Aceleración de procesado de imágenes de tráfico mediante diseño de algoritmos implementados en hardware

Trabajo Fin de Máster
Máster en Ingeniería Informática

Fernando Capellán Pizarroso

Dirigido por

Oscar Garnica Alcázar
Juan Lanchares Dávila

Departamento de Arquitectura de Computadores y Automática

Facultad de Informática

Universidad Complutense de Madrid

Curso académico 2016 – 2017

Convocatoria de febrero

Calificación: 6,5

Aceleración de procesado de imágenes de tráfico mediante diseño de algoritmos implementados en hardware

Trabajo fin de máster

Máster en Ingeniería Informática

Autor

Fernando Capellán Pizarroso

Directores

Oscar Garnica Alcázar
Juan Lanchares Dávila

Departamento de Arquitectura de Computadores y Automática

Facultad de Informática

Universidad Complutense de Madrid

Curso académico 2016 - 2017

Autorización

El abajo firmante, matriculado en el Máster de Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor en el presente Trabajo Fin de Máster: "Aceleración de procesado de imágenes de tráfico mediante diseño de algoritmos implementados en hardware", realizado durante los cursos académicos 2015-2016 y 2016-2017 bajo la dirección de Juan Lanchares Dávila y Óscar Garnica Alcázar en el Departamento de Arquitectura de Computadores y Automática, y a ña Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en internet y garantizar su preservación y acceso a largo plazo

Autor

Fernando Capellán Pizarroso

Agradecimientos personales

No me habría sido posible realizar este proyecto de no haber contado con el cariño y el apoyo de las personas que me han acompañado durante los cursos del Máster. En primer lugar, agradecer a mis directores Juan y Óscar por proponerme este proyecto, por todo lo aprendido y la experiencia de participar en congresos como ponente, y por guiarme, corregirme y aconsejarme durante la duración del mismo. En segundo lugar, a todos los profesores del Máster que han buscado desafiar nuestras mentes transmitiéndonos conocimientos de los que carecíamos al terminar la carrera. Y por último y no menos importante, a toda mi familia, amigos y compañeros, cuyo inestimable apoyo me ha acompañado tanto en los buenos momentos como en los no tan buenos.

A todos vosotros. Gracias.

Resumen

La iluminación adaptativa es un sistema de iluminación presente en un vehículo o medio de transporte que modifica su funcionamiento según las necesidades del conductor. Este comportamiento dinámico permite iluminar zonas específicas de la carretera para facilitar la conducción del vehículo en condiciones de baja visibilidad. Los sistemas de iluminación adaptativa actuales proporcionan las siguientes funcionalidades:

- Alumbrado lateral. Empleado a baja velocidad, para iluminar la zona que queda a los lados del vehículo cuando se va a realizar un giro cerrado o lento.
- Alumbrado en curva. Ilumina el trazado de una curva rotando la fuente de luz del vehículo.
- Cambio automático de luces. El vehículo utilizará las luces largas todo el tiempo que sea posible, ya que éstas proporcionan una mejor visibilidad de la carretera al conductor. El sistema alternará entre luces largas y luces de cruce si detecta otros vehículos a los que pueda deslumbrar, o si las condiciones de visibilidad son buenas y no hacen falta.

Como complemento a estos sistemas se ha buscado desarrollar un nuevo tipo de iluminación que proyecte un haz de luz sobre las señales de tráfico de la carretera. De este modo, el conductor del vehículo podrá leer dichas señales aun cuando se encuentre conduciendo durante la noche, sin riesgo de que los faros causen deslumbramientos a otros conductores.

La gente de la Facultad de Óptica de la UCM llevó a cabo la tarea de implementar dicho sistema en MATLAB. El objetivo de este trabajo es implementar parte del software de dicho sistema en hardware para acelerar la ejecución de los cálculos que lleva a cabo, ya que debe funcionar en tiempo real incorporado en un vehículo. La operación que se implementa en este proyecto es la transformación geométrica proyectiva bidimensional, que se utiliza para redimensionar una imagen obtenida de una cámara a la resolución requerida por el sistema de proyección de luz, sin causar distorsiones ni perder detalle.

Antes de implementar el sistema, se desarrolló primero en MATLAB un programa cuyo comportamiento fuese el mismo que el que se buscaba conseguir con el hardware. El objetivo de esta aproximación es comprender de antemano la magnitud del problema, para poder planificar cómo diseñar el sistema en hardware, y las operaciones que se deben realizar para obtener un resultado correcto. Tras comprobar la validez de dicho desarrollo, se procedió a diseñar el sistema usando lo aprendido previamente. Se tuvieron en cuenta las limitaciones con las que cuenta la placa FPGA, lo que permitió descartar aproximaciones menos óptimas. Una vez que la funcionalidad que debía tener el sistema quedó clara, se implementó el diseño en código VHDL mediante la

herramienta HDL Designer. Cuando se finalizó, se desarrolló un entorno de simulación para comprobar que su funcionamiento era el deseado. Los resultados obtenidos coinciden en su mayor parte con los que proporcionó el desarrollo software con MATLAB, por lo que se puede concluir que el diseño es válido.

Palabras clave: FPGA, Iluminación adaptativa, Transformación geométrica proyectiva, Diseño Hardware, Procesado de imagen.

Abstract

Adaptive Lighting refers to a lighting system of a car or another means of transport that alters its behaviour depending on the driver's needs. This dynamic behaviour allows lighting specific areas of the road to help the driver to see with low visibility conditions. Current adaptive lighting systems provide the following functionalities:

- Lateral lighting. Used at low speeds. It lights the areas at the sides of the vehicle when it is taking a sharp or a slow turn.
- Bend lighting. It lights the layout of the bend the vehicle is negotiating by turning the car headlights.
- Automatic light switching. The vehicle will employ its full beams as long as possible, because they enable the driver to see the road better. The system will switch to low beams when it detects any other vehicle it can dazzle, or when the visibility conditions are good enough and they aren't necessary.

A new type of adaptive lighting has been developed as a complement to the existing ones, which can irradiate a beam of light onto the surface of the traffic signals the vehicle comes across. Thanks to this, the driver of the vehicle will be able to read them even if he is driving at night, without the risk of dazzling other drivers.

The researchers at the Facultad de Óptica of the UCM implemented said system with MATLAB. The goal of this project is to translate part of that implementation into hardware to speed up the execution of the operations it performs, as it has to be able to work in real time while it is installed on a vehicle. The mathematical operation implemented in this project is the two-dimensional projective geometric transformation, which is used in order to rescale the picture taken by a camera into the resolution needed by the light projection system. Rescaling a picture with this approach prevents the picture from suffering distortion or losing detail.

Before implementing the system in hardware, its behaviour had to be replicated with a program written in MATLAB. The goal of this approach was to understand the difficulty of the problem at hand, to plan in advance how to design the hardware system and the operations that had to be performed to achieve an accurate result. After verifying the program's validity, the system was implemented with the knowledge attained from it. FPGA limitations had to be taken into account to discard less optimal solutions beforehand. Once the planning of the system functionality was over, the design was codified in VHDL with the HDL Designed tool. When the coding stage finished, a test environment was set up to check that its behaviour was the one expected. The results it provided matched greatly the ones from the MATLAB program, so it was concluded that the design was a success.

Keywords: FPGA, Adaptive Lighting, Projective Geometric Transformation, Hardware design, Image Processing.

Índice

Índice de figuras

Índice de tablas

Parte I

Estado del arte

Capítulo 1
Motivaciones del trabajo

Los medios de transporte están en continua evolución para ofrecer mejoras en la seguridad y el confort de sus usuarios. Elementos que antaño podrían considerarse lujos han pasado a ser incluidos en todo modelo de automóvil, por básico que éste sea, como por ejemplo la dirección asistida, el sistema de frenado antibloqueo o ABS, elementos de protección ante impactos como airbags o cinturones de seguridad, control de estabilidad del vehículo o ESP, etc.

Uno de estos elementos de seguridad es el de iluminación adaptativa. A diferencia de los métodos de iluminación estándar, que únicamente enfocan hacia delante del vehículo, los sistemas de iluminación adaptativa pueden alumbrar zonas de la vía específicas para que el conductor anticipe mejor los peligros que puedan presentarse. Estos sistemas permiten iluminar la vía de diferente forma, según las circunstancias en las que se encuentre el vehículo:

- Iluminación lateral, que alcanza un rango de apertura desde el frontal hacia cada lado del coche de hasta 80º. En la primera imagen de la figura 1 se puede ver la anchura del área que es posible iluminar. Complementa a los faros del vehículo para alumbrar una mayor área de la zona cuando éste se mueve a baja velocidad. Su principal uso es en ciudad, donde el vehículo puede tomar curvas muy cerradas a bajas velocidades. En este escenario, sus faros pueden no alcanzan a iluminar un posible paso de peatones u otro vehículo estacionado en las proximidades de la curva. Un ejemplo de su uso fuera de ciudad podría ser en una carretera de un puerto de montaña. Estas carreteras suelen ser estrechas, tener curvas muy cerradas para subir por la ladera de una montaña y dada su localización remota, contar con escasa iluminación. La luz puede iluminar la curva para que el conductor pueda trazarla con mayor seguridad. Esta luz se proporciona con un foco adicional que puede ser el de los faros antiniebla si el vehículo cuenta con ellos, o un faro extra que esté enfocado hacia el exterior del automóvil.
- Iluminación de curva, cuyo rango es mucho menor, pudiendo iluminar un área de tan solo 15º en cada dirección respecto al sentido de la marcha del coche. Esta luz se emplea cuando el vehículo circula a una velocidad mayor que la velocidad máxima en ciudad, por lo que está pensada para usarse en carretera, autopista y autovía. En este caso prima iluminar la vía a una mayor distancia aunque la apertura de la luz sea menor que en el caso anterior, como se puede apreciar en la segunda imagen de la figura 1. No requiere de focos adicionales, sino que se

emplean los faros del vehículo, que giran sobre el eje vertical gracias a un motor eléctrico. El grado de giro del faro es calculado por la centralita del vehículo según la velocidad a la que circula y la rotación del volante que ha realizado el conductor.

- Cambio automático de luces, que alterna entre luces largas y luces de cruce automáticamente. Su finalidad es utilizar las luces largas o de carretera del vehículo el mayor tiempo posible, ya que permiten alumbrar una distancia mayor y, en consecuencia, prevenir mejor los riesgos que puedan presentarse durante la circulación. Este sistema hace uso de un sensor o cámara colocada en el frontal del vehículo, normalmente en el soporte del espejo retrovisor interior, que detecta cuándo se aproxima un vehículo en sentido contrario que pueda ser deslumbrado. La última imagen de la figura 1 muestra cómo actúa este tipo de iluminación para evitar deslumbrar a otro vehículo que circule en sentido contrario.

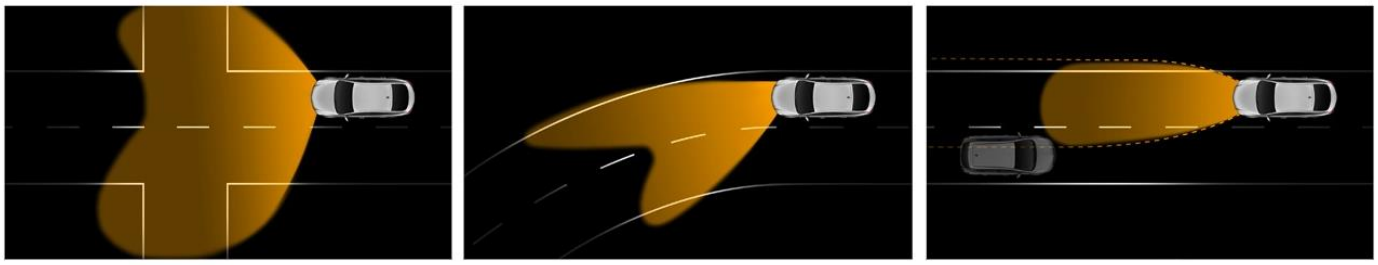

Figura 1. Los tres tipos de iluminación adaptativa explicados: alumbrado lateral, alumbrado en curva y cambio automático de luces.

Existen otras propuestas de iluminación adaptativa que complementan a las citadas anteriormente. Entre ella se encuentra un sistema cuyo objetivo es iluminar la superficie de las señales de tráfico que el vehículo encuentre a su paso, utilizando un foco especializado para no tener que emplear aquellos con los que ya cuenta y perder funcionalidad. La principal ventaja de este faro auxiliar es que el conductor pueda identificar las señales cuando las condiciones de iluminación sean escasas. Además, dados los avances en automoción y electrónica, se puede esperar conseguir que el vehículo acabe reconociendo la señal de tráfico mediante sistemas de visión por computador que incluyan los fabricantes en ellos. Esto puede ser especialmente atractivo para fabricantes de coches y empresas que tengan planes de crear automóviles autónomos, ya que un vehículo que pueda utilizar las señales que se encuentran presentes en la vía como complemento para adaptar su comportamiento tendrá un grado de seguridad mayor que aquel que sólo pueda hacerlo observando al resto de vehículos.

En la cátedra Valeo de la Facultad de Óptica de la Universidad Complutense de Madrid se ha desarrollado un software que detecta señales de tráfico en las imágenes captadas mediante una cámara incorporada en un vehículo. Esta información se transmite a un sistema de proyección de luz que enfoca únicamente a la zona que se desea iluminar, en concreto la superficie de la señal, lo que facilita al conductor del vehículo la lectura de las señales aun cuando la luz escasee. Este sistema se muestra en la figura 2, en la que están señaladas la cámara de infrarrojos que capta imágenes en condiciones de baja visibilidad y el sistema de proyección de luz que se encuentra debajo de ella.

Figura 2. Sistema de Iluminación Adaptativa de la cátedra Valeo. El ordenador que lo controla está instalado dentro del vehículo.

El problema de esta implementación es que ha sido realizada en software y escrita íntegramente en lenguaje MATLAB. Aunque es un lenguaje que ofrece gran variedad de recursos para tratamiento de imágenes y es muy versátil para trabajar con ellas, acarrea la desventaja de que la ejecución del código es muy lenta. Esto no sería un problema si fuese a ser usado en un entorno de investigación o académico, en el que se puede esperar a que el ordenador produzca los resultados deseados. Hay que tener en cuenta este problema ya que se pretende usar en un coche que puede desplazarse a una velocidad elevada, un escenario donde se dispone de escaso tiempo para tomar la imagen, procesarla para detectar la señal y reorientar la fuente de luz para alumbrar su superficie.

La figura 3 muestra la detección de las formas geométricas de las superficies de las señales en una foto tomada por el sistema. Este es el formato de fotografías que recibirán las implementaciones software y hardware para ser tratadas. Sobre las zonas blancas de la imagen se proyectará un haz de luz que las resalte para facilitar su lectura. El resultado en un escenario real es el que se puede ver en la figura 4, en una foto tomada por los investigadores de la Facultad de Óptica. Como se puede ver, las señales son perfectamente legibles en condiciones de nula visibilidad.

Así mismo, es importante mencionar la labor de tratamiento de imagen que realizó este equipo, ya que su sistema reconoce también el color de la superficie de la señal, por lo que el haz de luz que se proyecta sobre ella es del mismo color. Esto acentúa la saturación del color de la señal de tráfico en lugar de iluminarla con una luz blanca que pueda provocar destellos y deslumbramientos.

Figura 3. Superficies geométricas de señales de tráfico detectadas en una foto.

Figura 4. Señales de tráfico alumbradas por el sistema de proyección de luz del vehículo.

Figura 5. Comparación entre los resultados obtenidos con luces proyectadas de diferente color. En la foto de la izquierda, luz blanca y en la de la derecha, luz azul. La flecha de la señal queda más clara y definida al utilizar luz del mismo color que el fondo.

El objetivo de este trabajo es implementar parte del software de dicho sistema en hardware, de modo que los cálculos que realiza sean acelerados en gran medida, lo que le permitiría funcionar en tiempo real, según las especificaciones que deba cumplir. La parte que se va a implementar en hardware es la transformación geométrica proyectiva bidimensional, la cual permite al sistema redimensionar la imagen que obtiene la cámara a una resolución con la que puede trabajar el dispositivo de proyección de luz.

Capítulo 2

Transformación Geométrica Proyectiva Bidimensional

Como se ha explicado en el apartado anterior, nuestro sistema va a implementar en hardware la transformación geométrica proyectiva bidimensional para acelerar en gran medida su velocidad de ejecución. Dicha transformación es necesaria en un sistema de iluminación adaptativa porque permite transformar las imágenes de entrada obtenidas mediante cámaras instaladas en el vehículo a una resolución con la que puedan trabajar los actuadores del sistema. Los elementos actuadores son los que adaptan la rotación, amplitud e intensidad de los focos de luz según las necesidades del conductor en cada momento, lo que se conoce como Iluminación Adaptativa.

Una transformación geométrica es una operación que permite eliminar la distorsión geométrica de una imagen, causada por el equipo utilizado durante su captura, las condiciones de operación o el proceso de tratamiento de la imagen que se realiza con la finalidad de usarla en otro en otro momento. Esta transformación relaciona cada píxel de una imagen de entrada con otro de la imagen de salida, como ocurre en la figura 6. Para realizar los cálculos se utiliza una matriz, denominada matriz de transformación.

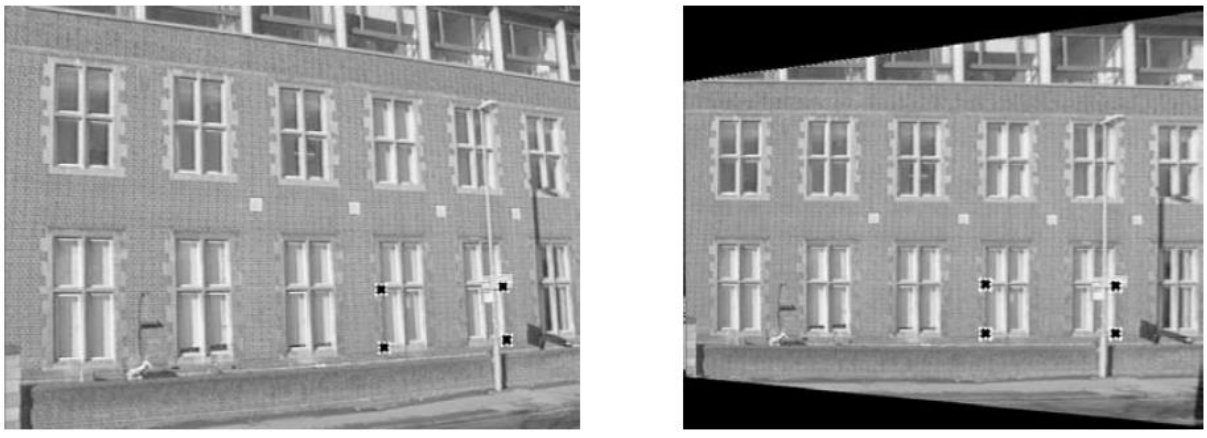

Figura 6. Ejemplo de eliminación de distorsión perspectiva. Tomando como referencia los puntos de las ventanas se obtiene la transformación a realizar, mapeándolos como un rectángulo. Imagen tomada de HARTLEY, ZISSERMAN (2003)

Esta correspondencia entre los píxeles de ambas imágenes se puede realizar de dos maneras diferentes:

1. Directa
2. Inversa

En la transformación directa, para cada píxel con coordenadas (x,y) de la imagen de entrada (IE) se calcula, mediante la transformación geométrica (Tform), las coordenadas (u,v) del píxel de la imagen de salida (IS) sobre el que se debe ubicar su valor. Es decir, escribir el valor de un píxel de la imagen de entrada en uno de la imagen de salida. Una vez calculado (u,v) se obtienen los píxeles de la imagen de salida de la siguiente forma IS(u,v)=IE(x,y).

$$(x, y) \xrightarrow{T_{form}} (u, v) \tag{1}$$

Figura 7. A cada píxel de la imagen de salida se le asigna el valor de un píxel, de varios o de ninguno de la imagen de entrada.

Uno de los problemas que implica esta transformación, como se puede apreciar en la figura 7, es que puede no establecerse una correspondencia entre todos los píxeles de la imagen de salida con los de entrada, por lo que pueden quedar zonas vacías que se rellenan con negro. El segundo problema que presenta es que el valor de un píxel puede ser sobrescrito múltiples veces, lo que provoca que se pierda información de la imagen original.

En la transformación inversa, para cada coordenada (u,v) de la IS se calcula la correspondiente coordenada (x,y) de la IE, de la que se tomará su valor, aplicando la función inversa de Tform (TformInv o T_{form}^{-1}).

Una vez calculadas (x,y), es recomendable aplicar una interpolación, ya que al transformar una imagen no realizando una aplicación 1 a 1 entre los píxeles de ambas se puede perder definición en los elementos que contiene. Esto es un problema común en el reescalado de imágenes en fotografía e ilustración digital. Como evidencia la figura 8, el logotipo de Firefox 2 adquiere bordes aserrados cuando es reescalado, mientras que sobre el de Firefox 3 se aplica una interpolación para suavizar los bordes de la imagen.

Figura 8. Comparativa entre el escalado de una imagen sin y con interpolación. Al aplicar la interpolación se suavizan no solo los bordes de la imagen sino también el resto de los píxeles mapeados.

La interpolación puede modificar el valor que se asigna a un píxel con la información de los píxeles que le rodean. Existen múltiples tipos de interpolación, y para elegir uno se debe tener en cuenta que cuanto mayor sea la precisión que aporte al resultado final, mayor será su también el tiempo que tomará calcularla. Los tres tipos de interpolación más utilizados los siguientes:

- Nearest Neighbor: el valor del punto corresponderá al del píxel más cercano a él. Es la interpolación más sencilla, ya que no requiere hacer cálculos complejos para aplicarla.
- Interpolación bilineal: se asigna a un punto la media ponderada de los valores de sus cuatro píxeles vecinos más cercanos. Los pesos de esta media ponderada se calculan según la distancia del nuevo píxel a cada píxel vecino conocido.
- Interpolación bicúbica: parecida a la interpolación bilineal, pero en este caso se utiliza la media ponderada de los valores de los 4x4 píxeles vecinos más cercanos. Los píxeles más cercanos tendrán un peso mayor en el cálculo de la media del nuevo píxel. Esta interpolación es la más usada en programas de edición de imagen profesional por ofrecer el mejor equilibrio entre el tiempo de cálculo que se tarda en realizar y la calidad resultante de la imagen que produce.

En la figura 9 se ofrece un ejemplo del funcionamiento de los tres tipos de interpolación con una imagen real.

Figura 9. Comparación de los suavizados obtenidos mediante los tres tipos de interpolaciones. La primera es la imagen original de tamaño 512x512 píxeles. La segunda es la misma imagen reducida hasta 128x128 píxeles. Las tres siguientes son imágenes reescaladas a la resolución original utilizando Nearest Neighbor, interpolación bilineal e interpolación bicúbica respectivamente.

Se puede apreciar cómo una mayor complejidad implica un mejor suavizado de la imagen. También se puede apreciar que debido a la reducción inicial se pierde nivel de detalle, algo que no pueden solucionar los algoritmos de interpolado.

En las explicaciones anteriores se habla de que un punto adquiere un valor, en lugar de hacerlo un píxel. Esto es así porque generalmente los valores obtenidos al realizar la transformación inversa son fraccionarios. Como el índice con el que se referencia un píxel de una imagen no puede ser un número decimal, se debe redondear al valor más cercano, pero esto no influye en la interpolación. La interpolación modifica el valor del píxel, no el índice sobre el que se va a escribir. Tras esta aproximación al píxel más cercano, se obtiene un nuevo punto con los valores (x',y'). El valor del píxel correspondiente a estas coordenadas debe interpolarse usando alguno de los métodos que hemos visto previamente, antes de ser escrito en la imagen de salida. Podemos generalizar este proceso de la siguiente forma: IS(u,v) = Interpolate(IE(x',y')).

$$(u, v) \xrightarrow{T_{form}^{-1}} (x, y) \tag{2}$$

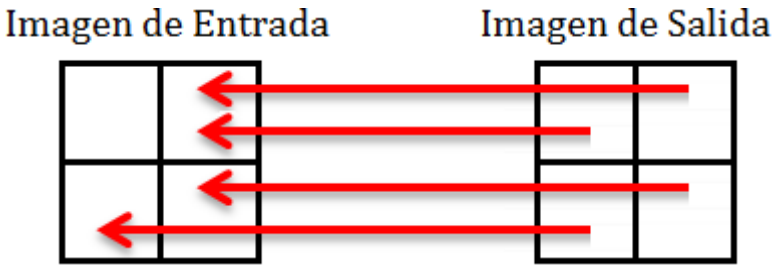

Figura 10. A cada píxel de la imagen de salida se le asigna el valor de un único píxel de la imagen de entrada.

Una transformación geométrica bidimensional puede ser de diferentes tipos:

- Rígida: únicamente admite rotaciones y traslaciones de la imagen
- Semejanza: incluye el escalado de la imagen, manteniendo los ángulos y la relación de las longitudes
- Afín: además de las anteriores, incluye la transformación de corte (escalado no uniforme en alguna dirección). El resultado mantiene los paralelismos en la imagen.
- Proyectiva: esta transformación mapea líneas a líneas, pero no mantiene necesariamente su paralelismo. Lo único que mantiene es la relación cruzada (cross-ratio) o relación entre las relaciones de las longitudes.

Es necesario realizar una transformación de alguno de estos tipos sobre las imágenes que proporciona la cámara, ya que tienen una resolución diferente a la imagen que puede proyectar el sistema de iluminación. Debido a esto, se necesita adaptar la imagen recabada a la resolución con la que puede trabajar el sistema de proyección de luz.

A continuación se exponen ejemplos gráficos de los diferentes tipos de transformaciones, para ilustrar las definiciones.

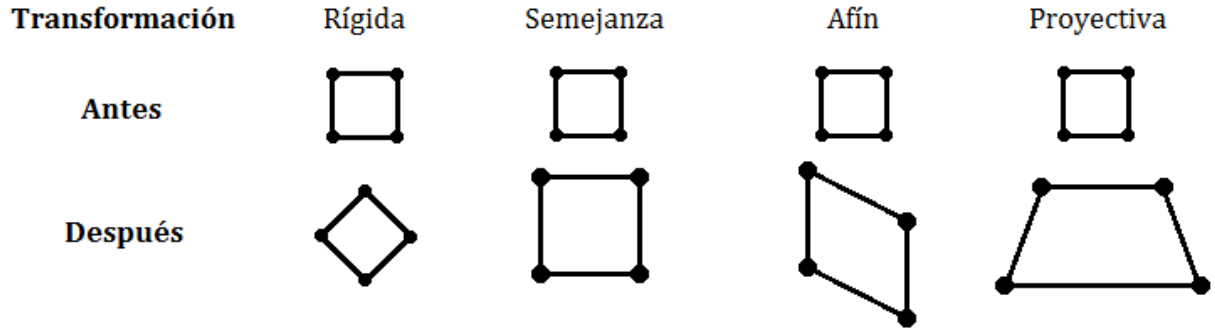

Figura 11. Ejemplos gráficos de los diferentes tipos de transformaciones geométricas.

La que utilizamos en este proyecto es una transformación de tipo proyectiva. Para expresarla se utilizan coordenadas homogéneas que representan un punto en el espacio proyectivo. La forma de generar las coordenadas homogéneas es la siguiente, a un punto (x, y) se le añade una nueva coordenada, lo que lo convierte en (x, y, 1). Dado que el factor de escala no es importante, (x, y, 1) es igual a (αx, αy, α) para todo valor de α diferente de 0, por lo que el punto (0, 0, 0) es inválido. De esta equivalencia proviene el nombre de coordenadas homogéneas.

Las ecuaciones que representan la transformación proyectiva que realizaremos utilizando coordenadas homogéneas son las siguientes:

$$\begin{pmatrix} u \\ v \\ s \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \rightarrow \begin{matrix} u \\ v \\ s \end{matrix} = \begin{matrix} ax + by + c \\ dx + ey + f \\ gx + hy + 1 \end{matrix} \quad (3)$$

Donde la matriz central corresponde a la matriz de transformación y los vectores (x,y,1) y (u,v,s) a las coordenadas homogéneas de los píxeles de las imágenes de entrada y salida respectivamente. Por tanto se puede deducir que representa una transformación directa. En caso de querer realizar una transformación inversa, los vectores de las coordenadas se intercambian de lugar y la matriz deberá transformarse en su inversa.

Vamos a ejemplificar cómo se realizaría una transformación geométrica inversa. Supongamos que tenemos una matriz de transformación con el siguiente formato:

$$\begin{pmatrix} 0{,}69140625 & 0{,}0009765625 & 0 \\ 0{,}0068359375 & 0{,}685546875 & 0 \\ 196{,}0322265625 & 72{,}6572265625 & 1 \end{pmatrix}$$

Esta matriz serviría tal como está para realizar una transformación directa, pero no inversa como queremos. En este caso es necesario invertirla, quedando de la siguiente forma:

$$\begin{pmatrix} 1{,}447265625 & -0{,}0029296875 & 0 \\ -0{,}0087890625 & 1{,}4609375 & 0 \\ -283{,}0009765625 & -105{,}62109375 & 1{,}0009765625 \end{pmatrix}$$

Como hemos comentado antes, lo único que hace falta ahora es multiplicar esta matriz por el vector que representa las coordenadas del píxel de la imagen de salida. Usamos un píxel de la imagen de salida porque sobre él queremos volcar el valor del píxel de la imagen de entrada que se encuentre en las coordenadas que calculemos en este paso. Dado que el píxel se puede identificar con dos coordenadas, en este caso podemos descartar lo que produzca la tercera componente del vector. Usaremos como ejemplo el valor (2,3,1) como coordenadas del píxel de IS. Las multiplicaciones serán:

$$\begin{aligned} 1{,}447265625 \cdot 2 + (-0{,}0029296875) \cdot 3 + 0 \cdot 1 &= 2{,}8857421875 \\ (-0{,}0087890625) \cdot 2 + 1{,}4609375 \cdot 3 + 0 \cdot 1 &= 4{,}365234375 \\ (-283{,}0009765625) \cdot 2 + (-105{,}62109375) \cdot 3 + 1{,}0009765625 \cdot 1 &= -881{,}8642578125 \end{aligned}$$

Tras redondear los resultados a los números enteros más cercanos, se puede concluir que sobre el píxel de la IS de las coordenadas (2,3) se debe volcar el valor del píxel de la IE de las coordenadas (3,4).

En este proyecto se va a implementar en hardware la operación de transformación geométrica proyectiva sobre una imagen, ya que la función Imwarp de MATLAB, que es la encargada de realizarla en la implementación del equipo de la Facultad de Óptica, se demora demasiado tiempo. El procedimiento consiste en lo siguiente: se obtiene una imagen de entrada (IE) representada como una matriz de píxeles con unas dimensiones 960x1280 y cuyo eje de coordenadas es (x,y) Esta imagen está compuesta por 1.228.800 píxeles. Se quiere construir una imagen de salida (IS) representada como una matriz con dimensiones 800x1280 y cuyo eje de coordenadas es (u,v) aplicando una determinada transformación geométrica (Tform). Es decir, obtener una imagen de menor tamaño, 1.024.000 píxeles, sin perder definición ni detalle. En nuestra aproximación utilizaremos la interpolación Nearest Neighbor, ya que la definición de la imagen no es un requisito indispensable, como sí lo es la velocidad a la que se pueda procesar, para lo que esta interpolación es la más adecuada.

Capítulo 3
La función Imwarp de MATLAB

La función Imwarp de MATLAB permite realizar una transformación geométrica sobre una imagen Esta función se utilizó en el trabajo original del equipo de la Facultad de Óptica para realizar su implementación software. Debido a que es la parte que más tarda en ser calculada, en este trabajo vamos a realizar una implementación hardware de la misma.

En nuestro proyecto, primero hemos realizado una implementación software de la función Imwarp, con dos objetivos en mente:

1. Estudiar el comportamiento de la función Imwarp, para poder descomponerla en rutinas más sencillas que se puedan implementar en módulos hardware directamente.
2. Utilizar el programa para crear estímulos de pruebas con los que verificar el hardware implementado una vez terminado el trabajo. Si los resultados que produzca coinciden razonablemente con los obtenidos mediante la implementación MATLAB, podemos concluir que su comportamiento es satisfactorio.

Antes de explicar el fragmento de código en el que se ha implementado el comportamiento de la función Imwarp, es necesario aclarar un tema relacionado con el valor de las coordenadas de la imagen que utilizan las funciones de transformación geométrica y su relación con los índices de las matrices que contienen las imágenes.

Existen varias maneras en las que se puede expresar la localización de un punto en la imagen:

Índices del píxel: en estos casos la imagen se trata como una malla de elementos discretos tal y como se puede ver en la figura 12:

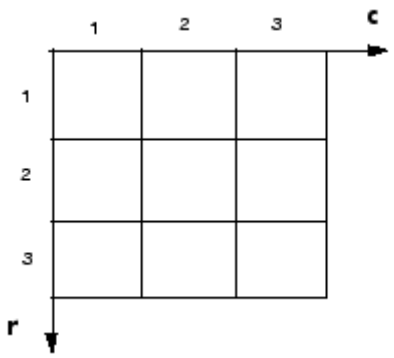

Figura 12. Cada elemento de la matriz que representa la imagen está identificado por dos componentes enteras.

Las filas se incrementan hacia abajo y las columnas se incrementan hacia la derecha. En estos casos existe una correspondencia uno a uno entre los índices de un píxel y los índices de la matriz que almacena la imagen, tal y como se tratan en MATLAB. Un ejemplo permite aclararlo: un píxel que está situado en la posición (3,2) de la imagen se corresponde con el valor almacenado en la posición (3,2) de la matriz MATLAB que contiene la imagen.

Coordenadas espaciales: existen varios tipos de coordenadas espaciales. En este proyecto nos vamos a centrar en las llamadas coordenadas intrínsecas. En esta forma de localización se utiliza un sistema de coordenadas continuas en lugar de índices discretos. En la imagen vemos un ejemplo:

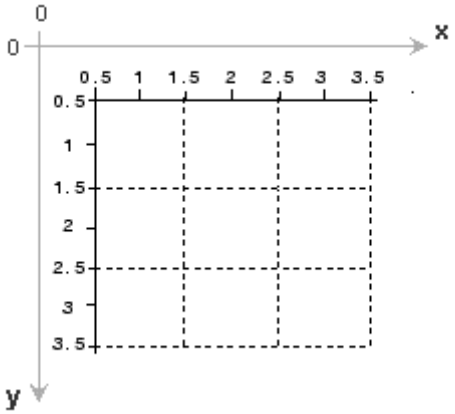

Figura 13. Cada componente de las coordenadas de un elemento de la matriz está identificado por un rango de valores de tamaño 1.

El origen de coordenadas comienza en el punto (0.5, 0.5) en lugar del clásico (1, 1) de MATLAB, y cada píxel tiene en su punto central las coordenadas enteras que le identifican.

La toolbox de procesado de imágenes de MATLAB utiliza por defecto este tipo de localización. Es necesario tener en cuenta que a diferencia de las coordenadas (x,y) normales, en este caso la coordenada *y* se incrementa hacia abajo.

Vamos a ver la relación que existe entre las dos maneras de expresar las coordenadas de un punto. Como se puede apreciar por la figura 13, las coordenadas (x,y) en el centro de cualquier píxel coinciden con los índices fila y columna de este píxel. Por ejemplo, el punto central de un píxel que se encuentra en la fila 5 columna 3 tiene como coordenadas espaciales x=3.0, y=5.0.

Es preciso notar que el orden cambia de índices de píxel a coordenadas espaciales. En los índices de píxel se respeta el orden de filas y columnas de MATLAB, es decir (f,c)=(5,3) mientras que en las coordenadas espaciales se expresa de la siguiente forma: (x,y)=(3.0 ,5.0).

A continuación pasamos a explicar los dos códigos MATLAB que implementamos para emular el comportamiento de la función Imwarp. El funcionamiento de la primera implementación MATLAB es el siguiente:

1. Mediante dos bucles anidados se recorren los índices de filas y columnas de la imagen de salida (f_out y c_out respectivamente).
2. Se obtienen nuevos índices de columna y fila (x e y respectivamente) realizando la transformación inversa con la matriz de transformación conocida (transformPointsInverse(tform,...)).
3. Los índices obtenidos son fraccionarios, por lo que son redondeados y se comprueba si entran dentro de los límites de la imagen original (ImagenEntrada). En este punto puede ocurrir:
    3.1. Que uno de los nuevos índices sea negativo o mayor que las dimensiones de la imagen de entrada; en ese caso se descartan ambos.
    3.2. Que los nuevos índices sean índices válidos dentro de la imagen original, por lo que se copia el valor de la posición a la que referencian de la imagen de entrada a la posición de la imagen de salida que nos encontramos tratando.

El código de la implementación fue el siguiente:

```
for f_out = 1:size(patron,1)%800
    for c_out = 1:size(patron,2)%1280
        [x,y] = transformPointsInverse(tform,c_out,f_out);
        if  round(x)>0  && round(y)>0 && 1281>round(x) && 961>round(y)
            ImagenSalida(f_out,c_out) = ImagenEntrada(round(y),round(x))
        end
    end
end
```

Esta implementación se descartó porque el tiempo de ejecución era de 101.232582 segundos lo que la hacía poco práctica en caso de tener que realizar modificaciones menores u obtener las imágenes de nuevo. La razón de una ejecución tan lenta es que MATLAB es mucho más eficiente realizando el cálculo con vectores y matrices que al hacer el cálculo individual sobre sus componentes. Por ello se revisó la implementación y se escribió una segunda versión. En la segunda implementación se sustituyen operaciones sobre componentes por operaciones matriciales. A continuación se describe esta segunda implementación:

1. Un único bucle sirve para iterar en todas las posiciones de la matriz de la imagen de salida. La variable rangePixels es un array de índices que va desde el índice 1 hasta el índice 800 * 1.280, o 1.024.000
2. En las variables row y col se toman los dos valores de una fila de la matriz IntCenterCartesianCoord. Esta matriz se crea de la siguiente forma:

    2.1. Se crea la matriz MatricialCoordinates. Esta matriz de dos columnas por 1.024.000 filas contiene parejas de índices para direccionar cada píxel de la imagen de salida. La primera columna hace referencia a la fila del píxel y la segunda a la columna. Tiene el siguiente formato:

Fila	Columna
1	1
2	1
…	…
799	1
800	1
1	2
2	2
…	…
799	1280
800	1280

Tabla 1. Matriz MatricialCoordinates.

    2.2. La matriz HomogeneousCoordinates se crea a partir de la anterior. Como se ha visto en el capítulo de las transformaciones geométricas, las coordenadas homogéneas de una imagen de entrada necesitan una tercera componente α. Como hay que multiplicar el resto de índices de la matriz por este α, se le dará el valor de 1, para que todos los índices queden como han sido tomados de MatricialCoordinates.
    En la tabla se añade una tercera columna de unos para cada pareja de coordenadas. Además, esta tabla se convierte en este punto a notación en punto fijo $Q_{22.10}$. Esto se hace en este paso porque aquí es donde se empieza a pensar en la ejecución de la implementación hardware, que va a usar esta misma notación para representar los datos. La notación en punto fijo se explica en un apartado más adelante en la memoria. La tabla resultante es la siguiente (recordemos que los números naturales que se muestran aquí son para mayor legibilidad, porque en la lógica del programa se tratan como números en punto fijo):

Fila	Columna	Auxiliar
1	1	1
2	1	1
...	...	...
799	1	1
800	1	1
1	2	1
2	2	1
…	…	...
799	1280	1
800	1280	1

Tabla 2. Matriz HomogeneousCoordinates.

2.3. A continuación se va a realizar la operación de transformación geométrica. Como hemos visto en el apartado donde se explican con detalle, para realizar una transformación geométrica proyectiva bidimensional hace falta tanto una matriz de transformación 3x3 como un vector vertical de 3 componentes que represente las coordenadas de cada píxel. Cada fila de la matriz HomogeneousCoordinates corresponde a las coordenadas de un píxel, por lo que es necesario trasponerla, para obtener una matriz en la que cada columna sea un vector vertical de 3 componentes que contenga coordenadas de un píxel. La matriz HomogeneousInvCoordinates se crea realizando la multiplicación de la matriz de transformación inversa por cada columna de la traspuesta de HomogeneousCoordinates. La matriz de transformación inversa utilizada es la misma utilizada de ejemplo en el apartado 3 para explicar cómo hacer una transformación proyectiva bidimensional inversa:

$$\begin{pmatrix} 1{,}447265625 & -0{,}0029296875 & 0 \\ -0{,}0087890625 & 1{,}4609375 & 0 \\ -283{,}0009765625 & -105{,}62109375 & 1{,}0009765625 \end{pmatrix}$$

En este caso, la matriz se encuentra representada en punto fijo, aunque aquí se represente con números reales para facilitar su lectura

La matriz resultante tiene el siguiente formato:

Nueva fila	1,4443359375	2,8916015625	...	1157,8095703125	1,44140625	...
Nueva columna	1,4521484375	1,443359375	...	-5,5703125	2,9130859375	...
Componente homogénea	-387,62109375	-670,6220703125	...	-226505,401367188	-493,2421875	...

Nueva fila	...	1152,615234375	1154,0625
Nueva columna	...	1862,9775390625	1862,96875
Componente homogénea	...	-361311,779296875	-361594,780273438

Tabla 3. Matriz HomogeneousInvCoordinates.

2.4. La siguiente tabla que se crea es CartesianCoordinates, cuyo formato es el mismo que la anterior tabla, pero suprimiendo la tercera fila, ya que tercera componente, añadida por estar utilizando coordenadas homogéneas, es innecesaria para los cálculos.

2.5. A continuación se crea el vector de dos filas y una columna Offset. Éste almacena el desplazamiento al que se debe someter la imagen una vez transformada para que quede en la misma posición que las que producen las funciones Imwarp y transformPointsInverse. El valor de este desplazamiento fue obtenido mediante prueba y error. También está expresado en punto fijo $Q_{22.10}$ y su valor es el siguiente:

Offset
95
292

Tabla 4. Vector Offset.

El 95 corresponde al número de filas que hay que desplazar la imagen hacia abajo, y la otra componente, 292 columnas que se debe desplazar la imagen hacia la derecha.

2.6. La siguiente matriz es CenterCartesianCoord. Para esta matriz se redondean las componentes de CartesianCoordinates y a cada columna se le resta el vector Offset. Para finalizar se traspone la matriz, lo que nos da como resultado una matriz de 2 columnas y 1.024.000 filas, cada una con las coordenadas del píxel de la IE del que se debería leer su valor

2.7. La última matriz es IntCenterCartesianCoord, la cual se crea transformando los valores en punto fijo $Q_{22.10}$ de CenterCartesianCoord a enteros.

3. Una vez que se cogen los dos valores de una fila de IntCenterCartesianCoord, se comprueba si entran dentro de los límites de la imagen. Es decir, si no son negativos o demasiado grandes. Si es así, la coordenada se descarta y se pasa a comprobar la siguiente.
4. Si no, se lee el valor de esta posición en la IE y se escribe en la posición de la IS actual, a la cual se puede acceder mediante la matriz MatricialCoordinates, que almacena en la posición idx las dos componentes de la coordenada de la IS sobre la que se debe escribir.

```
for idx=rangePixels
   row = IntCenterCartesianCoord(idx,1);
   col = IntCenterCartesianCoord(idx,2);
   if row > 0 && row < 961 && col > 0 && col < 1281
      ImagenQ22_10(MatricialCoordinates(idx,1),MatricialCoordinates(idx,2))
      = uint8(ImagenEntrada(row,col));
   end
end
```

El tiempo de ejecución de esta implementación es 3.669646 segundos, lo que la hace mucho más útil a la hora de realizar pruebas y pequeñas modificaciones en el algoritmo.

Tras la implementación se probaron diferentes valores del vector Offset, hasta que el resultado se consideró satisfactorio por su similitud con la imagen golden que proporcionaba la función Imwarp. El algoritmo resultante consiste en un conjunto de operaciones aritméticas sobre números en punto fijo por lo que su implementación en hardware es prácticamente directa.

Parte II
Implementación

Capítulo 4

Diferencias entre MATLAB y VHDL

Debido a que Matlab es un lenguaje de cálculo técnico y VHDL es un lenguaje de descripción de hardware, es normal que entre ambas aproximaciones existan diferencias. Estas diferencias pueden afectar a tipos o representación de datos, estructuras de almacenamiento de información, etc. Por otro lado las FPGA tienen recursos limitados lo que a su vez provoca que decisiones acertadas en MATLAB no se puedan aplicar en su traslación a VHDL. En este capítulo se van a explicar dichas diferencias y los motivos por las que aparecen.

4.1. Cálculo de las coordenadas y almacenamiento de imágenes

Como se ha explicado en el capítulo 3, el algoritmo de MATLAB crea una serie de matrices auxiliares en las que escribe los datos que va a utilizar para realizar los cálculos, como las coordenadas matriciales cada píxel de la imagen de salida, sus correspondientes coordenadas homogéneas, las coordenadas de la imagen de entrada una vez realizada la transformación, las coordenadas al realizar la traslación del píxel, etc. Planteando una analogía, sería como que el algoritmo genera tablas de consulta, o look-up tables, de donde leerá los valores que necesite en cada momento.

El algoritmo hardware, por contra, genera las coordenadas de los píxeles de la imagen de salida y las de los píxeles que les corresponden de la imagen de entrada sobre la marcha y no las almacena en ningún momento, ni durante el procesado ni al terminar las operaciones. Los resultados que obtiene los utiliza para ir copiando píxeles de la imagen de entrada en la imagen de salida, y después son descartados. Esta decisión de diseño se tomó porque los recursos de los que dispone la FPGA con la que se trabaja no son abundantes. Por otro lado esta aproximación tiene la ventaja de dotar al sistema de mayor flexibilidad en caso de que se decida cambiar de placa o la resolución de las imágenes de entrada o salida sea diferente a la estudiada en este trabajo.

Otra diferencia entre la aproximación MATLAB y la de VHDL es la forma en la que se almacenan las imágenes. En MATLAB no se tiene, en principio, ninguna limitación de espacio y por lo tanto el tamaño de las imágenes no supone ningún problema. En cambio, la FPGA que usamos sólo cuenta con 14.976 Kb de memoria BRAM onchip. Para almacenar las imágenes que procesa el sistema se necesita:

- Almacenar la imagen de entrada de 1280×960 píxeles
- Almacenar la imagen de salida de 1280×800 píxeles
- Cada píxel de las imágenes tiene un valor de 8 bits

$$\frac{(1280 \cdot 960 + 1280 \cdot 800) \cdot 8}{1024} = 17600\ Kb$$

Por lo tanto harían falta 17.600 Kilobits de memoria BRAM en la FPGA para almacenar la imagen de entrada y de salida. Esta es razón por la que no se utilizan módulos de memorias generadas con ISE Core Generator. Además de este problema el uso de estos cores presenta las siguientes dificultades:

- Tienen una capacidad fija. Una vez generados no se puede modificar si no se vuelven a generar.
- Utilizan recursos de la FPGA, reduciendo el área disponible para el resto del diseño.
- Los recursos de la FPGA son limitados, por lo que si las memorias no cupiesen en caso de redimensionarlas, habría que rediseñar el sistema para utilizar otro método de almacenamiento.

Esta es la razón por la que en esta primera aproximación al problema se decidió utilizar unos tipos de datos arrays (ver sección 5.1) para almacenar las imágenes de entrada y de salida, que tienen un comportamiento similar al de las memorias RAM generadas por el Core Generator: pueden recibir una dirección y transcurrido un ciclo de reloj devolver el dato ubicado en la posición a la que hace referencia, o pueden recibir un dato y una dirección y transcurrido un ciclo almacenarlo es la posición indicada. La imagen de entrada se obtiene de un archivo de texto, el cual no contiene la información de los píxeles en forma de matriz de *m·n datos*, sino que los almacena de forma lineal. Este formato sería el que se utilizaría en el caso de almacenarla en memoria, ya que es más eficiente. Para almacenar la imagen de salida en un fichero de texto se hace de la misma manera, con formato lineal. En MATLAB se utilizan estos archivos para construir las imágenes y mostrarlas visualmente al usuario. La forma de generar una imagen a partir de un array lineal es columna por columna de la imagen.

Otro tema que tiene que ver con la imagen es el tipo de almacenamiento utilizado para la misma. En MATLAB una imagen se representa como una matriz de píxeles, en la que cada píxel se localiza mediante dos coordenadas (x,y). En este proyecto utilizamos las coordenadas espaciales intrínsecas, en el cual la *x* hace referencia a la columna y la *y* a la fila del píxel. Sirva de ejemplo la matriz de la figura 14, en la que se representa una imagen de 12 píxeles, 4 de ancho por 3 de alto, y en la que el píxel con coordenadas (3,2) tiene un valor de 7:

1	2	3	4
5	6	7	8
9	10	11	12

Figura 14. Tabla de 4x3 elementos que representa una imagen.

En hardware no es conveniente utilizar esta estructura de memoria dado que es una implementación compleja. Lo normal es utilizar arrays lineales de vectores de 8 bits. En la figura 15 se muestra el mapa de memoria de la matriz del ejemplo anterior sobre una memoria de 12x8 bits. Como se ve por los elementos de cada celda, la matriz se ha tomado columna por columna:

1
5
9
2
6
10
3
7
11
4
8
12

Figura 15. Tabla de 1x12 elementos que representa una memoria lineal almacenando la anterior imagen.

Para acceder a cada píxel de la imagen hay que hacer una conversión entre el par de valores columna x y fila y que lo localizan en la matriz y la dirección de memoria que lo localiza en la memoria. Suponiendo que las coordenadas de la imagen en una matriz empiezan en (0,0), al contrario de lo que ocurre en MATLAB donde comienzan en (1,1), sabiendo que la primera dirección de una memoria hardware siempre es 0, y suponiendo que la matriz tiene F filas y C columnas, entonces para obtener la dirección de memoria (DM) se aplica la siguiente expresión:

$$DM = (x - 1) \cdot F + (y - 1) \tag{4}$$

Siguiendo el ejemplo anterior, para las coordenadas (x,y) = (3,2), la dirección de acceso a la memoria lineal sería:

$$DM = (3 - 1) \cdot 3 + (2 - 1) = 7$$

Teniendo en cuenta que la dirección de la primera palabra de la memoria lineal es 0x0 para la implementación hardware, el índice 7 corresponde a la octava posición de esta memoria, que contiene un valor de 7, como la posición (3,2) de la matriz bidimensional de la que se ha creado el array lineal que la almacena.

4.2. Representación de los datos

En MATLAB utilizamos la función numerictype para definir el tipo de datos de punto fijo (fixed-point-FI) que se va a utilizar y la función fimath que especifica la forma de operar con ese punto fijo. Obviamente no existe una traslación directa entre las funciones MATLAB y el VHDL, luego tanto el tipo numérico como las operaciones que se van a realizar sobre el tipo se tienen que especificar. El sistema debe ser capaz de hacer uso de un formato de datos que le permita:

- Operar con números que contengan parte decimal y no perder precisión en el proceso.
- Direccionar tantas posiciones como tengan las memorias en las que se van a leer y escribir los datos.

La representación de números en punto fijo consiste en multiplicar un número real por un factor lo suficientemente grande como para que el resultado sea un número entero. Para volver a obtener el valor real del número se deberá dividir por dicho factor. En HW un número en punto fijo se representa como un vector de bits, del cual una parte corresponde a la parte entera del número y la otra a la parte decimal. Por ejemplo, si se utilizan 10 bits de parte decimal, el factor por el que habrá que multiplicar un número real será $2^{10} = 1024$. Por lo tanto, si queremos representar el número 0,41895 en punto fijo habrá que realizar la siguiente operación $0{,}41895 * 1024 \simeq 429$. Lo normal es que el resultado obtenido tampoco sea un entero por lo tanto habrá que redondearlo al entero más cercano. La ventaja de esta notación es que al ser el número resultante un número entero en lugar de tener que utilizar operadores complejos como son los de punto flotante se pueden utilizar directamente operadores de enteros que son más sencillo de implementar. En las siguientes secciones se dan algunas nociones relacionadas con el punto fijo.

La notación Q describe el formato que tiene el punto fijo. La notación Q_n especifica que n es el número de dígitos que se encuentran a la derecha del punto, es decir, la longitud de la parte fraccionaria. Únicamente se determina la parte fraccional porque el factor de escala puede variar, por lo que no se especifica la longitud del vector de bits. En caso de necesitar hacerlo se utiliza una notación más compleja, $Q_{m.n}$, en la que m representa el número de bits de la parte entera y n el de la parte decimal. Además es necesario tener en cuenta que debe haber un bit reservado para representar el signo, lo que supone un número total de bits de $N = m + n + 1$.

Para calcular el valor decimal de un número binario de N bits con notación $Q_{m,n}$ se realiza la operación que se muestra en la figura 16:

b'_s	$b'_{m-1} \dots b'_0$	$b_{n-1}b_{n-2} \dots b_0$

$$(-1)^{b'_s} \times (2^{m-1}b'_{m-1} + \cdots + 2^0 b'_0 + 2^{-1}b_{n-1} + 2^{-2}b_{n-2} + \cdots + 2^{-n}b_0)$$

Figura 16. Formato $Q_{m,n}$

El principal desafío que plantea esta notación es definir el valor de m y n para un total de N de bits. Para ello se debe definir el rango dinámico del problema que nos concierne. Este rango depende del valor de m, y se obtiene mediante la expresión:

$$[-(2^m), 2^m - 2^{-n}]$$

donde 2^{-n} es la precisión máxima para la notación $Q_{m.n}$.

Para verlo con mayor claridad pondremos un ejemplo. Para notación $Q_{3.5}$ el rango será:

$$[-(2^3), 2^3 - 2^{-5}] = [-8, 7.96875]$$

y la precisión $2^{-5} = 0.03125$.

Para elegir el número de bits que se asignarán a *m* y a *n* se debe determinar cuál va a ser el mayor número positivo que se quiere representar en el sistema y qué precisión se desea con el mínimo error aceptable.

En nuestro caso, los cálculos se van a realizar sobre los índices de acceso a las memorias que almacenan las imágenes. El cálculo inicial que se realizó para obtener el número de dígitos necesarios para representar la parte entera fue el siguiente. Como la memoria que más datos almacenará es la de la imagen de entrada, de 960 por 1280 píxeles, pensamos que con un número con el que poder representar 1.228.800 posiciones bastaría. Dado que la potencia de 2 más pequeña que supere este número es 2^{21} = 2.097.151, se utilizaron 21 bits de parte entera y 11 bits de parte decimal en un principio, para completar el número de bits a una potencia de 2, con una longitud del vector de datos de 32 bits. En las simulaciones que se ejecutaron una vez finalizada la implementación del sistema se pudo apreciar que se producía un desbordamiento en las operaciones de cálculo de las coordenadas de los píxeles de entrada. Eso provocaba que se accediera a posiciones incorrectas de la imagen de entrada, volcando sobre la imagen de salida píxeles incorrectos. El resultado que se obtuvo fue una imagen en la que se repetía la parte de la izquierda a partir de cierto punto. Tras repasar los cronogramas de las simulaciones, se comprendió que la ejecución de las operaciones de multiplicación que calculaban el índice de lectura de la imagen de entrada provocaba el error, por lo que se incrementó la parte entera a 22 bits, lo que soluciona el problema. La notación Q elegida para implementar el problema en hardware fue por tanto $Q_{22.10}$

4.3. Aritmética de Punto Fijo

Debido a que los números en punto fijo son multiplicados por un factor de escala para transformarlos de números decimales en números enteros, no se opera con los números originales. Es necesario tener esto en cuenta al realizar una multiplicación o una división en punto fijo, ya que el factor altera el resultado correcto, y hay que realizar correcciones sobre él. Tomemos *a* y *b* como los números reales con los que queremos operar, y *f* como el factor de escala por el que se multiplican. Usaremos $A = a \cdot f$ y $B = b \cdot f$ como números en punto fijo. A continuación se explica con ejemplos cómo operar estos números:

- *Suma*:

  La operación de suma se realiza como una operación regular de suma binaria, y los números deben estar representados con la misma notación Q. Para sumar *A* y *B* lo haríamos así:

  $$A + B = a \cdot f + b \cdot f = (a + b) \cdot f \rightarrow a + b \simeq A + B$$

  Veámoslo con un ejemplo práctico: Supongamos que tenemos los siguientes números, 5 y 1,46137712324284. Estos números en la notación que manejamos, $Q_{22.10}$, son:

$$5 = 00000000000000000101.0000000000 = 5120_{(10)}$$

$$1{,}46137712324284 = 00000000000000000001.0111011000 = 1496_{(10)}$$

Y el factor de escala para 10 bits decimales es $2^{10} = 1024$

El resultado de la suma de los números decimales es:

$$5 + 1{,}46137712324284 = 6{,}46137712324284$$

Y el de los equivalentes tras multiplicar por el factor de escala:

$$5 \cdot 1024 + 1{,}46137712324284 \cdot 1024 \simeq 5120 + 1496 = 6616$$

Realizar la suma en el binario de punto fijo corresponde a:

$$\begin{array}{r} 00000000000000000101.0000000000 \\ +\ 00000000000000000001.0111011000 \\ \hline 00000000000000000110.0111011000 \end{array}$$

Resultado que si lo convertimos a un número decimal obtenemos lo siguiente:

$$00000000000000000110.0111011000 = 6616_{(10)} = 6{,}4609375 \cdot 1024$$

No es el número exacto que esperábamos pero es la aproximación más cercana que se puede conseguir con esta notación

- *Resta*:

  Para realizar una operación de resta se procede de la misma manera. Se utilizan dos número en punto fijo con la misma notación y se restan como en una operación binaria.

  $$A - B = a \cdot f - b \cdot f = (a - b) \cdot f \rightarrow a - b \simeq A - B$$

  Vamos a utilizar los mismos números que antes. Si restamos los números decimales obtenemos:

  $$5 - 1{,}46137712324284 = 3{,}53862287675716$$

  Y al restar sus equivalentes obtenidos multiplicando por el factor:

  $$5 \cdot 1024 - 1{,}46137712324284 \cdot 1024 \simeq 5120 - 1496 = 3624$$

  Si los restamos en formato de punto fijo obtenemos:

  $$\begin{array}{r} 00000000000000000101.0000000000 \\ -\ 00000000000000000001.0111011000 \\ \hline 00000000000000000011.1000101000 \end{array}$$

  Como se puede intuir, este resultado traducido de notación $Q_{22.10}$ será el resultado correcto:

  $$00000000000000000011.1000101000 = 3624_{(10)} = 3{,}5390625 \cdot 1024$$

  De nuevo, se obtiene una aproximación del resultado que se esperaba, ya que los bits decimales no ofrecen la misma precisión que la notación científica.

- *Multiplicación*:

  La multiplicación sin embargo, no es una operación tan intuitiva como las anteriores. Si utilizamos valores multiplicados por el factor de escala , este sería el resultado:

  $$A \cdot B = a \cdot f \cdot b \cdot f = (a \cdot b) \cdot f^2$$

  Obtendremos un resultado en el que se ha multiplicado por el factor de escala dos veces. Como sólo se puede multiplicar por él una única vez, hay que transformar el resultado, dividiendo por dicho factor:

  $$(a \cdot b) \cdot f^2 \rightarrow a \cdot b \simeq (A \cdot B) / f$$

  Vamos a seguir operando con los mismos números, 5 y 1,46137712324284 para ver esta operación con un ejemplo. El resultado de la multiplicación decimal será:

  $$5 \cdot 1{,}46137712324284 = 7{,}3068856162142$$

  Sin embargo, la multiplicación de los enteros correspondientes a su representación en punto fijo es muy diferente:

  $$5 \cdot 1024 \cdot 1{,}46137712324284 \cdot 1024 = 5120 * 1496 = 7.659.520$$

  Lo que corresponde a la siguiente multiplicación en binario:

  00000000000000000001010000000000

  * 00000000000000000000010111011000

  00000000000000000000000000000000000000011101001110000000000000

  Una multiplicación en binario produce un resultado con el doble de bits que el de sus multiplicandos. Como ambos tienen 32 bits, el resultado tiene el doble de bits que los operandos, 64 bits. Si traducimos este resultado a un formato más legible obtenemos:

  $$11101001110000000000000 = 7.659.520_{(10)} = 7480 \cdot 1024$$

  Se puede ver que este resultado dista mucho del correcto. Para obtener el resultado correcto de la operación a partir del que ya disponemos, se deben realizar dos operaciones sobre el resultado binario:

  1. Se debe aplicar un redondeo según el número de bits que usados para representar la parte decimal. Esto se lleva a cabo porque a continuación se va a ejecutar un desplazamiento a la derecha, ya que se quiere dividir por el factor, una potencia de 2, y si no se hiciera el redondeo se podría perder precisión. En nuestro caso son 10 bits para la parte decimal. Por ello se suma 1 al décimo bit, el más significativo de la parte decimal:

  00000000000000000000000000000000000000011101001110000000000000

  + 1000000000

  00000000000000000000000000000000000000011101001110001000000000

  2. Se eliminan los bits decimales al dividir por el factor de escala. Dado que es una potencia de 2, para dividir un número binario por ella se debe

desplazar a la derecha el vector de bits que lo representa. Como en este caso tenemos un factor de 1024, se debe desplazar 10 bits a la derecha.:

0000000000000000000000000000000000000001110100111000~~1000000000~~

3. Se conserva el bit de signo y tantos bits menos uno como la longitud de los operandos originales:

$0000000000000000000011101001110000 = 7480_{(10)} = 7{,}3046875 \cdot 1024$

Como se puede apreciar, el resultado es una aproximación al real. Dado un número limitado de bits con los que representar la parte decimal, 7,3046875 es el resultado más próximo al original, 7,3068856162142, que se puede obtener con la notación $Q_{22.10}$.

- *División*:

El caso de la división es parecido al de la multiplicación. Usemos el ejemplo de valores multiplicados por el factor de escala para verlo mejor:

$$A \,/\, B = a \cdot f \,/\, b \cdot f = (a \,/\, b)$$

Tras realizar la división hemos perdido el factor de escala en el resultado. Para obtenerlo de nuevo sería necesario multiplicar el resultado por el factor:

$$(a \,/\, b) \rightarrow a \,/\, b \simeq (A \,/\, B) \cdot f$$

Usemos los mismos números que en los apartados anteriores: 5 y 1,46137712324284, pero en este caso los vamos a invertir.

1,46137712324284 / 5 = 0,292275424648568

Si dividimos los números en binario que resultan de traducirlos a punto fijo debería resultar así:

0000000000000000000010111011000

/ 0000000000000000001010000000000

0000000000000000000000100101011

Y dividiendo los enteros que corresponden a sus representaciones binarias obtendremos esto:

$1{,}46137712324284 \cdot 1024 \,/\, 5 \cdot 1024 = 1496 \,/\, 5120 = 0{,}2921875$

El problema es que, al operar con estos enteros ya estaríamos operando con números en punto fijo, los cuales sólo pueden ser enteros, incluido el resultado. Únicamente cuando es traducido de vuelta desde punto fijo a decimal se pueden obtener números fraccionarios. Por ello es necesario buscar el valor correcto realizando las siguientes operaciones:

1. Se desplaza el dividendo a la izquierda tantos bits más uno como bits decimales tenga nuestra notación. Es decir, se multiplica al dividendo por el factor de escala, una potencia de 2, lo cual en binario se realiza fácilmente desplazando el vector de bits a la izquierda. Esta operación se hace antes de la propia operación de división ya que si no se podría perder precisión. En notación $Q_{22.10}$ se desplazaría 11 bits a la izquierda. 10111011000 se convertirá en 10111011000000000000000

2. Se realiza la división:

   10111011000000000000 / 1010000000000 = 1001010110

3. Se suma uno al resultado: 1001010110 + 1 = 1001010111
4. Se desplaza un bit hacia la derecha: el resultado es 100101011

00000000000000000000000100101011 = $299_{(10)}$ = 0,2919921875 · 1024

La cual es la mejor aproximación al resultado correcto que podemos obtener en punto fijo.

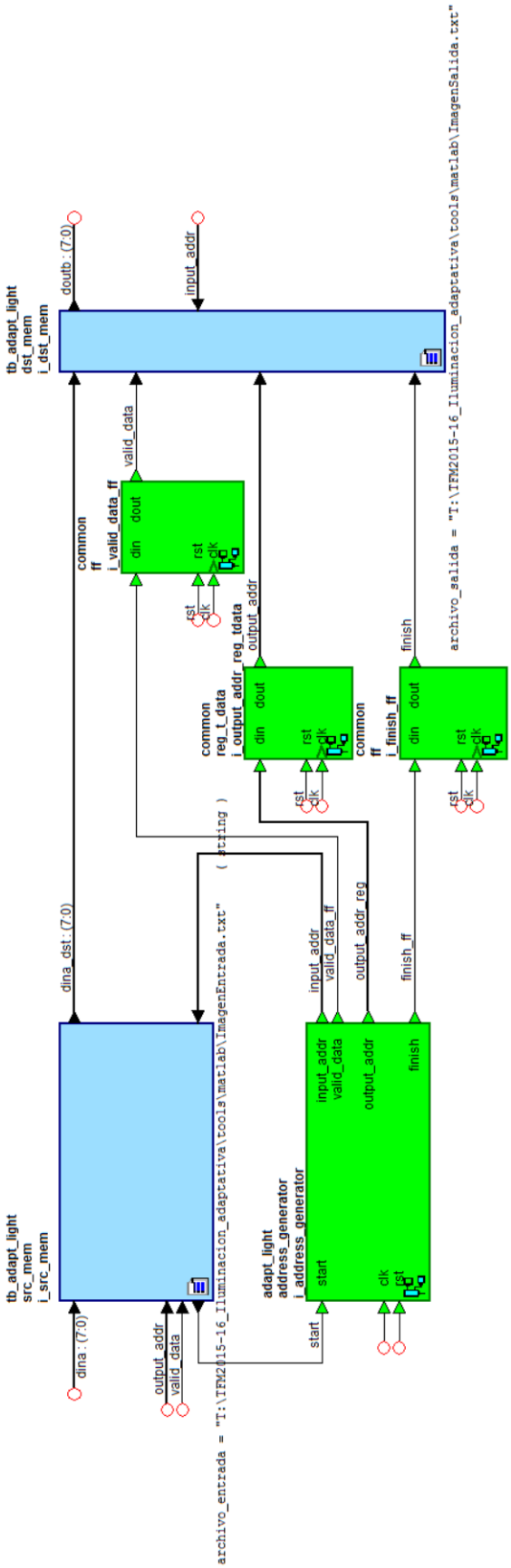

Figura 17. Esquema general del sistema completo.

Capítulo 5
Implementación hardware

En este capítulo se explica en detalle la implementación hardware de la función Imwarp de MATLAB. El diseño se puede descomponer en los siguientes módulos principales:

- Archivo *constants.vhd* en el que se especifican los tipos de datos y constantes utilizados en el diseño.
- Archivos *src_mem_beh.vhd* y *dst_mem_beh.vhd*. El primero contiene dos procedimientos: uno que lee una imagen de un archivo de texto y la almacena en un array y otro que proporciona el valor de una posición del array al recibir su dirección. El segundo también tiene dos procedimientos: uno para volcar el contenido de un array en un fichero de texto y otro con el que escribe un dato sobre una determinada posición del array, la cual se le indica como una dirección de memoria. Estos dos archivos sustituyen a las memorias en el sistema, ya que tienen su misma funcionalidad y evitan los problemas ya comentados en la sección 4.1.
- Archivo *address_generator_rtl.vhd*, que describe el componente que realiza el cálculo de las coordenadas de acceso a las memorias, aplica la transformación inversa y convierte las coordenadas en direcciones de acceso a ambas memorias, lo que permite realizar la copia de un pixel de la imagen de entrada en la imagen de salida.
- Módulos comunes. Son registros y otros módulos que se utilizan en diversas partes del diseño

5.1. Tipos de datos y constantes

En el archivo *constants.vhd* se definen tanto los tipos de datos como las constantes que usa la implementación. Los tipos de datos definidos por el usuario son los siguientes:

Nombre	Tipo	Uso
t_data	signed(data_width-1 downto 0)	Este tipo de datos implementa los números representados en punto fijo con notación $Q_{22.10}$ y se utiliza en las operaciones que calculan las coordenadas de las imágenes de entrada y de salida, así como en el cálculo de la transformada inversa.

t_3x3_matrix	array (0 to 8) of t_data	Vector de nueve elementos de tipo t_data que almacena la matriz de transformación
input_image_array	array (0 to 1228799) of std_logic_vector(7 downto 0)	Array que almacena la información de los píxeles de la imagen de entrada
output_image_array	array (0 to 1023999) of std_logic_vector(7 downto 0)	Array que sirve para almacenar los valores de los píxeles de la imagen de salida.

Tabla 5. Tipos de datos definidos por el usuario.

A continuación se muestra una lista de las constantes:

Nombre	Tipo	Valor	Explicación
data_width	natural	32	Ancho de los datos usados en el programa.
decimal_width	natural	10	Número de bits que representan la parte decimal de los valores.
dst_rows	natural	800	Número de filas de la imagen destino
dst_cols	natural	1280	Número de columnas de la imagen destino
src_rows	natural	960	Cantidad de filas de la imagen fuente
src_cols	natural	1280	Cantidad de columnas de la imagen fuente
tformInvMat	t_3x3_matrix	Matriz de la figura 18, en notación $Q_{22.10}$	Matriz de transformación, usada para realizar la transformación geométrica 2D de cada píxel
offset_rows	t_data	95, en notación Q22,10	Desplazamiento de filas necesario para que la imagen quede centrada. Este es uno de los valores del vector Offset que se ha explicado en el apartado de la implementación de MATLAB
offset_cols	t_data	292, en notación Q22,10	Desplazamiento para las columnas, que posiciona la imagen correctamente. Segundo valor de Offset, que sirve para desplazar la imagen a una posición deseada.

Tabla 6. Constantes del programa.

$$\begin{pmatrix} 1{,}447265625 & -0{,}0029296875 & 0 \\ -0{,}0087890625 & 1{,}4609375 & 0 \\ -283{,}0009765625 & -105{,}62109375 & 1{,}0009765625 \end{pmatrix}$$

Figura 18. Valor de la constante tformInvMat.

5.2. Almacenamiento de imágenes

Del almacenamiento de imágenes se encargan los módulos azules de la figura 17, *src_mem* y *dst_mem*. Como sus nombres indican, el primero se encarga de la imagen fuente (source) o imagen de entrada, y el segundo de la imagen destino (destination) o imagen de salida. A continuación se explican las características y particularidades de cada uno de ellos y de los procedimientos que contienen:

- src_mem: Lee de un archivo de texto una imagen con formato lineal. Esto quiere decir que cada línea del archivo contiene el valor de un único píxel de la imagen. Cuenta con una señal llamada *input_image*, de tipo *input_image_array* (explicado en la tabla 5), en la que vuelca el contenido del fichero. Para ello utiliza el siguiente procedimiento:

```
p_read : process
      variable l: line;
      variable s: std_logic_vector(7 downto 0);
      variable read_ok : boolean;
      variable i : integer := 0;

      begin

      while not(endfile(input_file)) loop
        readline(input_file,l);
            read(l,s,read_ok);
            assert read_ok
              report "Error during file reading: " & l.all
              severity warning;
            input_image(i)   <= s;
            i := i + 1;
      end loop;
      start <= '1' after 50 ns, '0' after 100 ns;
      wait;
end process p_read;
```

Este procedimiento se ejecuta antes de que el sistema comience a funcionar. En él se lee el fichero de texto línea por línea y se copian dichas líneas en posiciones consecutivas del array *input_image*. Una vez copiada la imagen de entrada entera, se activa la señal *start* del generador de direcciones (sección 5.3) y se desactiva poco después.

El segundo procedimiento que contiene es el que le hace funcionar como una memoria:

```
p_memory : process (clk)
begin
  if rising_edge(clk) then
     dina_dst <= input_image(to_integer(unsigned(input_addr)));
  end if;
end process p_memory;
```

En cada ciclo de reloj proporciona el dato que le es solicitado mediante una dirección de memoria. Este dato es el valor de una posición del array *input_image*

- dst_mem: Contiene una señal de tipo *output_image_array* (también explicado en la tabla 5 de tipos de datos) llamada *output_image*. Este módulo también funciona como una memoria, pero su función, al contrario que el anterior, no es proporcionar datos sino almacenarlos. Para ello cuenta con el siguiente procedimiento:

```
p_memory : process (clk)
begin
  if rising_edge(clk) then
      if valid_data = '1' then
      output_image(to_integer(unsigned(output_addr))) <= dina_dst;
      end if;
  end if;
end process p_memory;
```

  Su funcionamiento es sencillo: escribe el dato que recibe en la posición del array que se le indica mediante una dirección de memoria si se le notifica que dicho dato es válido. Esta notificación de validez se explica con mayor detalle más adelante en la sección 5.3.5.

  Además, una vez terminado el proceso de transformación de la imagen, debe volcar los datos del array a un fichero de texto para que el usuario pueda interpretarlos. Para hacer esto se utiliza el siguiente procedimiento:

```
p_write : process
variable px : line;

begin
wait until finish = '1';
      bucle: for j in 0 to dst_rows * dst_cols - 1 loop
            write(px, output_image(j));
            writeline(output_file, px);
      end loop bucle;
wait;
end process p_write;
```

  El cual empieza cuando el sistema ha activado la señal de finalización del procesado (*finish*). Lo que hace es escribir en un fichero el valor de cada posición del array *output_image* que contiene la imagen de salida en una línea del archivo de texto. Es decir, realiza una escritura lineal de la imagen.

Como estos módulos sirven para realizar las simulaciones y leer y escribir en ficheros de texto, no son sintetizables. Sólo se pueden utilizar en simulación.

5.3. Generador de direcciones

Es el módulo más importante del diseño. Su misión es proporcionar las direcciones para acceder a los arrays que contienen los módulos *src_mem* y *dst_mem*, *input_image* y *output_image*, comentados en el apartado anterior, que tienen la función de memorias. En cada ciclo aplica la transformación geométrica proyectiva a las coordenadas espaciales intrínsecas de la imagen de salida para obtener las coordenadas que corresponden de la imagen de entrada. Además, convierte las coordenadas de cada píxel de la IE en una dirección que permite leer un dato del array de la imagen de entrada *input_image* correctamente y genera una dirección que se usará para escribir ese dato en la posición correcta del array *output_image*, que contendrá la nueva imagen IS. En la tabla 7 se puede consultar su interfaz, y en la figura 19, el diagrama de bloques del mismo:

Nombre	Tamaño	Sentido	Comentario
clk	1 bit	entrada	Señal de reloj del sistema
rst	1 bit	entrada	Reset síncrono del sistema
start	1 bit	entrada	Señal disparadora de la unidad de control del módulo
input_addr	32 bits	salida	Dirección de la memoria que contiene la imagen de entrada a la que se busca acceder
valid_data	1 bit	salida	Señal que indica si el valor de input_addr es correcto y se puede usar para acceder a la matriz de la imagen de entrada
output_addr	32 bits	salida	Dirección de la memoria que almacena la imagen de salida, donde se quieren escribir los píxeles leídos
finish	1 bit	salida	Señal que indica que la transformación geométrica ha sido terminada

Tabla 7. Interfaz del módulo generador de direcciones.

Como se ha comentado al principio de la Sección 4, a diferencia de la implementación realizada en MATLAB, la implementación hardware no cuenta con ninguna tabla o matriz auxiliar que almacene las coordenadas para acelerar los accesos. Al comienzo del proyecto se pensó en crear una tabla de verdad que almacenara una entrada por cada píxel de la imagen de salida y cuya salida fuera la dirección correspondiente de la memoria que contiene la imagen de entrada de la que se debía leer el valor. Considerando la magnitud del problema, se estimó que no sería viable, ya que la tabla debería tener 1.024.000 entradas, cada una con un valor de 22 bits. Los problemas que esto acarrea son que usaría la mayor parte de los recursos de la FPGA y que no es reutilizable en caso de que cambiasen las dimensiones de la imagen o la matriz de transformación. Incluso podría ocurrir que la tabla no cupiese en la FPGA en caso de ser demasiado grande, por lo que se descartó la idea.

En su lugar se consideró que la mejor implementación sería realizar los cálculos sobre la marcha ya que el retardo que introducen no impide al sistema cumplir la restricción de tiempo, y el área que ocupa la implementación es mucho menor que si hiciera falta guardar una tabla de tal tamaño.

El funcionamiento es el siguiente:

1. Cuando el módulo *src_mem* ha copiado toda la imagen de entrada al array que funciona como memoria, activa la señal que inicia todo el proceso. El módulo generador de direcciones permanece inactivo hasta que reciba esta señal.
2. La unidad de control del módulo habilita la cuenta de filas y columnas, las cuales pueden tomar un valor máximo de 799 y 1279 respectivamente, una unidad menos que cada dimensión de la imagen de salida, porque como hemos comentado al principio, la transformación geométrica inversa recorre cada píxel de la imagen de salida.
3. Esta pareja de números toma dos caminos. En uno de ellos se les aplica la transformación geométrica inversa para generar las coordenadas de la dirección de la memoria de IE. En el otro los datos no son alterados.
4. Se comprueba que las coordenadas generadas, sobre las que se ha aplicado la transformación geométrica, entran dentro de los límites de la imagen de entrada, para que no se intente realizar un acceso a una dirección de memoria inexistente.
5. Se transforman ambas parejas de coordenadas, tanto la que se ha transformado y comprobado como la que se ha dejado sin modificar tras el paso 3, en una dirección que sirve para acceder a una memoria con formato lineal como hemos visto en la sección 4.1
6. Se manda al módulo *src_mem* la dirección de la memoria de entrada, la calculada tras la transformación geométrica. Al mismo tiempo se manda al módulo *dst_mem* la validación de dicha dirección, es decir, si es una dirección de lectura válida, y la dirección de la memoria para el array que almacena la imagen de salida. En este paso se solicita un dato a la memoria de entrada y se escribe sobre la memoria de salida si el dato es válido.
7. Cuando se han recorrido todas las filas y columnas, se activa la señal que avisa de la finalización del proceso. Esta señal la recibirá el módulo *dst_mem*, momento en el cual copiará el contenido del array que almacena la imagen de salida en un archivo de texto.

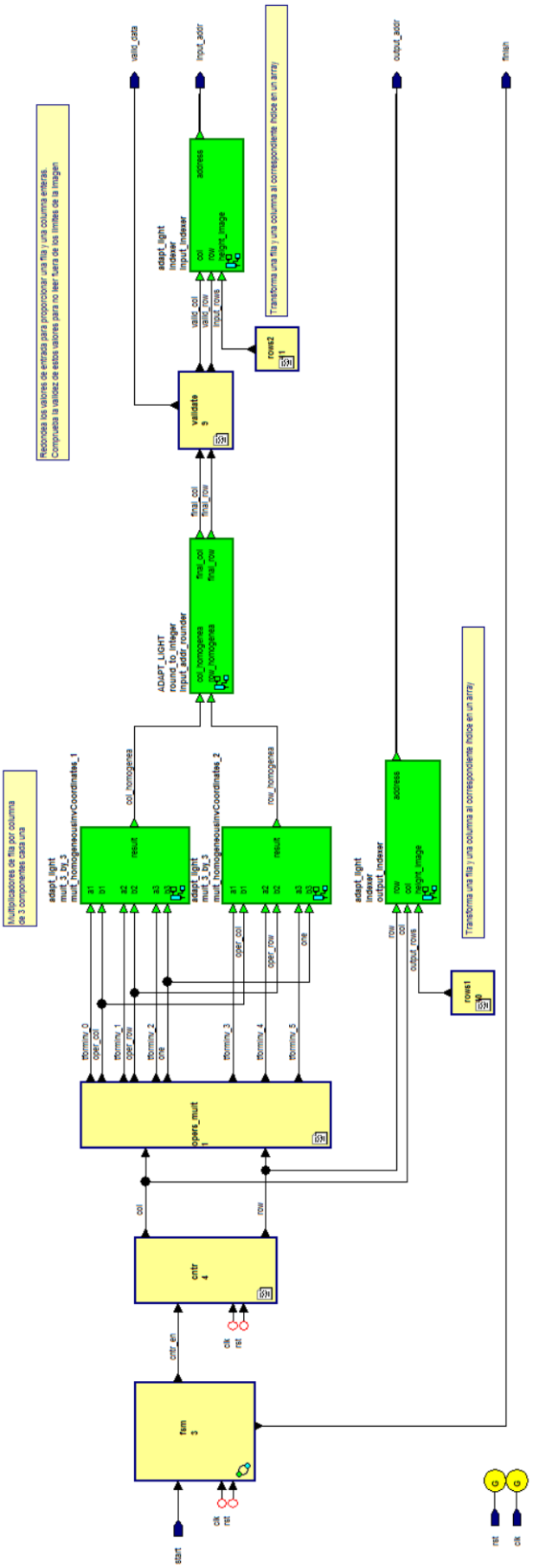

Figura 19. Diagrama de bloques del módulo address_generator.

5.3.1. Unidad de control

Es la máquina de estados que genera las señales necesarias para controlar el módulo. La figura 20 muestra el diagrama de transición de estados, y en la tabla 8 se describe cada uno de ellos en detalle:

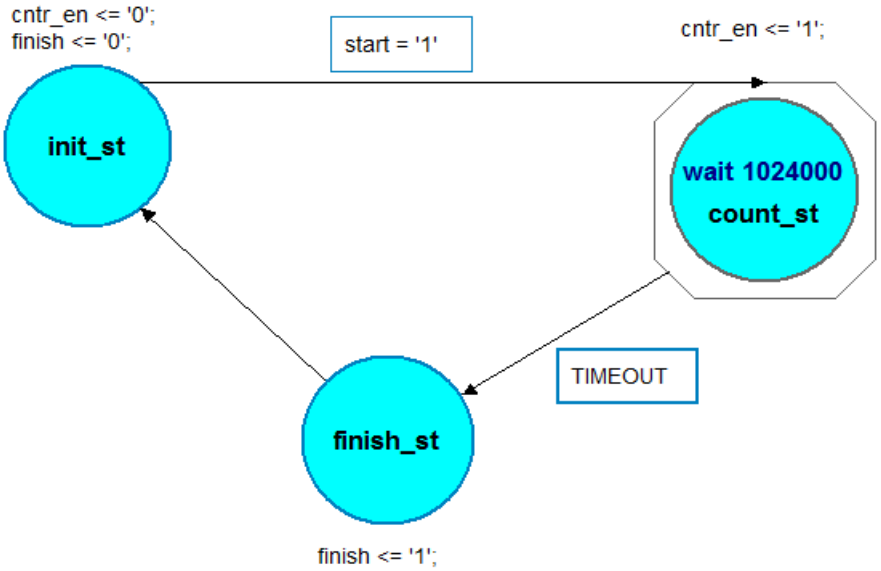

Figura 20. Unidad de control del módulo generador de direcciones.

Estado	Acción
init_st	El sistema se mantiene inactivo esperando la señal de comienzo de ejecución (start ← 1). Por seguridad se desactivan las señales de control de habilitación de cuenta (cntr_en ← 0) y de finalización del proceso (finish ← 0)
count_st	Estado que espera un número predefinido de ciclos de reloj. Debe esperar tantos ciclos de reloj como posiciones tenga la memoria de la imagen de salida IS. La imagen que almacena esta memoria tiene unas dimensiones de 800x1280 píxeles, por lo que para direccionar todas las posiciones hacen falta 1.024.000 direcciones. Es decir, la espera será de 1.024.000 ciclos de reloj. Habilita el conteo de filas y columnas que se transformarán en direcciones (cntr_en ←1)
finish_st	Una vez terminada la espera se notifica al exterior que la transformación ha finalizado activando la señal (finish ← 1). La cuenta no se desactiva hasta el ciclo siguiente, una vez que se vuelva al estado de inicio, para que el contador de filas y columnas sobrepase el límite y establezca sus valores a cero

Tabla 8. Estados de la unidad de control.

5.3.2. Generación de las filas y columnas de la imagen de salida

La máquina de estados activa la señal *cntr_en* que ataca al bloque encargado de generar las filas y columnas de la imagen de salida. La imagen se procesa por columnas, por lo que se utilizan dos condiciones *if-else* anidadas que incrementan dos contadores independientes. Como se ha comentado al explicar el funcionamiento del sistema, el contador de filas va a tomar un valor máximo de 799, mientras que el de columnas tomará como máximo 1279. Dado que la imagen se lee y almacena por columnas, el contador de filas será incrementado unidad a unidad hasta su valor máximo. En este momento, el contador de columnas se incrementará una única unidad y el contador de filas será reseteado a 0. Para implementar este bloque se ha definido los subtipos de datos:

- *subtype int1k is integer range 0 to 1279;*
- *subtype int8h is integer range 0 to 799;*

Como se puede observar los datos obtenidos en este paso son enteros, pero el cálculo de la transformada geométrica inversa necesita datos en punto fijo que nosotros hemos definido con el tipo *t_data*. Para convertir los enteros a punto fijo se aplican las siguientes expresiones:

- *oper_col <= to_signed(col, data_width) sll decimal_width;*
- *oper_row <= to_signed(row, data_width) sll decimal_width;*

La primera parte de la derecha de la expresión *to_signed(col, data_width)* convierte el entero en un vector de bits con signo de longitud *data_with*. Después el valor obtenido debe ser multiplicado por el factor *decimal_with* para convertirlo a punto fijo. En binario, una multiplicación por una potencia de dos se puede realizar mediante desplazamientos a la izquierda que es lo que hace el operando *sll*.

5.3.3. Multiplicación escalar: módulo mult_3_by_3

Una vez obtenidas las filas y columnas de la imagen de salida, y tras convertir sus tipos se tiene que aplicar la transformada geométrica inversa sobre ellas para obtener las filas y columnas de la imagen de entrada tal y como se puede ver en el ejemplo de la figura 21:

$$\begin{pmatrix} su \\ sv \\ sw \end{pmatrix} = \begin{pmatrix} 1{,}447265625 & -0.0029296875 & 0 \\ -0{,}0087890625 & 1{,}4609375 & 0 \\ -283{,}0009765625 & -105{,}62109375 & 1{,}0009765625 \end{pmatrix} \begin{pmatrix} col \\ row \\ 1 \end{pmatrix}$$

Figura 21. Esquema de la operación de multiplicación a realizar.

En esta expresión *su* y *sv* son la columna y la filas de la imagen de entrada que se asocian con la columna y fila de la imagen de salida. El valor de *sw* es innecesario en la transformación que nos ocupa, por lo que se puede desechar. Para llevar a cabo esta operación es necesario realizar dos productos escalares. El primero multiplica los elementos de la primera fila de la matriz de transformación inversa por las coordenadas espaciales homogéneas del píxel y el segundo en multiplica la segunda fila de la matriz por estas mismas coordenadas. No es necesario implementar una multiplicación completa de una matriz de 3x3 elementos por un vector de 3x1 elementos, ya que la operación de multiplicar la tercera fila de la matriz por el vector no es preciso realizarla.

La matriz de transformación es una constante de tipo t_3x3_matrix, que es un array de 9 elementos de tipo *t_data*.

La multiplicación escalar la realizan los módulos mult_3_by_3. Para ejecutar la operación se utilizan tres módulos *multiplier* que realizan cada una de las multiplicaciones necesarias de forma concurrente, cuyos resultados son sumados a posteriori. Estos módulos reciben los 6 elementos de la fila $\boldsymbol{F}$ y de la columna $\boldsymbol{C}$ a la vez, y realizan la operación: $\boldsymbol{F}_1 * \boldsymbol{C}_1 + \boldsymbol{F}_2 * \boldsymbol{C}_2 + \boldsymbol{F}_3 * \boldsymbol{C}_3$. Al ser módulos combinacionales realizan la operación en un único ciclo de reloj. Su composición interna se muestra en la figura 22:

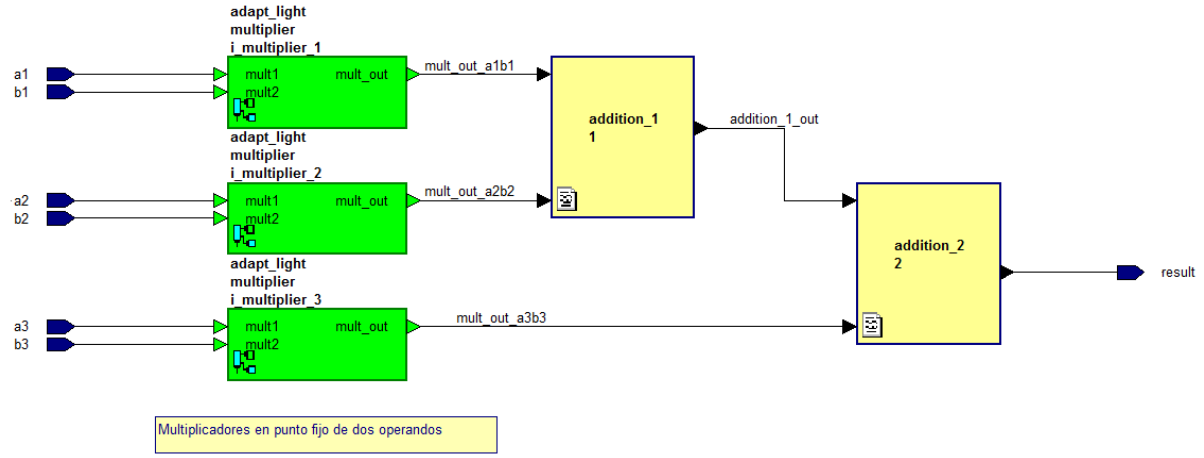

Figura 22. Diagrama de bloques del módulo mult_3_by_3.

El módulo multiplier, que se puede ver en la figura 23, realiza la multiplicación de dos números. Se compone de dos bloques, uno de ellos que realiza la operación de multiplicación propiamente dicha y el bloque round_fixed_point que realiza la corrección del resultado tal y como se explica en la sección 4.3.

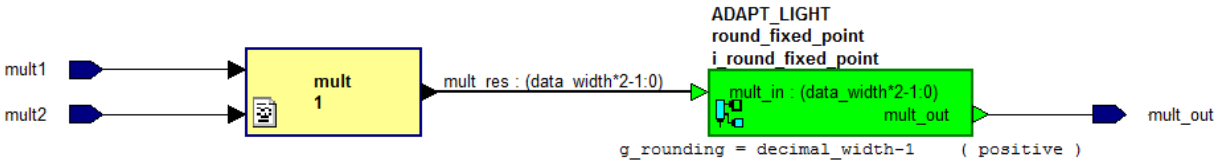

Figura 23. Diagrama de bloques del módulo multiplier.

5.3.4. Redondeo a enteros: módulo round_to_integer

En el diseño existen dos instancias del módulo mult_3_by_3, y cada una de ellos calcula una de las componentes de la coordenada de la nueva dirección de memoria. La primera calcula la columna y la segunda la fila. Pero este dato está expresado en notación $Q_{22.10}$ en punto fijo. Una vez obtenidas la fila y la columna de la imagen de entrada expresadas en punto fijo, hay que pasarlas al formato entro que es el que se utiliza para acceder a la memoria. Simplificando, habrá que dividir por el factor por el que se multiplicó en una etapa anterior. Además de esta división hay que realizar otras acciones que se explican a continuación.

1. En primer lugar se suma a las coordenadas obtenidas el offset que centra la imagen en la misma posición que la imagen golden obtenida mediante la función Imwarp, tal y como se explicó en la sección 3. A cada componente de la coordenada se le aplica un offset diferente: a las columnas se le resta el valor 292 y a las filas el valor 95.
2. La segunda etapa realiza un redondeo para no perder precisión al convertir el número a entero. Para ello, suma un 1 al bit decimal más significativo..
3. La última fase consiste en dividir el número por el factor de conversión, es decir desplazar el vector hacia la derecha tantas posiciones decimales como tenga nuestra notación (para $Q_{22.10}$ serían 10 posiciones). El vector resultante se va convierte al tipo *integer* o número entero, que es el formato que utiliza el módulo que crea las direcciones de memoria.

La figura 24 muestra las etapas en las que se divide este módulo:

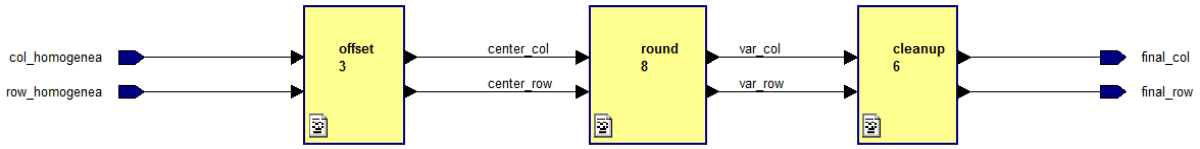

Figura 24. Diagrama de bloques del módulo round_to_integer.

5.3.5. Validación de los datos y creación de direcciones: módulo indexer

La validación de los datos sirve para saber si la columna y la fila que ha proporcionado el módulo de redondeo a entero entran dentro de los límites legibles de la imagen. En caso de que una de las componentes de la coordenada sea negativa o exceda los límites de la imagen, el dato que se lea en ese ciclo se marca como no válido. Al estar marcado, la memoria de la imagen de salida IS descarta dicho dato para no escribir nada incorrecto en la nueva imagen. Si la coordenada es válida, el dato leído si es escrito. Las coordenadas se traducen a una dirección de memoria, tanto si son válidas como si no. Esto tiene lugar en el último módulo, *indexer*.

El sistema cuenta con dos instancias de este módulo. Una de ellas recibe la columna y la fila que se ha obtenido realizando los cálculos de la transformación anteriormente explicados, y creará la dirección de memoria de la posición de la que se leerá la información del píxel de la imagen de entrada IE. La segunda recibe directamente la columna y la fila sin procesar que genera el contador de filas y columnas, explicado en la sección 5.3.2. Las direcciones que se creen a partir de ellas serán las posiciones de escritura sobre la memoria de la imagen de salida IS. Como hemos visto, la escritura se realiza por columnas de forma consecutiva, píxel tras píxel.

Como se ha comentado en el apartado de memorias, para crear la dirección hace falta la fila, la columna y la altura de la imagen, es decir, el número de filas de píxeles de las que consta. En la sección 4.1 Se explica con mayor detalle los cálculos que se realizan.

El flujo que siguen los datos se puede ver en la figura 25:

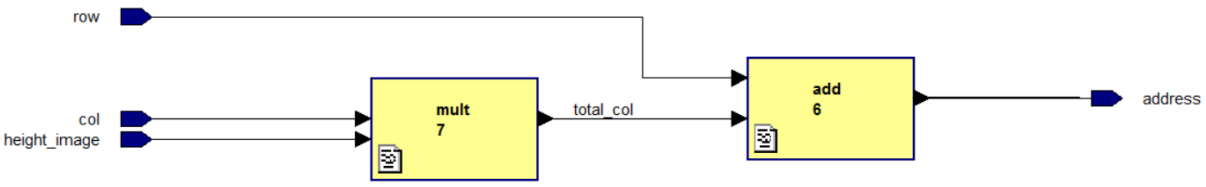

Figura 25. Diagrama de bloques del módulo indexer.

5.4. Módulos comunes

En el proceso de escritura y lectura de datos de las memorias, hace falta transmitir a las memorias las señales de control necesarias para leer o escribir el dato en una determinada dirección. La memoria que almacena la imagen de entrada tarda un ciclo de reloj en proporcionar el dato de la dirección que se le solicita, pero el módulo generador de direcciones produce todas las señales de control al mismo tiempo, por lo que es necesario retrasarlas un ciclo. Para ello se usan dos módulos sencillos:

- *Flip-flop (ff)*: registro síncrono que reproduce la entrada por el puerto de salida cuando ha transcurrido un ciclo de reloj. La señal de entrada y la de salida son de 1 bit cada una.
- *Registro (reg_t_data)*: el funcionamiento es el mismo al del flip-flop, manda el valor presente en el puerto de entrada por el de salida al transcurrir un ciclo de reloj. En este caso, los puertos de entrada y salida son de tipo *t_data*, es decir, son vectores con signo de un ancho de 32 bits cada uno.

La figura 26 presenta el diagrama de bloques de ambos componentes:

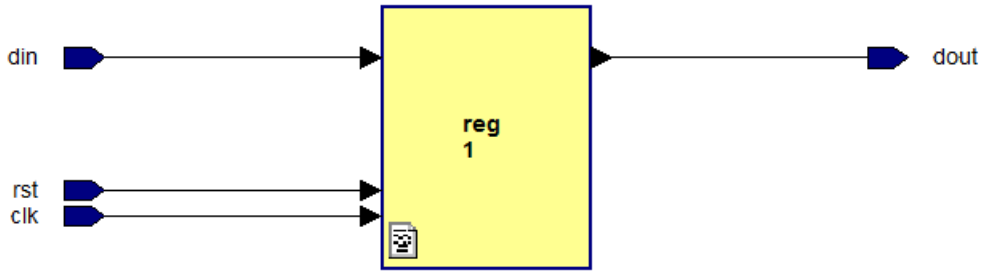

Figura 26. Diagrama de bloques de los módulos ff y reg_t_data.

Capítulo 6
Síntesis

La herramienta de síntesis se configuró con los siguientes parámetros:

- Family: virtex6
- Device: xc6vlx240t
- Package: ff1156
- Speed: -1

La herramienta proporciona un informe de síntesis del que se extraen los recursos de la placa que consumiría el diseño. Los recursos se muestran en la tabla 9:

Nombre	Número de módulos
Multiplicadores	8
Sumadores/Restadores	16
Contadores	3
Comparadores	4
Multiplexores	5
FSMs	1
Registros	43

Tabla 9. Recursos de la FPGA consumidos por la implementación.

Además se obtiene información de la frecuencia y de la temporización del sistema:

- Periodo mínimo: 3,558ns (Frecuencia máxima: 281,057MHz)
- Periodo mínimo de llegada de señales de entrada antes del reloj: 1,840ns
- Periodo máximo necesario para señales de salida tras el reloj: 26,060ns
- Máximo retardo combinacional: No se ha encontrado camino crítico

Los requisitos del sistema especificaban que debía ser capaz de procesar 30 imágenes o *frames* por segundo. Para procesar cada *frame* hacen falta 800 por 1280, 1.024.000 ciclos de reloj, sin contar los de la recepción de las señales de control. Para 30 imágenes harán falta 30.720.000 ciclos de reloj. Si el sistema funciona a su frecuencia máxima, 281,057 MHz, cada ciclo de reloj tendría un periodo de:

$$\frac{1}{281.057.000\ MHz} = 3{,}55799 \cdot\ 10^{-9} s = 3{,}56\ ns$$

Calcular 30 imágenes de dicha resolución tomaría al sistema:

$$30\ frames \cdot 1.024.000\ píxeles \cdot 3{,}55799 \cdot 10^{-9}\ s = 0{,}1093\ s$$

Apenas una décima de segundo, lo cual evidencia la alta capacidad de procesamiento de la implementación.

Capítulo 7
Resultados Experimentales

En este capítulo se detallan los resultados obtenidos mediante las diferentes implementaciones del sistema, tanto software como hardware.

La imagen en MATLAB se creó mediante tres métodos diferentes.

1. Mediante la función Imwarp. La finalidad de este método es obtener una imagen golden que sirva de referencia para comparar los resultados del resto de implementaciones con ella.

2. Mediante la función transformPointsInverse, de la toolbox de procesado de imágenes de MATLAB. En esta implementación se descompuso el comportamiento de la función Imwarp a un menor nivel.

3. Implementando directamente las operaciones matriciales de la transformación geométrica proyectiva en código MATLAB. Esta descomposición estaba compuesta por operaciones suficientemente sencillas como para ser traducidas a VHDL.

En las siguientes figuras se presentan las imágenes generadas por las tres implementaciones.

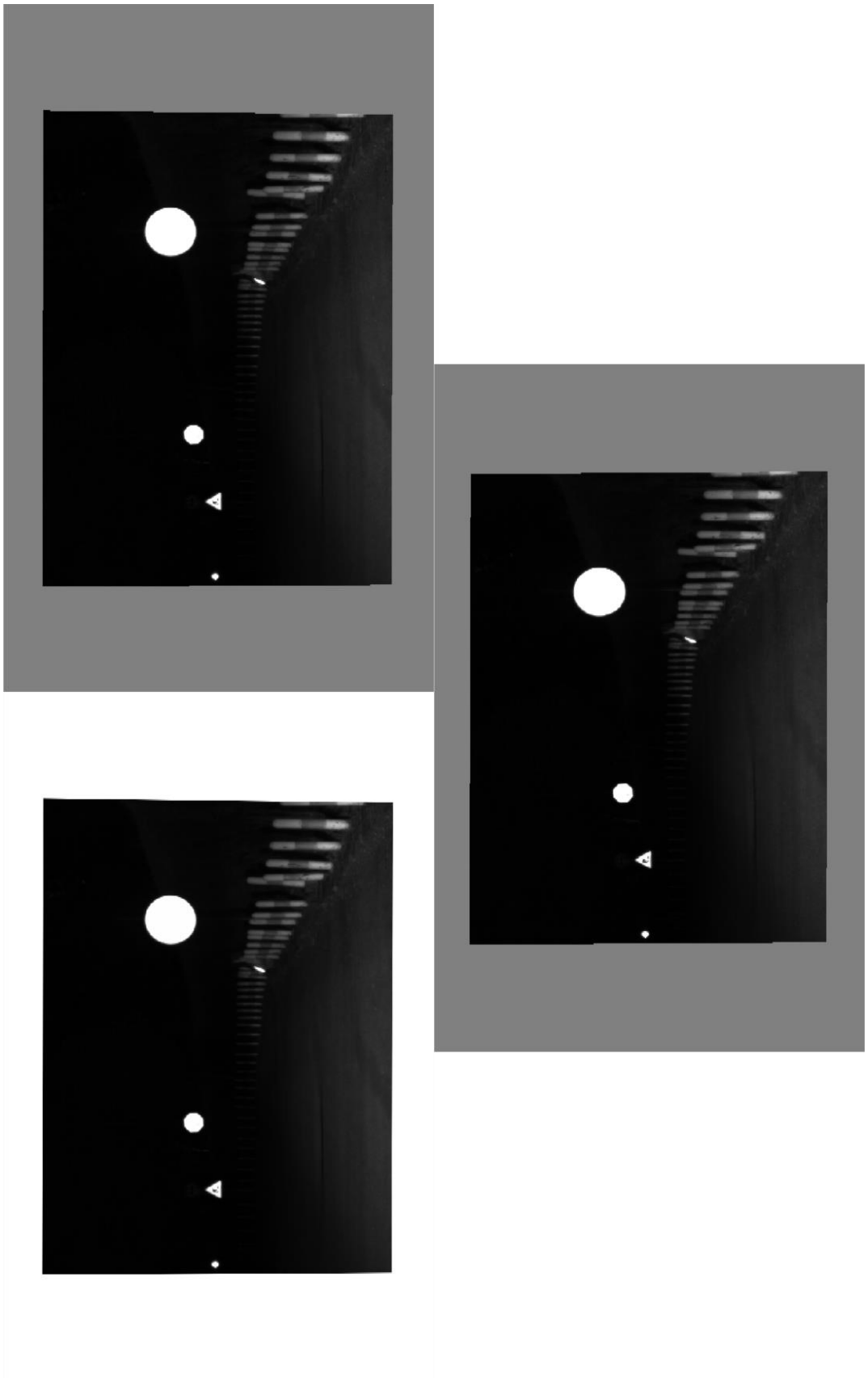

Figura 27. Resultados obtenidos con cada implementación MATLAB: Imwarp, transformPointsInverse y la descomposición realizada respectivamente.

Para ver las diferencias que existen entre ellas, se utilizó la función Imshowpair de MATLAB, con el parámetro *'diff'*, que superpone dos imágenes y resalta las zonas donde no coincidan. A continuación se muestran dos de estas comparaciones:

1. Comparación entre el resultado de la función Imwarp y el de la función transformPointsInverse (figura 28). La ausencia de una mayor área resaltada en blanco es consecuencia del resultado tan similar que han proporcionado ambas funciones.

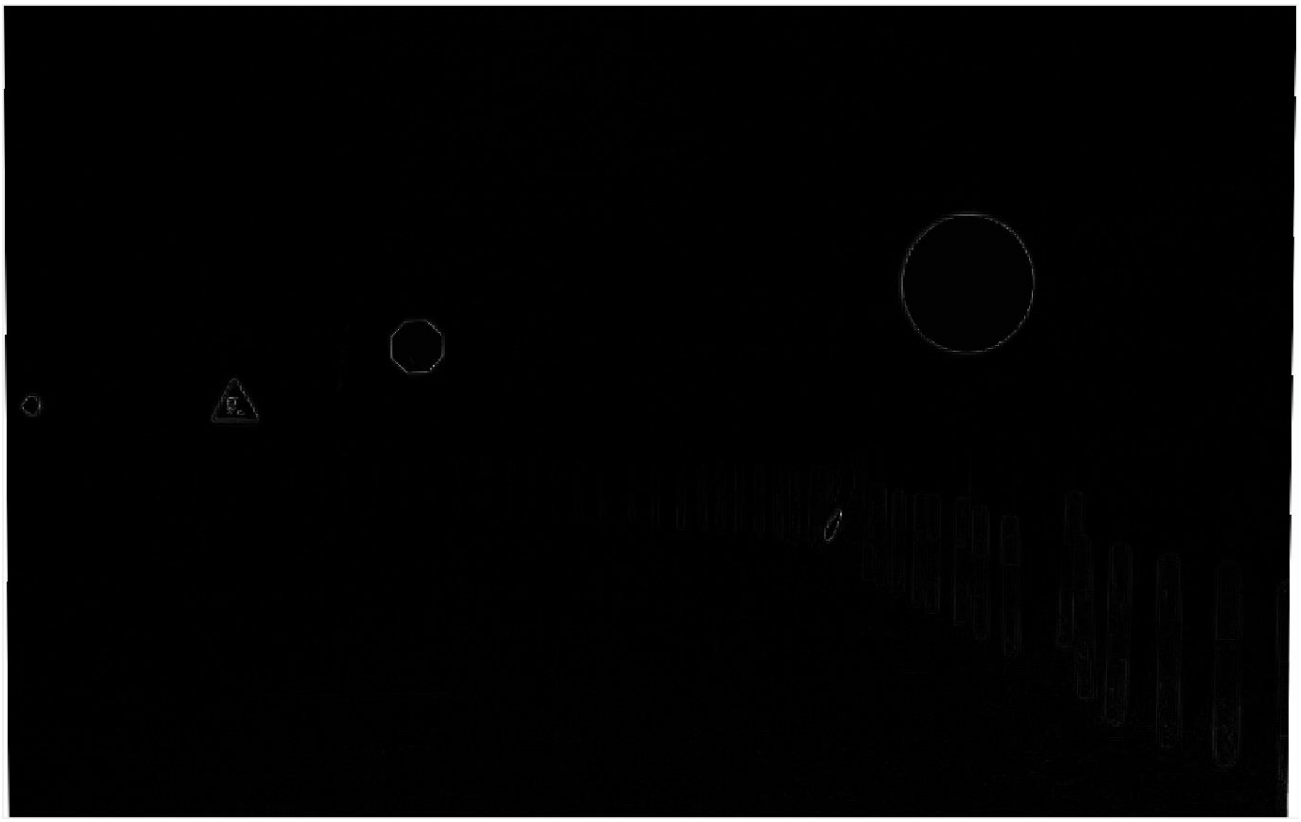

Figura 28. Diferencia entre resultados de Imwarp y transformPointsInverse.

2. Comparación entre el resultado de la función Imwarp (tomada como imagen golden) y el de la implementación realizada en código MATLAB (figura 29).

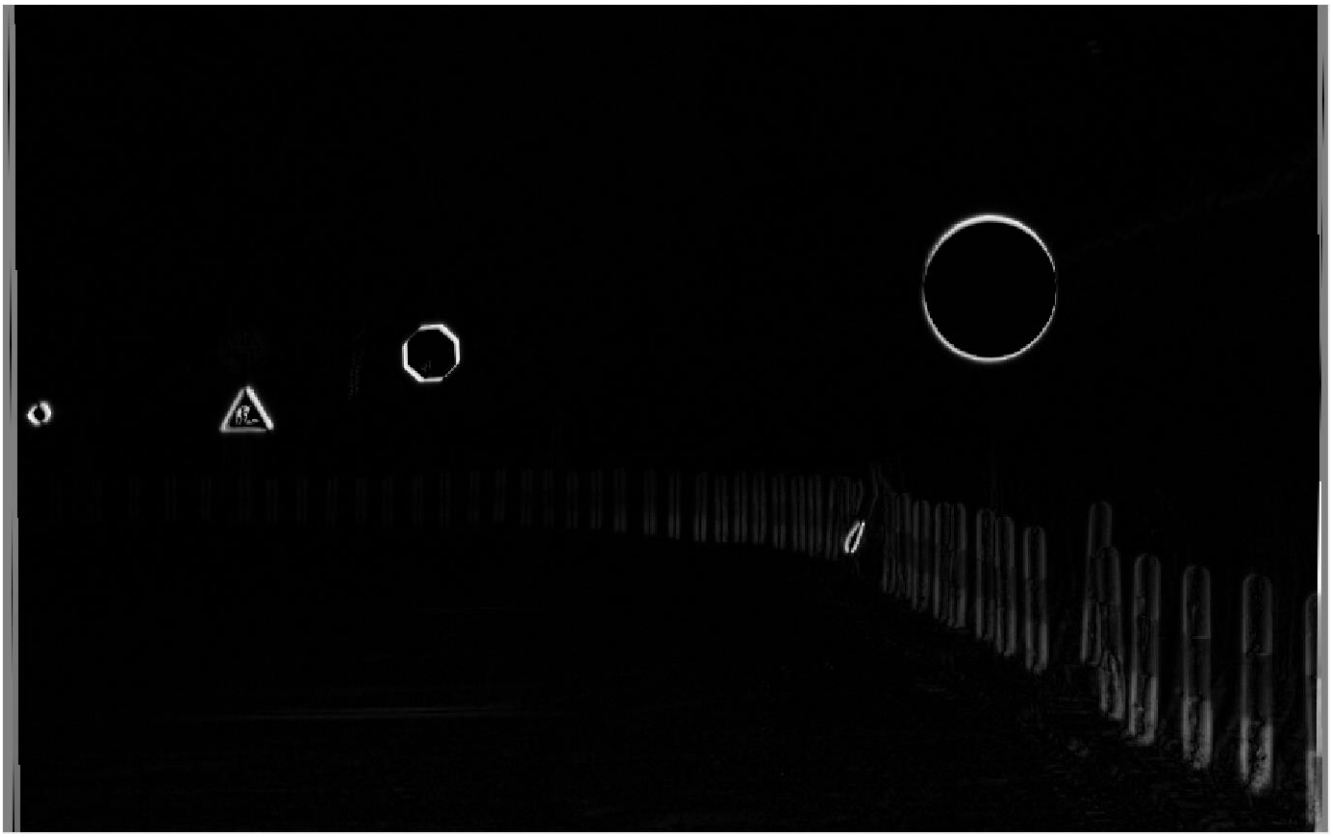

Figura 29. Diferencia entre resultados de Imwarp y la descomposición realizada.

A pesar de que las diferencias entre la imagen que produce la implementación realizada y la que produce la propia función Imwarp son apreciables, se consideró un margen de error válido para pasar a implementar en hardware, ya que el sistema de iluminación no enfocaría toda la superficie de la señal, por lo que no existiría peligro de alumbrar zonas no deseadas.

Tras realizar la implementación en VHDL, se procedió a la creación de un archivo de testbench con la herramienta HDL Designer. Este archivo contiene las memorias que hemos mencionado en capítulos anteriores, las cuales también realizan la lectura de los datos desde un fichero y la escritura de los resultados en otro. Para interpretar estos datos se procedió a crear un script en MATLAB que tomase el fichero donde se habían volcado los resultados, leyese su contenido y lo transformase en una imagen. Mediante el mismo método de comparación, podemos observar el resultado obtenido en hardware en comparación con la imagen golden de Imwarp (figura 30):

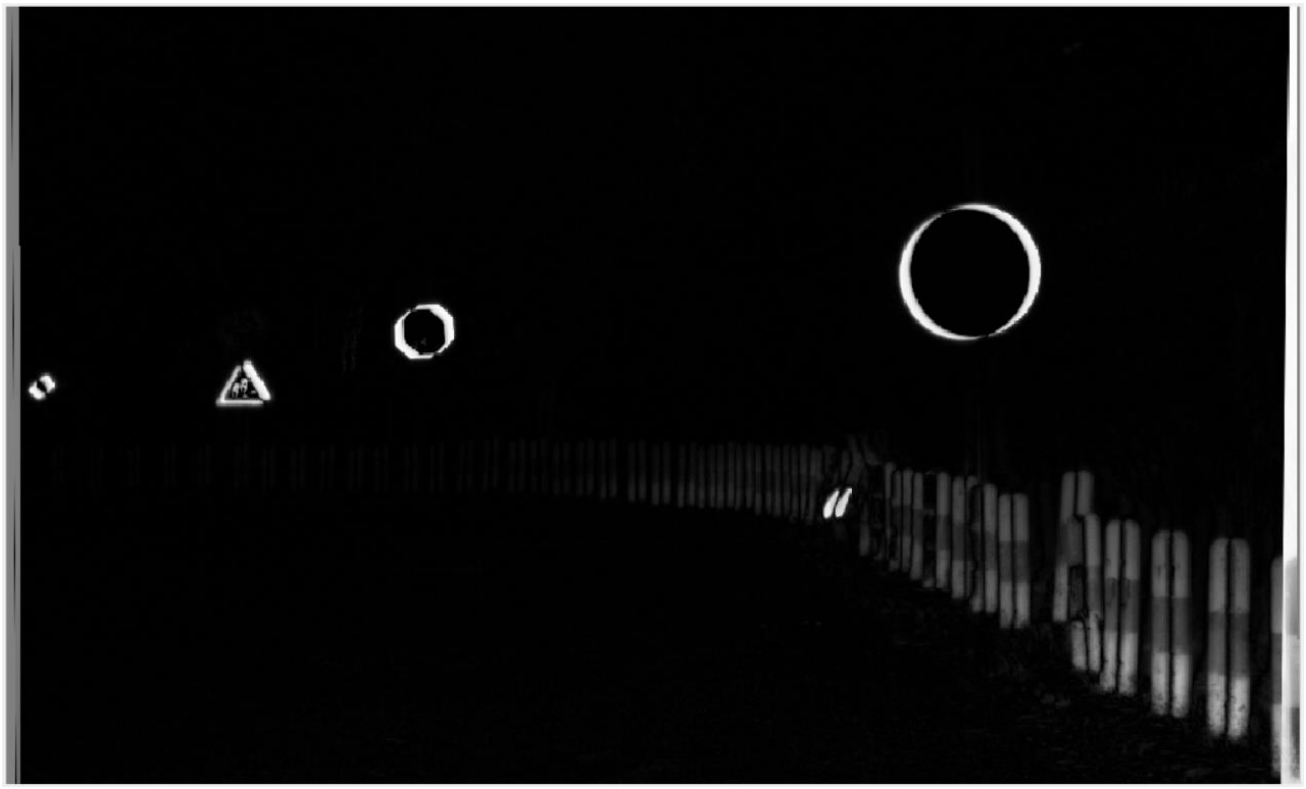

Figura 30. Diferencia entre resultados de Imwarp y la implementación hardware realizada.

Parte III
Conclusiones

Capítulo 8
Conclusiones

Este proyecto nace con la idea de complementar los sistemas actuales de seguridad de los automóviles mediante el uso de la Iluminación Adaptativa. Los sistemas actuales iluminan la vía de diferentes formas: a baja velocidad con un amplio área de luz; a mayor velocidad rotando los faros del coche levemente; alternando entre luces largas o cortas dependiendo de si el sistema detecta a otros usuarios de la vía que puedan ser deslumbrados.

El sistema propuesto por el grupo de investigación de la Facultad de Óptica no fue pensado como un sustituto de ellos, sino que los complementase y añadiese una funcionalidad nueva. Este sistema hace uso de las imágenes que una cámara graba de la vía frente al coche, con el objetivo de detectar las formas geométricas de las señales de tráfico y resaltar su superficie. A partir de estas imágenes, el sistema realiza una transformación geométrica que relaciona la imagen en la resolución que produce la cámara con una imagen a la resolución que pueda tratar el proyector de luz. Cuando el proyector obtiene una imagen válida, es capaz de irradiar luz en la dirección adecuada para alumbrar únicamente la superficie la señal de tráfico.

El sistema trata cada imagen píxel a píxel. Para cada posición de una imagen de salida se calcula qué posición de la imagen de entrada le corresponde, mediante la operación de transformación geométrica proyectiva. Para su implementación son necesarias dos memorias que almacenen las imágenes, así como una serie de módulos que realicen operaciones aritméticas en punto fijo.

El correcto funcionamiento del sistema ha sido verificado en las dos etapas principales del proyecto:

1. Mediante la implementación de las operaciones a realizar en código MATLAB, comparando los resultados con los obtenidos al utilizar la función Imwarp de la toolbox de procesamiento de imágenes.
2. Mediante la simulación del código VHDL escrito, utilizando un testbench que volcase los resultados a un fichero. Este fichero podía ser leído, la información que contenía ser transformada en una imagen, y ésta a su vez, ser comparada de la misma forma con el resultado de la función Imwarp.

La similitud entre las imágenes obtenidas fue prueba suficiente de que el sistema había sido implementado correctamente. Tras el análisis del informe de síntesis, que estimaba una frecuencia de trabajo máxima de 281,057 MHz, se calculó que el sistema sería capaz de procesar al menos 30 imágenes de la resolución indicada por segundo, requisito principal al que se enfrentaba el proyecto. En caso de que sea necesario aumentar el número de imágenes por segundo a tratar o modificar la resolución de éstas, se puede adaptar la implementación de forma sencilla y seguir satisfaciendo los requisitos dada su alta capacidad de procesamiento.

Capítulo 9
Conclusions

This project originated from the idea of complementing current security systems of automobiles with by using Adaptive Lighting. Current systems light the road in different ways: at low speeds with a broad spotlight; at higher speeds by turning the car headlights in the direction the car is turning to; by switching between low and high beams depending on whether the system detects any other road user in the vicinity who can be dazzled.

The system proposed by the researchers at the Facultad de Óptica was not conceived as a substitute for them, but rather as a complementary system which would add new functionality. This system makes use of pictures of the road taken by a camera which sits at the front of the car. It detects the geometric shape of a road signal and illuminates its surface. From these images, the system performs a geometric transformation that adapts the image of a certain resolution taken by the camera, into a picture with a resolution that the light projector can work with. When the projector gets a valid picture, it is able to illuminate only the surface of the traffic signal.

The system processes each image pixel by pixel. It calculates which position in an input image corresponds to each position of an output image, through the projective geometric transformation operation. Two memories to store the pictures as well as a series of modules that perform fixed-point arithmetic operations are needed for its implementation.

The proper operation of the system has been verified in both main stages of the project:

1. Through the implementation of the operations in MATLAB code, by comparing the results obtained with the results provided by the function Imwarp, of the image processing toolbox
2. Through simulations of the VHDL code implemented, by using a testbench that wrote the results to a text file. This file was read and its contents could then be transformed into a picture, which in turn could be compared in the same way with the results from the Imwarp function.

The similarity between the images was considered proof of a correct implementation. After analysing the synthesis report, which estimated the system's maximum working frequency at 281.057 MHz, it was concluded that the system would be able to process at least 30 frames per second, which was the main objective this project had to tackle. If

necessary, the number of frames per second to process of the resolution of the pictures can be changed easily in the implementation, and still increasing its workload, the system will be able to meet the objectives because of the high performance it has achieved.

Parte IV
Apéndice

Apéndice A
Herramientas utilizadas

En este apéndice se listan las herramientas utilizadas por el autor en el desarrollo de este proyecto.

- Desarrollo de implementaciones software y comparación de imágenes: Mathworks MATLAB R2015a
- Entorno Unix: Cygwin 2.4.1
- Cliente Git: SyntEvo SmartGit Version 7.1
- Diseño de hardware: Mentor Graphics HDL Designer Version 2015.1b
- Simulación del sistema: Mentor Graphics Questa Sim 10.1d
- Documentación del proyecto: Google Drive
- Escritura de la memoria: Google Docs

Agradecimientos

ESTE TRABAJO HA SIDO FINANCIADO POR LA CÁTEDRA VALEO-UCM DE LA FACULTAD DE ÓPTICA Y OPTOMETRÍA. DEPARTAMENTO DE ÓPTICA

Bibliografía

BEBIS, G., 2D/3D Geometric Transformations, *University of Nevada, Reno*, 2010

BERLITZ, S., Lighting roadmap 2020, *International Symposium on Automotive Lighting,* 2013

CAPELLÁN, F., PUICA S. F., SÁNCHEZ, D., GARNICA, A. O., LANCHARES, J. Diseño e implementación en FPGA de un filtro Kalman para aplicaciones biomédicas, *Trabajo de Fin de Grado en Ingeniería de Computadores (Universidad Complutense, Facultad de Informática, curso 2014/2015)*

GONZÁLEZ, R. C., WOODS, R. E., EDDINS, S. L., Digital Image Processing using MATLAB 2nd edition, *Gatesmart Publishing*, 2009

HARTLEY, R., ZISSERMAN, A., Multiple View Geometry in Computer Vision 2nd edition, *University of Oxford*, 2003

HEL-OR, Y., Geometric Operations and Morphing, *Interdisciplinary Center Herzliya*, 2010

HU, Y. H., Image Geometry and Geometric Transformation, ECE533 Digital Image Processing, *University of Wisconsin-Madison*, 2002

MILANOVA, M., Geometric Transformations, *University of Arkansas at Little Rock*, 2004

MICHENFELDER, S., NEUMAN, C., Pixel Lighting - An Automotive Lighting Research Head Lamp, *10th International Symposium on Automotive Lighting, Darmstadt,* 2013

RUBIO, M., ROMO, J., VÁZQUEZ, D., FERNÁNDEZ-BALBUENA, A. A., GARCÍA-BOTELLA A., Selective Automotive Lighting System, *11th International Symposium on Automotive Lighting, Darmstadt*, 2015

WALBERG, G., Geometric Transformation Techniques for Digital Images: a survey, *Technical Report CUCS-390-88*, 1988