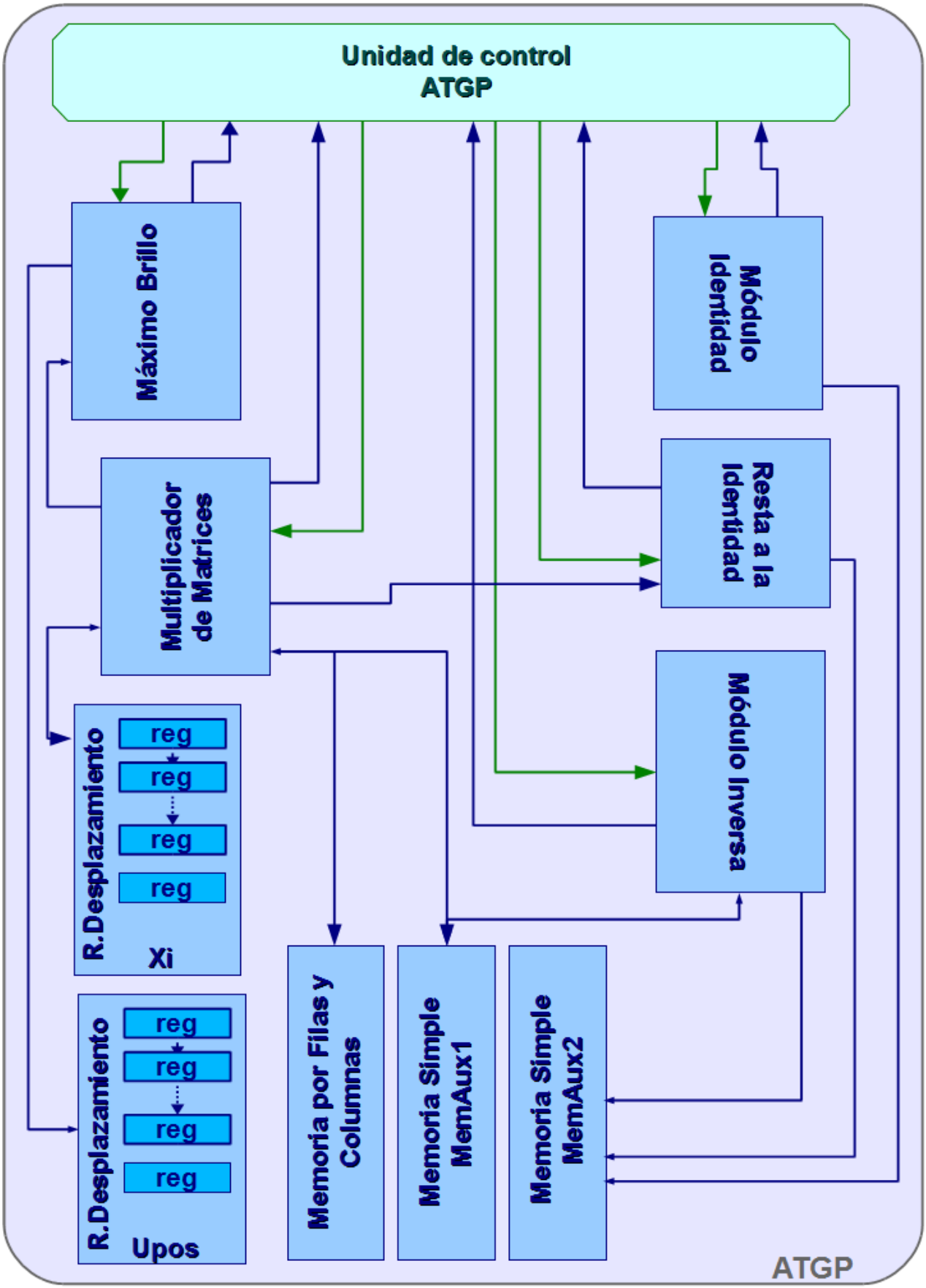


Figura 5.16. Esquema general del módulo ATGP.

La ruta de datos se compone de los siguientes módulos:

- *Multiplicador de matrices*: Módulo encargado de realizar las multiplicaciones de matrices, vectores y matrices por vectores.
- *Mayor brillo*: Módulo encargado de la obtención del píxel de mayor brillo y la posición de éste en *F*.
- *Restador a la matriz identidad*: Módulo dedicado a la operación de restar una fila de una matriz a su correspondiente fila de su matriz identidad
- *Módulo inversa*: Módulo que calcula la matriz inversa de una matriz cuadrada de dimensión dada.
- *Módulo identidad*: Módulo que escribe en una memoria la matriz identidad del tamaño deseado.
- *memAux1, memAux2*: Memorias simples utilizadas para los cálculos intermedios del algoritmo.
- *memU*: Memoria de lectura por filas y por columnas utilizada para almacenar el subespacio vectorial de los objetivos detectados.
- *Xi, Upos* Registros de desplazamiento usados para almacenar el conjunto de direcciones de los píxeles detectados como objetivos y para almacenar las componentes del cálculo de la proyección de los píxeles respecto $P_{\perp U}$ con objeto de suministrar al módulo *Mayor Brillo* la proyección completa de cada píxel.

Esta división en 5 módulos, los cuales han sido explicado con mayor detalle anteriormente, ha reducido considerablemente la complejidad de la unidad de control encargada de llevar a cabo la realización del algoritmo ATGP.

Esta implementación se basa en conectar las memorias que proporcionan los datos de entrada y aquellas que deben guardar el resultado a un determinado módulo y ordenar el inicio de la operación, según el paso del algoritmo en el que se encuentre, y permanecer a la espera de que finalicen las operaciones de este módulo para realizar el siguiente paso. En determinados pasos del algoritmo, la salida de un módulo actúa como entrada de otro.

De forma más detallada vemos como se ajusta la implementación al algoritmo mostrado en esta memoria, de manera similar a lo explicado para el proceso de cálculo de la inversa.

```
#Cálculo del píxel más brillante

max = 0
pos = 0
para i = 0 hasta r-1 hacer
       brillo = F[:,i]*F[:,i]^T
       si (max<=brillo) entonces
                     max = brillo
                     pos = i
       fsi
fpara
U[:,0] = F[:,pos]
Upos[0] = pos
```

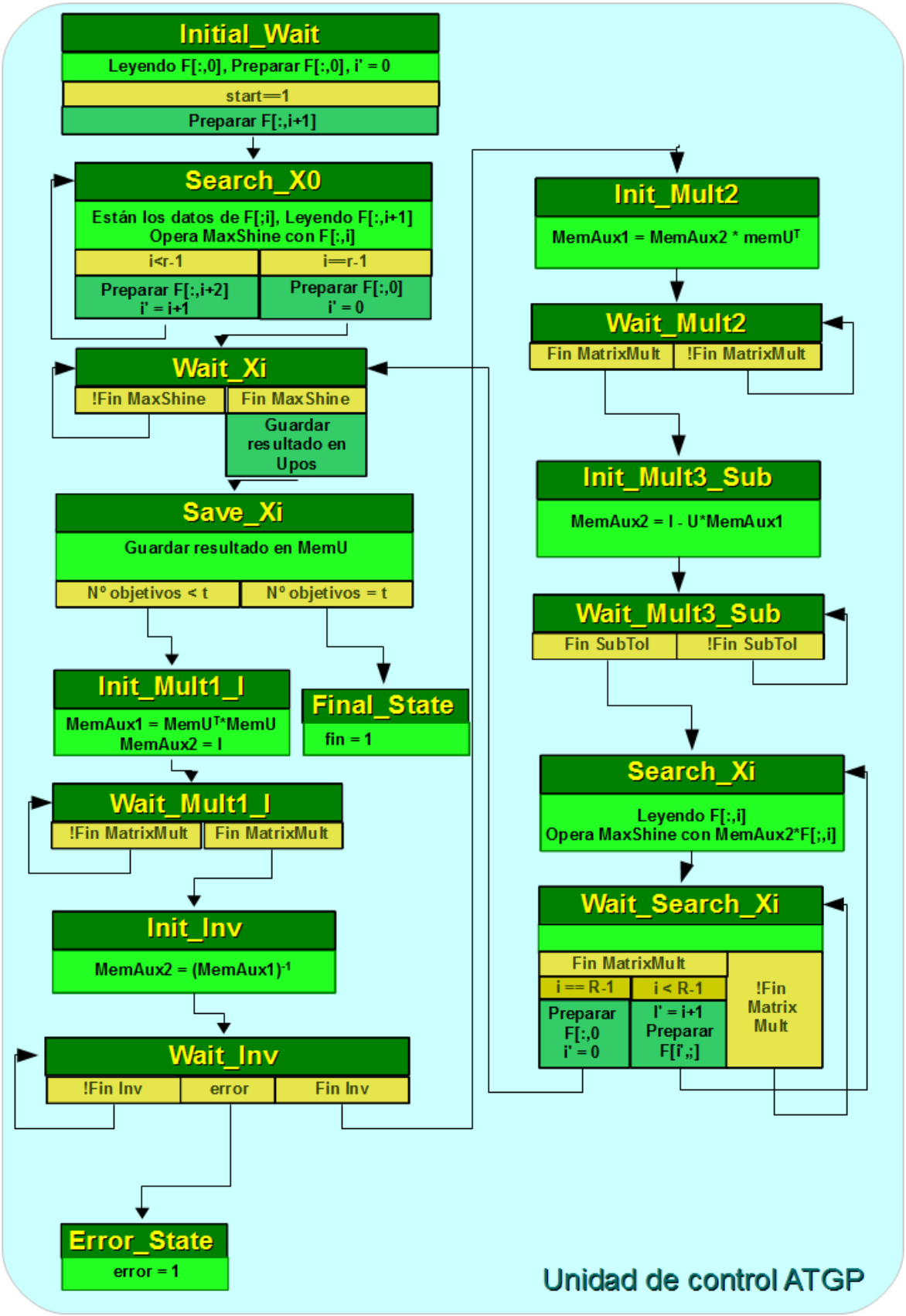


Figura 5.17. Esquema de la máquina de estados del módulo ATGP.

Conjunto de estados en los que se realiza el cálculo del píxel más brillante de *F* como inicialización de la matriz U de objetivos. Comprende los siguientes estados:

- **Search_X0**: Con los datos ya preparados por el estado *Initial_Wait*, la unidad de control cede el turno al módulo *Mayor brillo* que obtendrá el píxel más brillante de la matriz *F*. En este estado se van leyendo los píxeles de *F* e introduciendo en el módulo. Una vez introducidos todos los datos, se pasa al estado *Wait_Xi*.
- **Wait_Xi**: Estado de espera hasta que termine el módulo *Mayor brillo*. En este estado se lee de *F* el resultado proporcionado y se prepara para introducirlo en el registro de desplazamiento que almacenará el resultado del algoritmo, de manera que, en cuanto el módulo acabe, se guarda la posición en el registro, se realiza la lectura del píxel en esa posición y se transita al estado *Save_Xi*.
- **Save_Xi**: En este estado se procede a guardar en la memoria U el valor del píxel detectado como objetivo. Tras este estado, si hemos terminado de encontrar todos los objetivos pasaremos al estado *Final_State*, en otro caso pasaremos al estado *Init_Mult1_I*.

Estos dos últimos estados serán usados también en la segunda parte del algoritmo.

```
#Cálculo del resto de píxeles anómalos

para i = 1 hasta t-1 hacer

       #Cálculo de P⊥U

       aux1 = U[:,0..(i-1)]ᵀ*U[:,0..(i-1)]
       aux2 = I[0..(i-1),:]
```

En este conjunto de estados se realiza la multiplicación de U^T*U guardándose en memAux1 y en paralelo se inicializa la memoria memAux2 con la matriz identidad como preparación para el cálculo de $(U^T*U)^{-1}$. Los estados implicados en estas operaciones son:

- **Init_Mult1_I**: Estado en el cual la unidad de control cede el turno de operación a dos módulos:
  - *Multiplicador de matrices*: El cual se encarga de realizar la operación de multiplicar U^T por U almacenando este resultado en la memoria memAux1. Para ello, este módulo actúa en el modo 0, es decir, utiliza un sólo puerto, conectado a la lectura por columnas de la memoria U.
  - *Módulo identidad*: Este módulo inicializará de manera paralela a la multiplicación de matrices la matriz memAux2 con la matriz identidad.

  Una vez iniciados los cálculos, se pasa al estado *Wait_Mult1_I*.
- **Wait_Mult1_I**: Estado encargado de esperar a que terminen de operar los módulos *Multiplicador de matrices y Módulo identidad*. Cuando ambos hayan terminado pasaremos al estado *Init_Inv*.

```
inversa(aux1,aux2) #aux2=aux1[0..(i-1),:]⁻¹
```

Esta operación abarca dos estados:

- **Init_Inv**: Con el resultado de U^T*U almacenado en la memoria memAux1 y la identidad de esta matriz almacenada en la memoria memAux2, la unidad

de control cede el turno de operación al *Módulo inversa* que utilizará estas 2 memorias (memAux1 como K y memAux2 como KInv) para el cálculo de $(U^T*U)^{-1}$ que almacenará en la memoria memAux2. Una vez ordenado el cálculo, se avanza al estado *Wait_Inv*.

- **Wait_Inv**: Estado de espera para la finalización de cálculo de la inversa. Tras este estado pasamos al estado *Init_Mult2* salvo que el cálculo de la inversa haya producido un error, en ése caso pasaremos al estado *Error_State*.

```
aux1 = aux2[0..(i-1),:]*U[:,0..(i-1)]^T
```

Si la operación de cálculo de la inversa se ha realizado correctamente procedemos a realizar el cálculo de multiplicar el contenido de memAux2 = $(U^T*U)^{-1}$ por U^T. En esta operación se ven implicados los siguientes estados:

- **Init_Mult2**: Con los valores almacenados en memAux2 y en memU la unidad de control vuelve a ceder el control al módulo *Multiplicador de matrices* que calculará el producto del contenido de estas memorias en el orden descrito. En este caso, el modo de ejecución será el modo 1, usando ambos puertos. Por el puerto A se conectará a a la memoria memAux2 y por el puerto B a la lectura por filas de MemU, ya que en la operación que debe realizar, tiene que leer U^T por columnas, y las filas de la matriz equivalen a las columnas de su traspuesta. La memoria en que se escribirán los resultados será memAux1. Tras indicar la operación sigue el estado *Wait_Mult2*.
- **Wait_Mult2**: Estado que esperará a la finalización del cálculo de la multiplicación realizada por el *Multiplicador de matrices* en el estado anterior. Tras este estado pasamos al estado *Init_Mult3_Sub*.

```
aux2 = I[0-(n-1),:]-U[:,0..(i-1)]*aux1  #aux2=P⊥U
```

Realización de manera secuencial de la multiplicación del contenido de memU por la matriz $(U^T*U)^{-1}*U^T$, almacenada en memAux1, y la posterior resta del resultado de esta multiplicación a la matriz identidad, siendo guardado el resultado final en memAux3. De estas operaciones se encargan los siguientes estados:

- **Init_Mult3_Sub**: En este estado la unidad de control cede el turno de operación al módulo *Multiplicador de matrices* que realizará la operación previamente explicada de memU*memAux1. La salida del módulo multiplicador se conecta a la entrada del módulo *Restador a la matriz identidad*, de manera que según se vayan obteniendo los datos de la multiplicación se vaya calculando el resultado de restarlo a la matriz identidad y sea este último resultado el que se almacene en memAux2. Una vez iniciada la operación se pasa al estado *Wait_Mult3_Sub*.
- **Wait_Mult3_Sub**: Estado en el que se espera a la finalización del módulo *Restador a la matriz identidad*. Una vez haya terminado, en la memoria memAux2 estará almacenada la matriz $P_{\perp U}$. El siguiente estado es *Search_Xi*.

```
#Cálculo del pixel cuya proyección respecto de P⊥U tenga mayor módulo

      max=0
      pos=0
      para j=0 hasta r-1 hacer
                  x = aux2[0-(i-1),:]*F[:,j]
                  mod = x*x^T
                  si (max<=mod) entonces
```

```
                        max = mod
                        pos = i
                    fsi
        fpara
        U[i,:] = F[:,pos]
        UPos[i]=pos
fpara
```

Para este cálculo procedemos del siguiente modo. El módulo *Multiplicador de matrices* calcula la proyección de cada píxel respecto $P_{\perp U}$, siendo almacenado este resultado en un registro de desplazamiento. Cuando el módulo ha terminado manda operar a módulo *Mayor brillo* que, tomando como entrada el registro de desplazamiento, obtendrá el píxel que proyectado sobre $P_{\perp U}$ tiene mayor módulo y lo almacenará sobre la memoria memU. Si hemos llegado al último objetivo, al detectar el objetivo (t-1) el algoritmo habrá finalizado si no volveremos al inicio del bucle. Estas operaciones se desarrollan en los siguientes estados:

- **Search_Xi**: En este estado la unidad de control utiliza el módulo *Multiplicador de matrices* para calcular la proyección de un píxel de la matriz F sobre $P_{\perp U}$. (una multiplicación de píxel por $P_{\perp U}$ por cada vez que se manda operar a este módulo). Las componentes de la proyección son almacenadas en un registro de desplazamiento a la espera de obtener la proyección completa. Una vez mandado operar a este módulo se pasa al siguiente estado *Wait_Search_Xi*.
- **Wait_Search_Xi**: Este estado espera a que el módulo *Multiplicador de matrices* calcule la proyección del píxel sobre $P_{\perp U}$. Una vez el módulo ha terminado de operar pasamos al siguiente estado. Volveremos al estado *Search_Xi* si no hemos terminado de calcular las proyecciones sobre todos los píxeles, en caso contrario, pasaremos al estado *Wait_Xi*.
- **Wait_Xi**, **Save_Xi**: Estos estados ya han sido explicados. Esperan al término del módulo *Mayor brillo* y almacenan el resultado en el registro de posiciones resultado y en la memoria memU. Si aún no se han encontrado todos los objetivos, se vuelve al estado *Init_Mult1_I*, si no, habremos acabado la ejecución del algoritmo, pasando al estado *Final_State*.
- **Final_State**: Estado que indicará que el cálculo o detección de los *t* objetivos ha finalizado.

Capítulo 6. Resultados Experimentales

6.1.- Plataforma reconfigurable

La implementación *hardware* del algoritmo ATGP descrita en la sección anterior se ha llevado a cabo utilizando el lenguaje de descripción *hardware* VHDL. La utilización de este lenguaje, junto con el entorno de desarrollo *Xilinx ISE Design Suite* permite generar la matriz de configuración o *bitstream* necesaria para cargar dicho diseño en la FPGA.

Para comprobar el correcto funcionamiento del algoritmo sobre la plataforma, se ha simulado este diseño sobre una *Xilinx XC7VX690T*. Esta FPGA, de la familia de las *Virtex 7* pertenece a los últimos modelos del fabricante y ofrece los recursos necesarios para la implementación del algoritmo ATGP. Si bien el modelo de FPGA que cuenta con protección contra radiación (*Radiation Hardened,* RH) se trata de un modelo de prestaciones inferiores, se espera que en un futuro próximo aparezcan versiones de la FPGA utilizada protegidas contra radiación. En la Tabla 6.1 se ofrece una comparativa del modelo utilizado y el modelo protegido contra radiación.

	XQ5VFX130T	XQR5VFX130 (RH)
Celdas lógicas	693.120	131072
Slices CLB	108.300	20480
Distribución RAM máxima (Kb)	10.888	1580
Slices DSP48E	3600	320
Bloques de RAM 18 Kb	2.940	596
Bloques de RAM 36 Kb	1.470	298
Bloques de RAM máximos	52.920	10728
CMT	20	6
Puntos de anclaje PCI *Express*	3	3
GTH	80	0
Bloques XADC		0
Bancos Entrada/Salida	20	24
Puertos Entrada/Salida para usuario máximos	1000	836

Tabla 6.1. Características de la FPGA *Xilinx XC7VX690T*.

6.2.- Conjunto de imágenes hiperespectrales

En el presente trabajo fin de carrera se han utilizado dos imágenes hiperespectrales reales, ambas tomada por el sensor AVIRIS (desarrollado por el *Jet Propulsion Laboratory* de la NASA). Dichas imágenes son utilizadas frecuentemente en aplicaciones de detección de *targets*. A continuación se describen las imágenes hiperespectrales consideradas.

6.2.1.- AVIRIS WTC

La primera imagen utilizada en el presente trabajo fue adquirida por el sensor AVIRIS el día 16 de septiembre de 2001, justo cinco días después de los ataques terroristas que derrumbaron las dos torres principales y otros edificios del complejo *World Trade Center* (WTC) en la ciudad de Nueva York. Las características principales de la imagen aparecen resumidas en la Tabla 6.2. Conviene destacar que la resolución espacial de la imagen es muy elevada (1,7 metros por píxel) para lo que suele ser habitual en el caso de AVIRIS, en el que la resolución espacial habitual suele rondar los 20 metros por píxel. Esto se debe a que la imagen corresponde a un vuelo de baja altura, mediante el cual se pretendía obtener la mayor resolución espacial posible sobre la zona de estudio al contrario de otros estudios con el mismo sensor, en los que se pretende cubrir un área más extensa.

La Figura 6.1 muestra una composición en falso color de la imagen AVIRIS WTC. En dicha composición, se han utilizado las bandas espectrales localizadas en 1682, 1107 y 655 nanómetros como rojo, verde y azul, respectivamente. En función de la composición en falso color utilizada, las zonas de la imagen con predominancia de vegetación reciben tonalidades verdes, mientras que las zonas con fuegos aparecen con tonos rojos. El humo que proviene del área del WTC (enmarcada en el rectángulo de color rojo) y que se dirige al sur de la isla de Manhattan recibe un tono azul claro en la composición de falso color, debido a la elevada reflectancia del humo en la longitud de onda correspondiente a 655 nanómetros.

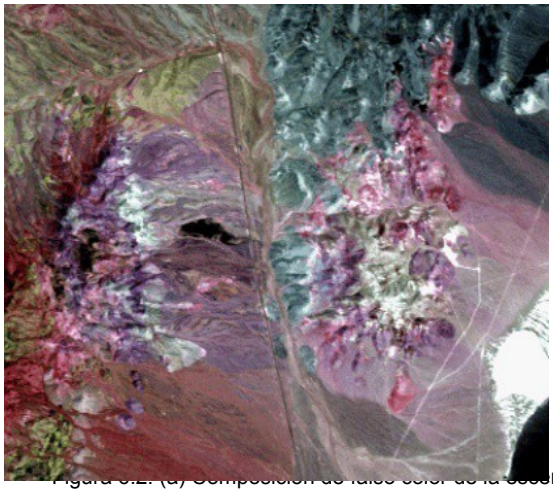
Líneas	614
Muestras	512
Bandas	224
Rango espectral	0,4 – 2,5 µm
Resolución espacial	1,7 metros/píxel
Tamaño	140 Mbytes aproximadamente

Tabla 6.2. Características de la imagen hiperespectral AVIRIS obtenida sobre la zona del *World Trade Center* en la ciudad de Nueva York, cinco días después del atentado terrorista del 11 de septiembre de 2001.

Figura 6.1. Composición en falso color de la imagen hiperespectral AVIRIS obtenida sobre la zona del WTC en la ciudad de Nueva York, cinco días después del atentado terrorista del 11 de septiembre de 2001. El recuadro en rojo marca la zona donde se sitúa el WTC en la imagen.

6.2.2.- AVIRIS CUPRITE

El segundo conjunto de datos corresponde a la conocida escena AVIRIS Cuprite (ver Figura 6.2(a)), recogida en el verano de 1997 y disponible *online* en unidades de reflectancia después de ser corregida atmosféricamente [37]. La Tabla 6.3 resume las características principales de la imagen. La porción utilizada en los experimentos corresponde a un subconjunto de 350 x 350 píxeles del sector, etiquetados como *f970619t01p02_r02_sc03.a.rfl* en los datos *online*, que cuenta con 224 bandas espectrales en el rango de 400 a 2500 nanómetros y un tamaño total de alrededor de 45 megabytes. Las bandas 1-3, 105-115 y 150-170 han sido eliminadas antes del análisis debido a la absorción por agua y la baja relación señal-ruido o *signal-to-noise ratio* (SNR) de estas bandas. La zona es bien conocida mineralógicamente, y tiene varios minerales expuestos de interés, incluyendo alunita, buddingtonita, calcita, caolinita y moscovita. Las firmas de referencia de suelo de los minerales mencionados (ver Figura 6.2(b)), disponibles en la biblioteca *U.S. Geological Survey Library* (USGS) [38] , se utilizarán para evaluar los *targets* detectados en este trabajo.



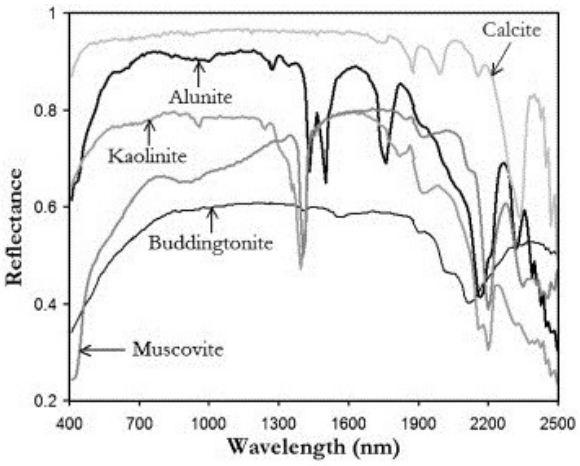


Figura 6.2. (a) Composición de falso color de la escena hiperespectral AVIRIS sobre la región minera de Cuprite en Nevada. (b) Firmas espectrales de los minerales en la librería *U.S. Geological Survey* utilizadas para la validación.

Líneas	350
Muestras	350
Bandas	188
Rango espectral	0,4 – 2,5 μm
Resolución espacial	15 metros/píxel
Tamaño	45 Mbytes aproximadamente

Tabla 6.3. Características de la imagen hiperespectral AVIRIS sobre la región minera de Cuprite en Nevada utilizada en los resultados experimentales.

6.3.- EVALUACIÓN DE LOS *TARGETS*

En este apartado se evalúa la precisión de los *targets* detectados por la implementación propuesta del algoritmo ATGP utilizando las imágenes hiperespectrales reales descritas anteriormente. Antes de comenzar, destacamos que nuestra implementación proporciona exactamente los mismos resultados que una versión *software* equivalente, por lo que podemos dar por válida la implementación *hardware* propuesta.

Primeramente, hemos partido de las imágenes hiperespectrales descritas en el apartado anterior y las hemos sometido a un proceso de reducción dimensional. Frecuentemente los píxeles de una imagen hiperespectral se ubican en un subespacio muy pequeño en comparación con el número de bandas disponibles. La identificación de este subespacio permite una correcta reducción de la dimensionalidad, que se traduce en una mejora en el rendimiento, en la complejidad de los algoritmos y en el almacenamiento de datos. En nuestro caso concreto hemos aplicado el algoritmo *Principal Component Analysis* (PCA) disponible en el software comercial *Environment for Visualizing Images* (ENVI) [39]. Las Figuras 6.3 y 6.4 muestran los autovalores para cada una de las bandas de la nueva imagen tras el análisis de componentes. Como cabría esperar, las primeras bandas son las que acumulan la mayor parte de la variabilidad de la imagen con un acentuado decremento. Finalmente, nuestras imágenes reducidas dimensionalmente constarán de 32 bandas, reteniendo un 99.99% de la información de la imagen AVIRIS WTC y un 99.96% en el caso de la imagen AVIRIS Cuprite.

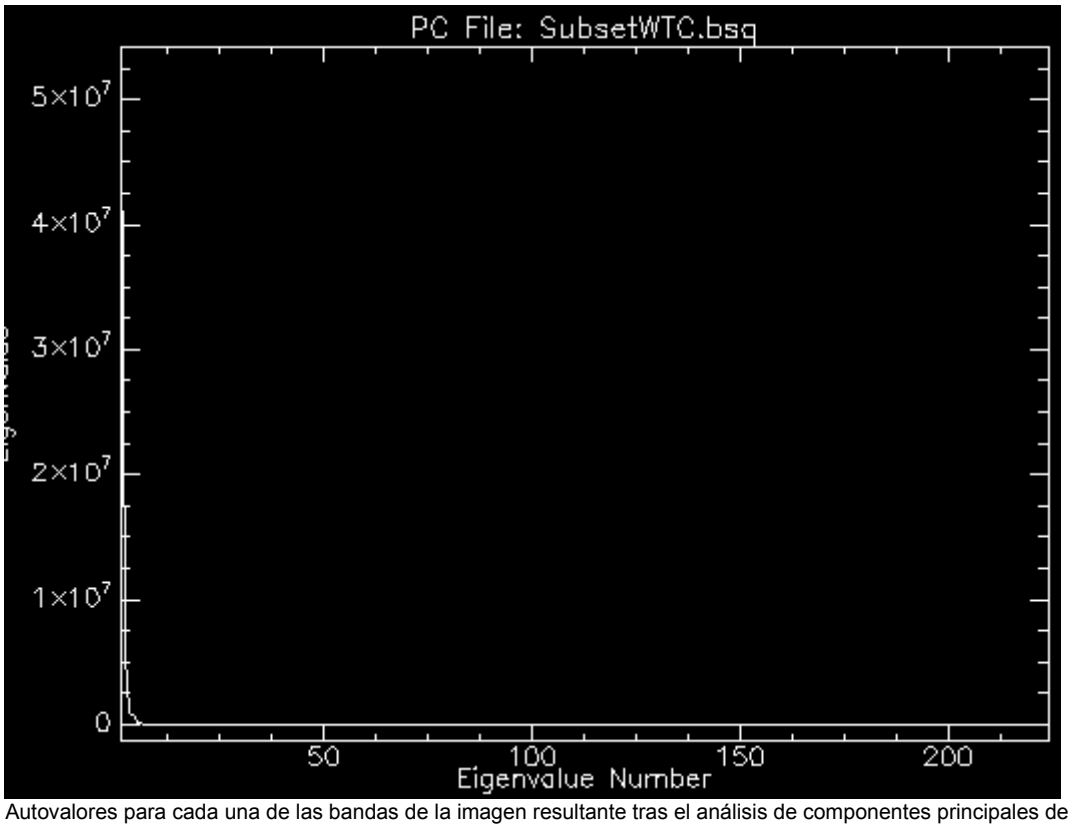


Figura 6.3. Autovalores para cada una de las bandas de la imagen resultante tras el análisis de componentes principales de la imagen AVIRIS WTC.

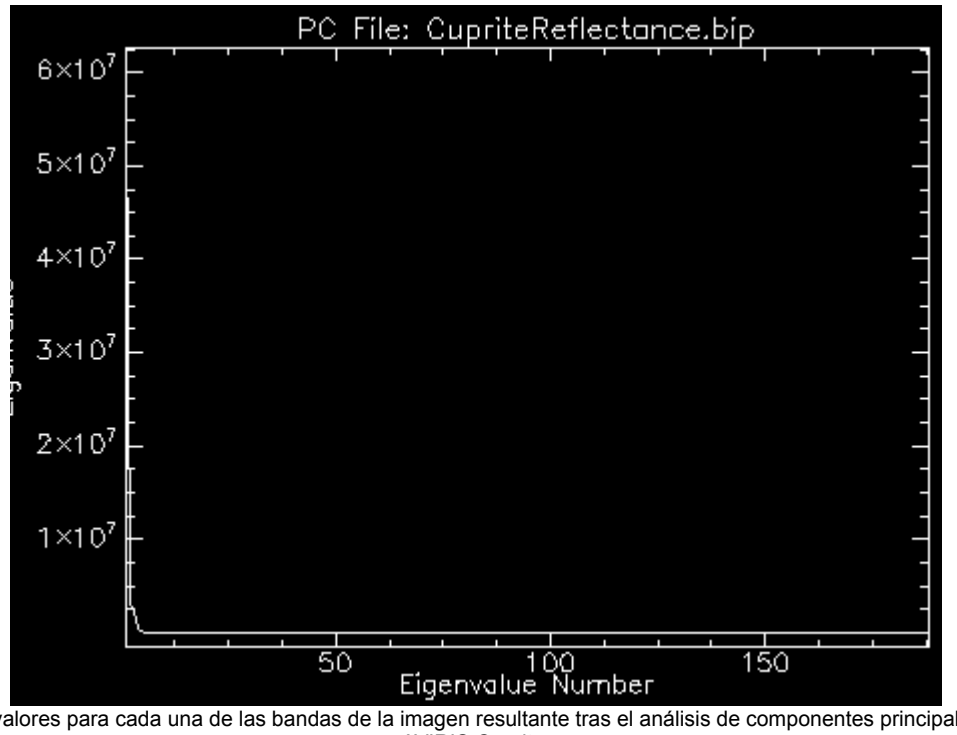


Figura 6.4. Autovalores para cada una de las bandas de la imagen resultante tras el análisis de componentes principales de la imagen AVIRIS Cuprite.

La precisión del algoritmo va a ser evaluada por su capacidad de detectar automáticamente los puntos calientes del incendio en la imagen AVIRIS WTC y los minerales de referencia en la escena AVIRIS Cuprite. El número de *targets* a detectar se ha fijado en 30 para la imagen AVIRIS WTC y en 19 para la imagen AVIRIS Cuprite después de calcular la dimensionalidad virtual de los datos.

Existe una extensa información de referencia, recopilada por el USGS, sobre la escena AVIRIS WTC [40]. En este proyecto fin de carrera, utilizamos el mapa termal del USGS [41] donde se muestra la localización de los puntos calientes (que pueden considerarse anomalías) en el área del WTC, mostrados en rojo brillante, naranja y amarillo en la Figura 6.5(a). El mapa está centrado en la región donde cayeron las dos torres y en el rango de temperaturas que va desde los 700F a los 1300F. La Tabla 6.4 muestra información adicional disponible por el USGS sobre los puntos calientes (incluyendo localización y temperatura). Dicho mapa termal es el que utilizamos para obtener la información sobre el terreno (ver Figura 6.5(b)) que serán los puntos que utilicemos para evaluar la correcta detección de *targets*.

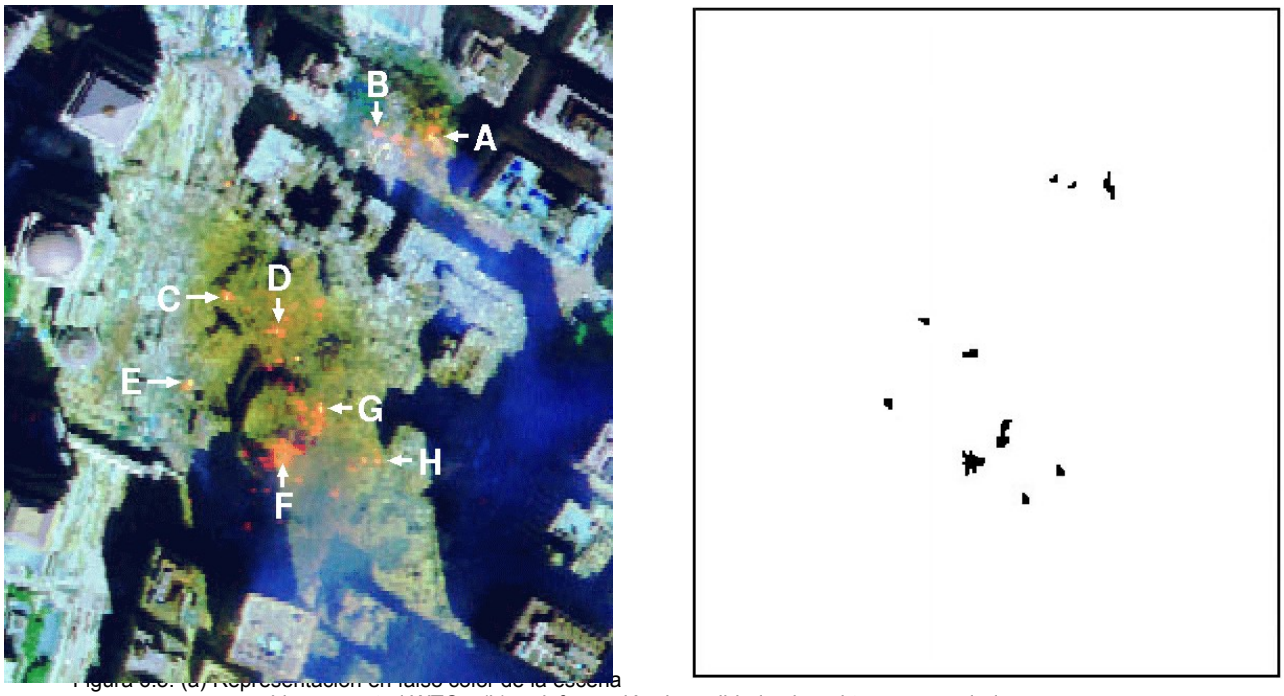


Figura 6.5. (a) Representación en falso color de la escena hiperespectral WTC y (b) su información de realidad sobre el terreno asociado.

Punto Caliente	Latitud (Norte)	Longitud (Este)	Temperatura (Kelvin)
'A'	40º 42' 47,18"	74º 00' 41,43"	1000
'B'	40º 42' 47,14"	74º 00' 43,53"	830
'C'	40º 42' 42,89"	74º 00' 48,88"	900
'D'	40º 42' 41,99"	74º 00' 46,94"	790
'E'	40º 42' 40,58"	74º 00' 50,15"	710
'F'	40º 42' 38,74"	74º 00' 46,70"	700
'G'	40º 42' 39,94"	74º 00' 45,37"	1020
'H'	40º 42' 38,60"	74º 00' 43,51"	820

Tabla 6.4. Propiedades de los puntos calientes etiquetados en la Figura 6.5(a).

En el caso de la escena AVIRIS WTC la Figura 6.6(a) muestra los 30 *targets* detectados en el total de la imagen. Centrándonos en la zona de interés, la Figura 6.6(b) muestra los *targets* detectados donde se sitúa el WTC. La Tabla 6.5 muestra los valores de ángulo espectral (AE) (en grados) entre los píxeles más similares detectados como *targets* por el algoritmo ATGP y los datos de los píxeles situados en las posiciones conocidos, etiquetadas de la "A" a la "H", en la imagen AVIRIS WTC. El número de píxeles objetivos a detectar se establece en t = 30 después de calcular la dimensionalidad virtual (VD) de los datos. Como se muestra en la Tabla 6.5, el algoritmo ATGP extrae *targets* similares espectralmente, a los objetivos reales conocidos por la información del terreno. La implementación propuesta es capaz de detectar a la perfección los objetivos marcados como "A" y "C", aunque tuvo más dificultades en la detección del resto de *targets*. Estos resultados se corresponden con los anteriormente publicados en la literatura [42] .

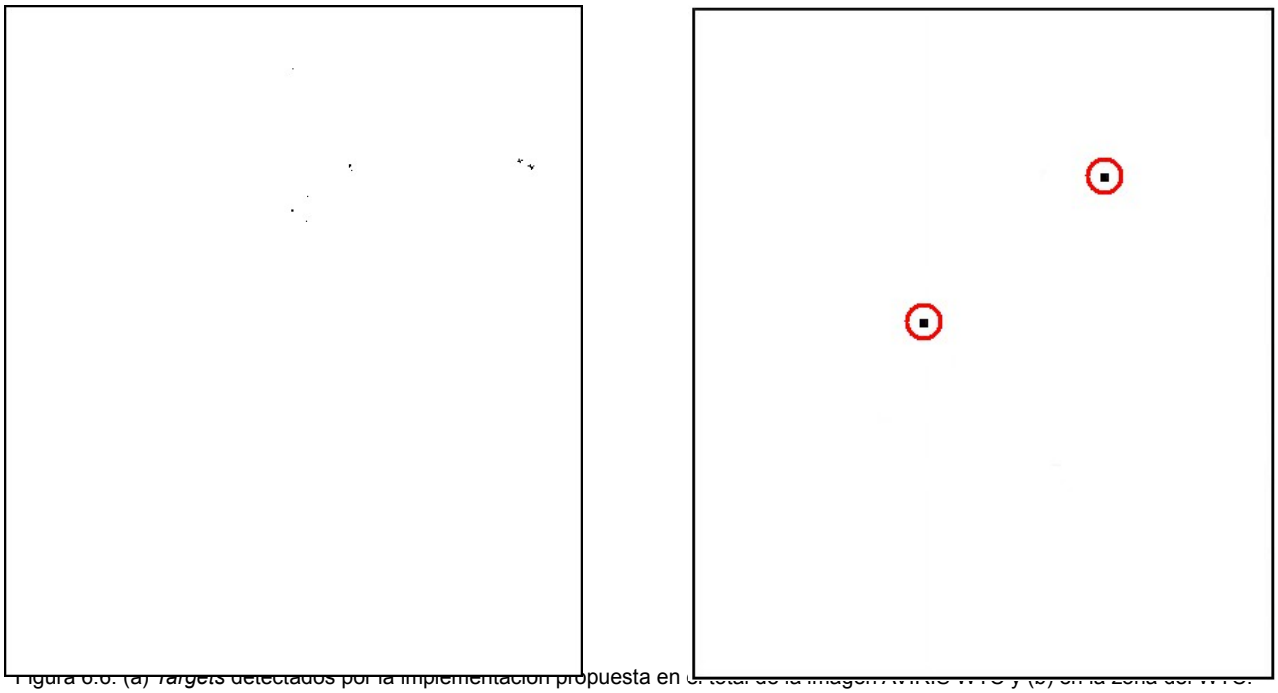


Figura 6.6. (a) *Targets* detectados por la implementación propuesta en el total de la imagen AVIRIS WTC y (b) en la zona del WTC.

A	B	C	D	E	F	G	H
0,00º	14,32º	0,00º	27,41º	20,29º	7,11º	4,13º	31,22º

Tabla 6.5. Valores de ángulo espectral (en grados) entre los *targets* detectados por la implementación propuesta y los *targets* de tierra conocidos en la escena AVIRIS World Trade Center.

El mismo experimento se realizó para la imagen AVIRIS Cuprite. La Figura 6.7 muestra los *targets* detectados por la implementación *hardware* propuesta. Al igual que en la imagen AVIRIS WTC, aumentando el número de *targets* a detectar se mejoran los resultados de detección, aunque basándonos en de la dimensionalidad virtual de los datos, hemos decidido detectar 19 *targets*. Estos resultados se corresponden con los anteriormente publicados en la literatura [42].

La Tabla 6.6 muestra los valores de AE (en grados) entre los píxeles más similares detectados como *targets* por la implementación propuesta, y los datos de los píxeles localizados en las posiciones conocidas de la imagen AVIRIS Cuprite donde se encuentran los minerales de interés.

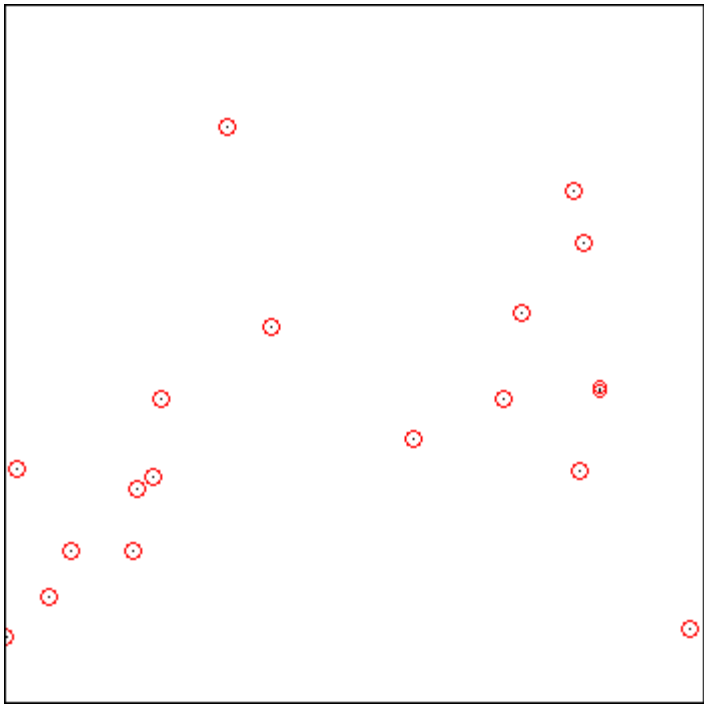


Figura 6.7. *Targets* detectados por la implementación propuesta en la imagen AVIRIS Cuprite.

Alunita	Buddingtonita	Calcita	Caolinita	Moscovita
4,86º	4,17º	9,22º	10,72º	5,31º

Tabla 6.6. Valores de ángulo espectral (en grados) entre los *targets* detectados por la implementación propuesta y los *targets* de tierra conocidos en la escena AVIRIS Cuprite.

6.4.- EVALUACIÓN DEL RENDIMIENTO

La implementación llevada a cabo permite el procesamiento de imágenes de cualquier tamaño, con un número máximo de 32 bandas, pudiendo detectar hasta 32 objetivos. En la Tabla 6.7 y 6.8 pueden observarse los recursos empleados por el módulo ATGP desarrollado para la FPGA XC7VX690T y las restricciones temporales derivadas del diseño.

	Usados	Disponibles	Porcentaje uso
Registros *slice*	13.370	866.400	1%
Slices LUT	292.478	433.200	67%
Pares LUT-FF usados al completo	5.957	299.891	1%
Bloques RAM/FIFO	1.088	1.470	74%
BUFG/BUFGCTRL	2	32	6%
DSP48E1	478	3600	13%

Tabla 6.7 Resumen de recursos utilizados en la implementación para la FPGA XC7VX690T.

Como puede observarse en la Tabla 6.7, los recursos disponibles en la FPGA XC7VX690T son suficientes para la implementación desarrollada, siendo los bloques de memoria y los *slices* de LUT aquellos con mayor porcentaje de ocupación. Como se ha dicho anteriormente, el uso de bloques de memoria en diseños para FPGA tiene un gran impacto en los recursos empleados, pues el número de bloques de RAM es muy limitado en comparación con los otros recursos. Gracias a la reutilización de memorias llevada a cabo en el diseño desarrollado se ha conseguido reducir este impacto al mínimo. Otro recurso ampliamente usado en el diseño propuesto son los DSP48E1s. Estos elementos *hardware* presentes en algunos modelos de FPGA, son pequeños elementos de procesamiento lógico usados en los módulos operadores de PF para generar los *IP-Cores* más eficientemente y con menor impacto en el tiempo mínimo de ciclo. Estos elementos se usan en los sumadores, restadores y multiplicadores del módulo ATGP.

Teniendo lo anterior en cuenta, las limitaciones del diseño propuesto, es decir, el número de bandas de la imagen a procesar y el número de *targets* a detectar, podrían ampliarse antes de sobrepasar los recursos de la FPGA XC7VX690T. No obstante, en este proyecto no te ha considerado esta posibilidad, pues la mayoría de las implementaciones propuestas en la literatura [42,43] gozan de las mismas limitaciones y, en nuestro caso, aumentar el número máximo de bandas inflige penalizaciones en el tiempo de ejecución.

La Tabla 6.9 muestra los tiempos de ejecución del módulo desarrollado basándose en los ciclos de reloj necesarios para la completa ejecución del algoritmo. El número de ciclos de reloj se ha calculado en base a las siguientes fórmulas, siendo *b* el número de bandas:

$$T_{MaxShine} = 3 + \log_2 b$$

$$T_{InvModule}(M_{(nxn)}) = \begin{cases} sin = 1 \rightarrow 9 \\ sin > 1 \rightarrow 2 + [(8 + 2*(n-1)*n + 1)] + [2*n + 7] \end{cases}$$

$$T_{MatrxMult\_mode0}(A_{(nxn)} * A^T_{(nxn)}) = 2 + n*(n+1) + \log_2 b$$

$$T_{MatrxMult\_mode1}(A_{(nxl)} * B_{(lxm)}) = 2 + n*m + \log_2 b$$

$$\begin{aligned} T_{ATGP} = {} & 2 + r + T_{MaxShine} + \\ & + \sum_{i=1}^{t} (T_{MatrixMult\_mode0}(AA_{(ixi)} * A^T_{(ixi)}) + 1) + (T_{InvModule}(A_{(ixi)}) + 1) + \\ & + (T_{MatrixMult\_mode1}(A_{(ixi)} * B_{(ixb)}) + 1) + (T_{MatrixMult\_mode1}(A_{(bxi)} * B_{(ixb)}) + 2) + \\ & + (T_{MatrixMult\_mode1}(A_{(bxi)} * B(ix1)) + 1) * r + T_{MaxShine} + 1 \end{aligned}$$

	Restricciones
Periodo de ciclo mínimo (Tiempo de ciclo)	13,806ns
Frecuencia máxima	72,433MHz
Tiempo mínimo de *set-up*	7,550ns
Tiempo de retardo máximo entre entradas y salidas	1,537ns

Tabla 6.8. Restricciones temporales del diseño desarrollado.

	AVIRIS WTC 614x512 píxeles 32 bandas 30 *targets*		AVIRIS Cuprite 350x350 píxeles 32 bandas 19 *targets*	
	Nº Ciclos	Tiempo	NºCiclos	Tiempo
Implementación en FPGA	365.055.719 Ciclos	5,041054423670 9995 segundos	88355061 Ciclos	1,220095037349 segundos

Tabla 6.9. Tiempos de ejecución y número de ciclos del módulo desarrollado en el procesamiento de las imágenes AVIRIS WTC y AVIRIS Cuprite.

En la Tabla 6.10 pueden observarse los tiempos de ejecución del módulo propuesto en comparación con distintas implementaciones del algoritmo ATGP presentes en la literatura, así como el *speedup* respecto de éstas:

- **Implementación *software***: corresponde a la ejecución de un programa compilado con GCC (compilador C/C++ por defecto en GNU) y el *flag* de optimización –O3 activado para explotar la localidad de los datos y evitar cálculos redundantes. El programa es ejecutado en uno de los núcleos de un microprocesador *Intel i7 920* a 2,67GHz conectado a una placa base *ASUS P6T7 WS SuperComputer* [42].
- **Implementación GPU:** corresponde a la ejecución de una versión del ATGP modificada para explotar el paralelismo de los cálculos del algoritmo. La implementación se ejecuta sobre dos GPU distintas, una *NVIDIA GeForce GTX 580* y una *NVIDIA Tesla C1060*, ambas conectadas a un microprocesador *Intel i7 920* a 2,67GHz y una placa base ASUS *P6T7 WS SuperComputer* [42].
- **Implementación *clúster*:** corresponde a la ejecución de una versión del ATGP modificada para explotar el paralelismo de los cálculos del algoritmo. La ejecución se lleva a cabo sobre uno de los nodos de un *clúster Beowulf.* Cada nodo esta compuesto por un microprocesador *Intel Xeon* a 2,26GHz y 12GB de memoria SDRAM DDR3. El sistema operativo utilizado fue *Scientific Linux CERN SLC* versión 4.6 (*Berylium*) [43].

Como se puede observar, la implementación en FPGA supone una mejora apreciable en tiempo de ejecución a las alternativas mostradas. Cabe destacar, que los tiempos conseguidos permiten el procesamiento a tiempo real de las imágenes obtenidas con el sensor AVIRIS, ya que éste recoge 512 píxeles cada 8,3 ms, lo que implica procesar la imagen AVIRIS WTC en menos de 5,09 s y la imagen de AVIRIS Cuprite en menos de 1,98 s si quiere realizarse un cálculo a tiempo real, y la implementación propuesta realiza en procesamiento de las imágenes en 5,0410 s y 1,2200 s respectivamente.

		Imágenes			
		AVIRIS WTC		AVIRIS Cuprite	
Plataforma		Tiempo	*Speedup*	Tiempo	*Speedup*
Implementación FPGA		5,0410 s	1	1,2200 s	1
Implementación *software*		512,1120 s	101,5893	87,9820 s	72,1163
Implementación GPU	*NVIDIA Tesla C1060*	51,4626 s	10,2088	9,0032 s	7,37
	NVIDIA GeForce GTX 580	10,5747 s	2,0977	1,9947 s	1,635
Implementación *Clúster*	1 núcleo	153,65 s	30,4800	27,23 s	22,3196
	2 núcleos	77,47 s	15,3679	13,83 s	11,3360
	4 núcleos	38,85 s	7,7068	6,99 s	8,5278
	8 núcleos	20,43 s	4,0527	3,71 s	3,0409
	16 núcleos	10,93 s	2,1682	2,01 s	1,6475

Tabla 6.10. Comparativa de tiempos de ejecución del algoritmo ATGP-OSP.

CAPÍTULO 7. CONCLUSIONES Y TRABAJO FUTURO

En este proyecto fin de carrera se ha llevado a cabo el diseño e implementación del algoritmo ATGP y comprobado los resultados de su ejecución en una FPGA. Como parte del diseño, se ha realizado una adaptación del algoritmo adecuándolo al flujo habitual de un diseño específico *hardware*, minimizando en la medida de lo posible la cantidad de almacenamiento interno requerido para la ejecución del algoritmo.

Dicha implementación ha sido validada por su equivalente *software*, mostrando unas tasas de detección de *targets* más que suficientes para su utilización en aplicaciones de análisis de imágenes hiperespectrales.

Los tiempos de ejecución obtenidos con el diseño propuesto muestran una mejora en tiempo de ejecución respecto implementaciones del mismo algoritmo en otras plataformas, llegando a permitir el procesamiento a tiempo real de las imágenes hiperespectrales obtenidas con el sensor AVIRIS.

Por tanto, con este proyecto queda demostrada la viabilidad del uso de plataformas de *hardware* reconfigurable en misiones de observación remota de la Tierra como medio para reducir el coste y tiempo de la transmisión de los datos adquiridos por los sensores, pudiendo ser procesados antes y enviar sólo los resultados obtenidos. Todo esto se consigue además con un coste reducido añadiendo una carga energética, volumétrica y de masa muy pequeñas en comparación con otras alternativas presentes en la literatura.

La continuación natural del trabajo desarrollado en este proyecto sería el diseño de una interfaz entre el módulo ATGP y el sensor hiperespectral, así como mejorar el diseño del propio módulo ATGP en busca de explotar el paralelismo en los cálculos realizados.

Otra posible vía de trabajo incluiría la implementación del módulo ATGP pero usando el proceso de ortonormalización de Gram-Schmidt, puesto que evita el cálculo de la matriz inversa y éste es un cálculo muy costoso a la hora de realizarlo en plataformas *hardware*.

BIBLIOGRAFÍA

[1] NASA. Jet Propulsion Laboratory. Available: http://www.jpl.nasa.gov/

[2] C. González Calvo, "Procesamiento a bordo de imágenes hiperespectrales de la superficie terrestre mediante hardware reconfigurable," 2012.

[3] A. Plaza, P. Martinez, R. Perez, and J. Plaza, "A quantitative and comparative analysis of endmember extraction algorithms from hyperspectral data," Ieee Transactions on Geoscience and Remote Sensing, vol. 42, pp. 650-663, Mar 2004.

[4] D. Manolakis, D. Marden, and G. A. Shaw, "Hyperspectral image processing for automatic target detection applications," Lincoln Laboratory Journal, vol. 14, pp. 79-116, 2003.

[5] C.-I. Chang and Q. Du, "Estimation of number of spectrally distinct signal sources in hyperspectral imagery," Geoscience and Remote Sensing, IEEE Transactions on, vol. 42, pp. 608-619, 2004.

[6] A. Paz and A. Plaza, "Cluster versus GPU implementation of an orthogonal target detection algorithm for remotely sensed hyperspectral images," in Cluster Computing (CLUSTER), 2010 IEEE International Conference on, 2010, pp. 227-234.

[7] S. Bernabe, A. Plaza, S. López, and R. Sarmiento, "Parallel implementation of a hyperspectral unmixing chain: Graphic processing units versus multi-core processors," in Geoscience and Remote Sensing Symposium (IGARSS), 2012 IEEE International, 2012, pp. 3463-3466.

[8] D. Landgrebe, "Hyperspectral image data analysis," Signal Processing Magazine, IEEE, vol. 19, pp. 17-28, 2002.

[9] U. S. G. Survey. USGS Imagery. Available: http://www.usgs.gov/

[10] J. P. L. NASA. ASTER Image Gallery. Available: http://asterweb.jpl.nasa.gov/

[11] A. Goetz and B. Kindel, "Comparison of unmixing results derived from AVIRIS, high and low resolution, and HYDICE images at Cuprite, NV," in AVIRIS Airborne Geoscience Workshop Proceedings, 1999.

[12] J. W. Boardman, F. A. Kruse, and R. O. Green, "Mapping target signatures via partial unmixing of AVIRIS data," 1995.

[13] R. Green and B. Pavri, "AVIRIS in-flight calibration experiment, sensitivity analysis, and intraflight stability," in JPL AVIRIS Workshop, 2000.

[14] NASA. New Millennium Program. Available: http://nmp.nasa.gov/

[15] G. Motta, F. Rizzo, and J. A. Storer, "Compression of hyperspectral imagery," in Data Compression Conference, 2003. Proceedings. DCC 2003, 2003, pp. 333-342.

[16] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," ACM Computing Surveys (csuR), vol. 34, pp. 171-210, 2002.

[17] J. Turley, "Soft computing reconfigures designer options-Flexible hardware requires new design paradigms," Computer Design, vol. 36, pp. 76-77, 1997.

[18] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Physical unclonable functions and public-key crypto for FPGA IP protection," in Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on, 2007, pp. 189-195.

[19] T.-H. Li, S.-J. Chang, and Y.-X. Chen, "Implementation of human-like driving skills by autonomous fuzzy behavior control on an FPGA-based car-like mobile robot," Industrial Electronics, IEEE Transactions on, vol. 50, pp. 867-880, 2003.

[20] Xilinx. (2012). Xilinx ships the world's first heterogeneous 3D FPGA. Available: http://www.eetimes.com/electronics-products/electronic-product-reviews/fpga-pld-products/4374071/Xilinx-ships-the-world-s-first-heterogeneous-3D-FPGA

[21] K. Bondalapati and V. K. Prasanna, "Reconfigurable computing systems," Proceedings of the IEEE, vol. 90, pp. 1201-1217, 2002.

[22] M. Ahrens, A. El Gamal, D. Galbraith, J. Greene, S. Kaptanoglu, K. Dharmarajan, et al., "An FPGA family optimized for high densities and reduced routing delay," in Custom Integrated Circuits Conference, 1990., Proceedings of the IEEE 1990, 1990, pp. 31.5/1-31.5/4.

[23] H.-C. Hsieh, W. S. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erickson, et al., "Third-generation architecture boosts speed and density of field-programmable gate arrays," in Custom Integrated Circuits Conference, 1990., Proceedings of the IEEE 1990, 1990, pp. 31.2/1-31.2/7.

[24] Xilinx. (2000). Using Block SelectRAM + Memory in Spartan II FPGAs. Available: http://www.xilinx.com

[25] Xilinx. (2009). Virtex 4 Documentation. Available: http://www.xilinx.com/support/documentation/ip_documentation/ppc405_virtex4.pdf

[26] Verilog. IEEE Standard Verilog Hardware Description Language. http://www.verilog.com/IEEEVerilog.html, 2008.

[27] VHDL. IEEE VHDL Analysis and Standarization Group. http://www.vhdl.org/vasg, 2006.

[28] S. Bhunia, M. Tabib-Azar, and D. Saab, "Ultralow-Power Reconfigurable Computing with Complementary Nano-Electromechanical Carbon Nanotube Switches," in Proceedings of the 2007 Asia and South Pacific Design Automation Conference, 2007, pp. 86-91.

[29] W. R. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, et al., "Demystifying 3D ICs: the pros and cons of going vertical," Design & Test of Computers, IEEE, vol. 22, pp. 498-510, 2005.

[30] M. Hariyama and M. Kameyama, "A Multi-Context FPGA Using a Floating-Gate-MOS Functional Pass-Gate and Its CAD Environment," in Circuits and Systems, 2006. APCCAS 2006. IEEE Asia Pacific Conference on, 2006, pp. 1803-1806.

[31] Z. Li, K. Compton, and S. Hauck, "Configuration caching management techniques for reconfigurable computing," in Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on, 2000, pp. 22-36.

[32] J. Resano, D. Mozos, and F. Catthoor, "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware," in Proceedings of the conference on Design, Automation and Test in Europe-Volume 1, 2005, pp. 106-111.

[33] J. A. Clemente, J. Resano, C. Gonzalez, and D. Mozos, "A hardware implementation of a run-time scheduler for reconfigurable systems," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 19, pp. 1263-1276, 2011.

[34] R. O. Reynolds, P. H. Smith, L. S. Bell, and H. U. Keller, "The design of mars lander cameras for mars pathfinder, mars surveyor'98 and mars surveyor'01," Instrumentation and Measurement, IEEE Transactions on, vol. 50, pp. 63-71, 2001.

[35] M. Kifle, M. Andro, Q. K. Tran, G. Fujikawa, and P. P. Chu, "Toward a dynamically reconfigurable computing and communication system for small spacecraft," in Proceedings of the 21st International Communication Satellite Systems Conference & Exhibit (ICSSC 2003), Yokohama, Japan, 2003.

[36] A. Plaza and C.-I. Chang, "Impact of initialization on design of endmember extraction algorithms," Geoscience and Remote Sensing, IEEE Transactions on, vol. 44, pp. 3397-3407, 2006.

[37] J. P. L. NASA. AVIRIS Spectrometer. Available: http://aviris.jpl.nasa.gov

[38] U. S. G. Survey. USGS Digital Spectral Library. Available: http://speclab.cr.usgs.gov/spectral-lib.html

[39] Exelis. ENVI software. Availibre: http://www.exelisvis.com/ProductsServices/ENVI/ENVI.aspx

[40] U. S. G. Survey. AVIRIS WTC Library. Available: http://speclab.cr.usgs.gov/wtc/

[41] U. S. G. Survey. AVIRIS WTC Hotspots. Available: http://pubs.usgs.gov/of/2001/ofr-01-0429/hotspot.key.tgif.gif

[42] S. Bernabé, S. López, A. Plaza, and R. Sarmiento, "GPU implementation of an automatic target detection and classification algorithm for Hyperspectral Image Analysis," 2012.

[43] S. Bernabé and A. Plaza, "Commodity Cluster-Based Parallel Implementation of an Automatic Target Generation Process for Hyperspectral Image Analysis," in Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on, 2011, pp. 1038-1043.